Python 101 Documentation

Wydanie 0.5

Centrum Edukacji Obywatelskiej

22 maj 2022

Spis treści

Niniejsze materiały to poprawiona i uzupełnionan dokumentacja do szkoleń z języka Python realizowanych w ramach projektu Koduj z Klasą prowadzonych przez Fundację Centrum Edukacji Obywatelskiej w latach 2014-2017.

Początkowe materiały zakładały wykorzystanie języka Python w wersji 2. W wersji obecnej wszędzie, gdzie to możliwe, używamy Pythona 3.

ROZDZIAŁ 1

Pobieranie dokumentacji

- Wersja HTML offline
- Wersja źródłowa w GitHubie

ROZDZIAŁ 2

Przygotowanie do szkoleń

Jeżeli na szkoleniach chcesz wykorzystywać swój komputer, musisz go przygotować.

2.1 System i oprogramowanie

Nasze materiały zakładają wykorzystanie języka Python w większości w wersji 3.x, w dwóch przypadkach (*Gra robotów* i częściowo *Minecraft Pi*) wymagana jest wersja 2.x. Mogą być realizowane w dowolnym systemie operacyjnym, jednak proponujemy systemy Linux, w których Python 3.x i często 2.x są obecne domyślnie i nie ma problemów z instalacją dodatkowych narzędzi i bibliotek.

Do realizacji materiałów można również wykorzystać system *Linux Live* przeznaczony do instalacji na pendrajwach. Uruchamia się z napędu USB na większości komputerów i ma możliwość zapamiętywania zmian i naszej pracy. Podczas realizacji scenariuszy wykorzystujących Pythona będziemy korzystać z różnych narzędzi:

- pip – instalator pakietów Pythona, podstawowe narzędzie służące do zarządzania pakietami Pythona zgromadzonymi w repozytorium PyPI (Python Package Index);
- git – konsolowy klient systemu wersjonowania kodu umożliwiający korzystanie z repozytoriów w serwisie Github;
- sqlite3 – konsolowa powłoka dla baz SQLite3, umożliwia przeglądanie schematów tabel oraz zarządzanie bazą za pomocą języka SQL;
- ipython i qtconsole – rozszerzone interaktywne konsole Pythona.

Na kolejnych stronach wyjaśniamy, jak je zainstalować i wykorzystywać w systemie operacyjnym.

2.1.1 Przygotowanie systemu Linux

Jeżeli nie masz zainstalowanego systemu Linux, możesz wykorzystać wersję *Linux Live*. Jeżeli masz Linuksa lub planujesz go zainstalować na dysku, czytaj dalej.

Dystrybucje

Najwygodniej pracować w systemie Linux zainstalowanym na dysku twardym, np. obok albo zamiast MS Windows. Polecamy dystrybucje oparte na Debianie, na których przetestowaliśmy scenariusze:

- Linux Mint 20.04
- Ubuntu 20.04 LTS.
- MX Linux 21

Środowisko graficzne (zob. środowisko graficzne) dowolne.

Wskazówki dotyczące instalacji:

- Windows i Linux na jednym dysku;
- Zainstaluj Linuksa;
- Instalacja Ubuntu;
- Linux Mint Installation Guide

Interpreter, narzędzia i pakiety

W Linuksach interpreter Pythona 3.x zainstalowany jest domyślnie. Wymagane pakiety Pythona i/lub wersję Pythona 2.x, a także narzędzia dodatkowe w razie potrzeby instalujemy za pomocą systemowego menedżera pakietów `apt`. Pakiety można również instalować przy użyciu instalatora pakietów Pythona pip.

Informacja: Polecenie `sudo` oznacza, że do instalacji potrzebne są uprawnienia administracyjne, czyli w praktyce należy być zalogowanym na koncie użytkownika utworzonym podczas instalacji systemu.

- Aktualizacja bazy oprogramowania i instalacja podstawowych narzędzi:

```
~$ sudo apt update
~$ sudo apt install python3-pip python3-venv git sqlite3
```

- Ogólnosystemowa instalacja rozszerzonych powłok:

```
~$ sudo apt install python3-qtconsole python3-tk python3-sip python3-pyqt5
```

- Ogólnosystemowa instalacja dodatkowych pakietów:

```
~$ sudo pip3 install matplotlib
~$ sudo pip3 install pygame
~$ sudo pip3 install flask flask-wtf peewee sqlalchemy flask-sqlalchemy␣
↪django
```

Wskazówka: Zamiast ogólnosystemowej instalacji rozszerzonych powłok i pakietów zalecamy instalację w *środowisku wirtualnym* dostępną dla zwykłego użytkownika.

Informacja:

- Nazwy pakietów w różnych dystrybucjach mogą się nieco różnić od podanych.

- System *Debian* w domyślnej konfiguracji nie wykorzystuje mechanizmu podnoszenia uprawnień `sudo`, wtedy polecenia instalacji należy wydawać z konta użytkownika *root*.

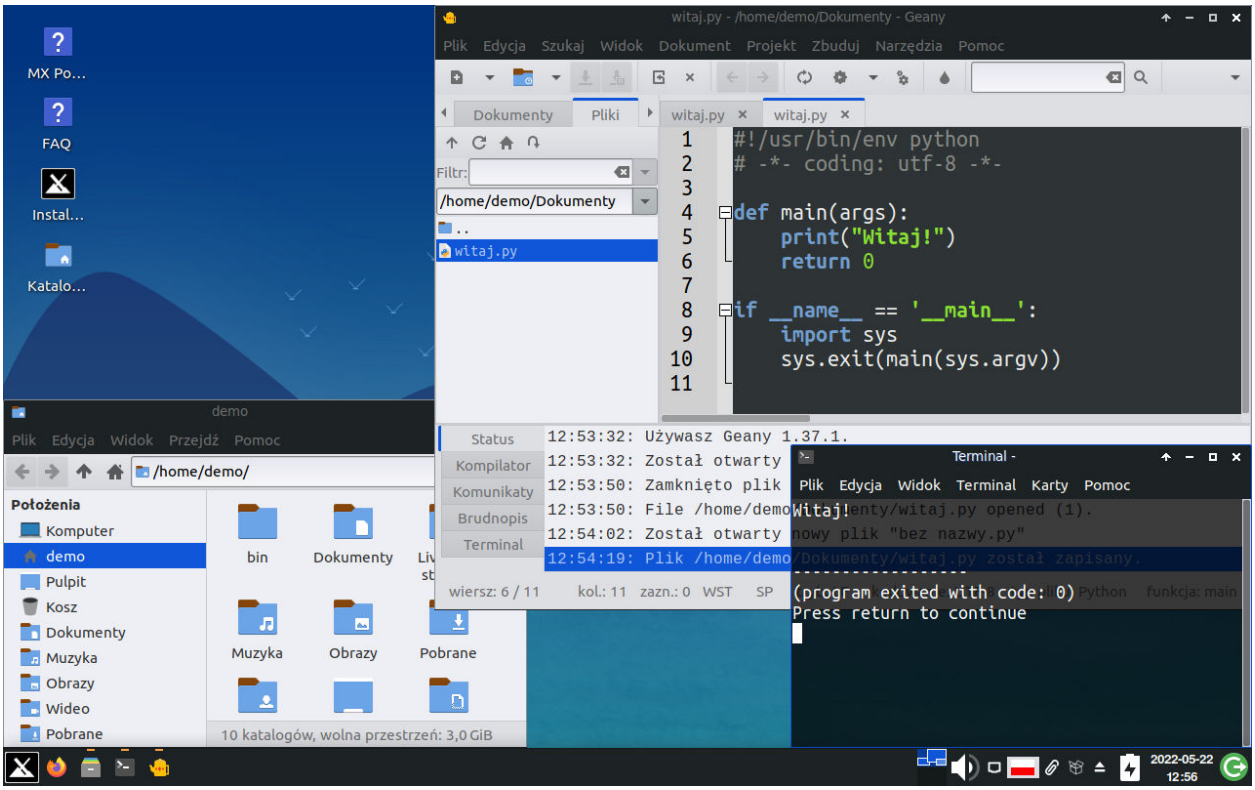
Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.1.2 Linux Live USB

Klucz USB z systemem w wersji *live* pozwala na uruchomienie komputera, testowanie i pracę bez ingerowania w dane na twardym dysku (np. inne systemy). Dystrybucje *live* można zainstalować w maszynie wirtualnej lub na dysku twardym.

Do realizacji scenariuszy i codziennej pracy, dla nauczycieli i uczniów proponujemy dystrybucję **MX Linux** (opartą na Debianie). System zawiera wszystkie wymagane narzędzia, umożliwia doinstalowywanie programów i pakietów, po wstępnej konfiguracji pozwala na zapisywanie ustawień, skryptów i dokumentów.

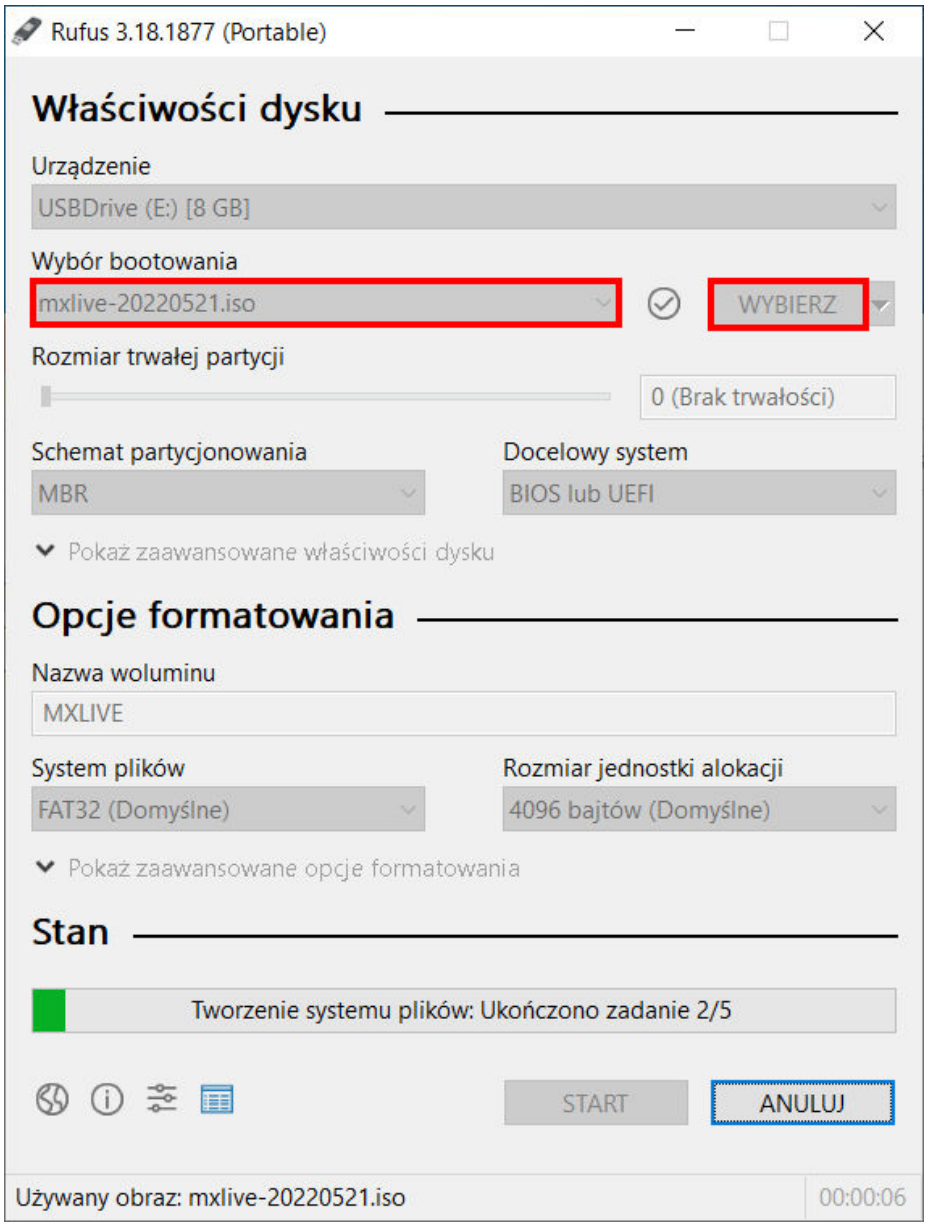


Rys. 2.1: MXLinux 22.01 XFCE 64-bit

W systemie Windows

- Pobieramy obraz iso MXLinux2201.iso (2,49GB).
- Pobieramy program Rufus.
- Wpinamy pendrajwa o pojemności min. 4 GB.

- Uruchamiamy *Rufusa*, upewniamy się, że na liście "Urządzenie" wybrany jest właściwy pendrajw, klikamy przycisk "Wybierz" i wskazujemy ściągnięty obraz iso. Klikamy "Start" i czekamy na napis "Gotowe".

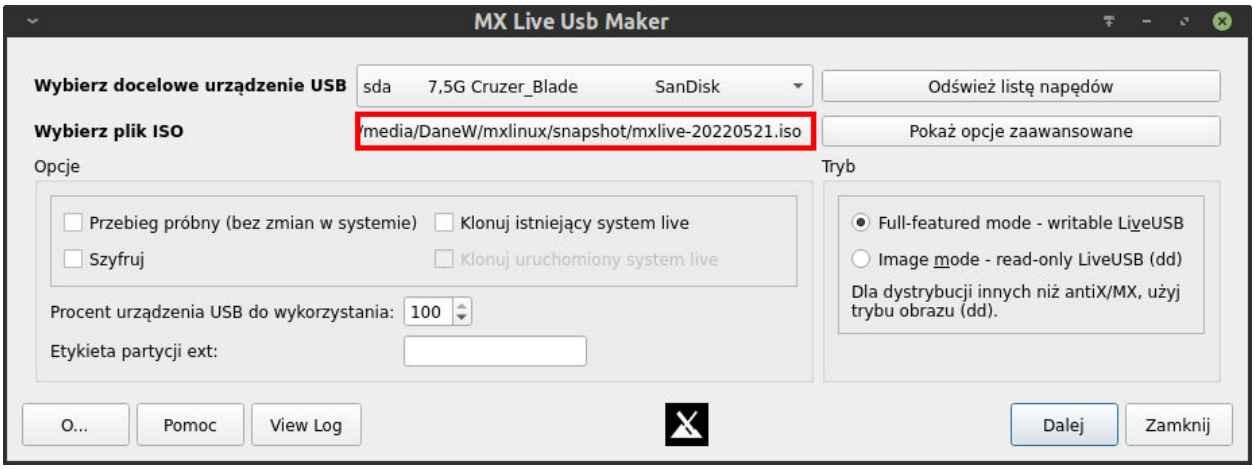


W Linuksie

- Pobieramy obraz iso MXLinux2201.iso (2,49GB).
- Pobieramy archiwum programu live-usb-maker-qt-21.11 appimage i rozpakowujemy go w dowolnym katalogu.
- Wpinamy pendrajwa o pojemności min. 4 GB.
- Rozpakowany program uruchamiamy w terminalu:

```
~$ sudo ./live-usb-maker-qt-21.11glibc2.28-x86_64.AppImage
```

- Po uruchomieniu programu klikamy przycisk “Wybierz” i wskazujemy ściągnięty obraz iso. Klikamy “Dalej” i czekamy na nagranie obrazu.



MX Linux Live USB

Login i hasło domyślnego użytkownika to: **demo**.

Ustawienie języka

Po uruchomieniu komputera z przygotowanego klucza USB zobaczymy menu startowe bootmenedżera:

- wybieramy pozycję **Language - Keyboard - Timezone / Language / lang=pl_PL: Polski - Polish**.
- wracamy do głównego menu **Back to main menu**, wybieramy **Advanced Options / Save options: / grubsave Save options (LiveUSB only) -> GRUB menu**. Dzięki temu ustawienia języka zostaną zapamiętane.

Zapamiętywanie zmian

MX Linux Live USB może być aktualizowany, można również instalować w nim dodatkowe programy, np. środowiska IDE do programowania. Zmiany mogą być zapamiętywane po włączeniu odpowiednich opcji *persistence* dostępnych w menu startowym bootmenedżera: **Advanced Options / Persistence option:**. Możemy zapamiętywać zmiany w systemie (**root**) lub **/** i w katalogu użytkownika (**home**).

- Proponujemy wybrać **persist_all**, następnie wracamy do głównego menu i uruchamiamy system.
- Podczas startu wyświetlą się prośby o utworzenie plików *rootfs* i *homefs*, w których zapisywane będą zmiany. W zależności od rozmiarów klucza USB można zaakceptować rozmiary domyślne lub wybrać niestandardowe, np. 2GB.
- Na ewentualne pytanie, czy utworzyć *live-usb swap file* (plik wymiany) możemy odpowiedzieć “nie”.
- Na ewentualne pytanie, czy skopiować pliki do *home persistence* odpowiadamy “tak”.
- W razie potrzeby podajemy nowe hasła dla użytkownika *root* i *demo*.

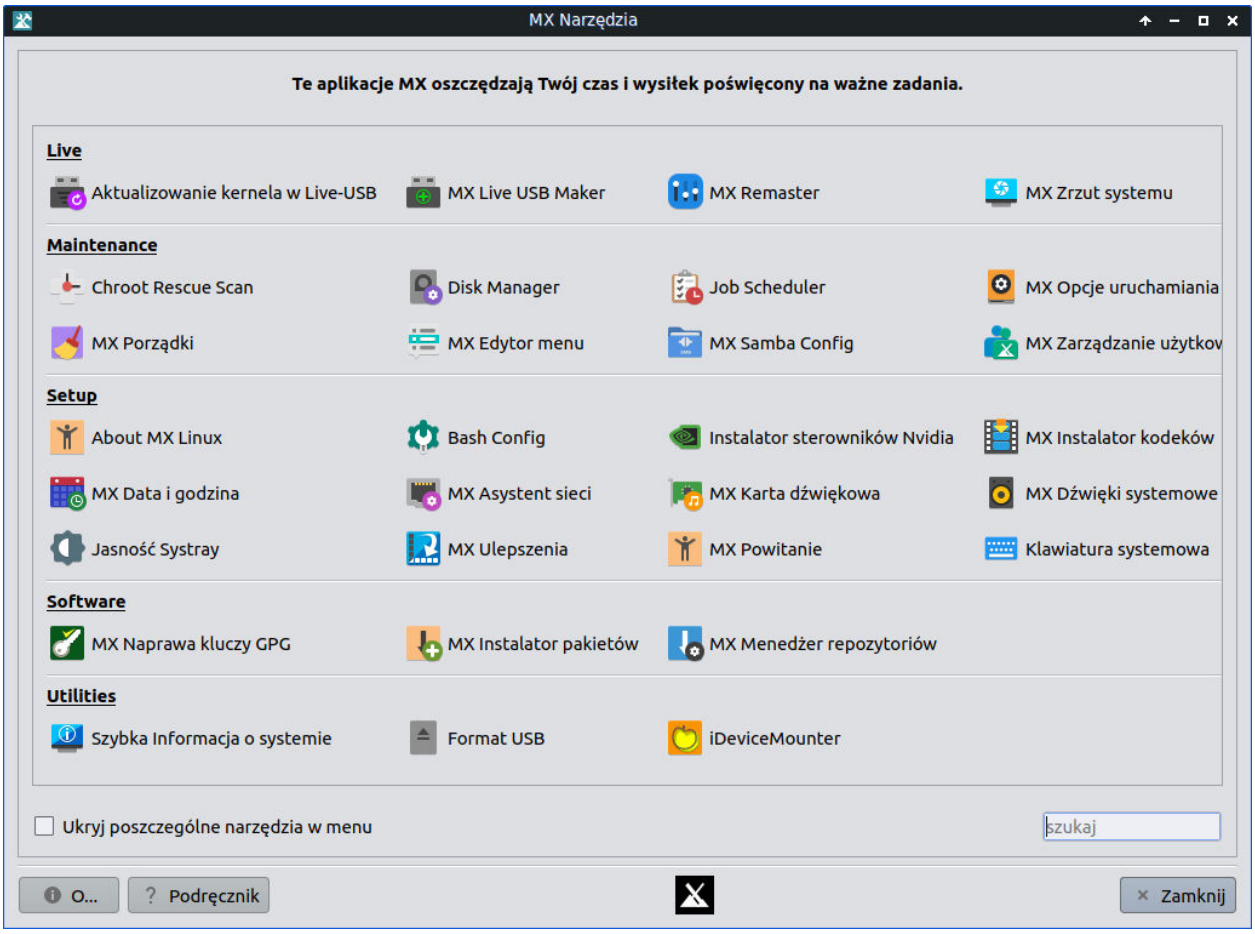
Informacja: W zależności od sposobu utworzenia klucza USB z Linuksem Live maksymalny rozmiar plików przechowujących zmiany może być ograniczony przez wykorzystywany system plików, np. FAT32 obsługuje pliki do 4GB. Jeżeli korzystamy z systemu EXT4, ogranicza nas tylko rozmiar klucza USB.

Niezależnie od trybu *persistence* pliki zapisane w katalogu *Live-usu-storage* zapisywane są na pendrajwie.

Remastering

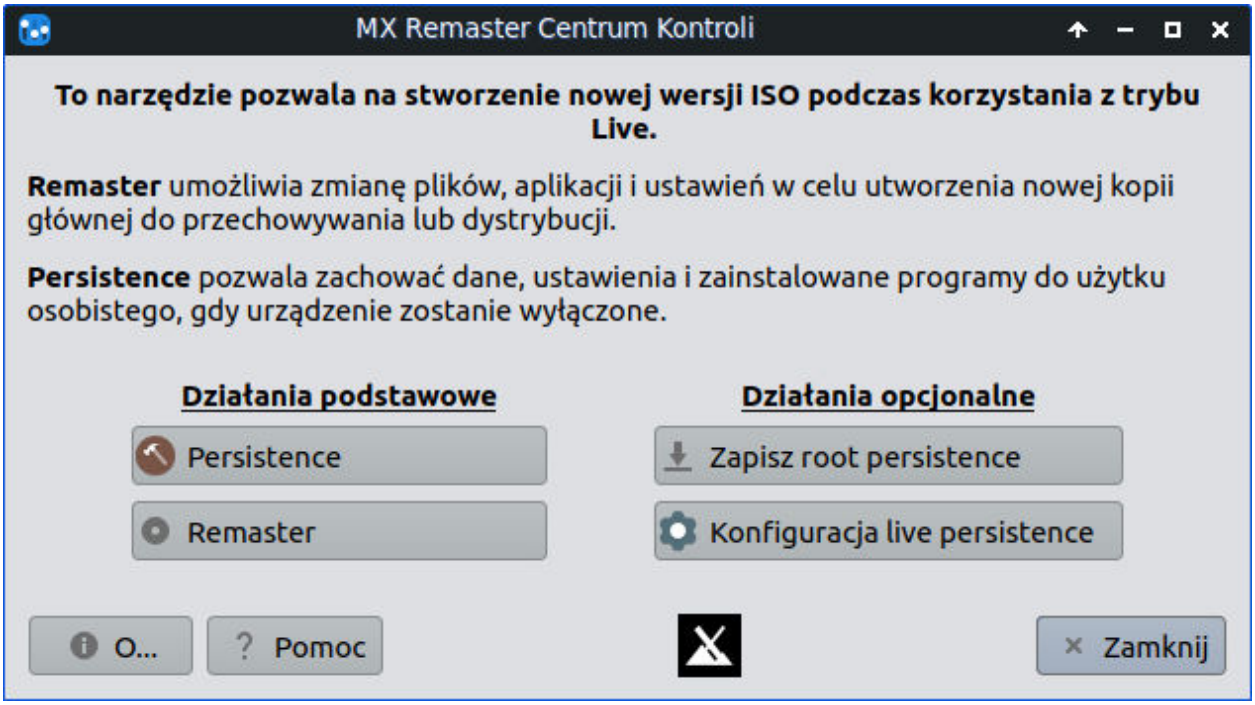
Opcja *persistence* uwzględniająca zmiany w *root*, czyli utworzenie pliku *rootfs*, pozwala zapamiętywać zaktualizowane i dodane pakiety, ale jesteśmy ograniczeni rozmiarem wspomnianego pliku. **Remastering** pozwala zaktualizować wersję live, czyli zapisać aktualny stan systemu na Linux Live USB i zwolnić miejsce zajmowane przez zmiany zapisane w trybie *persistence*.

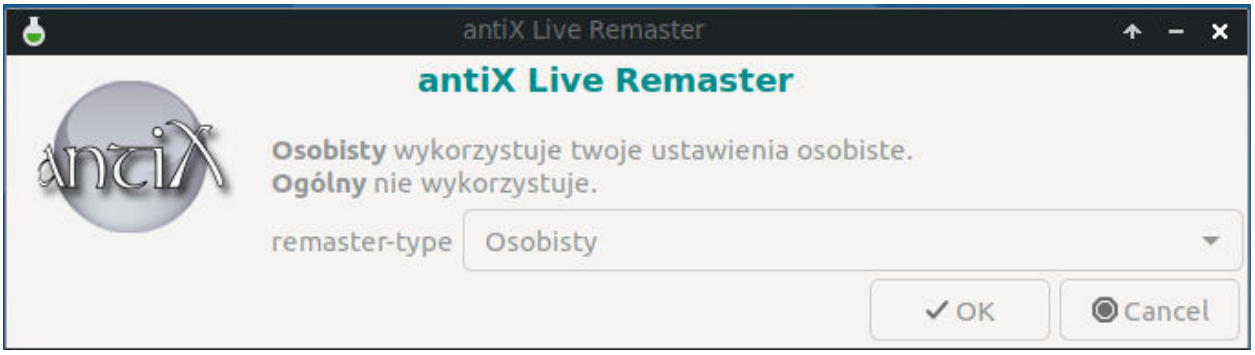
- Uruchamiamy aplikację MX Narzędzia i wybieramy MX Remaster.

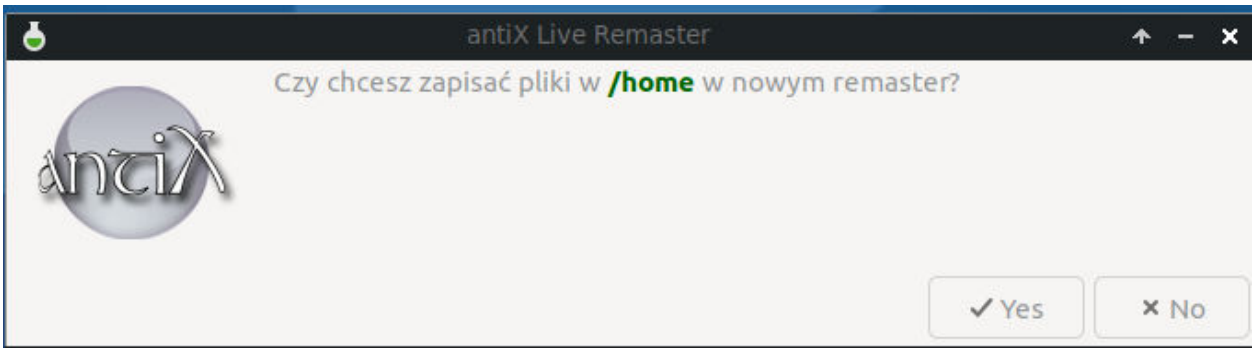


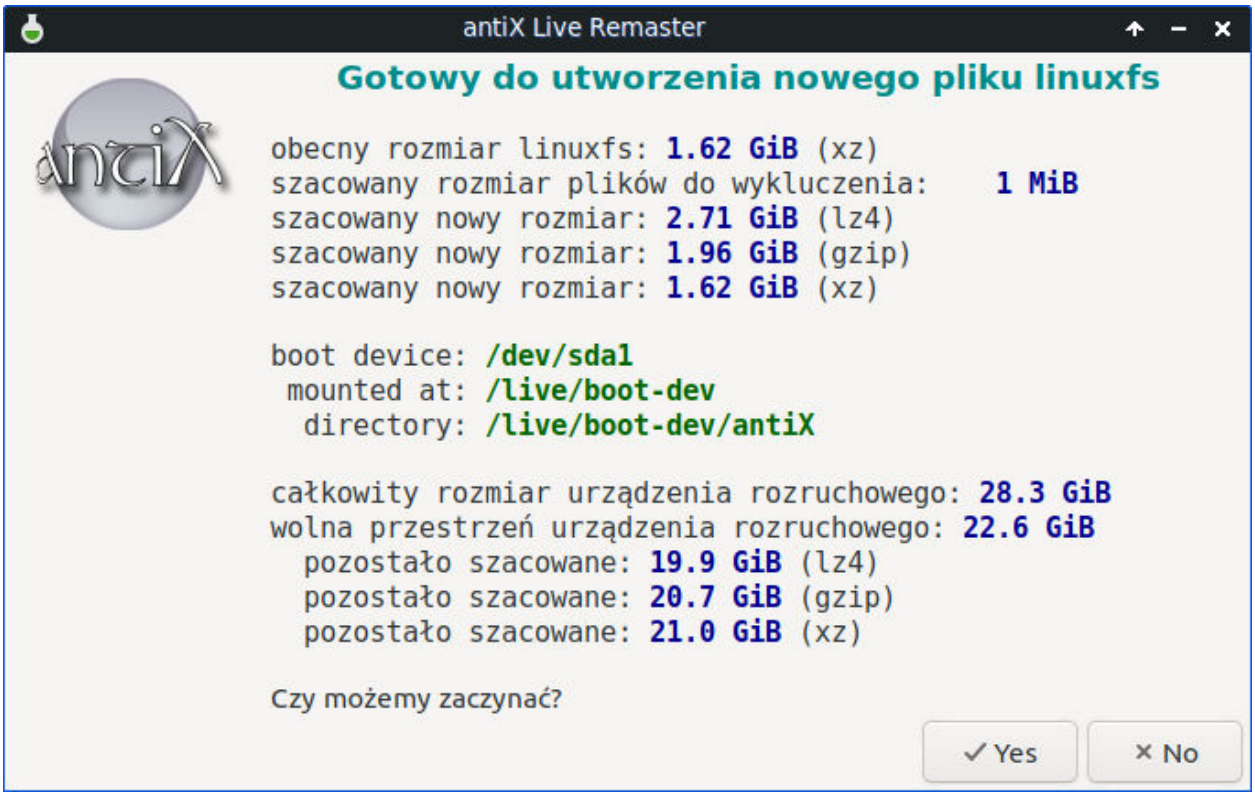
- W oknie “MX Remaster Centrum Kontroli” klikamy “Remaster”.
- Jako “remaster-type” wybieramy “Osobisty”.
- Na pytanie, czy chcemy zapisać pliki w */home* klikamy “Yes”.
- W oknie podsumowującym klikamy “Yes”.

Po zakończeniu operacji na pendrajwie w katalogu `antiX` zostanie utworzony nowy plik `linuxfs`. Poprzednią wersję zapisaną w pliku `linuxfs.old` można usunąć, aby zwolnić miejsce na pendrajwie.


MX Remaster Centrum Kontroli
To narzędzie pozwala na stworzenie nowej wersji ISO podczas korzystania z trybu Live.
Remaster umożliwia zmianę plików, aplikacji i ustawień w celu utworzenia nowej kopii głównej do przechowywania lub dystrybucji.
Persistence pozwala zachować dane, ustawienia i zainstalowane programy do użytku osobistego, gdy urządzenie zostanie wyłączone.
Działania podstawowe
Działania opcjonalne
Persistence
Remaster
Zapisz root persistence
Konfiguracja live persistence
O...
Pomoc
Zamknij


antiX Live Remaster
antiX Live Remaster
Osobisty wykorzystuje twoje ustawienia osobiste.
Ogólny nie wykorzystuje.
remaster-type
Osobisty
OK
Cancel


antiX Live Remaster
Czy chcesz zapisać pliki w /home w nowym remaster?
Yes
No



Materiały archiwalne

Zalecamy używanie dystrybucji MX Linux Live, poniżej zamieszczamy jednak linki do obrazu iso dystrybucji Porteus przygotowanej na potrzeby realizacji projektu KzK w 2018 r.

- porteus322XFCE.iso (597MB)

Informacja: Wszystkie wersje zawierają edytor Geany. Dodatkowe programy w postaci modułów (np. IDE SublimeText3, PyCharm Professional) są albo w obrazie albo do pobrania i dodania.

Zobacz również:

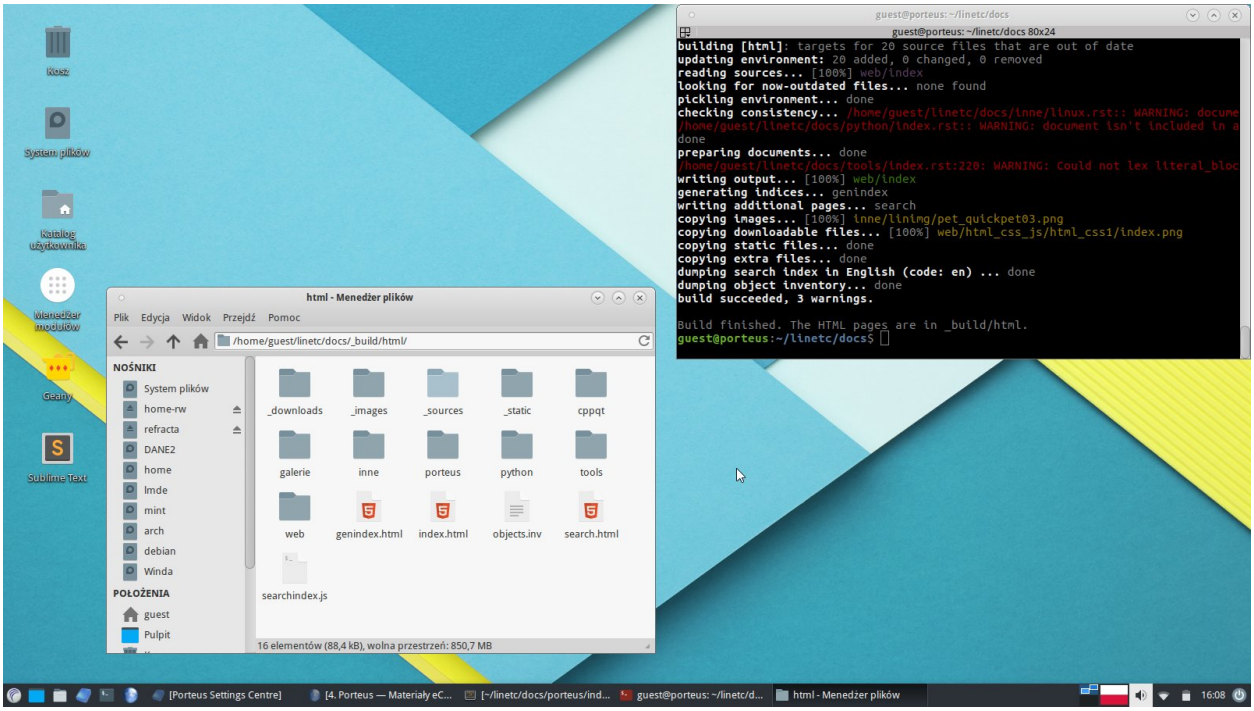
- Pierwsze uruchomienie Porteusa,
- Moduły w Porteusie,

W maszynie wirtualnej

Dystrybucję *LxPupXenial* łatwo uruchomić w dowolnym systemie za pomocą tzw. maszyny wirtualnej.

1. Pobieramy program VirtualBox w wersji dla naszego systemu i instalujemy.
2. Pobieramy maszynę wirtualną z LxPupXenial (1,1 GB) w formacie OVA.
3. Uruchamiamy VirtualBox, wybieramy polecenie “Plik/Importuj urządzenie wirtualne” i wskazujemy ściągnięty w poprzednim kroku plik. Po zaimportowaniu maszyny klikamy “Uruchom”.

LxPupXenial można też zainstalować w VirtualBoksie samemu. Aby to zrobić, uruchamiamy aplikację i tworzymy nową maszynę wirtualną:



Rys. 2.2: Porteus 3.2.2 XFCE 64-bit

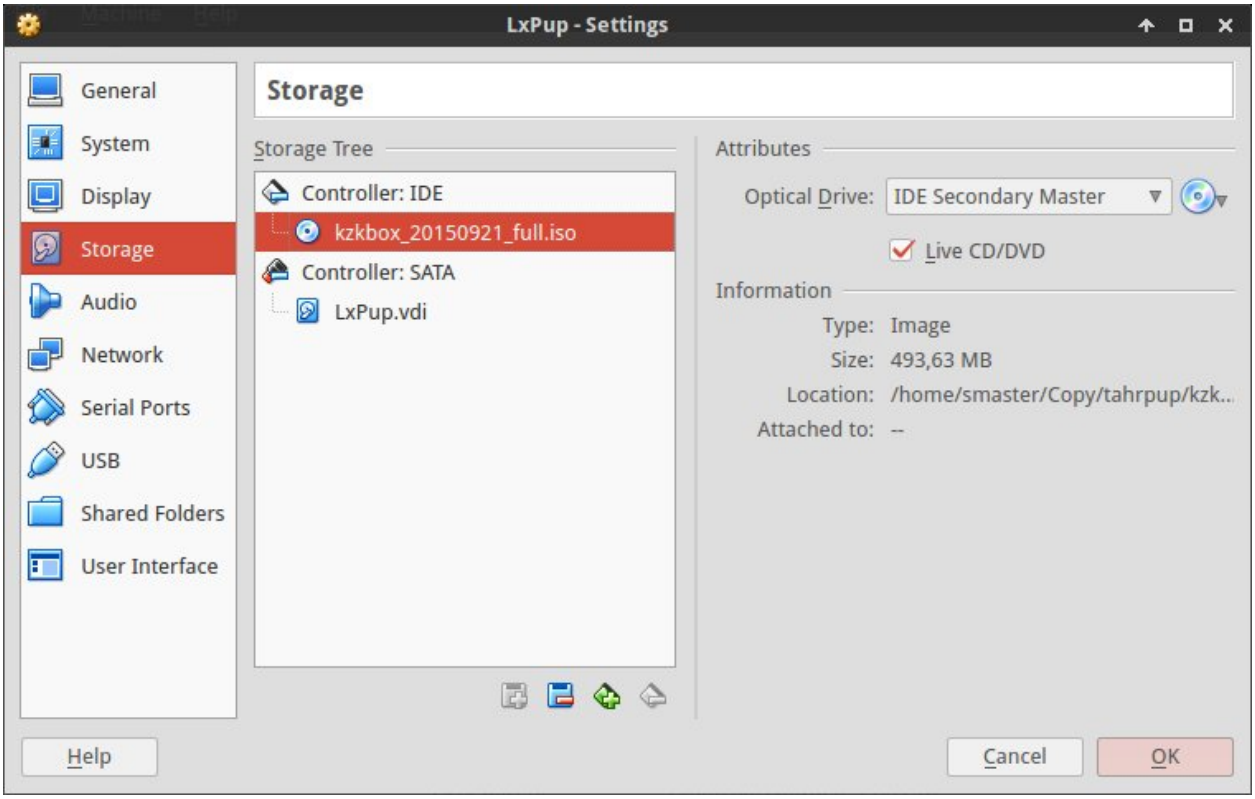
- nazwa – np. “LxPup”, typ – *Linux*, wersja – *Ubuntu (32-bit)*;
- rozmiar pamięci – min. 1024MB
- tworzymy dysk twardy VDI o stałym rozmiarze min. 2048MB

Po utworzeniu maszyny w sekcji “Storage” jako dysk rozruchowy wskazujemy ściągnięty obraz iso dystrybucji, np. `kzkbox20160922_full.iso`:

Uruchamiamy maszynę, ale na ekranie rozruchowym systemu podajemy dodatkowe parametry uruchomieniowe: `puppy pmedia=cd pfix=ram`:

Po uruchomieniu systemu zamykamy kreatora konfiguracji, w przypadku problemów z rozdzielczością przechodzimy do trybu pełnoekranowego (`HOST+F` lub menu *View/Full screen Mode*) i uruchamiamy instalatora poleceniem *Start/Konfiguracja/Puppy uniwersalny instalator*.

1. W oknie “Instaluj” wybieramy *Uniwersalny instalator*;
2. W kolejnym wybieramy *Wewnętrzny (IDE lub SATA) dysk twardy*;
3. Następnie wskazujemy dysk *sda ATA VBOX HARDDISK* za pomocą ikony;
4. Kolejne okno umożliwi uruchomienie edytora GParted, za pomocą którego założymy i sformatujemy partycję systemową;
5. W edytorze GParted wybieramy kolejno:
   (a) w menu *Urządzenie/Utwórz tablicę partycji*, kolejne okno potwierdzamy *Zastosuj*;
   (b) Klikamy nieprzydzielone miejsce prawym klawiszem i wybieramy *Nowa*, wybieramy “Partycja główna” i system “Ext4”, zatwierdzamy *Dodaj*;
   (c) Następnie wybieramy *Edycja/Zastosuj wszystkie działania* lub klikamy ikonę “zielonego ptaszka”;



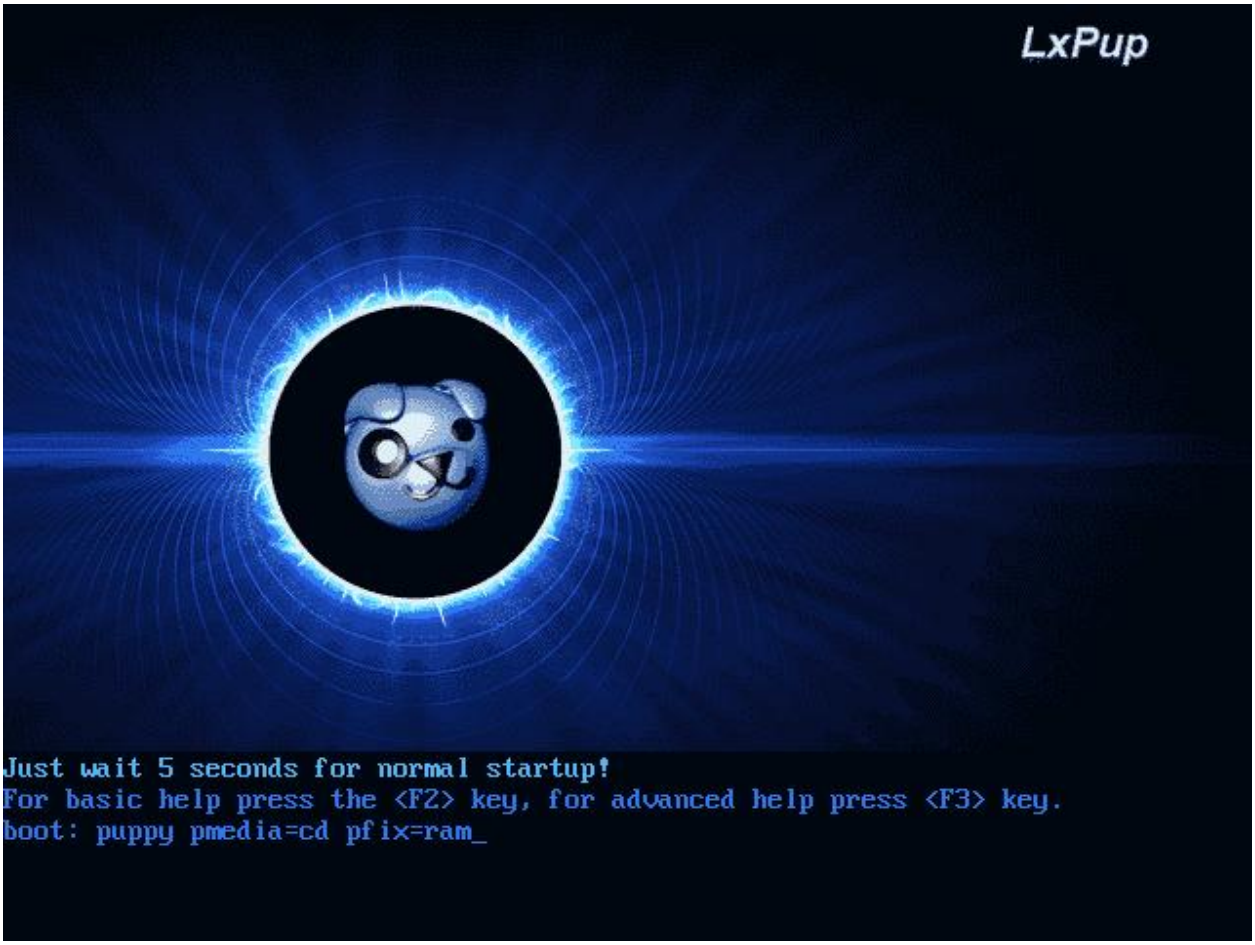
(d) Na koniec klikamy utworzoną partycję prawym klawiszem, wybieramy *Zarządzaj flagami*, zaznaczamy opcję "boot" i zatwierdzamy *Zamknij*; w efekcie powinniśmy zobaczyć co następuje:

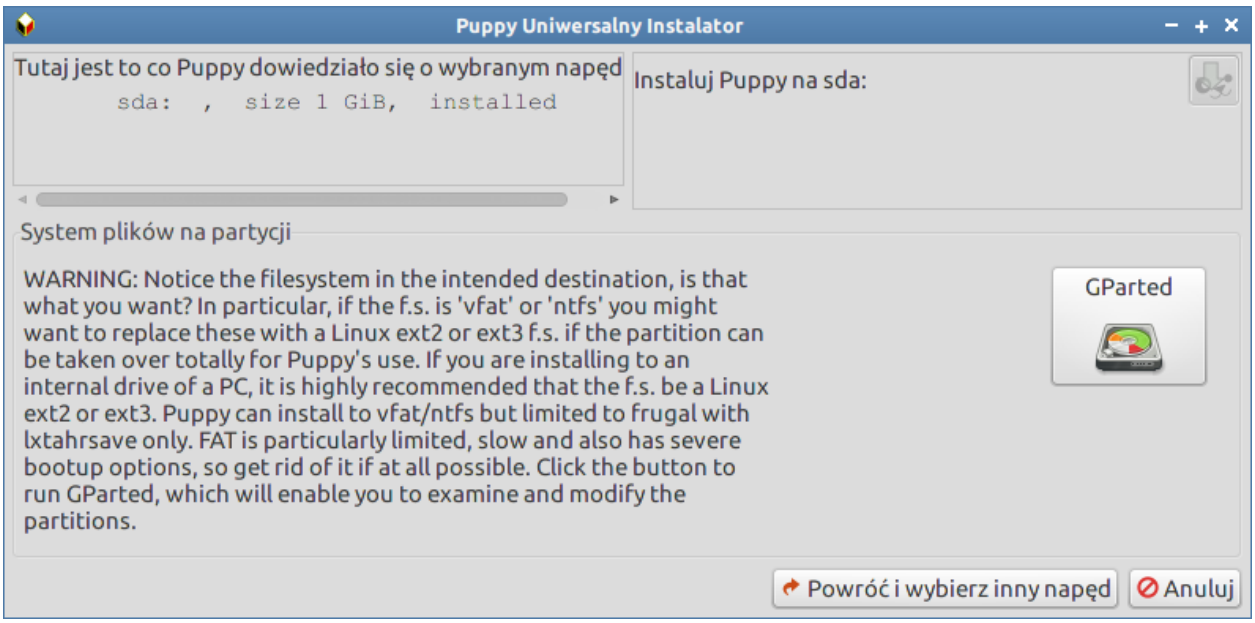
6. Po zamknięciu edytora GParted, ponownie wskazujemy dysk "sda", a w kolejnym, powtórzonym oknie klikamy ikonę w prawym górnym rogu obok napisu "Instaluj Puppy na sda1";
7. W kolejnym oknie potwierdzamy instalację przyciskiem *OK*;
8. W następnym klikamy przycisk *CD*, aby wskazać położenie plików systemowych, i jeszcze raz potwierdzamy przyciskiem "OK";
9. W kolejnym oknie wybieramy *OSZCZĘDNY* tryb instalacji – system będzie zachowywał się tak, jakby był zainstalowany na pendrajwie; następne wyjaśnienia potwierdzamy *OK*;
10. Podajemy nazwę katalogu, w którym znajdą się pliki systemowe, np. "lxpup";
11. Po skopiowaniu plików wybieramy instalację bootmenedżera *grub4dos* przyciskiem *Tak*;
12. W oknie instalacyjnym Grub4Dos zaznaczamy opcje zgodnie ze zrzutem:
13. W kolejnym oknie zatwierdzamy listę wykrytych systemów *OK*, a w następnym potwierdzamy instalację bootmenedżera w MBR;
14. Na koniec zamykamy informację o udanej instalacji:

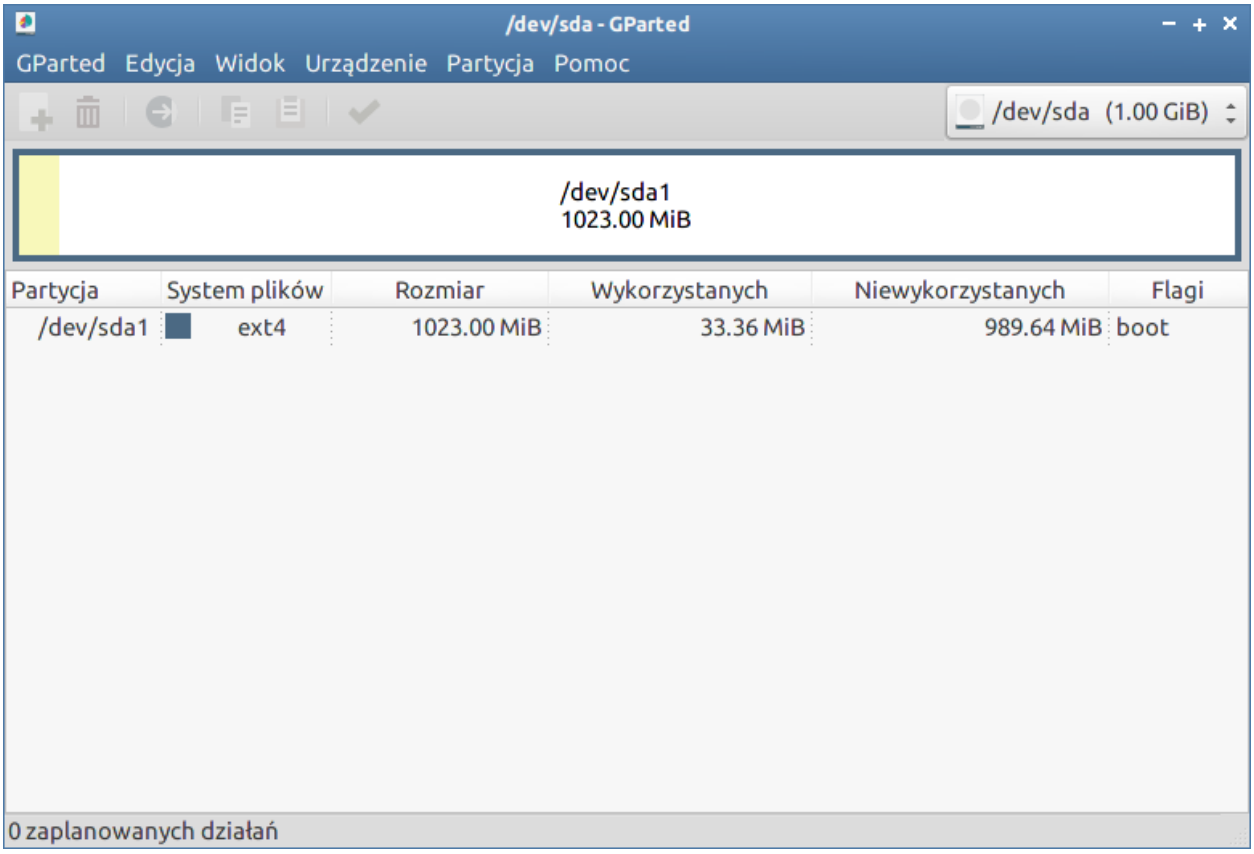
Zamykamy LxPup (*Start/Zamknij*), usuwamy plik obrazu iso z wirtualnego napędu i możemy uruchomić LxPupTahr w maszynie wirtualnej:

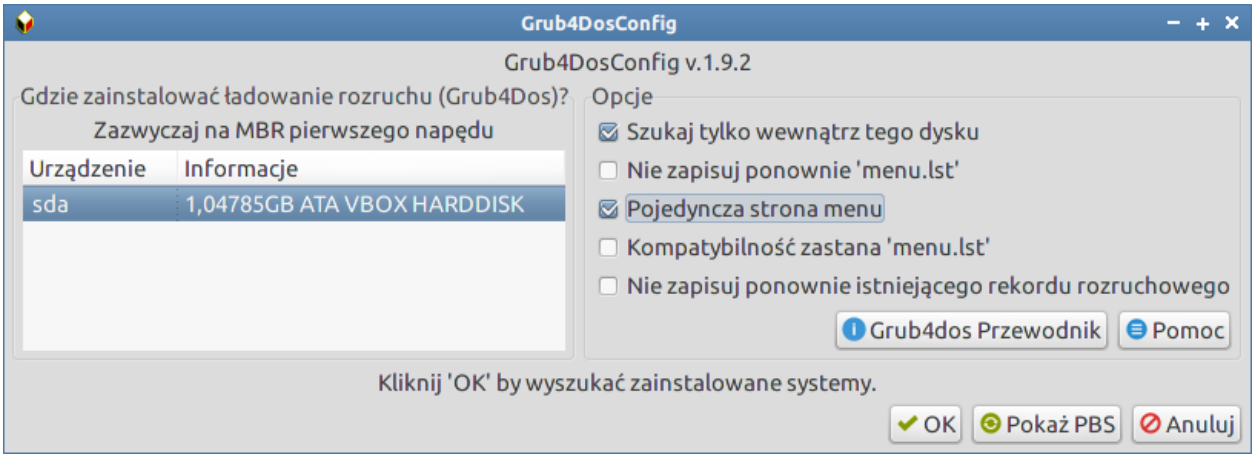
System zainstalowany w ten sposób działa tak samo jak zainstalowany na kluczu USB, a więc wymaga potwierdzenia konfiguracji wstępnej i utworzenia pliku zapisu. Zob.: *Pierwsze uruchomienie*!!!

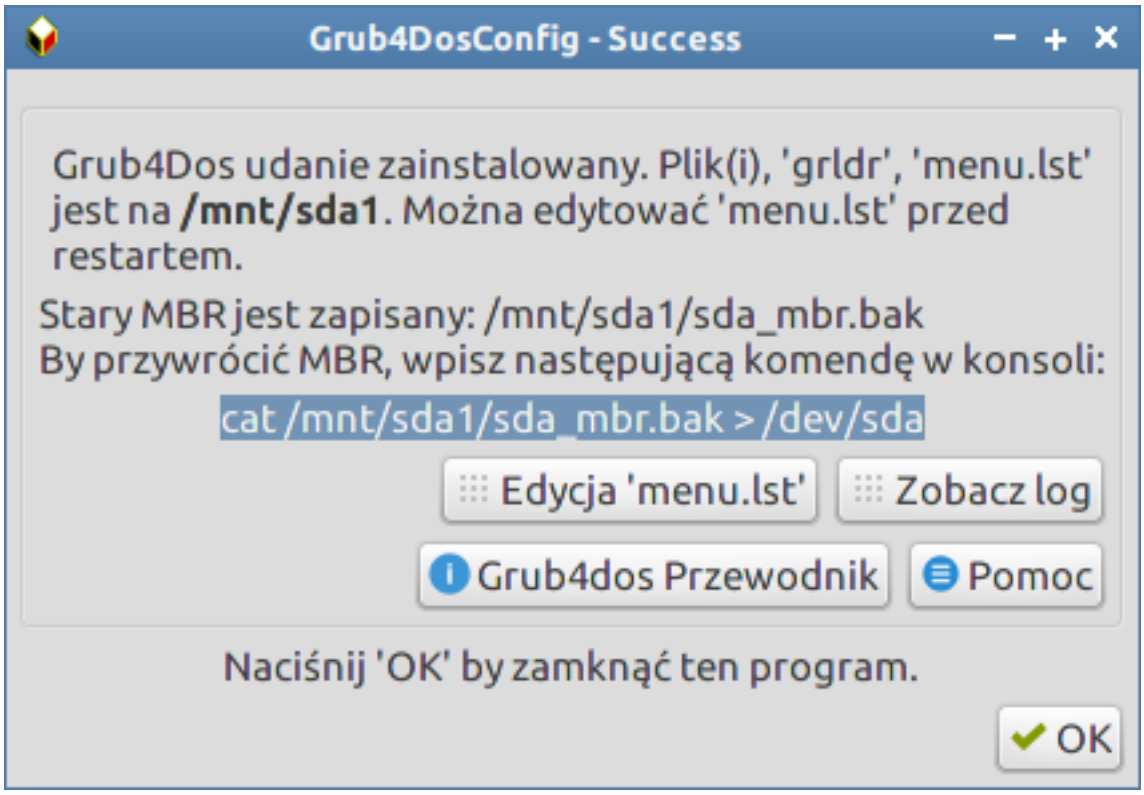
Wskazówka: Za pomocą VirtualBoksa można zainstalować dowolną inną dystrybucję Linuksa z pobranego obrazu

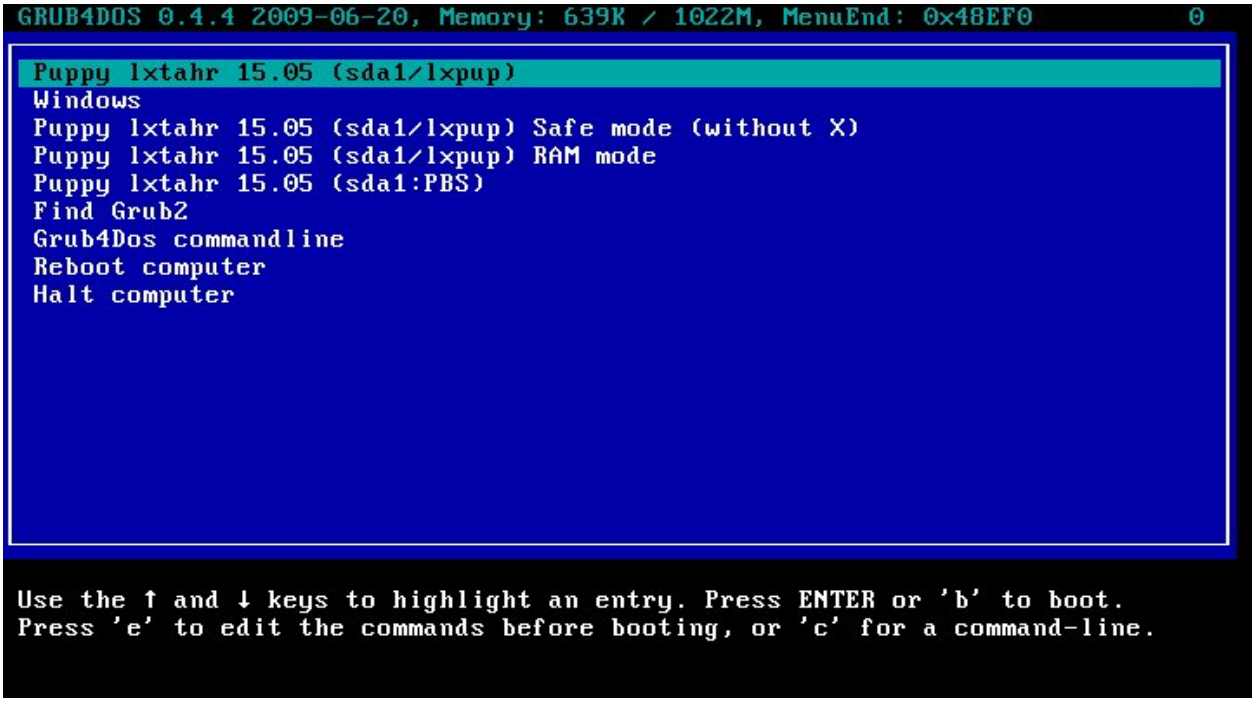

LxPup
Just wait 5 seconds for normal startup!
For basic help press the <F2> key, for advanced help press <F3> key.
boot: puppy pmedia=cd pfix=ram_


Puppy Uniwersalny Instalator
Tutaj jest to co Puppy dowiedziało się o wybranym napęd
sda: , size 1 GiB, installed
Instaluj Puppy na sda:
System plików na partycji
WARNING: Notice the filesystem in the intended destination, is that what you want? In particular, if the f.s. is 'vfat' or 'ntfs' you might want to replace these with a Linux ext2 or ext3 f.s. if the partition can be taken over totally for Puppy's use. If you are installing to an internal drive of a PC, it is highly recommended that the f.s. be a Linux ext2 or ext3. Puppy can install to vfat/ntfs but limited to frugal with lxtahrsave only. FAT is particularly limited, slow and also has severe bootup options, so get rid of it if at all possible. Click the button to run GParted, which will enable you to examine and modify the partitions.
GParted
Powróć i wybierz inny napęd
Anuluj


/dev/sda - GParted
GParted Edycja Widok Urządzenie Partycja Pomoc
/dev/sda (1.00 GiB)
/dev/sda1
1023.00 MiB
Partycja System plików Rozmiar Wykorzystanych Niewykorzystanych Flagi
/dev/sda1 ext4 1023.00 MiB 33.36 MiB 989.64 MiB boot
0 zaplanowanych działań


Grub4DosConfig
Grub4DosConfig v.1.9.2
Gdzie zainstalować ładowanie rozruchu (Grub4Dos)?
Zazwyczaj na MBR pierwszego napędu
Urządzenie Informacje
sda 1,04785GB ATA VBOX HARDDISK
Opcje
Szukaj tylko wewnątrz tego dysku
Nie zapisuj ponownie 'menu.lst'
Pojedyncza strona menu
Kompatybilność zastana 'menu.lst'
Nie zapisuj ponownie istniejącego rekordu rozruchowego
Grub4dos Przewodnik
Pomoc
Kliknij 'OK' by wyszukać zainstalowane systemy.
OK
Pokaż PBS
Anuluj


Grub4DosConfig - Success
Grub4Dos udanie zainstalowany. Plik(i), 'grldr', 'menu.lst' jest na /mnt/sda1. Można edytować 'menu.lst' przed restartem.
Stary MBR jest zapisany: /mnt/sda1/sda_mbr.bak
By przywrócić MBR, wpisz następującą komendę w konsoli:
cat /mnt/sda1/sda_mbr.bak > /dev/sda
Edycja 'menu.lst'
Zobacz log
Grub4dos Przewodnik
Pomoc
Naciśnij 'OK' by zamknąć ten program.
OK


GRUB4DOS 0.4.4 2009-06-20, Memory: 639K / 1022M, MenuEnd: 0x48EF0
Puppy lxtahr 15.05 (sda1/lxpup)
Windows
Puppy lxtahr 15.05 (sda1/lxpup) Safe mode (without X)
Puppy lxtahr 15.05 (sda1/lxpup) RAM mode
Puppy lxtahr 15.05 (sda1:PBS)
Find Grub2
Grub4Dos commandline
Reboot computer
Halt computer
Use the ↑ and ↓ keys to highlight an entry. Press ENTER or 'b' to boot.
Press 'e' to edit the commands before booting, or 'c' for a command-line.

iso. Taka instalacja zadziała jak "normalny" system, a więc umożliwi aktualizację i instalację oprogramowania, a także zapis tworzonych dokumentów.

Wskazówka: W przypadku problemów z działaniem myszy w wirtualnym systemie, warto spróbować wyłączyć ewentualną automatyczną integrację kursora za pomocą skrótu `HOST+I`. Klawisz `HOST` to wskazany w menu *File/Preferences/Input/Virtual Machine* klawisz umożliwiający sterowanie wirtualną maszyną. Dla polskiej klawiatury można ustawić np. prawy CTRL.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

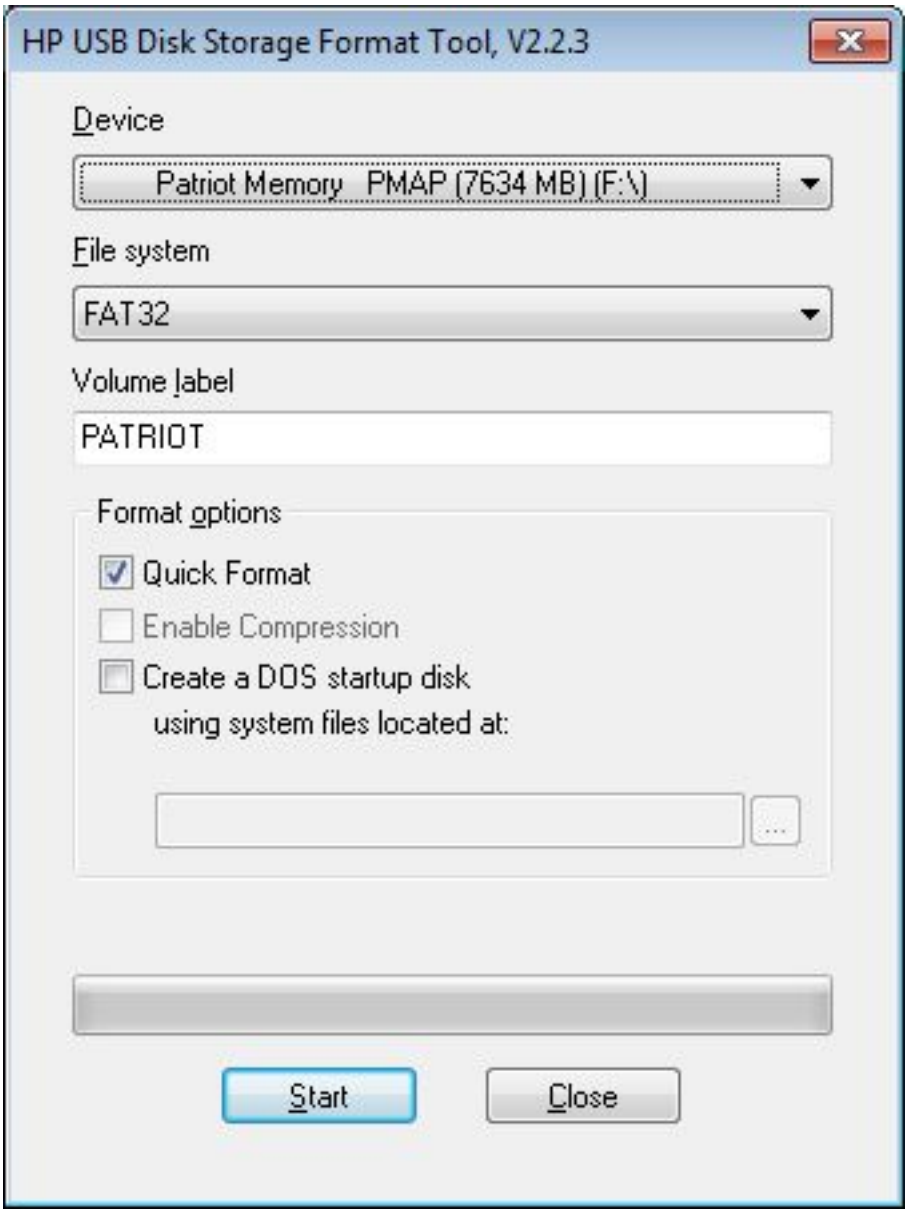
Autorzy Patrz plik "Autorzy"

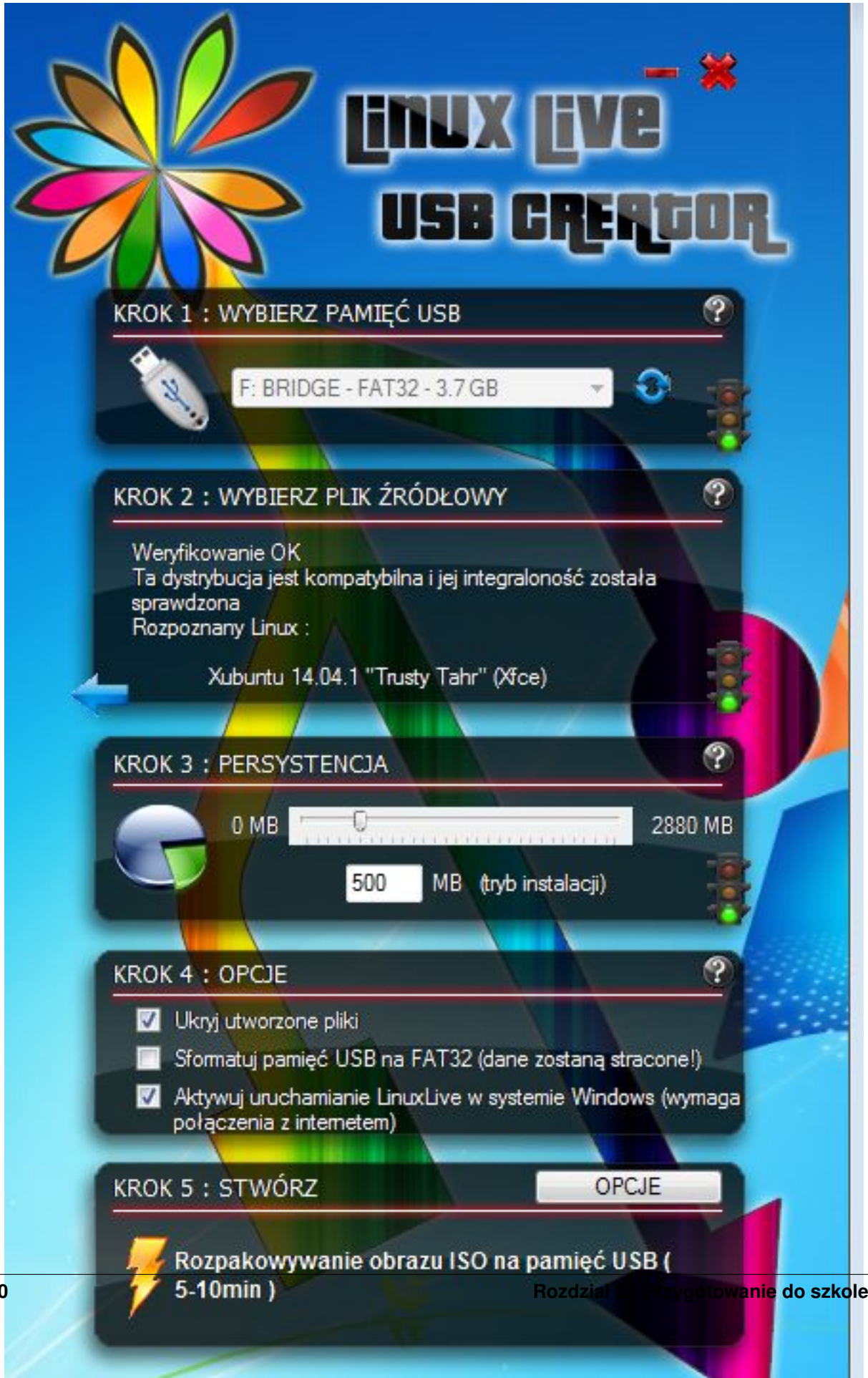
Problemy

Jeśli nie da się uruchomić komputera za pomocą przygotowanego klucza, przeczytaj poniższe wskazówki.

1. Zanim uznasz, że pendrajw nie działa, przetestuj go na innym sprzęcie!
2. W niektórych komputerach możliwość uruchamiania z napędu USB trzeba odblokować w BIOS-ie. Odpowiedniego ustawienia poszukaj np. w opcji "Boot order".
3. Starsze komputery stacjonarne mogą wymagać wejścia do ustawień BIOSU (zazwyczaj klawisz `F1`, `F2` lub `DEL`) i ustawienia pendrajwa (o ile zostanie wykryty) jako urządzenia startowego zamiast np. dysku twardego czy cdromu. Opuszczając BIOS zmiany należy zapisać! Komputer restartujemy bez usuwania klucza USB.
4. W przypadku komputerów stacjonarnych, jeżeli nie działają frontowe gniazda USB, podłącz klucz z tyłu!
5. Niebootujący pendrajw można najpierw sformatować:
   - Windows: użyj programu HP-USB-Disk-Storage-Format-Tool jako administrator;
   - W Linuksie wydaj polecenie: `mkfs.vat /dev/sdb1`, zwróć uwagę na właściwą nazwę partycji (*sdb1*)!

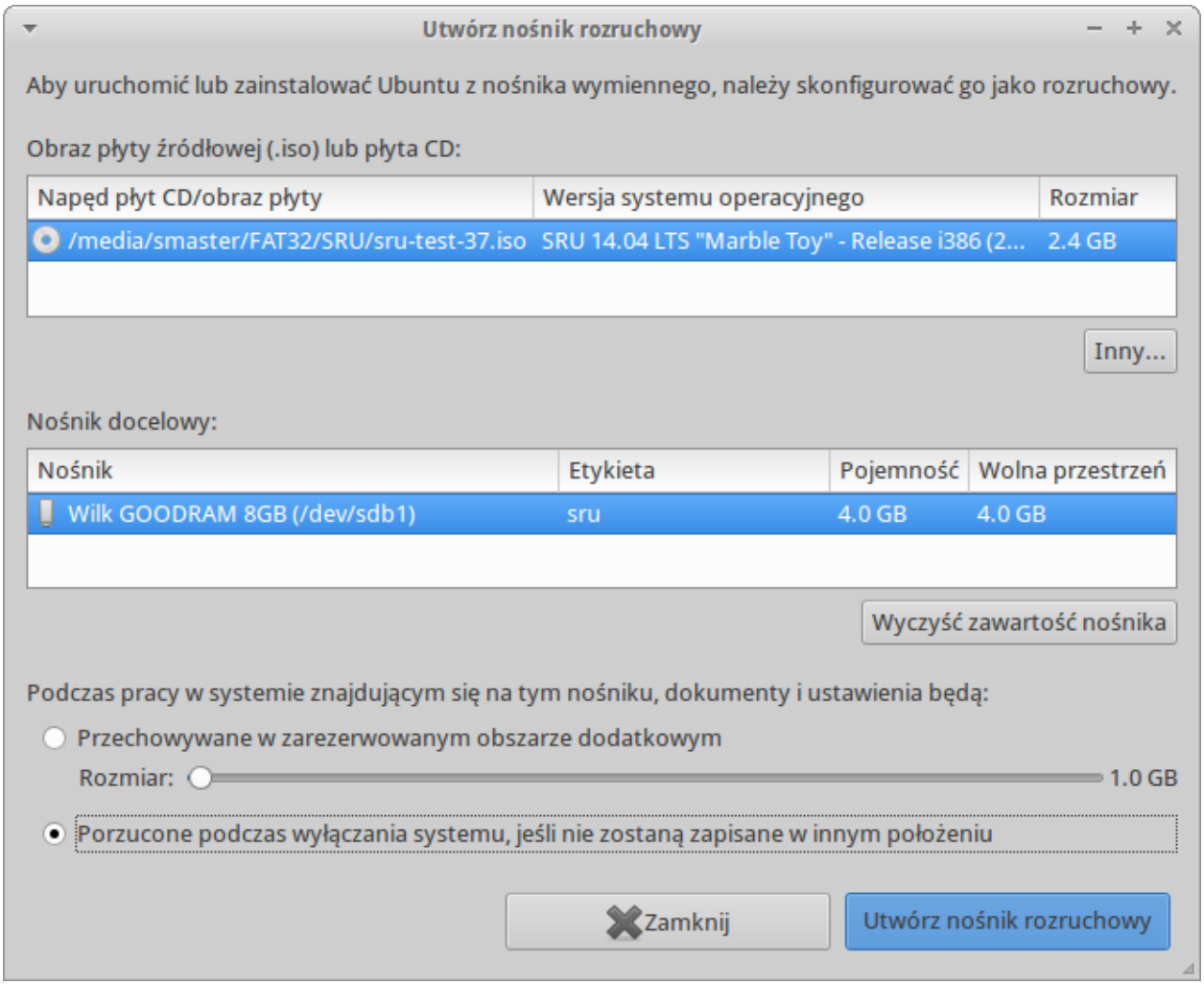
   Nagraj jeszcze raz wybrany obraz iso.
6. W Windows wypróbuj narzędzie Linux Live USB Creator. Użyj go do nagrania obrazu *Xubuntu lub LxPupXenial*. Po uruchomieniu klikij "Opcje", wybierz polski język interfejsu. Skonfiguruj program zgodnie z podanym zrzutem, czyli: wskaż klucz USB, wybierz obraz iso i określamy rozmiar pliku "casper-rw" (persystencji) na min. 512MB. Poprawność konfiguracji oznaczana jest przez zapalone zielone światła! Naciśnij ikonę błyskawicy i czekaj. Uwaga: program może poprosić o hasło administratora, aby wgrać sektor rozruchowy.
7. W Windows możesz wypróbować narzędzie Universal USB Installer polecane przez producenta Ubuntu, który udostępnia również instrukcję. Użyj do nagrania dystrybucji *Xubuntu*.
8. Spróbuj z innym pendrajwem.
9. Zmień maszynę, być może jest za stara lub za nowa!
10. Przygotuj pendrajwa na innym komputerze!
11. Jeżeli masz BIOS UEFI z włączonym mechanizmem SecureBoot, co stanowi normę dla laptopów z preinstalowanym Windows 7/8/10... po 2012 r., spróbuj wyłączyć zabezpieczenie w biosie. Możesz zajrzeć do instrukcji:
    - pomoc Ubuntu
    - pomoc Microsoft
    - wsparcie HP


HP USB Disk Storage Format Tool, V2.2.3
Device
Patriot Memory PMAP (7634 MB) (F:\)
File system
FAT32
Volume label
PATRIOT
Format options
Quick Format
Enable Compression
Create a DOS startup disk
using system files located at:
Start
Close


LINUX LIVE
USB CREATOR
KROK 1 : WYBIERZ PAMIĘĆ USB
F: BRIDGE - FAT32 - 3.7 GB
KROK 2 : WYBIERZ PLIK ŹRÓDŁOWY
Weryfikowanie OK
Ta dystrybucja jest kompatybilna i jej integraloność została sprawdzona
Rozpoznany Linux :
Xubuntu 14.04.1 "Trusty Tahr" (Xfce)
KROK 3 : PERSYSTENCJA
0 MB
2880 MB
500
MB (tryb instalacji)
KROK 4 : OPCJE
Ukryj utworzone pliki
Sformatuj pamięć USB na FAT32 (dane zostaną stracone!)
Aktywuj uruchamianie LinuxLive w systemie Windows (wymaga połączenia z internetem)
KROK 5 : STWÓRZ
OPCJE
Rozpakowywanie obrazu ISO na pamięć USB (5-10min)

12. W Ubuntu i pochodnych można użyć programu **usb-creator-gtk**, który powinien być zainstalowany domyślnie. Jeśli nie, wydajemy polecenia: `sudo apt-get update && sudo apt-get install usb-gtk-creator`.

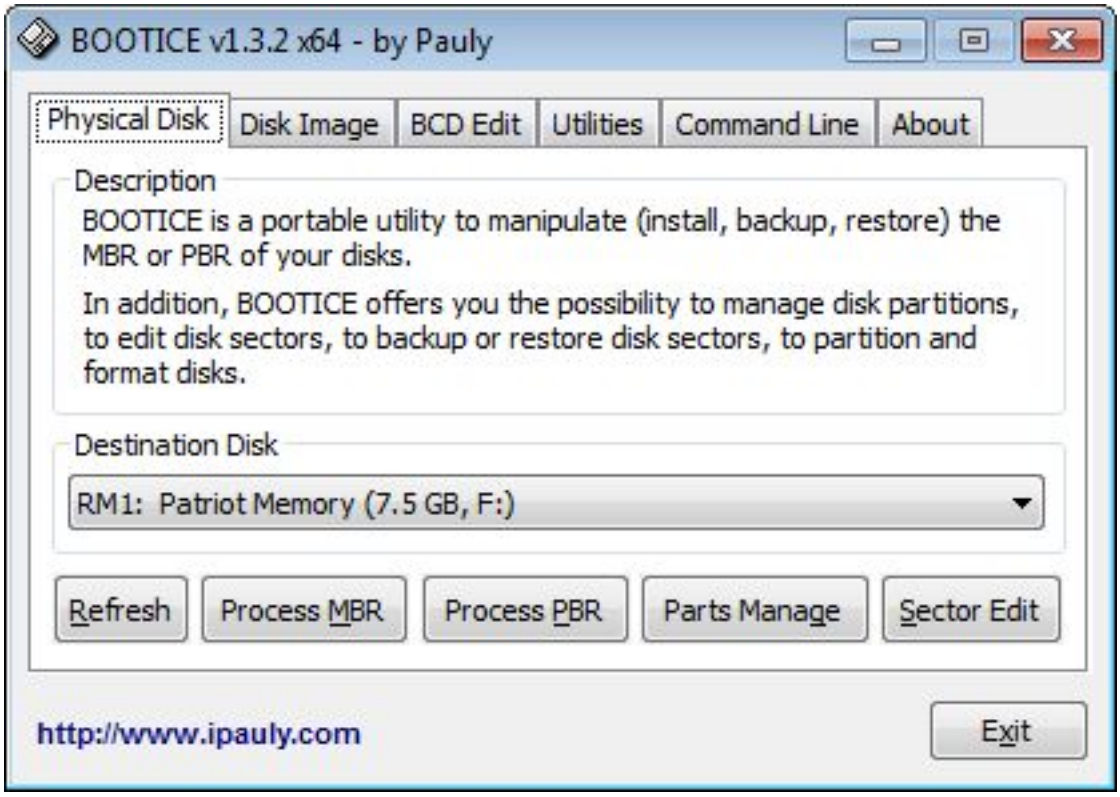
    Po uruchomieniu kreatora poleceniem `usb-creator-gtk` wydanym w terminalu klikamy przycisk "Inny" i wskazujemy obraz iso wybranego systemu, w polu "Nośnik docelowy" wybieramy partycję podstawową pendrajwa (np. /dev/sdb1). Wybieramy opcję "Przechowywanie pracy...", jeżeli dane użytkownika mają być przechowywane w pliku i na pendrajwie nie tworzyliśmy dodatkowej partycji, w przeciwnym wypadku zaznaczamy opcję drugą "Porzucone podczas wyłączania...", która mimo nazwy spowoduje zapisywanie ustawień na dodatkowej partycji ext4 o etykiecie "home-rw".



Inne narzędzia

- Bootice – opcjonalne narzędzie do różnych operacji na dyskach. Za jego pomocą można np. utworzyć, a następnie odtworzyć kopię MBR pendrajwa.

Wskazówka: Narzędzia udostępniane w serwisie *dobreprogramy.pl* domyślnie ściągane są przy użyciu dodatkowej aplikacji ukrytej pod przycieskiem "Pobierz program". Jest ona całkowicie zbędna, sugerujemy korzystanie z przycisku "Linki bezpośrednie" i wybór odpowiedniej wersji (32-/64-bitowej), jeżeli jest dostępna.



Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

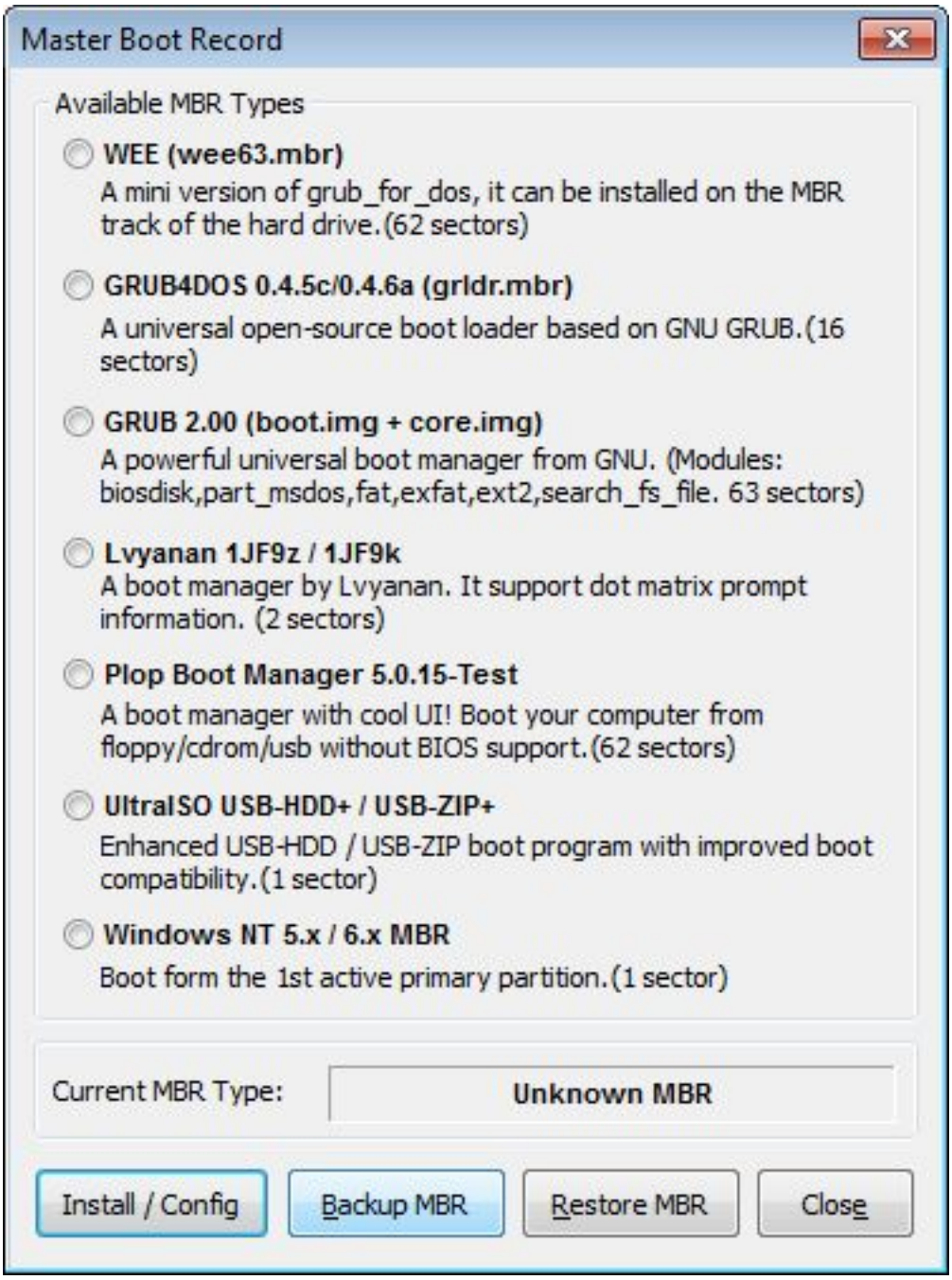
Linux Live USB – opcje

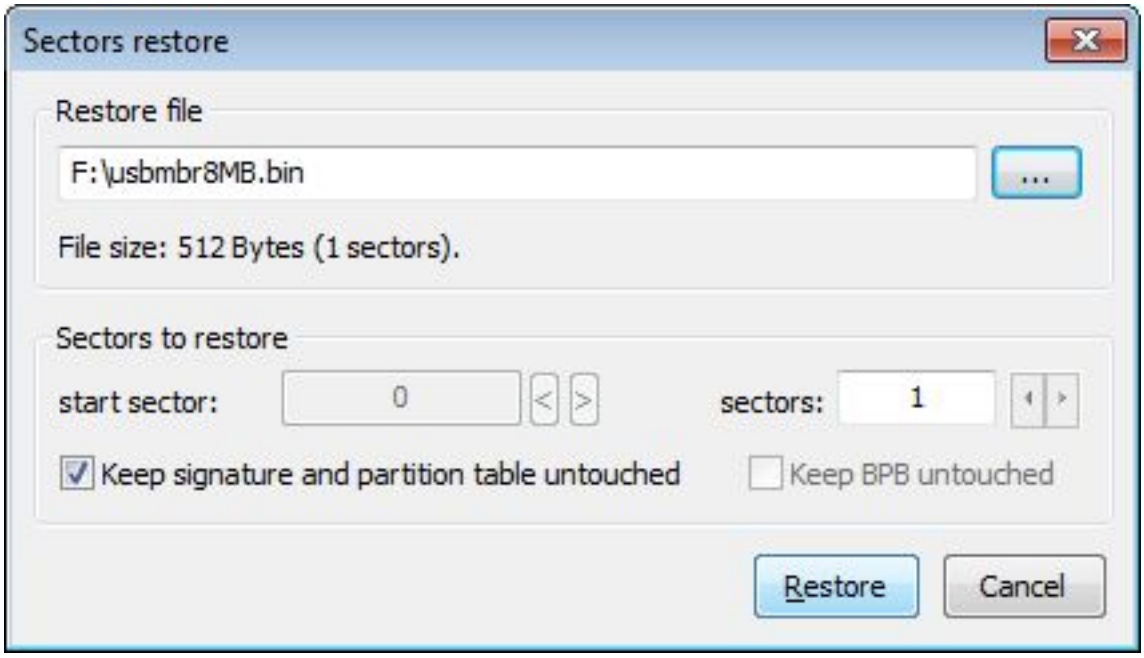
System na kluczu USB

Jeżeli dysponujemy startowym nośnikiem (np. CD/DVD) z systemem Linux Mint, Xubuntu czy FREE_DESKTOP możemy uruchomić normalną instalację, podpiąć nośnik USB, założyć na nim (w trakcie instalacji) partycję Ext4 i wskazać ją jako miejsce instalacji systemu. Trzeba również zainstalować menedżer startowy GRUB w MBR takiego napędu.

Wskazówka: Załóżmy, że uruchamiamy Xubuntu z płyty DVD na komputerze z jednym twardym dyskiem. Instalator oznaczy go jako `sda(x)`, a podłączony klucz USB jako `sdb(x)`, co poznać będzie można po rozmiarze i obecnych na nich partycjach. Na dysku `sdb` tworzymy co najmniej jedną partycję Ext4, jako cel instalacji systemu, czyli punkt montowania katalogu głównego `/` wskazujemy partycję `/dev/sdb1`, natomiast jako miejsce instalacji GRUB-a wybieramy `/dev/sdb`.

Po uruchomieniu tak zainstalowanego systemu wszystkie dokonywane zmiany będą zapamiętywane. Można system aktualizować, można instalować nowe oprogramowanie i zapisywać swoje pliki.


Master Boot Record
Available MBR Types
WEE (wee63.mbr)
A mini version of grub_for_dos, it can be installed on the MBR track of the hard drive.(62 sectors)
GRUB4DOS 0.4.5c/0.4.6a (grldr.mbr)
A universal open-source boot loader based on GNU GRUB.(16 sectors)
GRUB 2.00 (boot.img + core.img)
A powerful universal boot manager from GNU. (Modules: biosdisk,part_msdos,fat,exfat,ext2,search_fs_file. 63 sectors)
Lvyanan 1JF9z / 1JF9k
A boot manager by Lvyanan. It support dot matrix prompt information. (2 sectors)
Plop Boot Manager 5.0.15-Test
A boot manager with cool UI! Boot your computer from floppy/cdrom/usb without BIOS support.(62 sectors)
UltraISO USB-HDD+ / USB-ZIP+
Enhanced USB-HDD / USB-ZIP boot program with improved boot compatibility.(1 sector)
Windows NT 5.x / 6.x MBR
Boot form the 1st active primary partition.(1 sector)
Current MBR Type:
Unknown MBR
Install / Config
Backup MBR
Restore MBR
Close



Kopia klucza USB

Jeżeli dysponujemy już nośnikiem startowym USB, możemy łatwo go skopiować. Żeby operację przyśpieszyć, zwłaszcza jeśli chcemy wykonać kilka kopii, można na początku utworzyć obraz danych zawartych na pendrajwie.

W Linuksie

Posługujemy się poleceniem `dd` wydanym w katalogu domowym:

```
~$ sudo dd if=/dev/sdb of=obrazusb.img bs=1M
```

Ciąg `/dev/sdb` w powyższym poleceniu oznacza napęd źródłowy, `obrazusb.img` to dowolna nazwa pliku, do którego zapisujemy odczytaną zawartość.

Informacja: Linux oznacza wykryte napędy jako `/dev/sd[a-z]`, a więc pierwszy dysk twardy oznaczony zostanie jako `sda`. Po podłączeniu klucza USB otrzyma on nazwę `sdb`. Kolejny podłączony napęd USB będzie dostępny jako `sdc`. Nazwę napędu USB możemy sprawdzić po wydaniu podanych niżej poleceń. Pierwsze z nich wyświetli w końcowych liniach ostatnio dodane napędy w postaci ciągu typu `sdb:sdb1`. Podobne wyniki powinno zwrócić polecenie drugie.

```
~$ mount | grep /dev/sd
~$ dmesg | grep /dev/sd
```

Po utworzeniu obrazu podłączamy napęd docelowy i dokładnie ustalamy jego oznaczenie, ponieważ wcześniejesze **dane z napędu docelowego zostaną usunięte**. Jeżeli napęd został zamontowany, czyli jego zawartość została automatycznie pokaza w menedżerze plików, musimy go odmontować za pomocą polecenia `Odmontuj` (nie mylić z `Wysuń`!). Następnie wydajemy polecenie:

```
~$ sudo dd if=obrazusb.img of=/dev/sdc bs=4M; sync
```

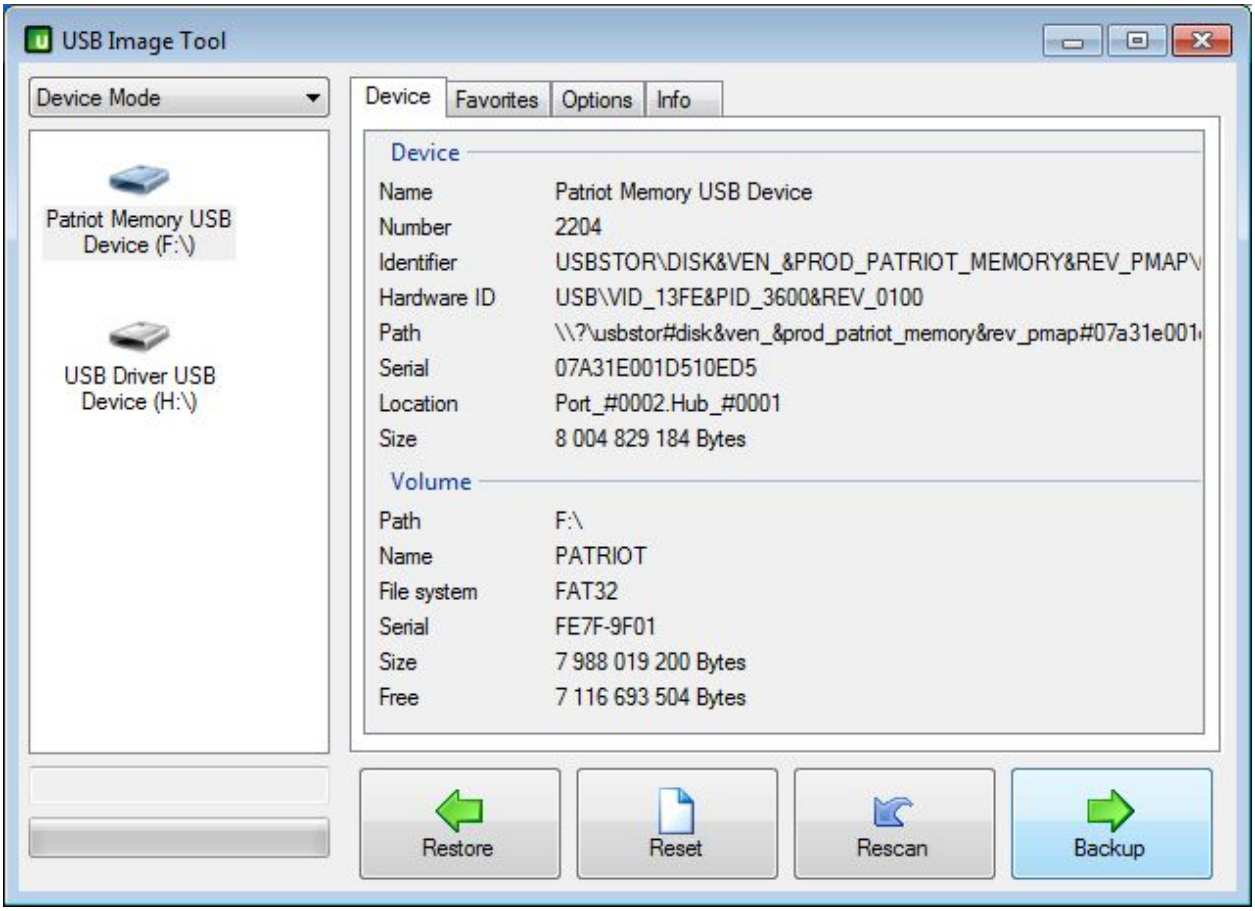
Możliwe jest również **kopiowanie zawartości klucza USB od razu na drugi klucz** bez tworzenia obrazu na dysku. Po podłączeniu obu pendrajwów i ustaleniu ich oznaczeń wydajemy polecenie:

```
~$ sudo dd if=/dev/sdb of=/dev/sdc bs=4M; sync
```

- gdzie `sdb` to nazwa napędu źródłowego, a `sdc` to oznaczenie napędu docelowego.

W MS Widows

- USB Image Tool – narzędzie do robienia obrazów dysków USB i nagrywania ich na inne pendrajwy.

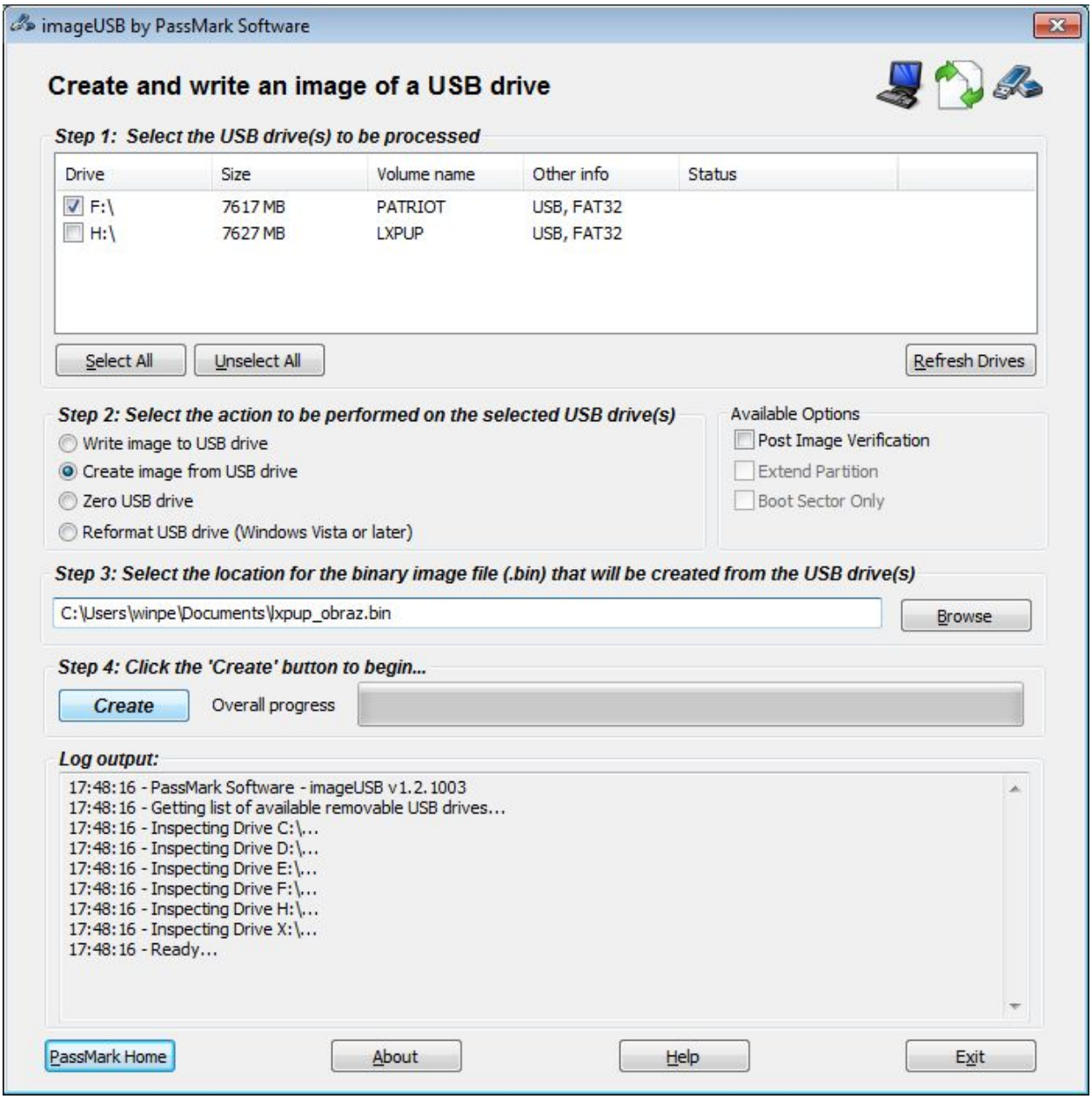


- Image USB – świetny program do tworzenia obrazów napędów USB i nagrywania ich na wiele pendrajwów jednocześnie.

Wskazówka: Narzędzia udostępniane w serwisie *dobreprogramy.pl* domyślnie ściągane są przy użyciu dodatkowej aplikacji ukrytej pod przycieskiem "Pobierz program". Jest ona całkowicie zbędna, sugerujemy korzystanie z przycisku "Linki bezpośrednie" i wybór odpowiedniej wersji (32-/64-bitowej), jeżeli jest dostępna.

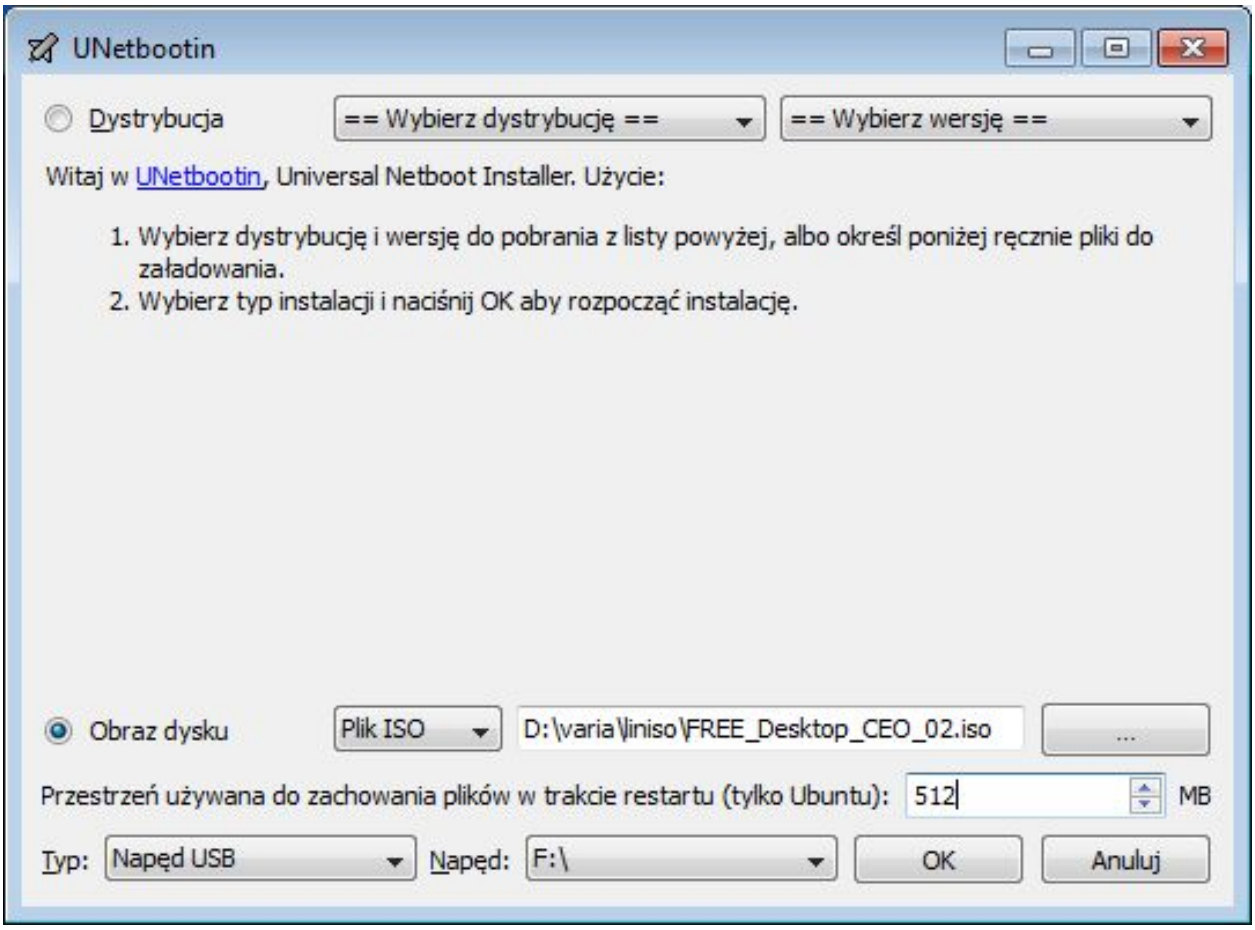
Linux-live USB – różne systemy

W trybie live mogą być również instalowane na pendrajwach różne dystrybucje Linuksa, np. Xubutnu 16.04 LTS czy Linux Mint 18, oparte na stabilnym wydaniu systemu Ubuntu. Do realizowania naszych scenariuszy wymagają


imageUSB by PassMark Software
Create and write an image of a USB drive
Step 1: Select the USB drive(s) to be processed
Drive
Size
Volume name
Other info
Status
F:\
7617 MB
PATRIOT
USB, FAT32
H:\
7627 MB
LXPUP
USB, FAT32
Select All
Unselect All
Refresh Drives
Step 2: Select the action to be performed on the selected USB drive(s)
Write image to USB drive
Create image from USB drive
Zero USB drive
Reformat USB drive (Windows Vista or later)
Available Options
Post Image Verification
Extend Partition
Boot Sector Only
Step 3: Select the location for the binary image file (.bin) that will be created from the USB drive(s)
C:\Users\winpe\Documents\xpup_obraz.bin
Browse
Step 4: Click the 'Create' button to begin...
Create
Overall progress
Log output:
17:48:16 - PassMark Software - imageUSB v1.2.1003
17:48:16 - Getting list of available removable USB drives...
17:48:16 - Inspecting Drive C:\...
17:48:16 - Inspecting Drive D:\...
17:48:16 - Inspecting Drive E:\...
17:48:16 - Inspecting Drive F:\...
17:48:16 - Inspecting Drive H:\...
17:48:16 - Inspecting Drive X:\...
17:48:16 - Ready...
PassMark Home
About
Help
Exit

doinstalowania części narzędzi i bibliotek. Wymienione systemy bardzo dobrze nadają się do zainstalowania jako system główny lub drugi na dysku twardym komputera. Można to zrobić za pomocą pendrajwów live. Aby wgrać system na pendrajwa:

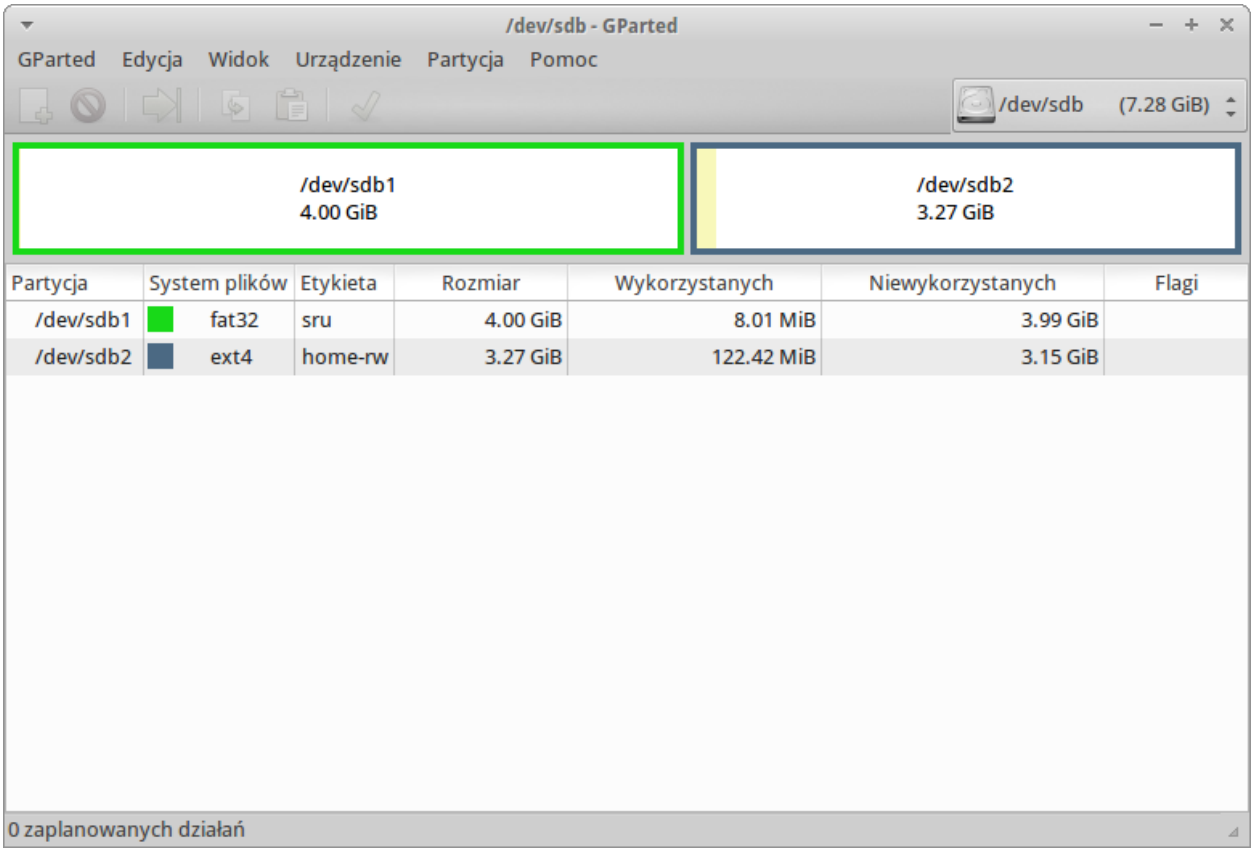
- Pobieramy wybrany obraz iso:
  - Xubuntu
  - Linux Mint 18
- Pobieramy program Unetbootin.
- Wpinamy pendrajwa o pojemności min. 4GB.
- Po uruchomieniu programu *Unetbootin* zaznaczamy opcję "Obraz dysku", klikamy przycisk "..." i wskazujemy pobrany obraz. W polu "Przestrzeń używana do zachowania plików..." wpisujemy min. **512**. W polu "Napęd:" wskazujemy pendrajwa i klikamy "OK". Czekamy w zależności od wybranej dystrybucji i prędkości klucza USB od 5-25 minut.



Informacja: Jeżeli nagrywamy obraz *Xubuntu* lub *Minta* możemy na pendrajwie utworzyć dodatkową partycję typu Ext4 o dowolnej pojemności, ale obowiązkowej etykiecie "home-rw". Zostanie ona wykorzystana jako miejsce montowania i zapisywania plików użytkownika. W takim wypadku pole "Przestrzeń używana do zachowania plików..." pozostawiamy puste!

Dodatkową partycję utworzysz przy użyciu programu **gparted**. Instalacja: `sudo apt-get update && sudo apt-get install gparted`. Niestety za pomocą standardowych narzędzi MS Windows nie utworzymy partycji

Ext4. Ostateczny układ partycji powinien wyglądać tak jak na poniższym zrzucie:



Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

LxPup – obsługa

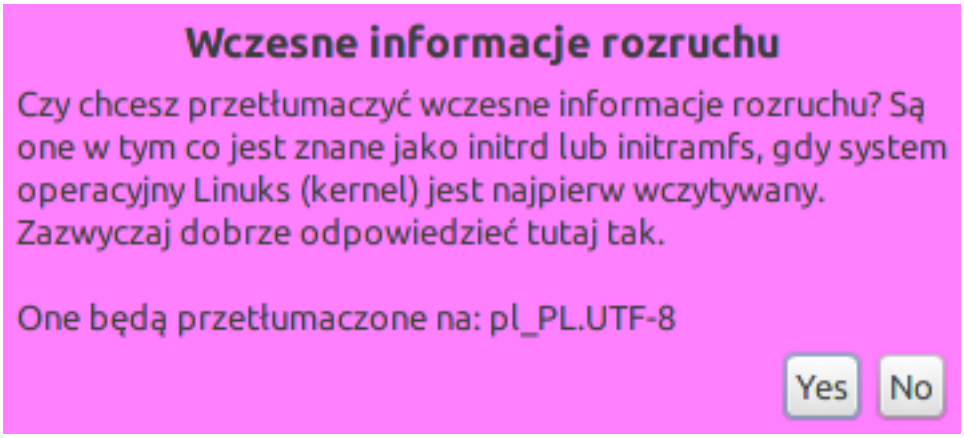
Spis treści

– *Wskazówki*

Pierwsze uruchomienie

Po pierwszym uruchomieniu zatwierdzamy okno kreatora ustawień przyciskiem "Ok" i zamykamy kreatora połączenia z internetem. Następnie **zamykamy system i tworzymy** plik zapisu (ang. *savefile*), w którym przechowywane będą wprowadzane przez nas zmiany: konfiguracja, instalacja programów, utworzone dokumenty.

Na początku potwierdzamy tłumaczenie informacji rozruchowych.



Dalej klikamy "Zapisz", następnie "administrator". Wybieramy partycję oznaczającą pendrajwa: w konfiguracjach z 1 dyskiem twardym będzie ona oznaczona najczęściej *sdb1* (kierujemy się rozmiarem i typem plików: vfat).

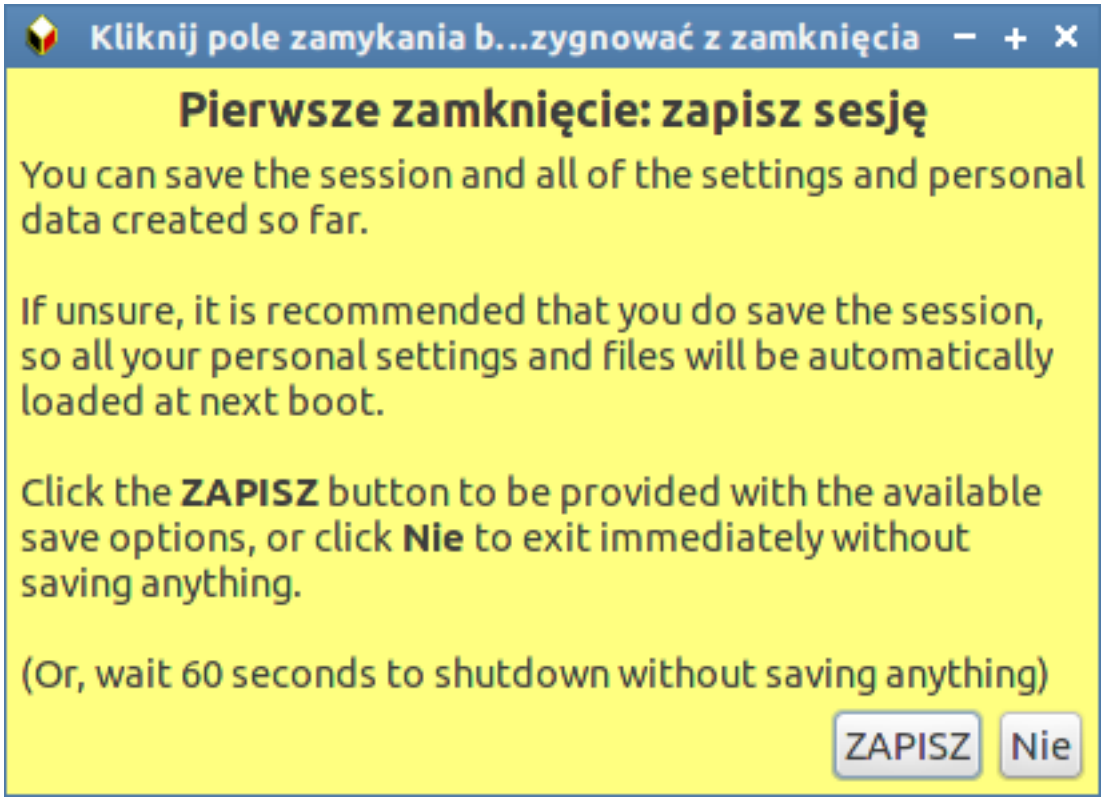
Następnie wybieramy szyfrowanie i system plików. Sugerujemy brak szyfrowania, domyślny system *ext4* i początkowy rozmiar **512MB**.

Opcjonalnie rozszerzamy domyślną nazwę i potwierdzamy zapis.

Należy spokojnie poczekać na utworzenie pliku i wyłącznie komputera. Po ponownym uruchomieniu system będzie gotowy do pracy :-)

System wersji *FULL* zawiera:

- spolszczone prawie wszystkie elementy systemu;
- zaktualizowane listy oprogramowania;
- zaktualizowaną i spolszczoną przeglądarkę Pale Moon (otwartoźrodłówa, oparta na Firefoksie);
- fonty Ubuntu oraz podstawowe z Windows;
- podstawowe pakiety narzędziowe: *python-pip*, *python-virtualenv*, *git*;
- wszystkie biblioteki Pythona wymagane w poszczególnych scenariuszach;
- środowisko programistyczne *Geany IDE*, a także *PyCharm Professional* i *Sublime Text* jako *pakiety SFS*, które trzeba załadować;
- serwer Etherpada Lite – narzędzia do współpracy online;
- skonfigurowany interfejs LXDE;
- skonfigurowane skróty klawiszowe.



Połączenie z internetem

System *LxPupXenial* domyślnie wczytuje się w całości do pamięci RAM i uruchamia środowisko graficzne LXDE z zalogowanym użytkownikiem *root*, czyli administratorem w systemach linuksowych. Na początku będziesz chciał nawiązać połączenie z internetem.

Z menu "Start/Konfiguracja" uruchamiamy *Internet kreator połączenia*, klikamy "Wired or wireless LAN", w następnym oknie wybieramy narzędzie "Simple Network Setup".

Po jego uruchomieniu powinniśmy zobaczyć listę wykrytych interfejsów, z której wybieramy *eth0* dla połączenia kablowego, *wlan0* dla połączenia bezprzewodowego. W przypadku *eth0* połączenie powinno zostać skonfigurowane od razu, natomiast w przypadku *wlan0* wskazujemy jeszcze odpowiednią sieć, metodę zabezpieczeń i podajemy hasło.

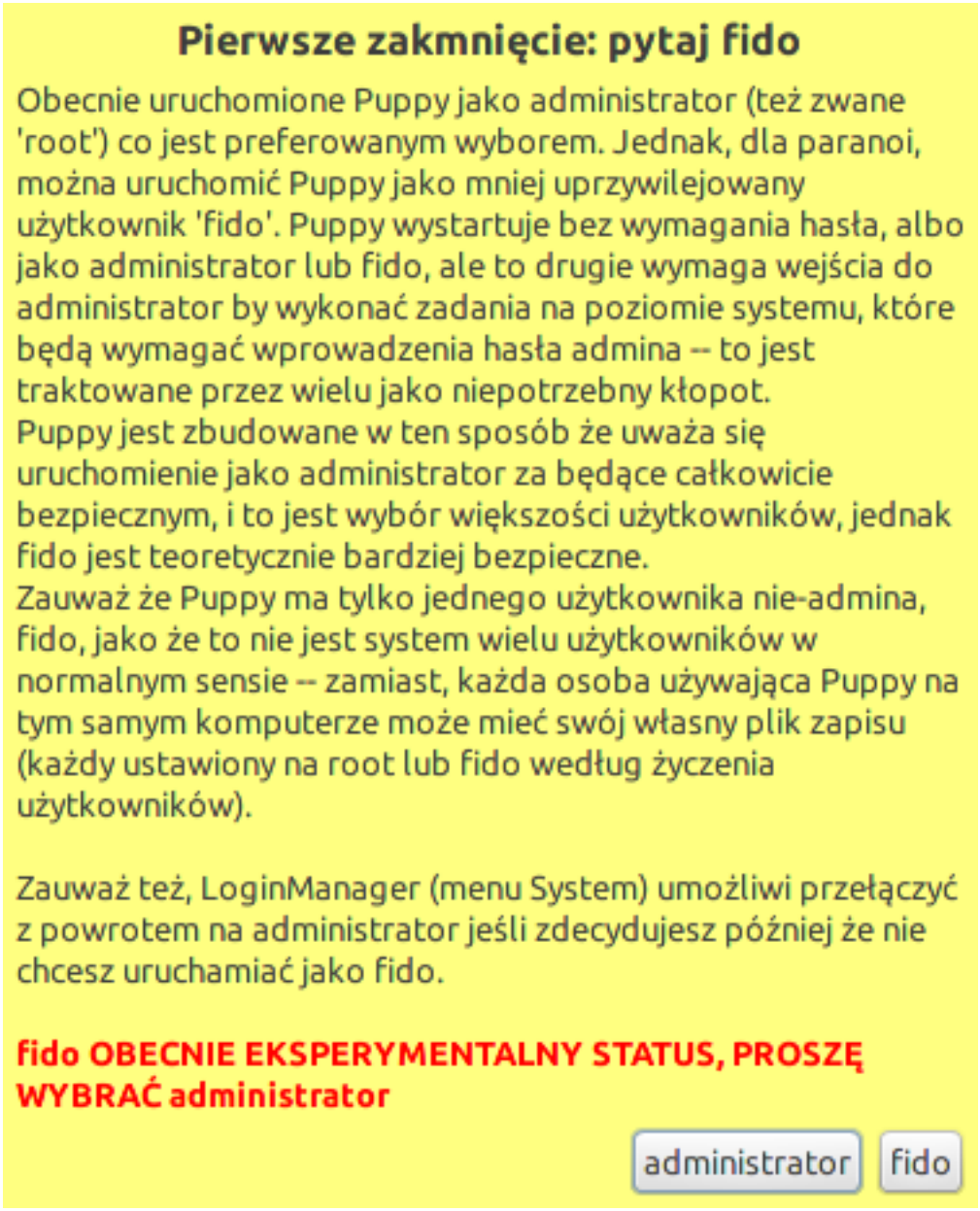
Jeżeli uzyskamy połączenie, w oknie "Network Connection Wizard/Kreator Połączenia Sieci" zobaczymy aktywne interfejsy. Sugerujemy kliknąć "Cancel/Anuluj", a w ostatnim oknie informacyjnym "Ok".

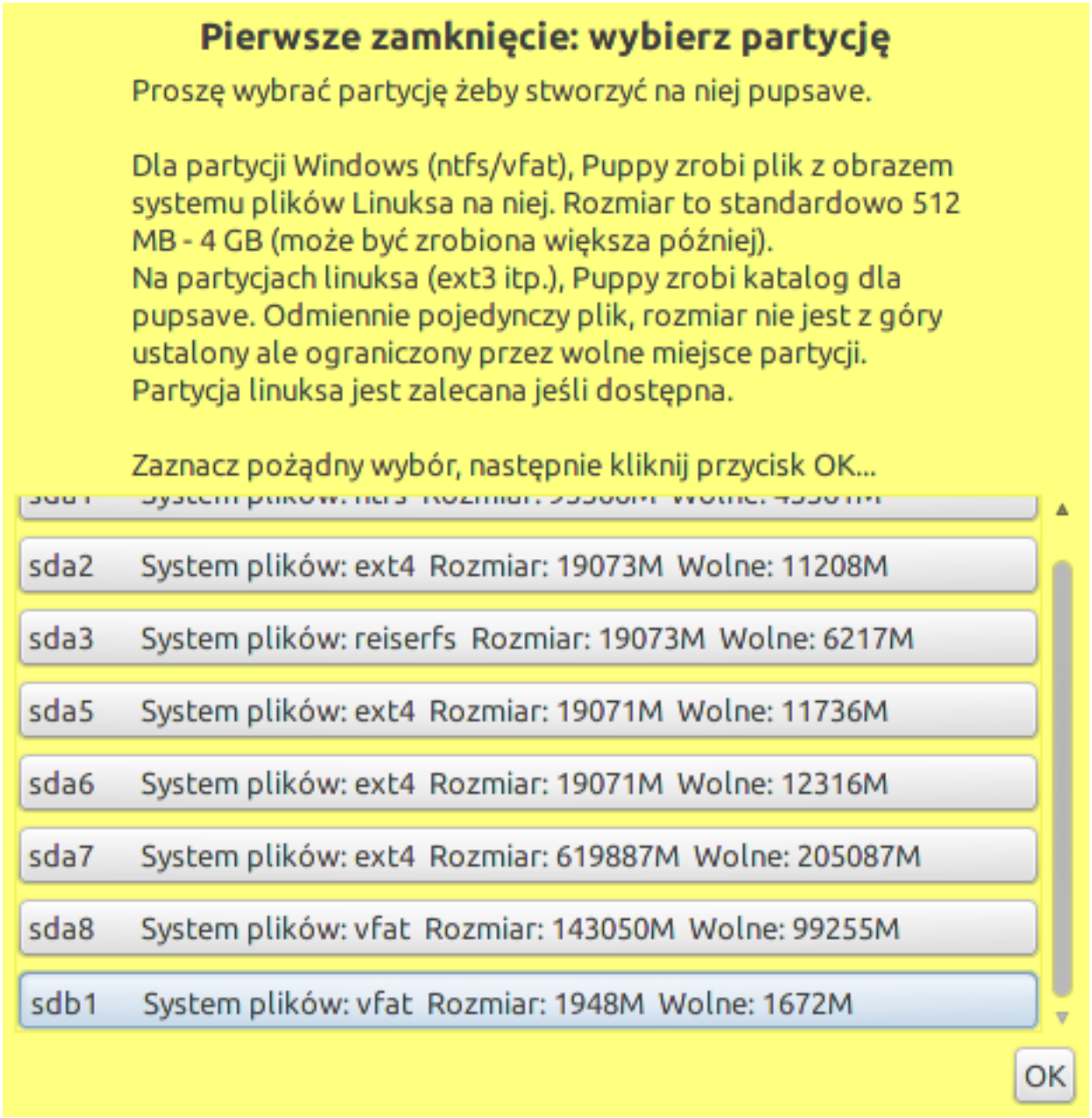
Równie proste i dobre są dwa pozostałe narzędzia, tzn. **Frisbee** i **Network Wizard**.

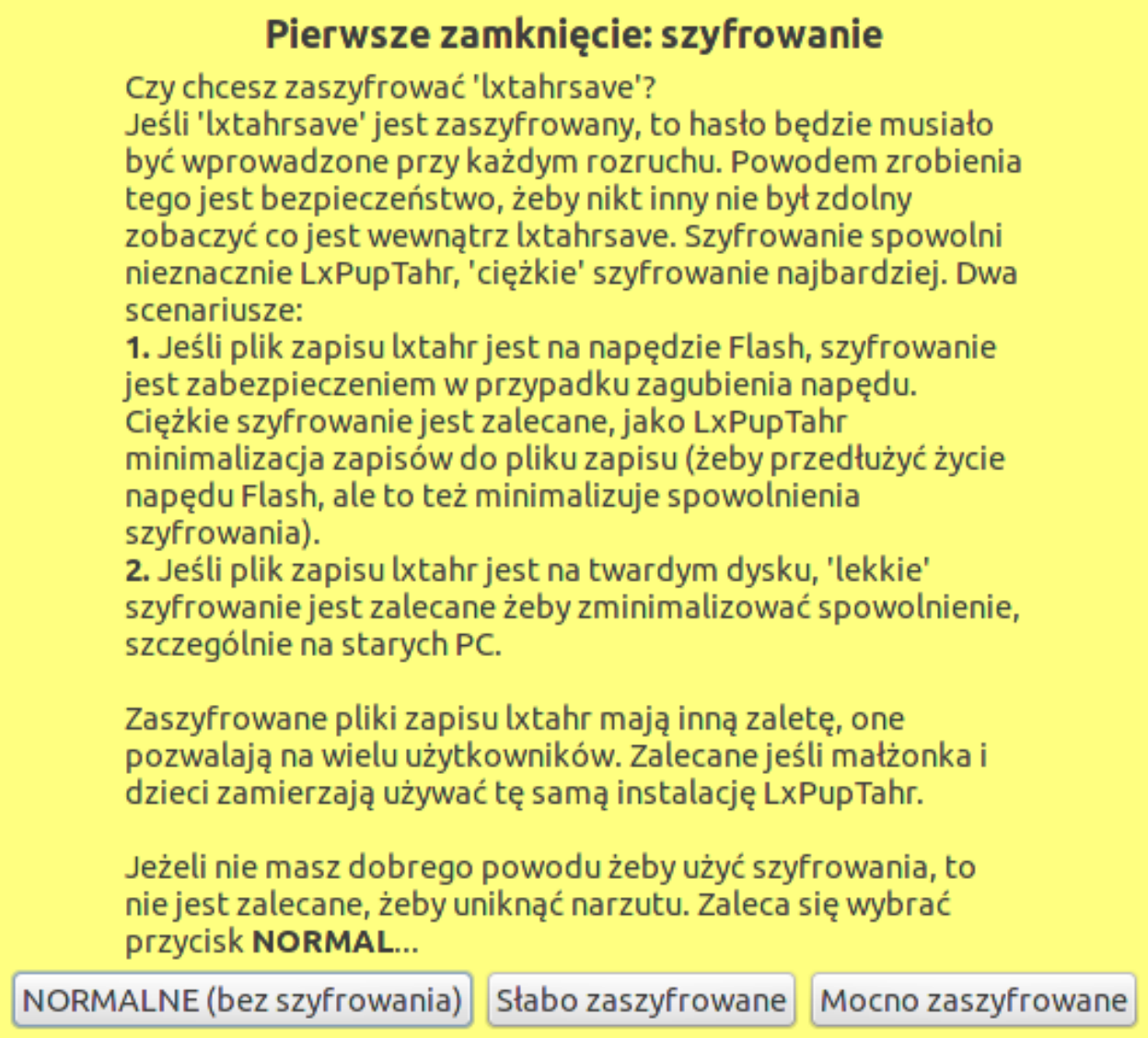
Pliki SFS i PET

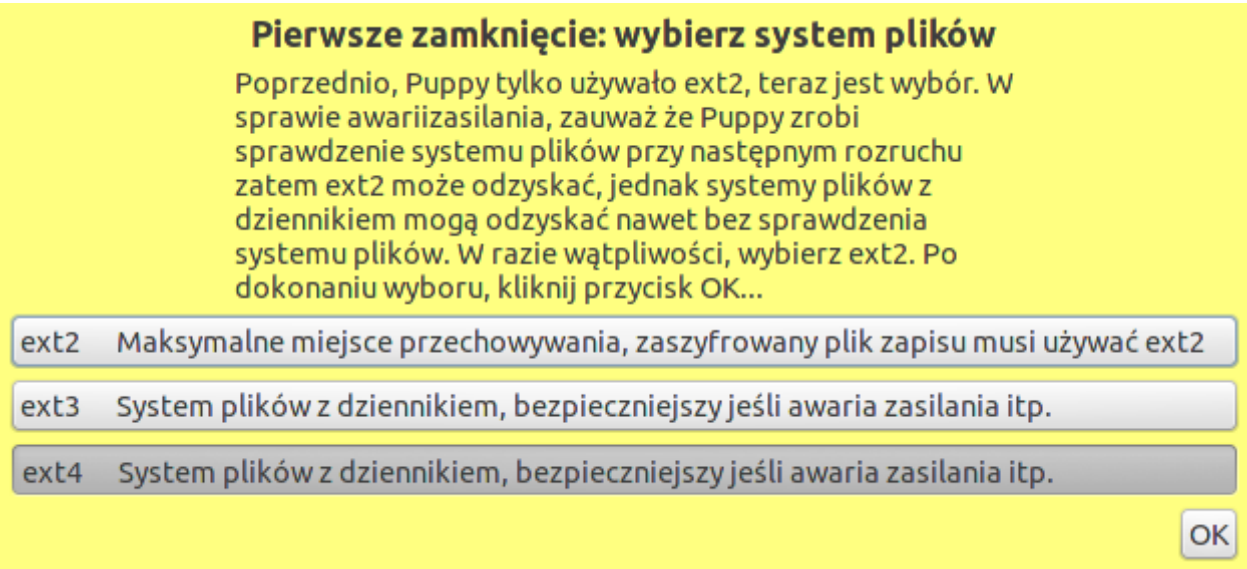
LxPup oferuje dwa dedykowane formaty plików zawierających oprogramowanie. Edytory *PyCharm* i *SublimeText3*, a także *serwer Etherpad* umożliwiający wspólne redagownie dokumentów w czasie rzeczywistym przygotowaliśmy w formie plików SFS. W wersji FULL są one już dołączone. Jeżeli ściągneliśmy obraz BASE lub chcemy mieć ostatnią dostępną wersję, ściągamy poniższe pliki:

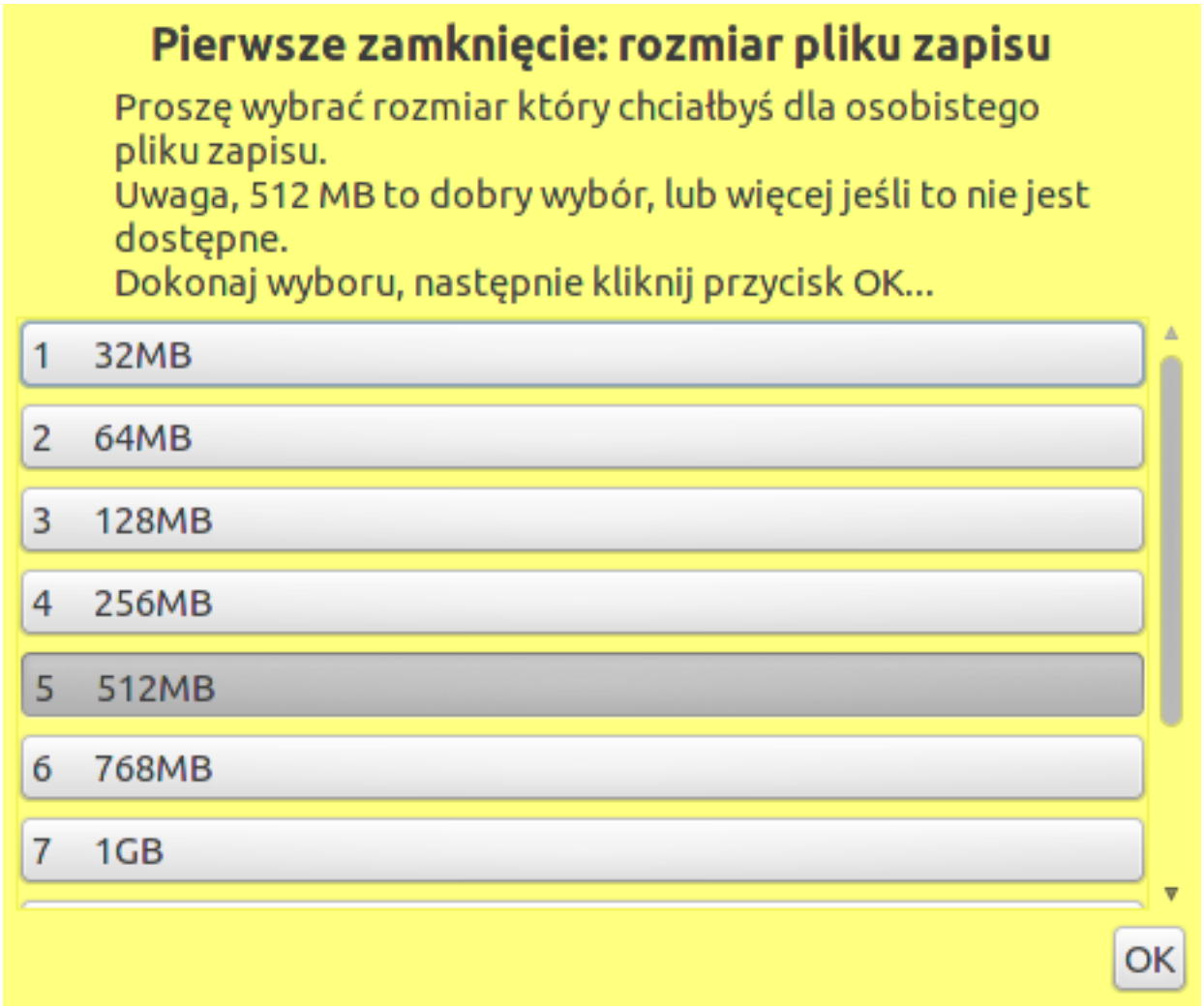
- PyCharm 2016.2
- Java JRE 1.8.65
- Sublime Text 3.126

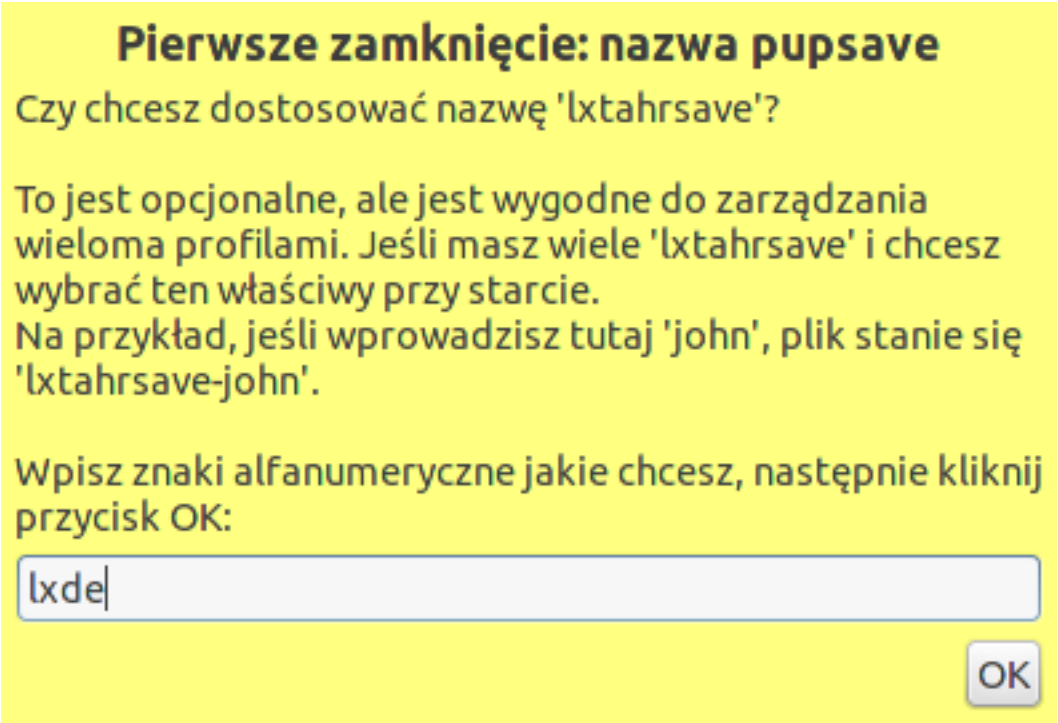

Pierwsze zakmnięcie: pytaj fido
Obecnie uruchomione Puppy jako administrator (też zwane 'root') co jest preferowanym wyborem. Jednak, dla paranoi, można uruchomić Puppy jako mniej uprzywilejowany użytkownik 'fido'. Puppy wystartuje bez wymagania hasła, albo jako administrator lub fido, ale to drugie wymaga wejścia do administrator by wykonać zadania na poziomie systemu, które będą wymagać wprowadzenia hasła admina -- to jest traktowane przez wielu jako niepotrzebny kłopot.
Puppy jest zbudowane w ten sposób że uważa się uruchomienie jako administrator za będące całkowicie bezpiecznym, i to jest wybór większości użytkowników, jednak fido jest teoretycznie bardziej bezpieczne.
Zauważ że Puppy ma tylko jednego użytkownika nie-admina, fido, jako że to nie jest system wielu użytkowników w normalnym sensie -- zamiast, każda osoba używająca Puppy na tym samym komputerze może mieć swój własny plik zapisu (każdy ustawiony na root lub fido według życzenia użytkowników).
Zauważ też, LoginManager (menu System) umożliwi przełączyć z powrotem na administrator jeśli zdecydujesz później że nie chcesz uruchamiać jako fido.
fido OBECNIE EKSPERYMENTALNY STATUS, PROSZĘ WYBRAĆ administrator
administrator
fido

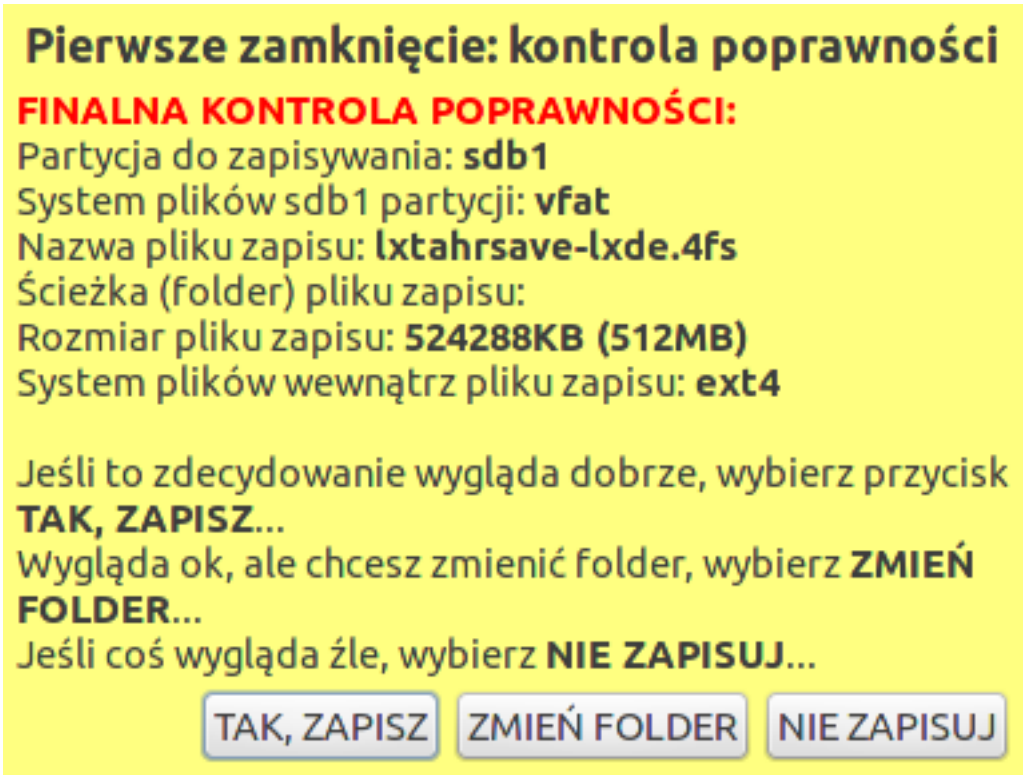

Pierwsze zamknięcie: wybierz partycję
Proszę wybrać partycję żeby stworzyć na niej pupsave.
Dla partycji Windows (ntfs/vfat), Puppy zrobi plik z obrazem systemu plików Linuksa na niej. Rozmiar to standardowo 512 MB - 4 GB (może być zrobiona większa później).
Na partycjach linuksa (ext3 itp.), Puppy zrobi katalog dla pupsave. Odmiennie pojedynczy plik, rozmiar nie jest z góry ustalony ale ograniczony przez wolne miejsce partycji.
Partycja linuksa jest zalecana jeśli dostępna.
Zaznacz pożądny wybór, następnie kliknij przycisk OK...
sda2 System plików: ext4 Rozmiar: 19073M Wolne: 11208M
sda3 System plików: reiserfs Rozmiar: 19073M Wolne: 6217M
sda5 System plików: ext4 Rozmiar: 19071M Wolne: 11736M
sda6 System plików: ext4 Rozmiar: 19071M Wolne: 12316M
sda7 System plików: ext4 Rozmiar: 619887M Wolne: 205087M
sda8 System plików: vfat Rozmiar: 143050M Wolne: 99255M
sdb1 System plików: vfat Rozmiar: 1948M Wolne: 1672M
OK

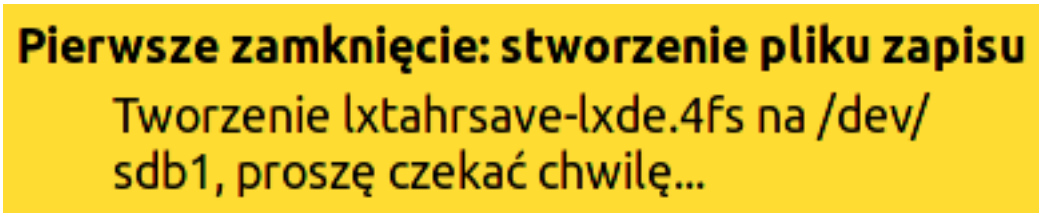

Pierwsze zamknięcie: szyfrowanie
Czy chcesz zaszyfrować 'lxtahrsave'?
Jeśli 'lxtahrsave' jest zaszyfrowany, to hasło będzie musiało być wprowadzone przy każdym rozruchu. Powodem zrobienia tego jest bezpieczeństwo, żeby nikt inny nie był zdolny zobaczyć co jest wewnątrz lxtahrsave. Szyfrowanie spowolni nieznacznie LxPupTahr, 'ciężkie' szyfrowanie najbardziej. Dwa scenariusze:
1. Jeśli plik zapisu lxtahr jest na napędzie Flash, szyfrowanie jest zabezpieczeniem w przypadku zagubienia napędu. Ciężkie szyfrowanie jest zalecane, jako LxPupTahr minimalizacja zapisów do pliku zapisu (żeby przedłużyć życie napędu Flash, ale to też minimalizuje spowolnienia szyfrowania).
2. Jeśli plik zapisu lxtahr jest na twardym dysku, 'lekkie' szyfrowanie jest zalecane żeby zminimalizować spowolnienie, szczególnie na starych PC.
Zaszyfrowane pliki zapisu lxtahr mają inną zaletę, one pozwalają na wielu użytkowników. Zalecane jeśli małżonka i dzieci zamierzają używać tę samą instalację LxPupTahr.
Jeżeli nie masz dobrego powodu żeby użyć szyfrowania, to nie jest zalecane, żeby uniknąć narzutu. Zaleca się wybrać przycisk NORMAL...
NORMALNE (bez szyfrowania)
Słabo zaszyfrowane
Mocno zaszyfrowane

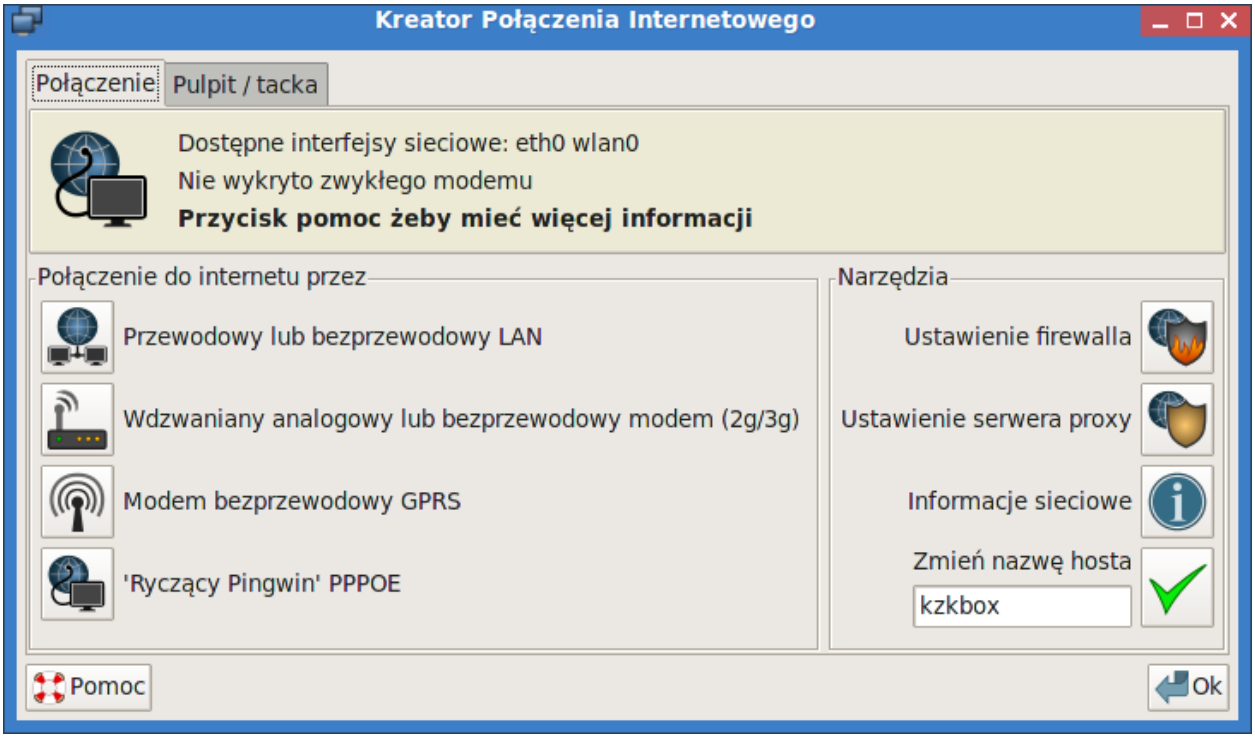

Pierwsze zamknięcie: wybierz system plików
Poprzednio, Puppy tylko używało ext2, teraz jest wybór. W sprawie awariizasilania, zauważ że Puppy zrobi sprawdzenie systemu plików przy następnym rozruchu zatem ext2 może odzyskać, jednak systemy plików z dziennikiem mogą odzyskać nawet bez sprawdzenia systemu plików. W razie wątpliwości, wybierz ext2. Po dokonaniu wyboru, kliknij przycisk OK...
ext2 Maksymalne miejsce przechowywania, zaszyfrowany plik zapisu musi używać ext2
ext3 System plików z dziennikiem, bezpieczniejszy jeśli awaria zasilania itp.
ext4 System plików z dziennikiem, bezpieczniejszy jeśli awaria zasilania itp.
OK


Pierwsze zamknięcie: rozmiar pliku zapisu
Proszę wybrać rozmiar który chciałbyś dla osobistego pliku zapisu.
Uwaga, 512 MB to dobry wybór, lub więcej jeśli to nie jest dostępne.
Dokonaj wyboru, następnie kliknij przycisk OK...
1 32MB
2 64MB
3 128MB
4 256MB
5 512MB
6 768MB
7 1GB
OK

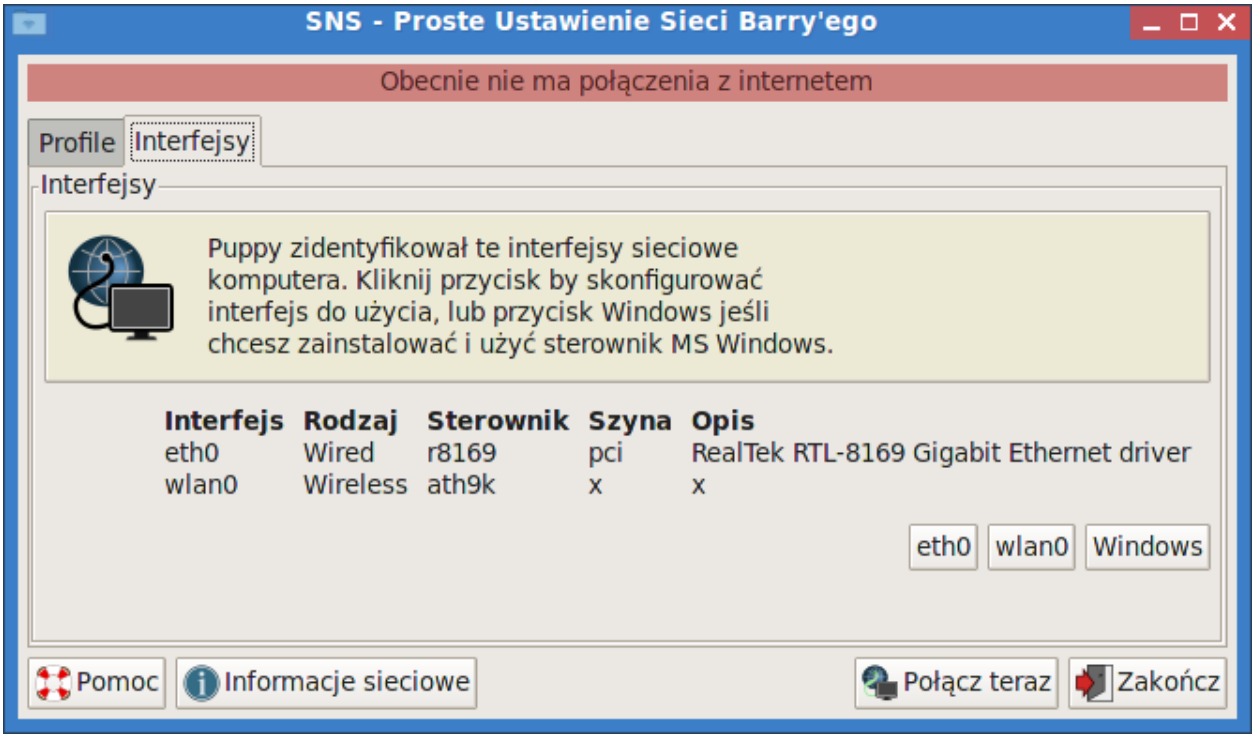

Pierwsze zamknięcie: nazwa pupsave
Czy chcesz dostosować nazwę 'lxtahrsave'?
To jest opcjonalne, ale jest wygodne do zarządzania wieloma profilami. Jeśli masz wiele 'lxtahrsave' i chcesz wybrać ten właściwy przy starcie.
Na przykład, jeśli wprowadzisz tutaj 'john', plik stanie się 'lxtahrsave-john'.
Wpisz znaki alfanumeryczne jakie chcesz, następnie kliknij przycisk OK:
lxde
OK


Pierwsze zamknięcie: kontrola poprawności
FINALNA KONTROLA POPRAWNOŚCI:
Partycja do zapisywania: sdb1
System plików sdb1 partycji: vfat
Nazwa pliku zapisu: lxtahrsave-lxde.4fs
Ścieżka (folder) pliku zapisu:
Rozmiar pliku zapisu: 524288KB (512MB)
System plików wewnątrz pliku zapisu: ext4
Jeśli to zdecydowanie wygląda dobrze, wybierz przycisk TAK, ZAPISZ...
Wygląda ok, ale chcesz zmienić folder, wybierz ZMIEŃ FOLDER...
Jeśli coś wygląda źle, wybierz NIE ZAPISUJ...
TAK, ZAPISZ
ZMIEŃ FOLDER
NIE ZAPISUJ


Pierwsze zamknięcie: stworzenie pliku zapisu
Tworzenie lxtahrsave-lxde.4fs na /dev/
sdb1, proszę czekać chwilę...


Kreator Połączenia Internetowego
Połączenie
Pulpit / tacka
Dostępne interfejsy sieciowe: eth0 wlan0
Nie wykryto zwykłego modemu
Przycisk pomoc żeby mieć więcej informacji
Połączenie do internetu przez
Przewodowy lub bezprzewodowy LAN
Wdzwaniany analogowy lub bezprzewodowy modem (2g/3g)
Modem bezprzewodowy GPRS
'Ryczący Pingwin' PPPOE
Narzędzia
Ustawienie firewalla
Ustawienie serwera proxy
Informacje sieciowe
Zmień nazwę hosta
kzkbox
Pomoc
Ok


Kreator Połączenia Sieci
To Puppy ma następujące wybory narzędzia GUI które można wybrać do połączenia do sieci (i internetu)...
Simple Network Setup
To jest narzędzie prostego ustawienia sieci stworzone przez BarregoK. Nie tak wiele funkcji jak Kreator Sieci, na przykład wymaga sieci by mieć serwer DHCP (zdecydowana większość przypadków). Największą funkcją jest ekstremalna łatwość użycia. Zalecane dla większości ludzi.
Frisbee
To jest narzędzie opracowane przez Jemimaha. W większości dla bezprzewodowej łączności dla laptopów, ale także może obsłużyć połączenie ethernet. Bardzo dobre dla roamingu bezprzewodowego.
Network Wizard
To jest najbardziej wyrafinowane narzędzie dla ustawienia sieci w Puppy. Bazuje na 'Klasycznym' Kreatorze Sieci oryginalnie napisanym przez BarregoK, ale świetnie zwiększonym przez Dougala, Rarsę, Shinobara i innych. Jest dla obu połączeń - przewodowego (ethernet) i bezprzewodowego, z większą liczbą funkcji konfiguracji niż Pwireless i SNS.
Anuluj



- Etherpad Lite

Pobrane pliki umieszczamy w katalogu głównym pendrajwa. W działającym systemie dostępny jest on w ścieżce `/mnt/home`, którą należy wpisać w pole adresu menedżera plików:

Załadowanie modułu sprowadza się do dwukrotnego kliknięcia wgranego pliku i wybraniu "Zainstaluj SFS":

Można również użyć programu *Start/Konfiguracja/SFS-Ładowanie w locie* lub polecenia `sfs_load` w terminalu. W oknie dialogowym z rozwijalnej listy wybieramy plik sfs i klikamy "Załaduj":

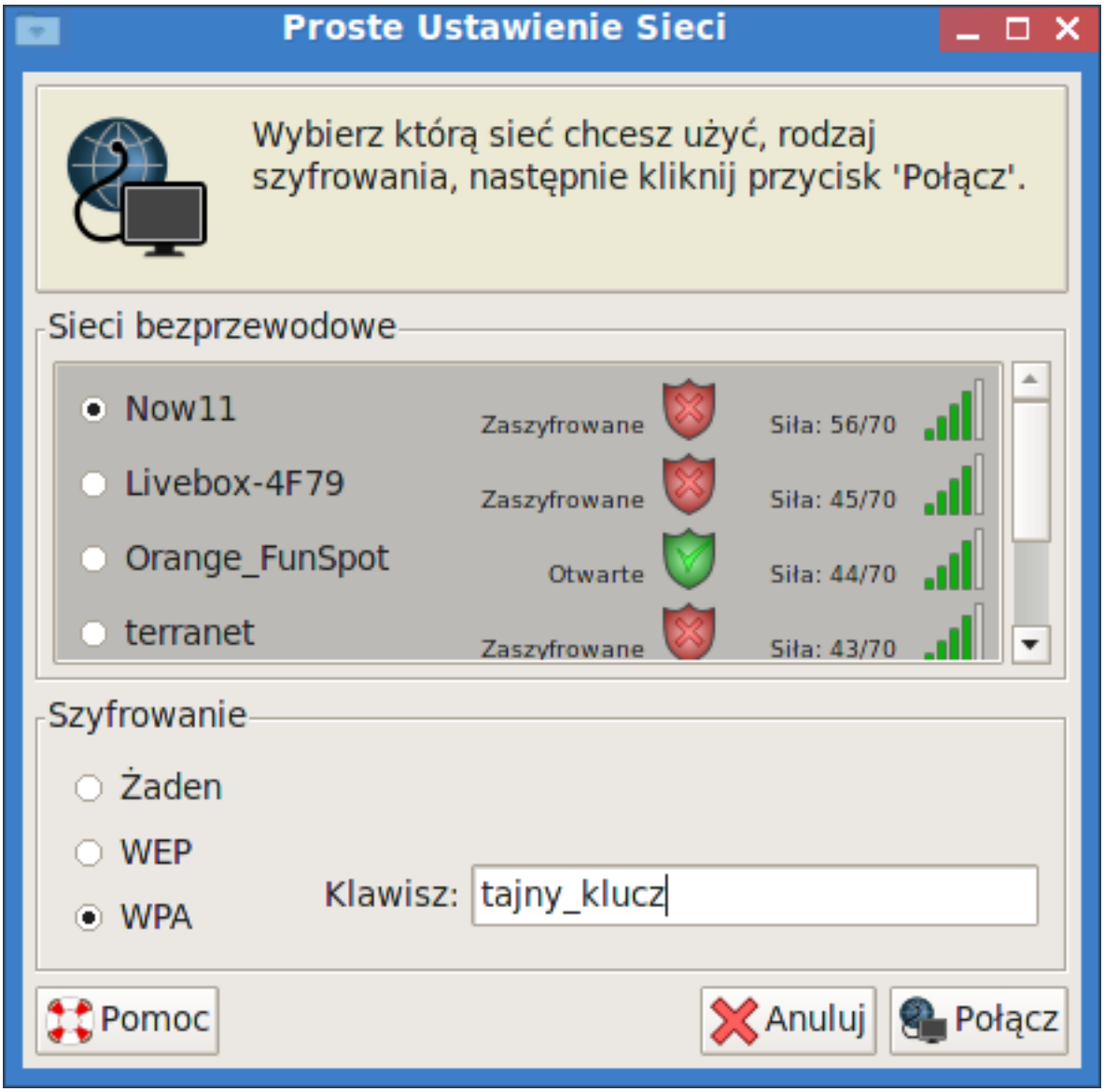
Po załadowaniu plików warto zrestartować menedżer okien: *Start/Zamknij/Restart WM*. Jeżeli nie potrzebujemy już danego programu lub chcemy go zaktualizować, pakiet SFS możemy też wyładować.

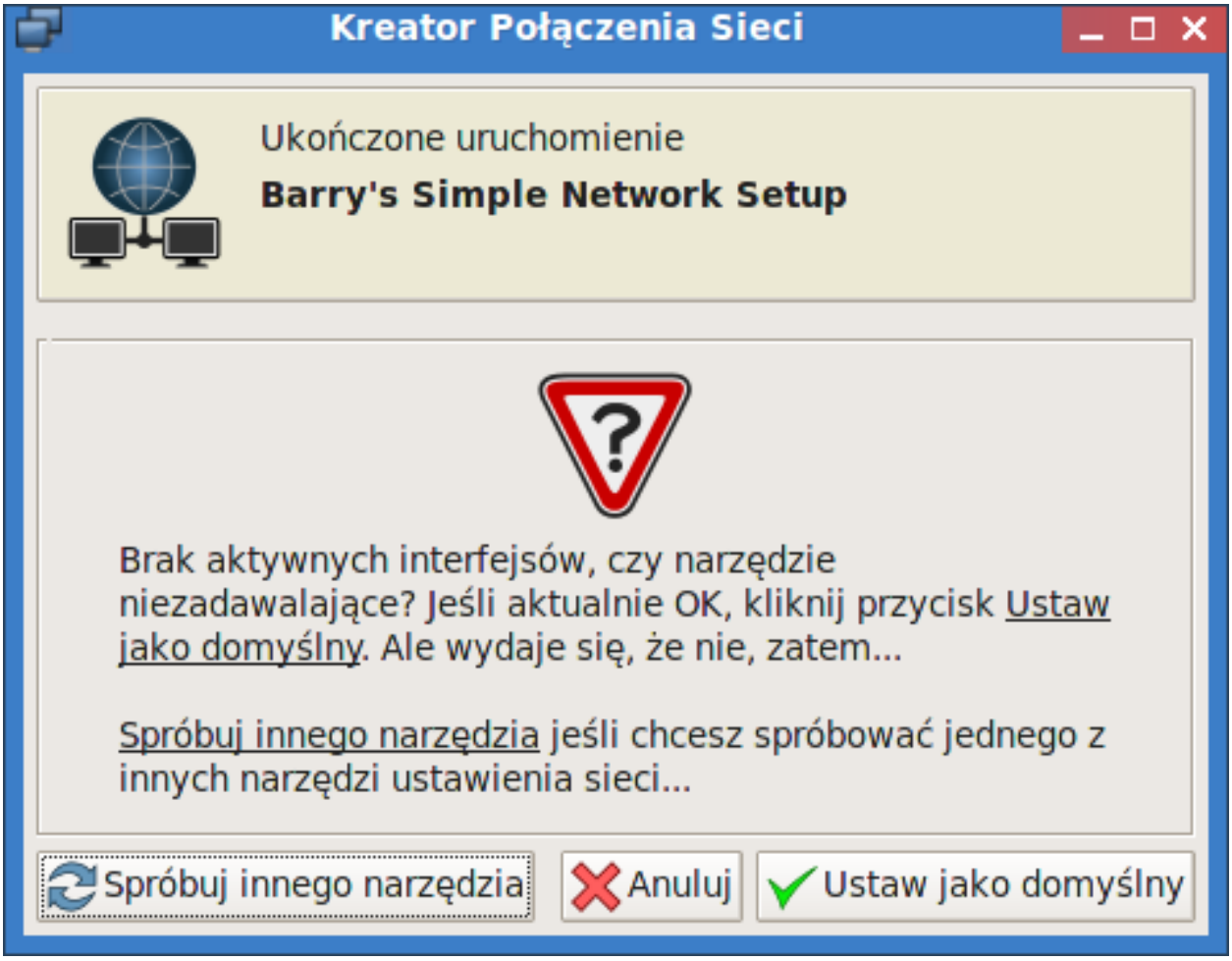
Drugi format dedykowany dla LxPupa to paczki w formacie PET, dostępne np. na stronie pet_packages. Ściągamy je, a następnie instalujemy dwukrotnie klikając (uruchomi się narzędzie *petget*).

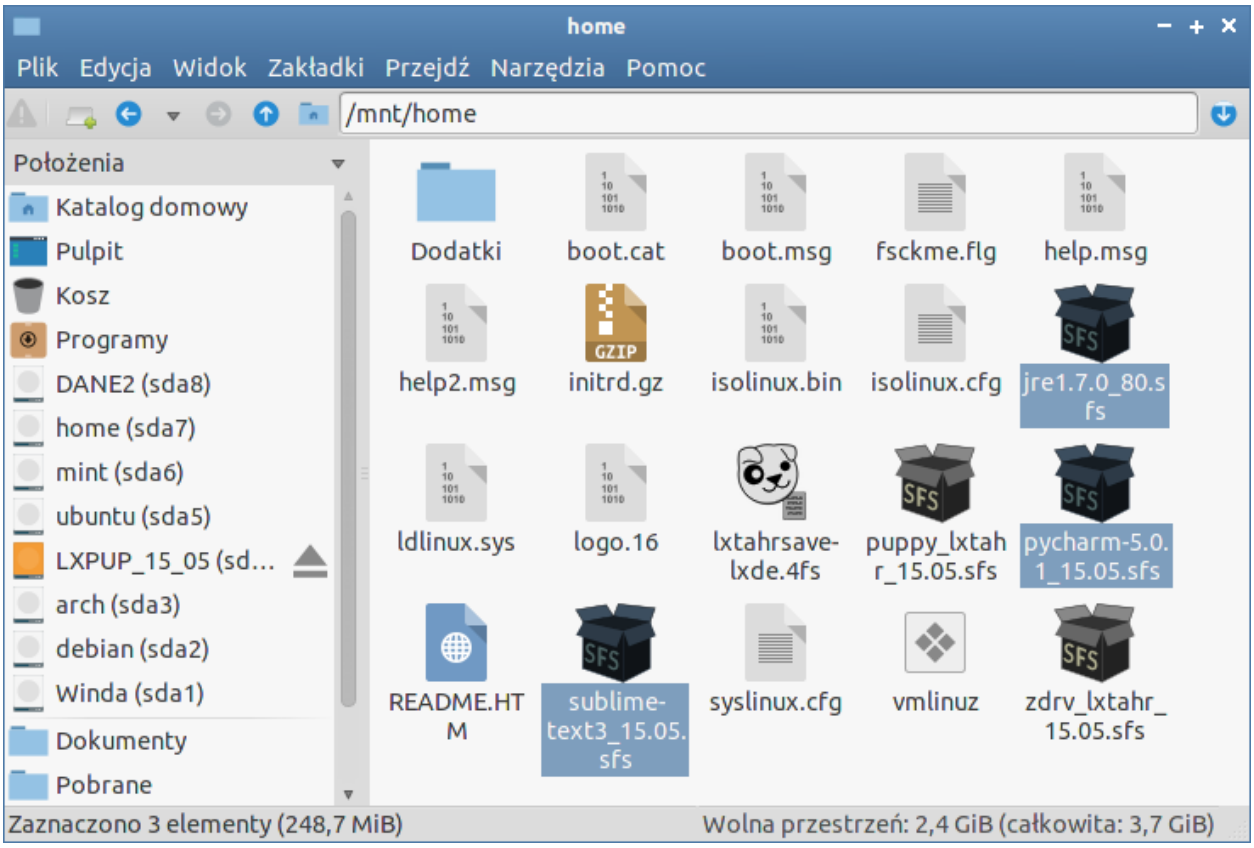
Informacja: W wersji LxPupTahr (ale nie w LxPupXenial) aktualizacje oraz programy w formatach SFS/PET przygotowywane przez społeczność można przeglądać i instalować za pomocą programu **Start/Konfiguracja/Quickpet tahr**. System aktualizujemy klikając "tahrpup updates". Później możemy zainstalować np. Chrome'a, Gimpa czy Skype'a.

Menedżer pakietów

Aby doinstalować jakiś pakiet (program), uruchamiamy **Start/Konfiguracja/Puppy Manager Pakietów**. Aktualizujemy listę dostępnych aplikacaji: klikamy ikonę ustawień obok koła ratunkowego, w następnym oknie zakładkę "Aktualizuj bazę danych" i przycisk "Aktualizuj teraz". Po uruchomieniu okna terminala klawiszem ENTER potwierdzamy aktualizację repozytoriów. Na koniec zamykamy okno aktualizacji przyciskiem "OK", co zrestartuje menedżera pakietów.


Proste Ustawienie Sieci
Wybierz którą sieć chcesz użyć, rodzaj szyfrowania, następnie kliknij przycisk 'Połącz'.
Sieci bezprzewodowe
Now11
Zaszyfrowane
Siła: 56/70
Livebox-4F79
Zaszyfrowane
Siła: 45/70
Orange_FunSpot
Otwarte
Siła: 44/70
terranet
Zaszyfrowane
Siła: 43/70
Szyfrowanie
Żaden
WEP
WPA
Klawisz:
tajny_klucz
Pomoc
Anuluj
Połącz


Kreator Połączenia Sieci
Ukończone uruchomienie
Barry's Simple Network Setup
Brak aktywnych interfejsów, czy narzędzie niezadawalające? Jeśli aktualnie OK, kliknij przycisk Ustaw jako domyślny. Ale wydaje się, że nie, zatem...
Spróbuj innego narzędzia jeśli chcesz spróbować jednego z innych narzędzi ustawienia sieci...
Spróbuj innego narzędzia
Anuluj
Ustaw jako domyślny



Po ponownym uruchomieniu PPM, wpisujemy nazwę szukanego pakietu w pole wyszukiwania, następnie klikamy pakiet na liście, co dodaje go do kolejki. W ten sposób możemy wyszukać i dodać kilka pakietów na raz. Na koniec zatwierdzamy instalację przyciskiem “Do it!”

Przeglądarka WWW

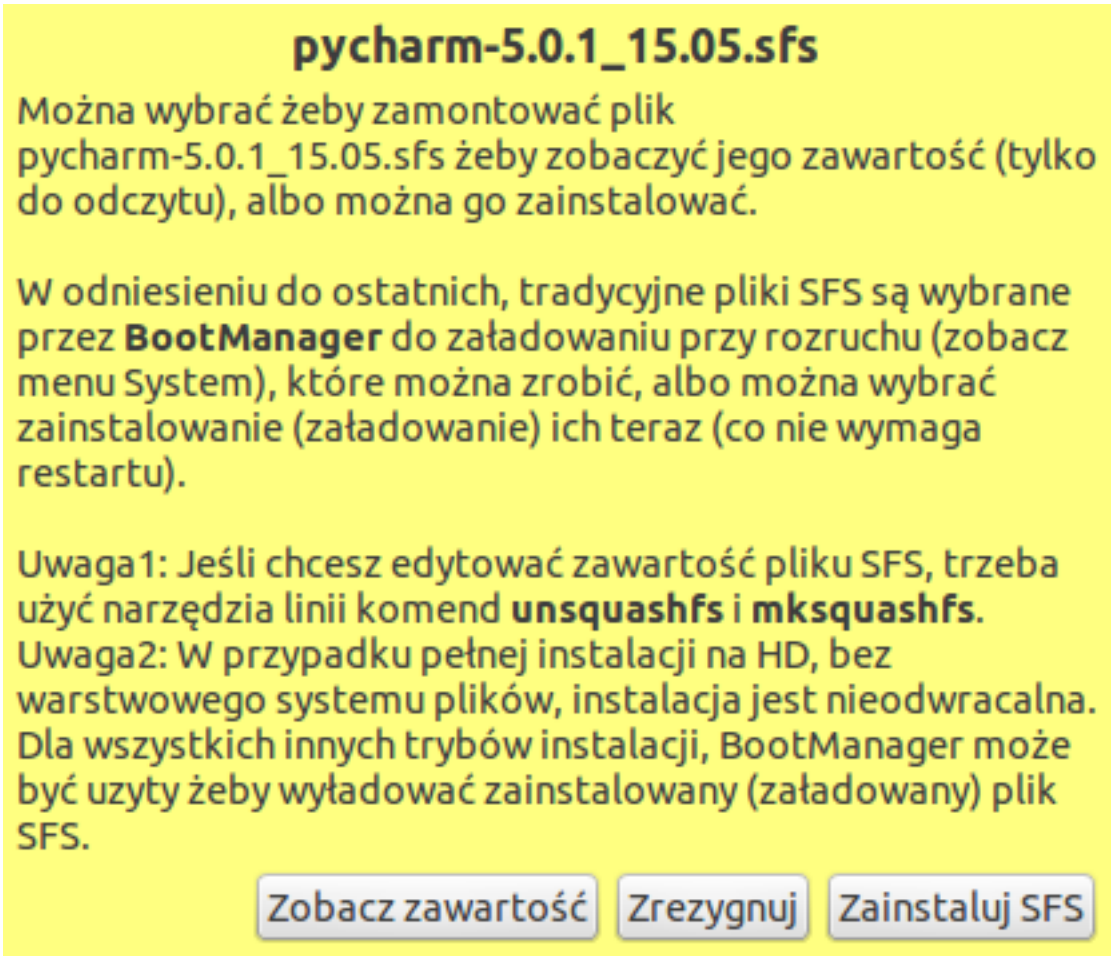
Domyślną przeglądarką jest PaleMoon, otwartoźródłowa odmiana oparta na Firefoksie. Od czasu do czasu warto ją zaktualizować wybierając **Start/Internet/Update Palemoon**

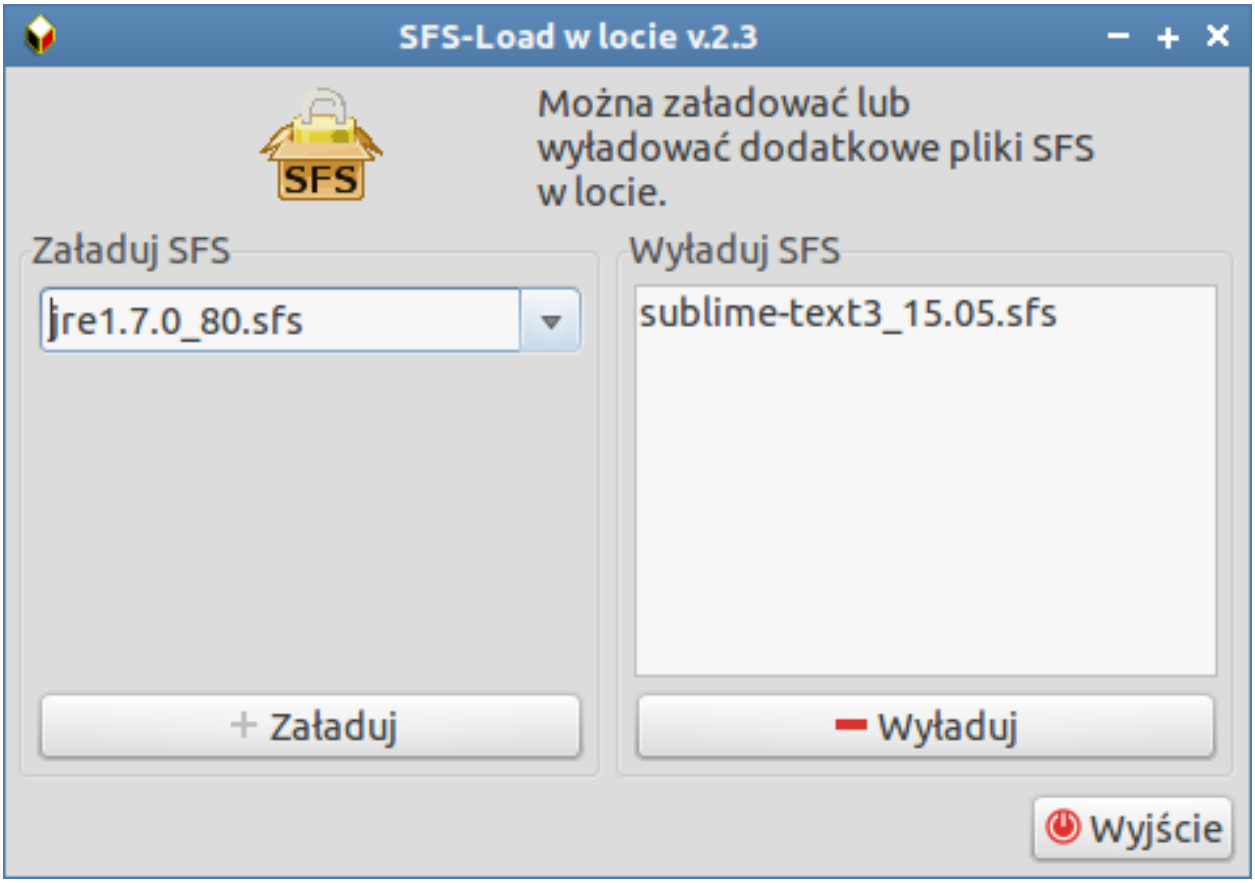
Domyślne katalogi

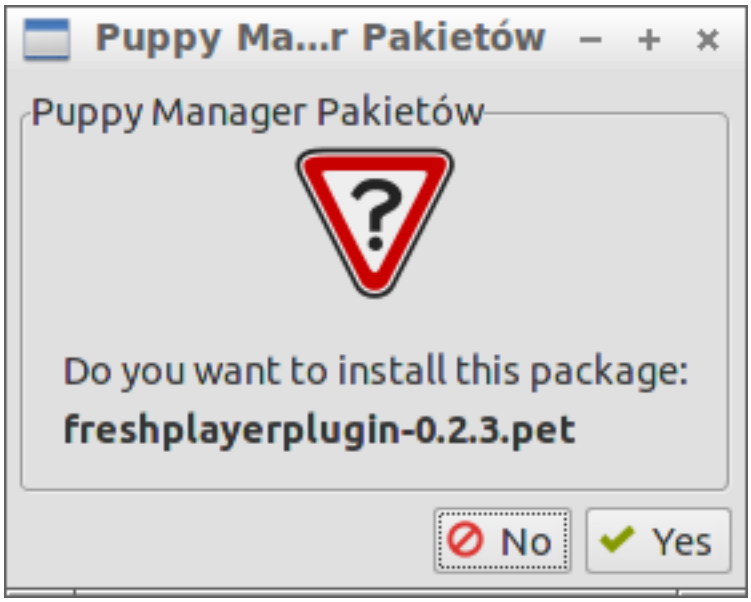
- `/root/my-documents` lub `/root/Dokumenty` – katalog na dokumenty
- `/root/Pobrane` – tu zapisywane są pliki pobierane z internetu
- `/root/my-documents/clipart` lub `/root/Obrazy`– katalog na obrazki
- `/root/my-documents/tmp` lub `/root/tmp` – katalogi tymczasowe
- `/root/LxPupUSB` lub `/mnt/home` – ścieżki do głównego katalogu napędu USB
- `/usr/share/fonts/default/TTF/` – dodatkowe czcionki TrueType, np. z MS Windows

Skróty klawiaturowe

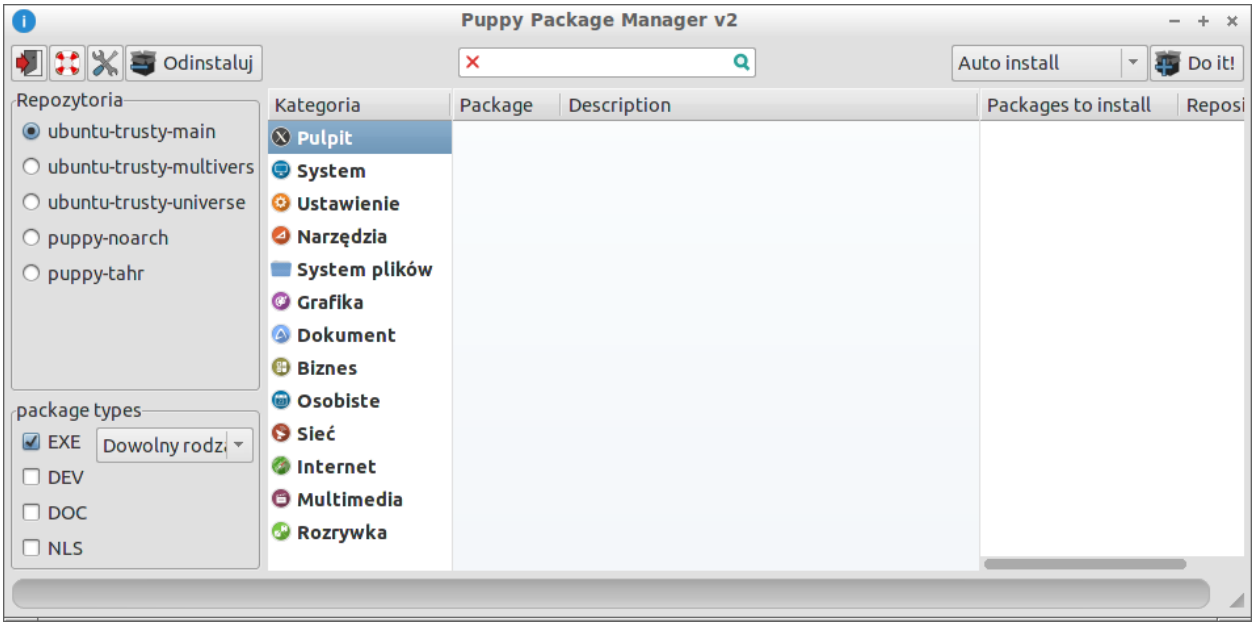
Oznaczenia: C – Control, A – Alt, W - Windows (SuperKey).

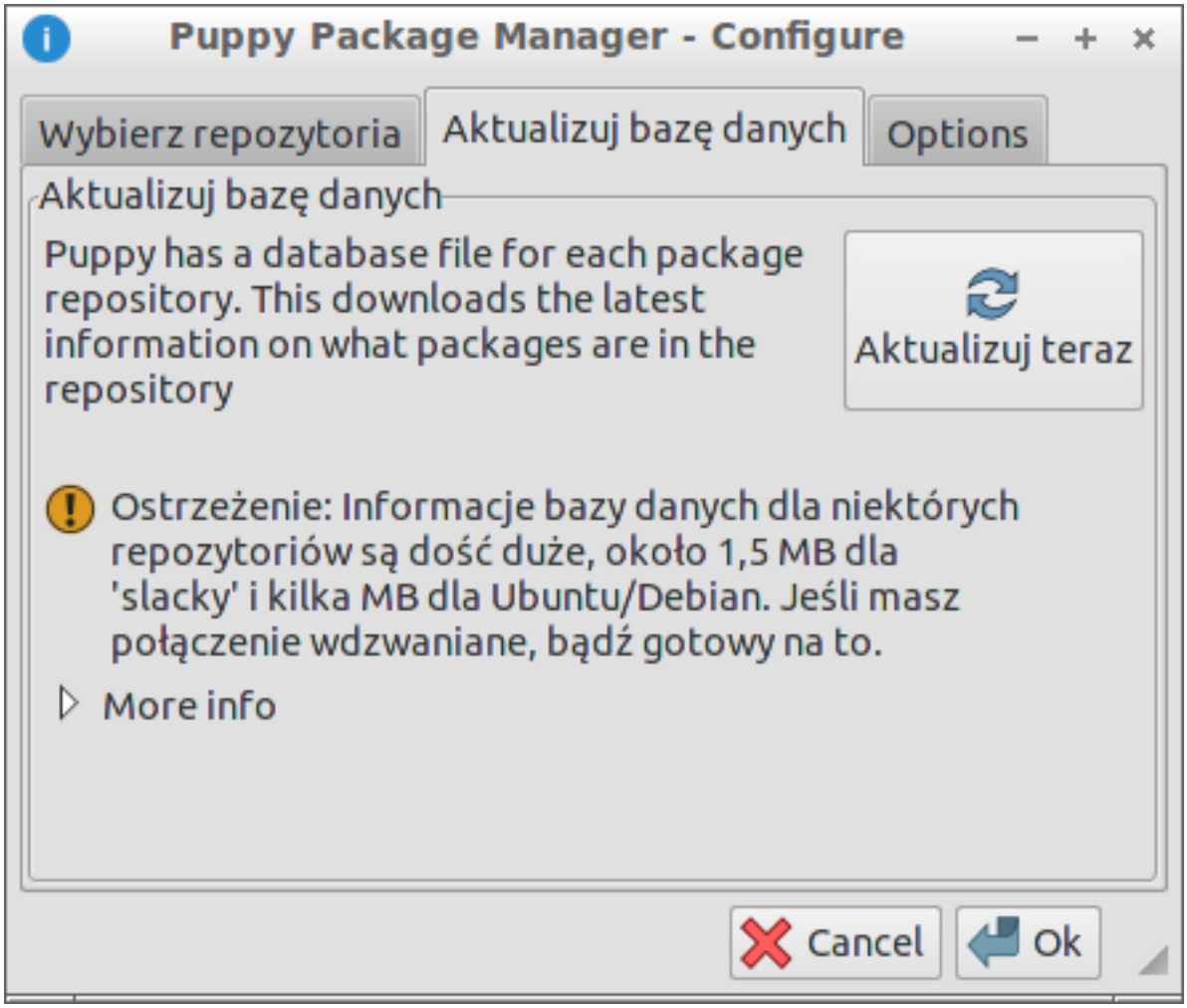

pycharm-5.0.1_15.05.sfs
Można wybrać żeby zamontować plik pycharm-5.0.1_15.05.sfs żeby zobaczyć jego zawartość (tylko do odczytu), albo można go zainstalować.
W odniesieniu do ostatnich, tradycyjne pliki SFS są wybrane przez BootManager do załadowaniu przy rozruchu (zobacz menu System), które można zrobić, albo można wybrać zainstalowanie (załadowanie) ich teraz (co nie wymaga restartu).
Uwaga1: Jeśli chcesz edytować zawartość pliku SFS, trzeba użyć narzędzia linii komend unsquashfs i mksquashfs.
Uwaga2: W przypadku pełnej instalacji na HD, bez warstwowego systemu plików, instalacja jest nieodwracalna.
Dla wszystkich innych trybów instalacji, BootManager może być uzyty żeby wyładować zainstalowany (załadowany) plik SFS.
Zobacz zawartość
Zrezygnuj
Zainstaluj SFS

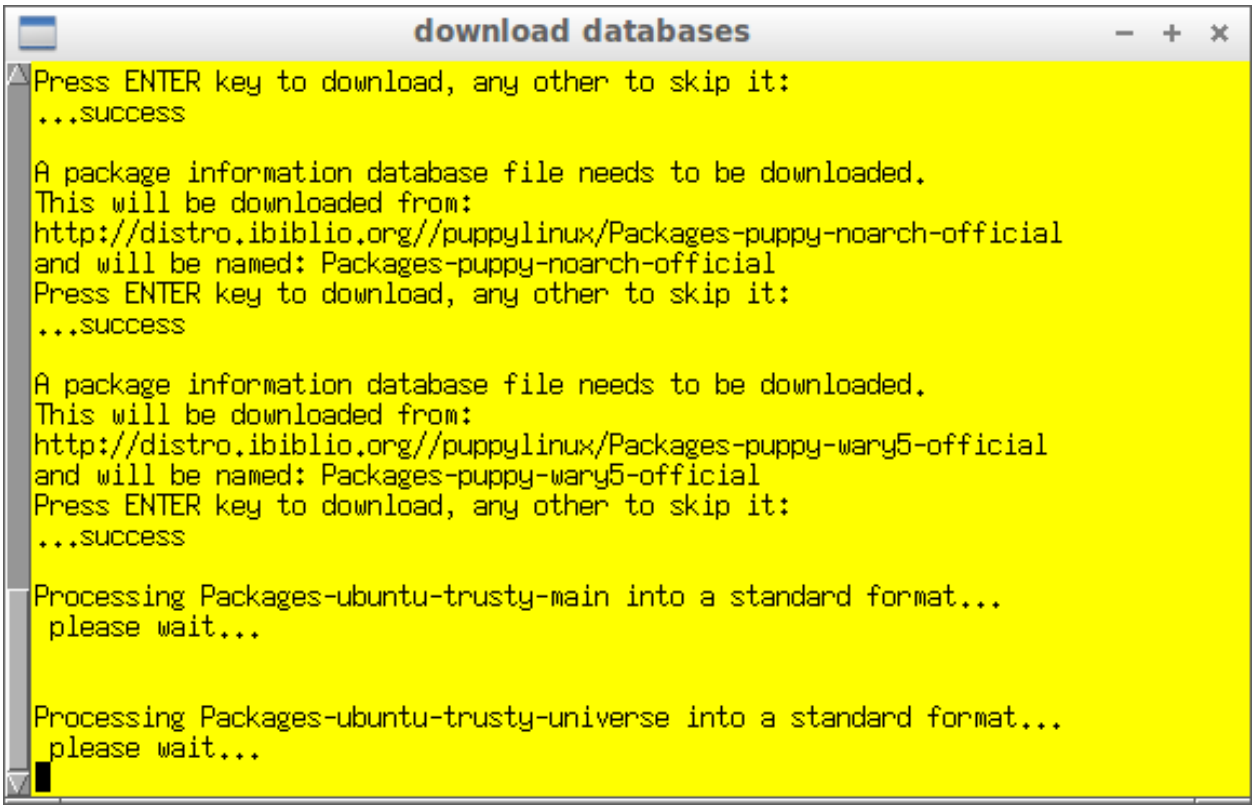

SFS-Load w locie v.2.3
SFS
Można załadować lub
wyładować dodatkowe pliki SFS
w locie.
Załaduj SFS
jre1.7.0_80.sfs
Załaduj
Wyładuj SFS
sublime-text3_15.05.sfs
Wyładuj
Wyjście

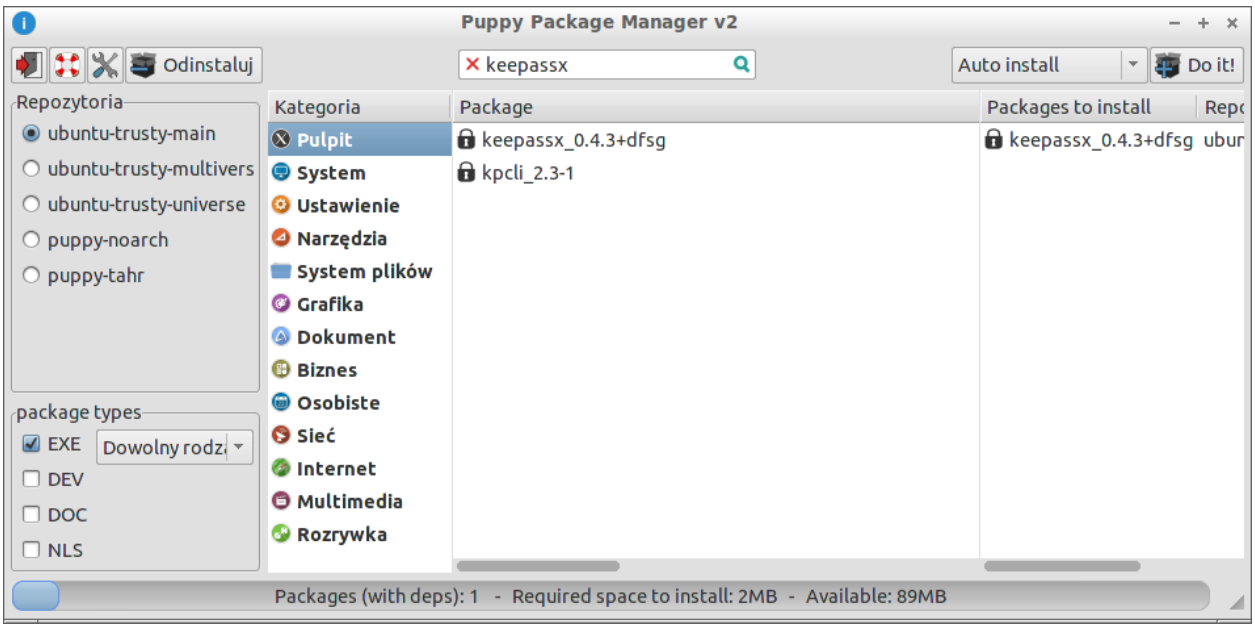

Puppy Ma...r Pakietów
Puppy Manager Pakietów
Do you want to install this package:
freshplayerplugin-0.2.3.pet
No
Yes


Quickpet Tahr
help
quickpet size
PPM
Info
Browsers
Internet
Audio
Video
Graphics
Games
Office&other
Useful pets
desktops
Drivers
Welcome to Quickpet
Quickpet enables you to install popular apps quickly.
more apps can be found in the puppy package manager
tahrpup updates
changing kernels
puppy search engine
Anuluj


Puppy Package Manager v2
Odinstaluj
Auto install
Do it!
Repozytoria
ubuntu-trusty-main
ubuntu-trusty-multivers
ubuntu-trusty-universe
puppy-noarch
puppy-tahr
package types
EXE
Dowolny rodz
DEV
DOC
NLS
Kategoria
Package
Description
Packages to install
Reposi
Pulpit
System
Ustawienie
Narzędzia
System plików
Grafika
Dokument
Biznes
Osobiste
Sieć
Internet
Multimedia
Rozrywka


Puppy Package Manager - Configure
Wybierz repozytoria
Aktualizuj bazę danych
Options
Aktualizuj bazę danych
Puppy has a database file for each package repository. This downloads the latest information on what packages are in the repository
Aktualizuj teraz
Ostrzeżenie: Informacje bazy danych dla niektórych repozytoriów są dość duże, około 1,5 MB dla 'slacky' i kilka MB dla Ubuntu/Debian. Jeśli masz połączenie wdzwaniane, bądź gotowy na to.
More info
Cancel
Ok


download databases
Press ENTER key to download, any other to skip it:
...success
A package information database file needs to be downloaded.
This will be downloaded from:
http://distro.ibiblio.org//puppylinux/Packages-puppy-noarch-official
and will be named: Packages-puppy-noarch-official
Press ENTER key to download, any other to skip it:
...success
A package information database file needs to be downloaded.
This will be downloaded from:
http://distro.ibiblio.org//puppylinux/Packages-puppy-wary5-official
and will be named: Packages-puppy-wary5-official
Press ENTER key to download, any other to skip it:
...success
Processing Packages-ubuntu-trusty-main into a standard format...
please wait...
Processing Packages-ubuntu-trusty-universe into a standard format...
please wait...


Puppy Package Manager v2
Odinstaluj
keepassx
Auto install
Do it!
Repozytoria
ubuntu-trusty-main
ubuntu-trusty-multivers
ubuntu-trusty-universe
puppy-noarch
puppy-tahr
package types
EXE
Dowolny rodz
DEV
DOC
NLS
Kategoria
Pulpit
System
Ustawienie
Narzędzia
System plików
Grafika
Dokument
Biznes
Osobiste
Sieć
Internet
Multimedia
Rozrywka
Package
keepassx_0.4.3+dfsg
kpcli_2.3-1
Packages to install
keepassx_0.4.3+dfsg
Repo
ubur
Packages (with deps): 1 - Required space to install: 2MB - Available: 89MB

- **C+A+Left** – puplpit lewy
- **C+A+Right** – pulpit prawy
- **Alt + Space** – menu okna
- **C+Esc** – menu start
- **C+A+Del** – menedżer zadań
- **W+f** – menedżer plików (pcmanfm)
- **W+t** – terminal (LXTerminal)
- **W+e** – Geany IDE
- **W+s** – Sublime Text 3
- **W+p** – PyCharm IDE
- **W+w** – przeglądarka WWW (Palemoon)
- **W+Góra**, **W+Dół**, **W+Lewo**, **W+Prawo**, **W+C**, **W+Alt+Lewo**, **W+Alt+Prawo** – sterowanie rozmiarem i położeniem okien

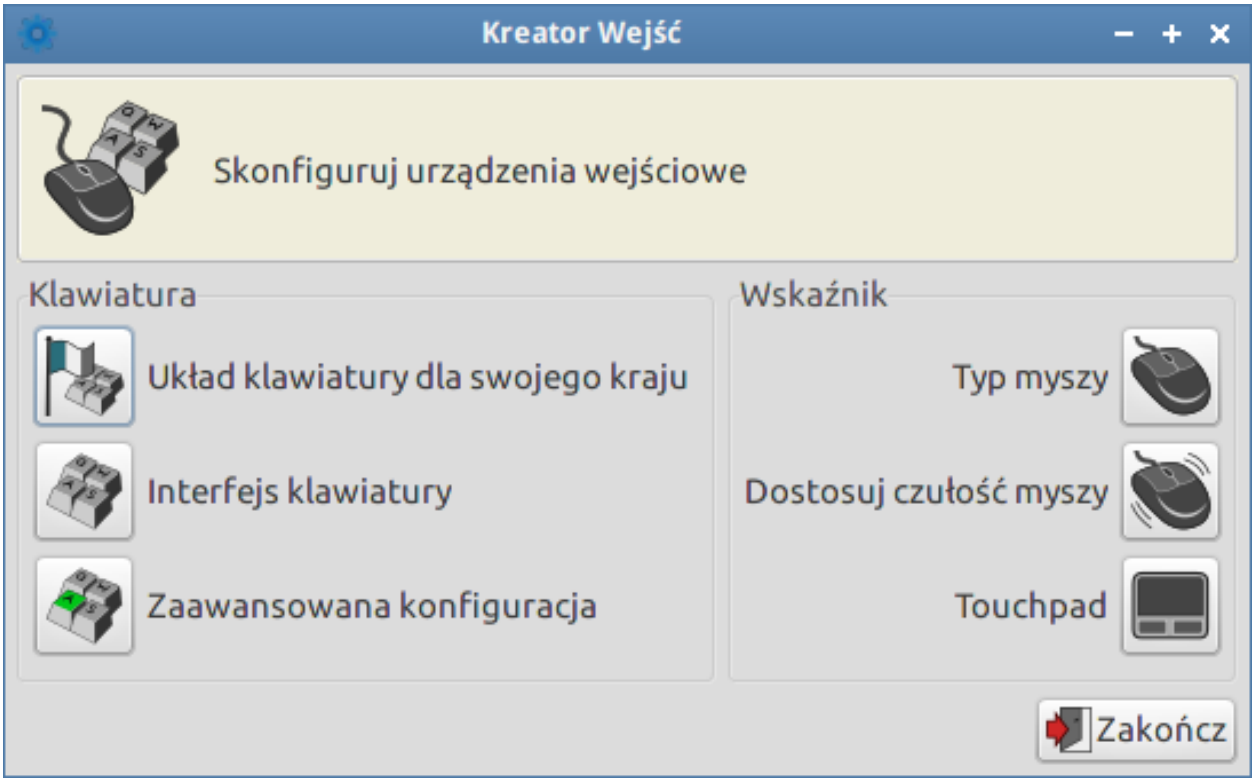
---

Wskazówka: Jeżeli skróty nie działają, ustawiamy odpowiedni model klawiatury. Procedura jest bardzo prosta. Uruchamiamy "Ustawienia Puppy" (pierwsza ikona obok przycisku Start, lub "Start/Konfiguracja/Wizard Kreator"), wybieramy "Mysz/Klawiatura". W następnym oknie "Zaawansowana konfiguracja", potwierdzamy "OK", dalej "Model klawiatury" i na koniec zaznaczamy **pc105**. Pozostaje potwierdzenie "OK" i jeszcze kliknięcie przycisku "Tak" w poprzednim oknie, aby aktywować ustawienia.

Konfiguracja LXDE

- **Wygląd, Ikony, Tapeta, Panel**: Start/Pulpit/Zmiana wyglądu.
- **Ekran(y)**: Start/System/System/Ustawienia wyświetlania.
- **Czcionki**: Start/Pulpit/Desktop/Manager Czcionki.
- **Wygładzanie czcionek**: Start/Pulpit/Desktop/Manager Czcionki, zakładka "Wygląd", "Styl hintingu" 1.
- **Menedżer plików**: Edycja/Preferencje w programie.
- **Ustawienia Puppy**: Start/Konfiguracja/Wizard Kreator
- **Internet kreator połączenia**: Start/Konfiguracja
- **Zmiana rozmiaru pliku zapisu**: Start/Akcesoria
- **Puppy Manager Pakietów**: Start/Konfiguracja
- **Quickpet tahr**: Start/Konfiguracja
- **SFS-załadowanie w locie**: Start/Konfiguracja/SFS-Załadowanie w locie
- **QuickSetup ustawienia pierwszego uruchamiania**: Start/Konfiguracja
- **Restart menedżera okien** (RestartWM): Start/Zamknij
- **WM Switcher** – switch windowmanagers:
- **Startup Control – kontrola aplikacji startowych**: Start/Konfiguracja
- **Domyślne aplikacje**: Start/Pulpit/Preferowane programy


Ustawienie Puppy
To jest podstawowy przewodnik ustawień dla Puppy.
Więcej konfiguracji można znaleźć w menu **Pulpit** i
Ustawienia.
Kreatory
Język / Kraj
Ustawienia startowe
Mysz / Klawiatura
Dźwięk
Grafika / Ekran
Akceleracja grafiki
Internet
Drukarka
Wyjście



- **Terminale** Start/Akcesoria
- **Ustawienie daty i czasu**: Start/Pulpit

Wskazówki

1. Dwukrotne kliknięcie – menedżer plików PcManFm domyślnie otwiera pliki i katalogi po pojedynczym kliknięciu. Jeżeli chcielibyśmy to zmienić, wybieramy "Edycja/Preferencje".
2. Jeżeli po uruchomieniu system nie wykrywa podłączonego monitora czy rzutnika, wybieramy "Start/Zamknij/Restart WM" – po restarcie menedżera okien obraz powinien pojawić się automatycznie. Możemy go dostosować wybierając "Start/System/Sytem/Ustawienia wyświetlania".
3. Jeżeli po uruchomieniu systemu nie działą ani myszka, ani klawiatura, restarujemy system i uruchamiamy go ponownie podając opcje **puppy pfix=nox**, co uruchomi system w trybie konsoli (bez okienek). Następnie wydajemy polecenie *xorgwizard* i wybieramy opcje domyślne.

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

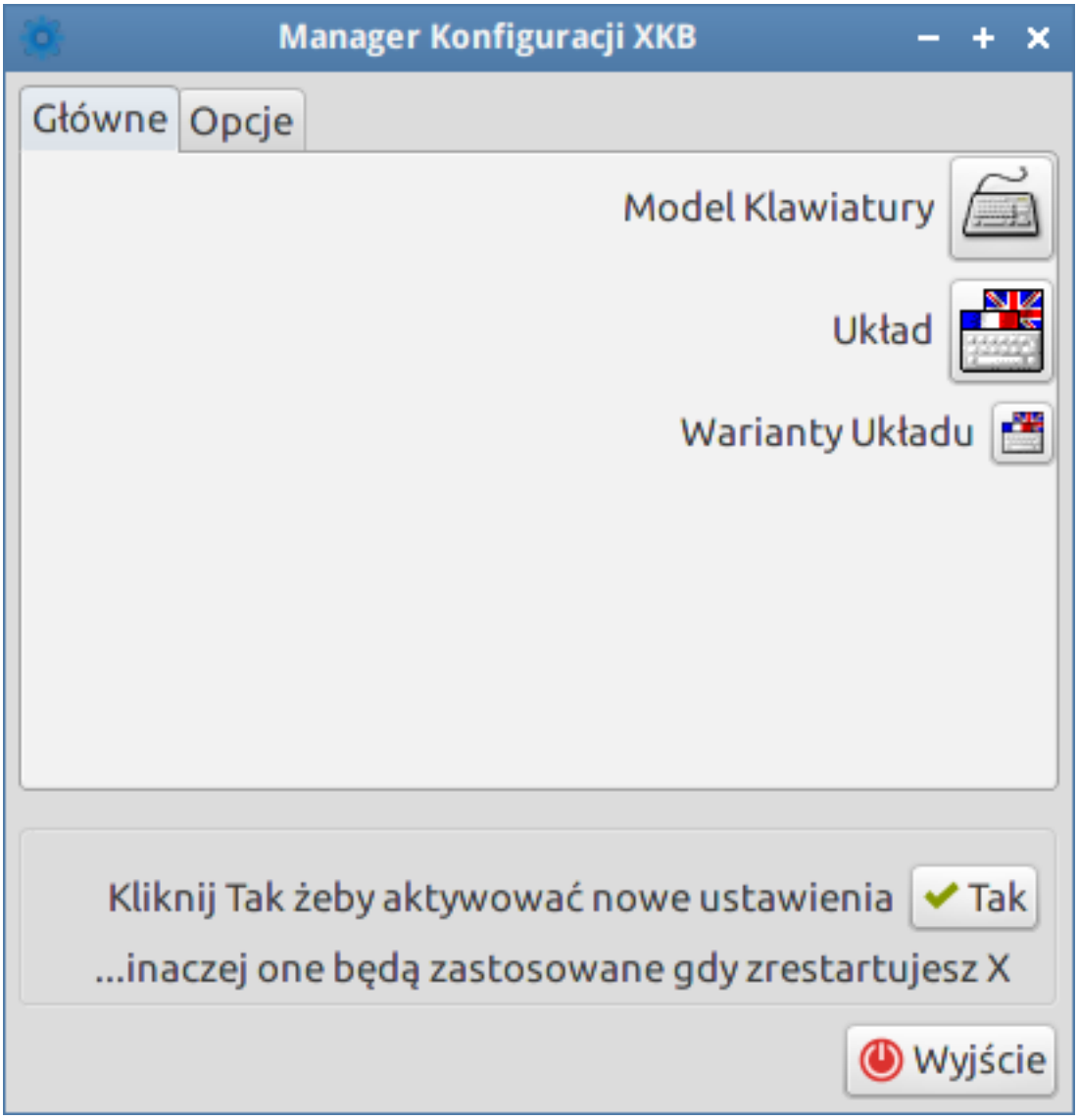
   Autorzy Patrz plik "Autorzy"

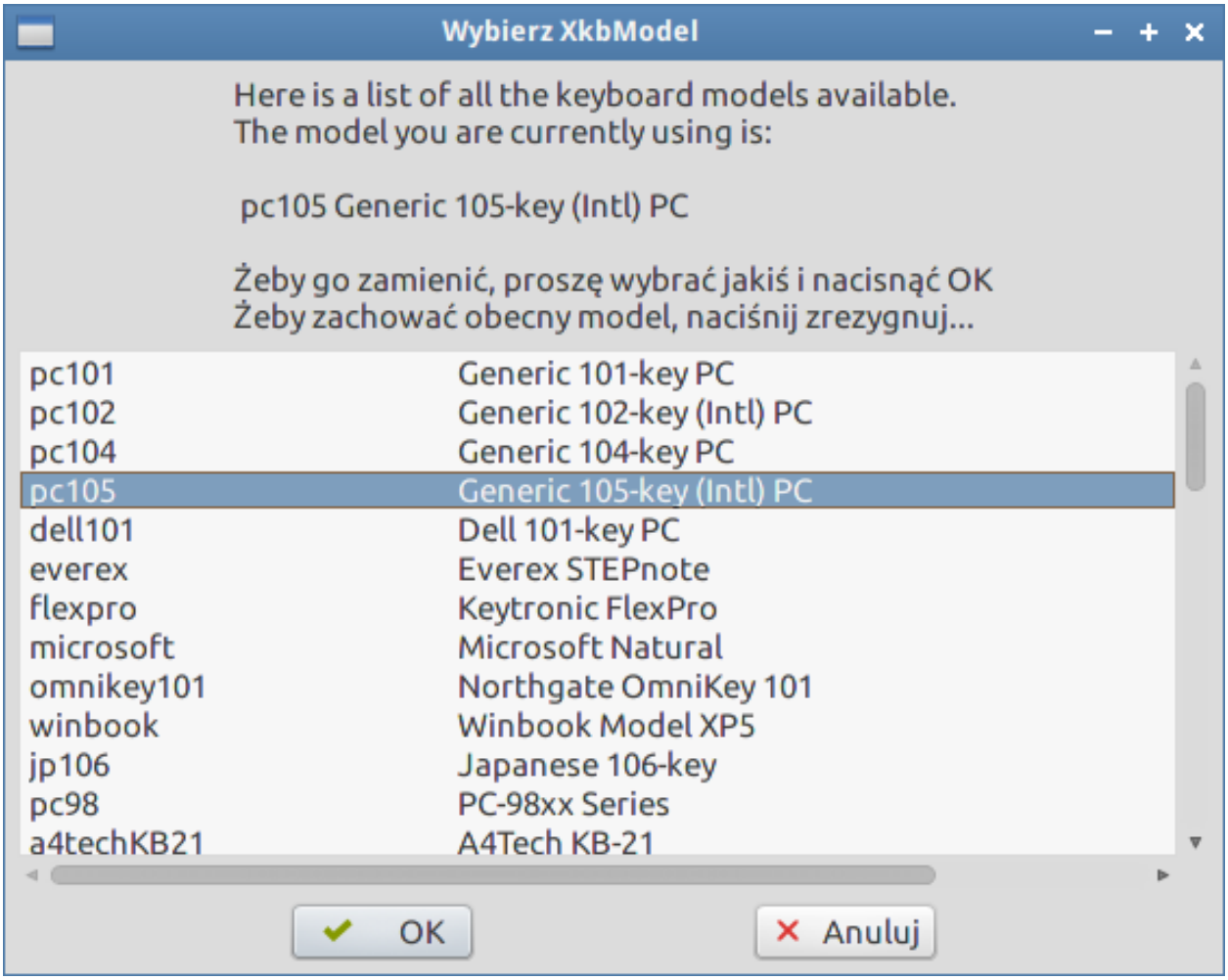
Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

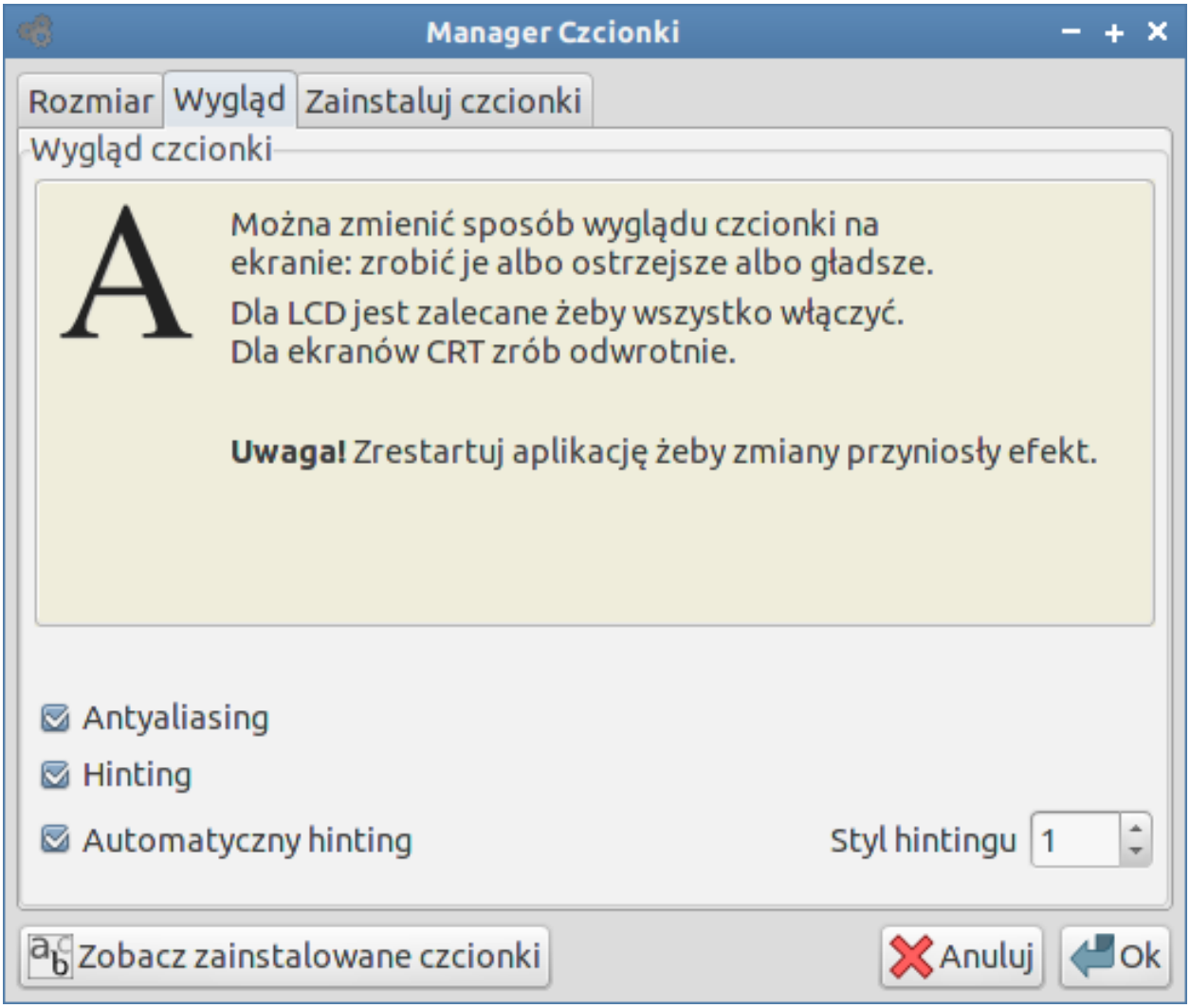
Autorzy Patrz plik "Autorzy"

2.1.3 Przygotowanie systemu Windows

Interpreter Pythona


Manager Konfiguracji XKB
Główne
Opcje
Model Klawiatury
Układ
Warianty Układu
Kliknij Tak żeby aktywować nowe ustawienia
Tak
...inaczej one będą zastosowane gdy zrestartujesz X
Wyjście

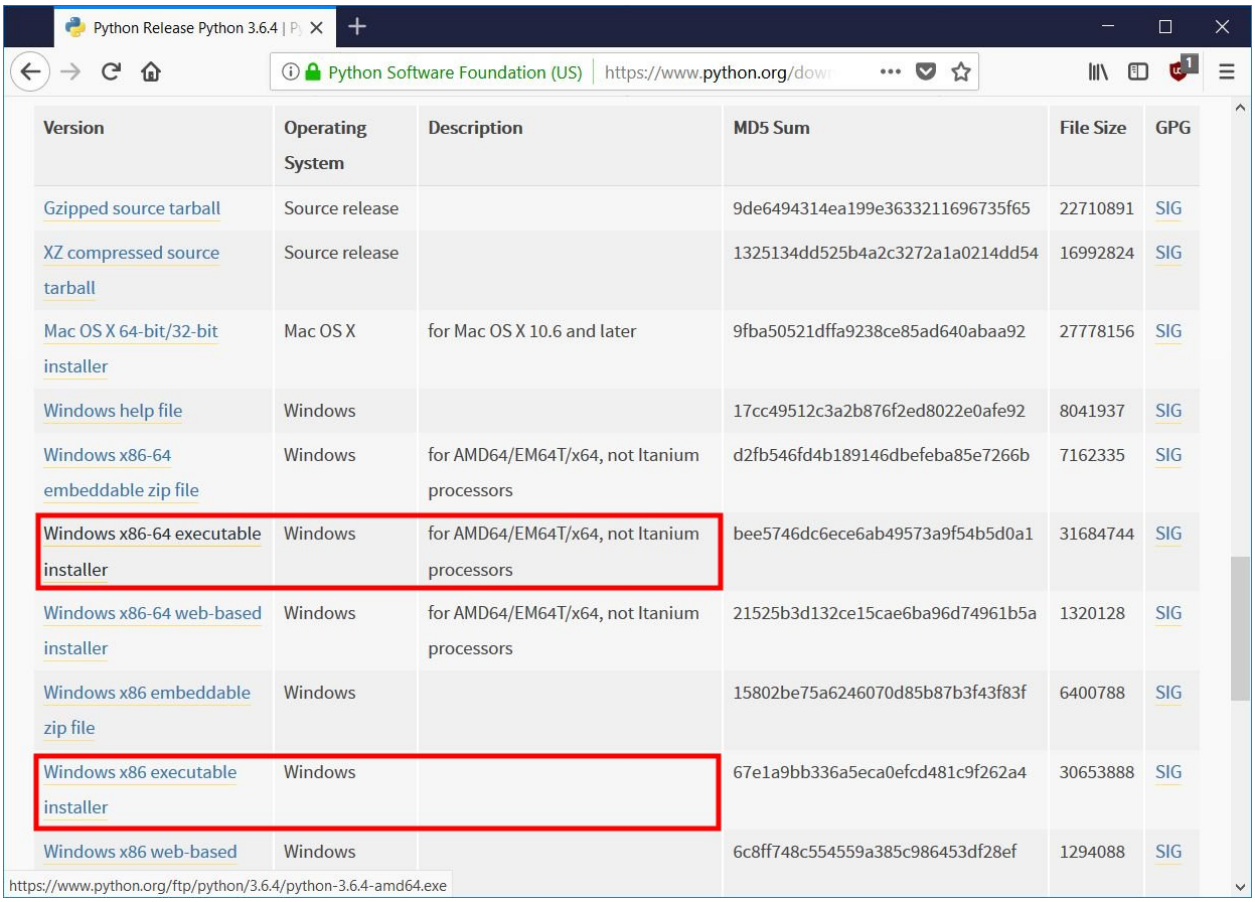

Wybierz XkbModel
Here is a list of all the keyboard models available.
The model you are currently using is:
pc105 Generic 105-key (Intl) PC
Żeby go zamienić, proszę wybrać jakiś i nacisnąć OK
Żeby zachować obecny model, naciśnij zrezygnuj...
pc101 Generic 101-key PC
pc102 Generic 102-key (Intl) PC
pc104 Generic 104-key PC
pc105 Generic 105-key (Intl) PC
dell101 Dell 101-key PC
everex Everex STEPnote
flexpro Keytronic FlexPro
microsoft Microsoft Natural
omnikey101 Northgate OmniKey 101
winbook Winbook Model XP5
jp106 Japanese 106-key
pc98 PC-98xx Series
a4techKB21 A4Tech KB-21
OK
Anuluj



Rys. 2.3: Wygładzanie czcionek

Informacja: Przed rozpoczęciem instalacji Pythona zaktualizuj sytem.

Na stronie Python Releases for Windows klikamy link *Last Python 3 Release - ...* i pobieramy instalator `Windows executable installer` w wersji *x86-64* (64-bitowej).



Version	Operating System	Description	MD5 Sum	File Size	GPG
Gzipped source tarball	Source release		9de6494314ea199e3633211696735f65	22710891	SIG
XZ compressed source tarball	Source release		1325134dd525b4a2c3272a1a0214dd54	16992824	SIG
Mac OS X 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	9fba50521dffa9238ce85ad640abaa92	27778156	SIG
Windows help file	Windows		17cc49512c3a2b876f2ed8022e0afe92	8041937	SIG
Windows x86-64 embeddable zip file	Windows	for AMD64/EM64T/x64, not Itanium processors	d2fb546fd4b189146dbefeba85e7266b	7162335	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64, not Itanium processors	bee5746dc6ece6ab49573a9f54b5d0a1	31684744	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64, not Itanium processors	21525b3d132ce15cae6ba96d74961b5a	1320128	SIG
Windows x86 embeddable zip file	Windows		15802be75a6246070d85b87b3f43f83f	6400788	SIG
Windows x86 executable installer	Windows		67e1a9bb336a5eca0efcd481c9f262a4	30653888	SIG
Windows x86 web-based	Windows		6c8ff748c554559a385c986453df28ef	1294088	SIG

Wskazówka: Podczas instalacji zaznaczamy opcję “Add Python.exe to Path” i wybieramy “Customize installation”.

Na końcu instalacji można aktywować opcję “Disable path length limit”.

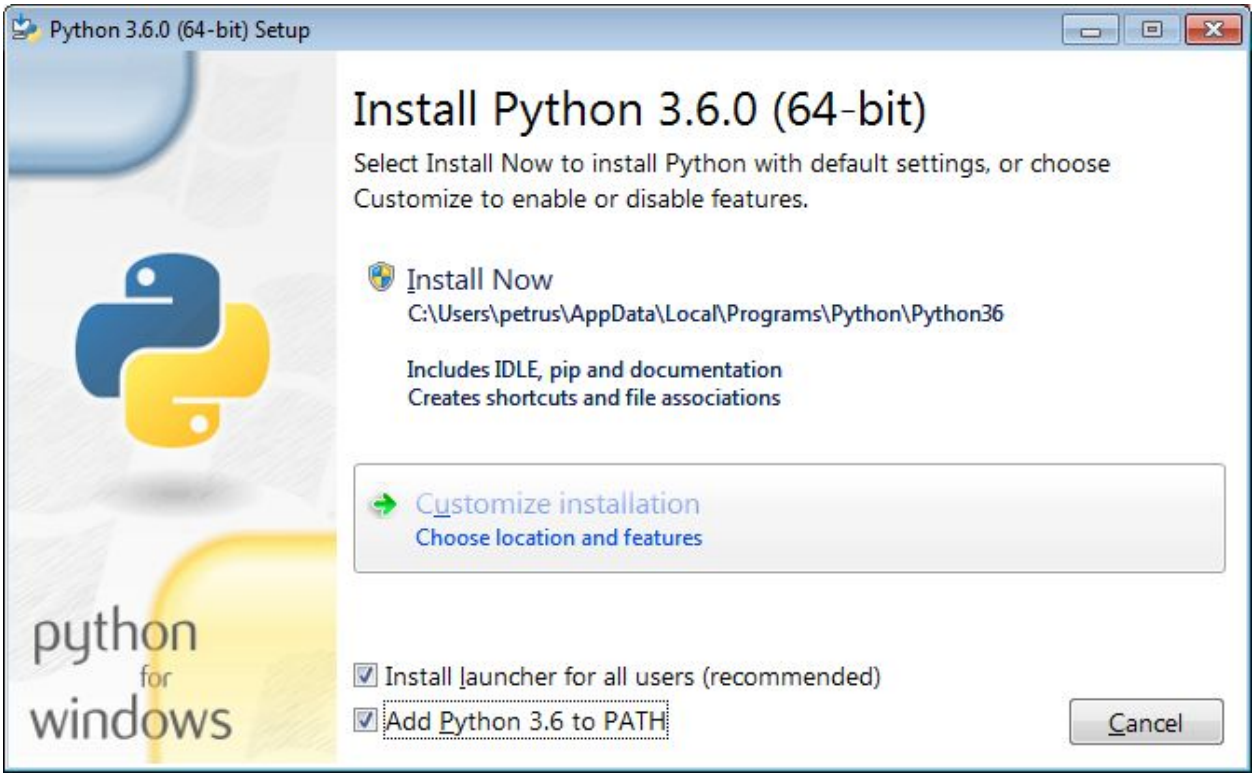
Podczas pierwszego uruchomienia możemy zobaczyć komunikat zapory systemowej. Zezwalamy na dostęp wybierając sieci prywatne:

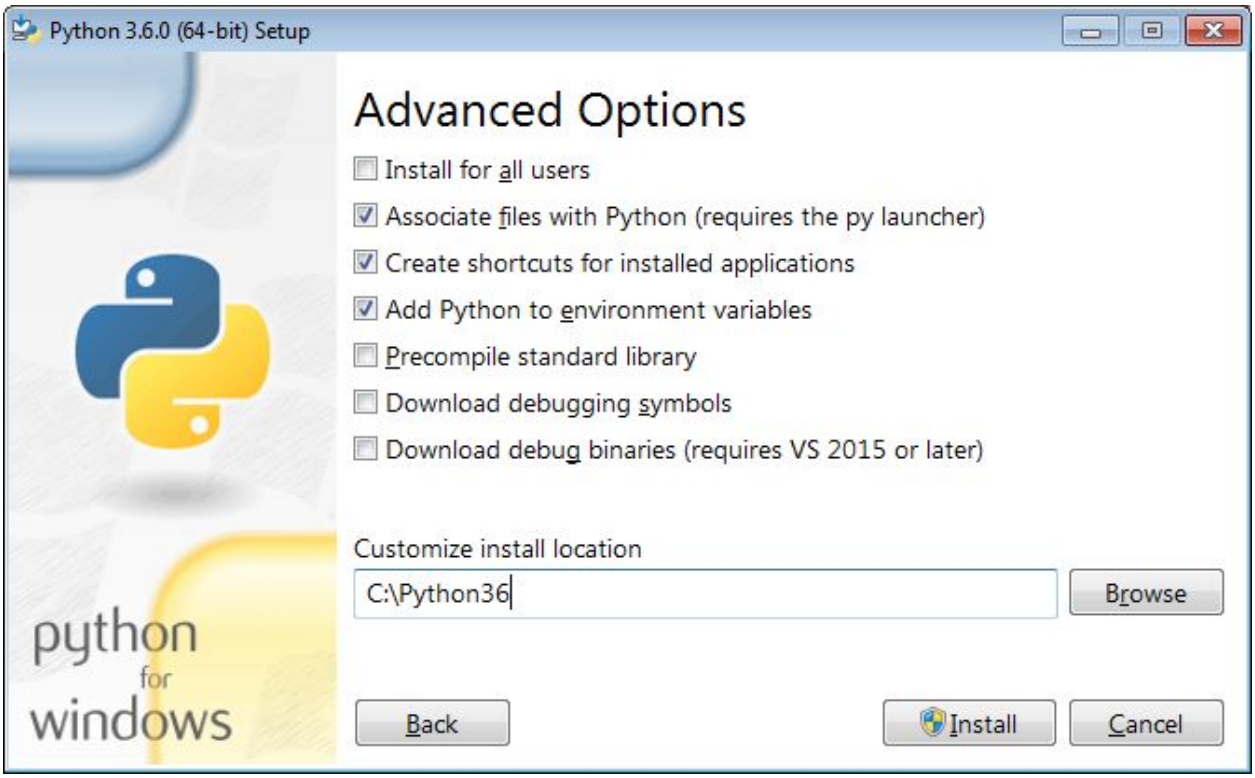
Narzędzia

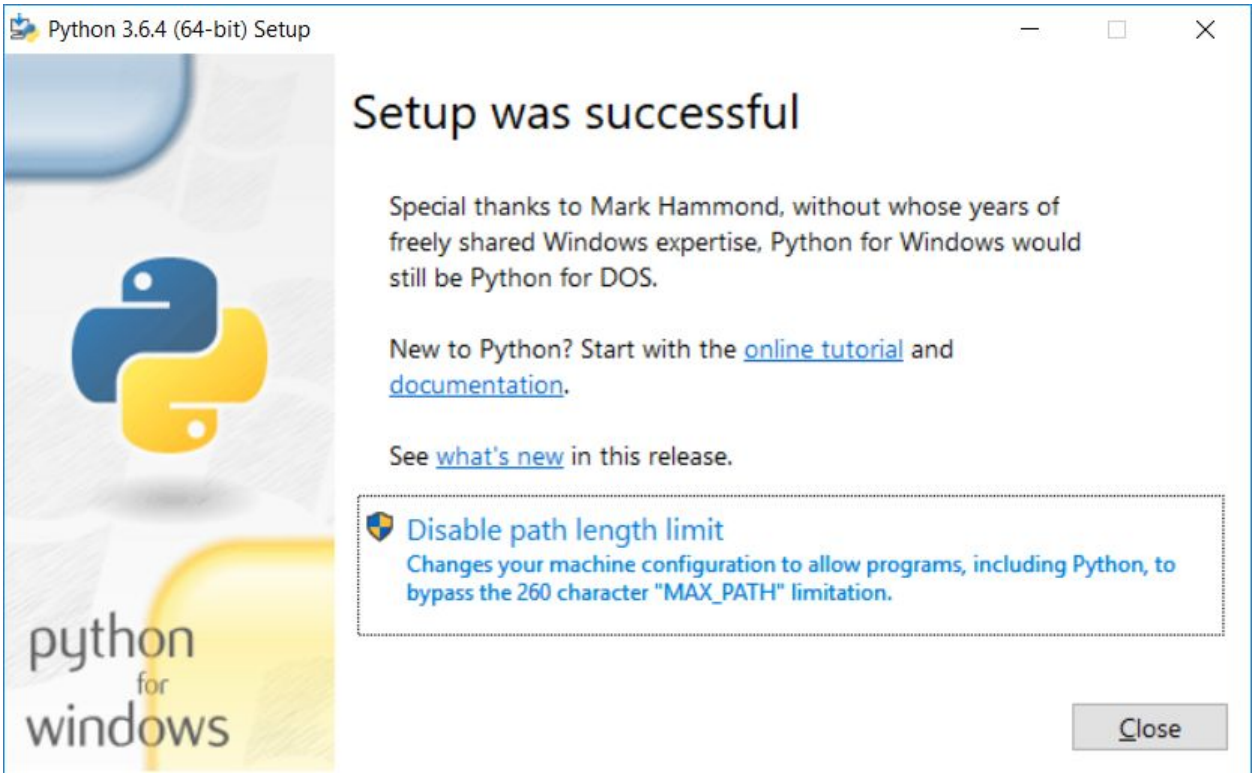
Git

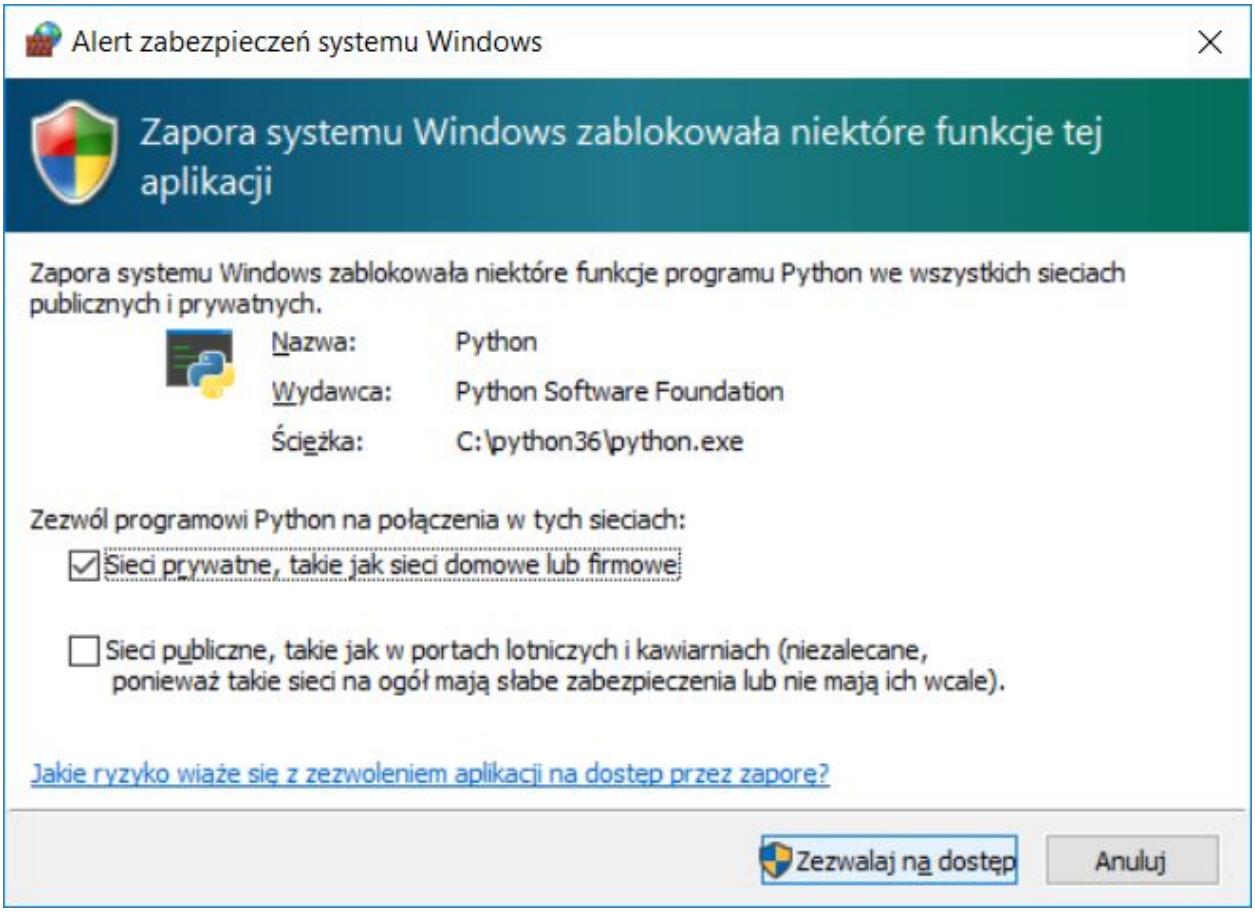
Podstawowego klienta w wersji 64-bitowej pobieramy ze strony Downloading Git i instalujemy, zaznaczając wszystkie opcje.

Alternatywna metoda instalacji, jak również zasady pracy z repozytoriami omówione zostały w osobnym *dokumencie*. Gorąco zachęcamy do jego przejrzenia.


Python 3.6.0 (64-bit) Setup
Install Python 3.6.0 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\petrus\AppData\Local\Programs\Python\Python36
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
Install launcher for all users (recommended)
Add Python 3.6 to PATH
Cancel


Python 3.6.0 (64-bit) Setup
Advanced Options
Install for all users
Associate files with Python (requires the py launcher)
Create shortcuts for installed applications
Add Python to environment variables
Precompile standard library
Download debugging symbols
Download debug binaries (requires VS 2015 or later)
Customize install location
C:\Python36
Browse
python
for
windows
Back
Install
Cancel


Python 3.6.4 (64-bit) Setup
Setup was successful
Special thanks to Mark Hammond, without whose years of freely shared Windows expertise, Python for Windows would still be Python for DOS.
New to Python? Start with the online tutorial and documentation.
See what's new in this release.
Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.
python for windows
Close


Alert zabezpieczeń systemu Windows
Zapora systemu Windows zablokowała niektóre funkcje tej aplikacji
Zapora systemu Windows zablokowała niektóre funkcje programu Python we wszystkich sieciach publicznych i prywatnych.
Nazwa: Python
Wydawca: Python Software Foundation
Ścieżka: C:\python36\python.exe
Zezwól programowi Python na połączenia w tych sieciach:
Sieci prywatne, takie jak sieci domowe lub firmowe
Sieci publiczne, takie jak w portach lotniczych i kawiarniach (niezalecane, ponieważ takie sieci na ogół mają słabe zabezpieczenia lub nie mają ich wcale).
Jakie ryzyko wiąże się z zezwoleniem aplikacji na dostęp przez zaporę?
Zezwalaj na dostęp
Anuluj

Rozszerzona konsola

W wierszu poleceń wydajemy następujące polecenia:

```
pip install ipython qtconsole pyqt5
```

SQLite3

Ze strony SQLite Download Page, z sekcji *Precompiled Binaries for Windows* ściągamy powłokę dla 64-bitowej wersji Windows. Przykładowe archiwum `sqlite-dll-win64-x64-3380500.zip` należy rozpakować, najlepiej do katalogu systemowego (`C:\Windows\System32`), żeby był dostępny z każdej lokalizacji.

Biblioteki

Wskazówka: W przypadku bibliotek warto rozważyć instalację w *środowisku wirtualnym* dostępną dla zwykłego użytkownika.

PyQt

Qtconsole wymaga bibliotek PyQt. W 64-bitowej wersji Windowsa w wierszu poleceń wydajemy polecenie:

```
pip install python-qt5
```

PyGame

Jest to moduł wymagany m.in. przez scenariusze gier. W przypadku Windows 32-bitowego ze strony PyGame pobieramy plik pygame-1.9.1.win32-py2.7.msi i instalujemy:

W przypadku wersji 64-bitowej wchodzimy na stronę http://www.lfd.uci.edu/~gohlke/pythonlibs i pobieramy pakiet `pygame-1.9.3-cp36-cp36m-win_amd64.whl` (dla Pythona 3.6). Następnie otwieramy terminal w katalogu z zapisanym pakietem i wydajemy polecenie:

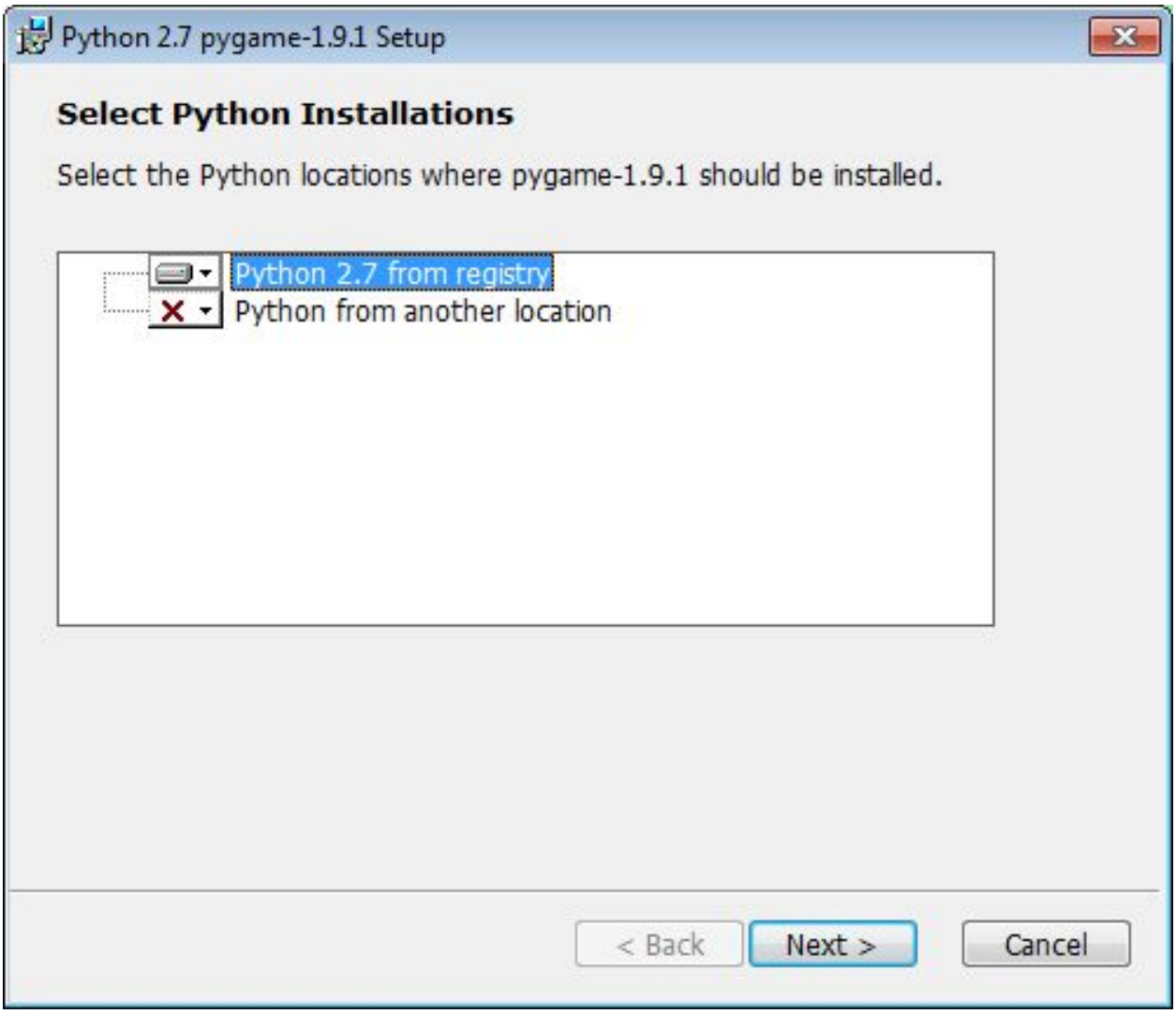
```
pip install pygame-1.9.2b1-cp27-cp27m-win_amd64.whl
```

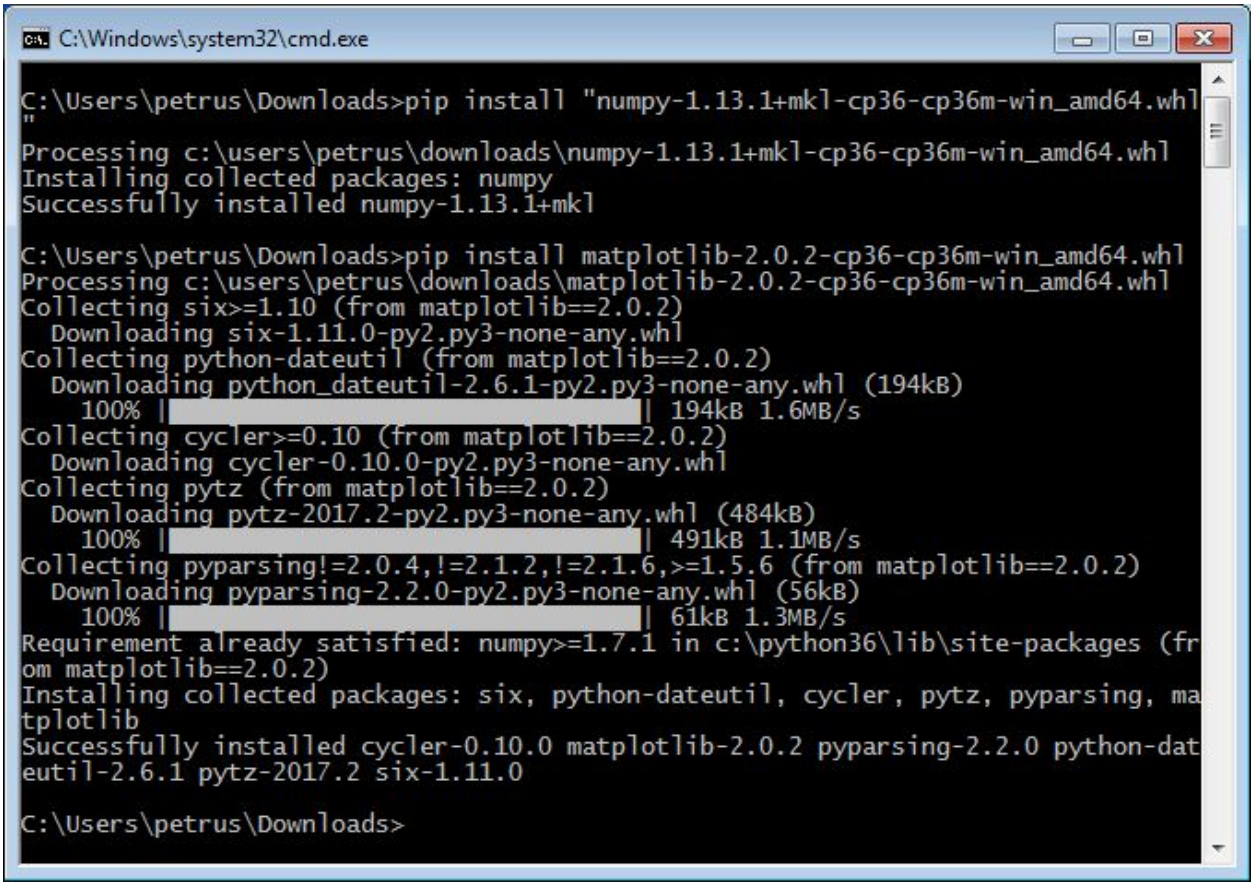
Matplotlib

Wejdź na stronę http://www.lfd.uci.edu/~gohlke/pythonlibs i pobierz pakiety `numpy` oraz `matplotlib` w formacie `whl` dostosowane do wersji Pythona i Windows. Np. jeżeli mamy *Pythona 3.6.x* i *Windows 64-bit*, pobierzemy: `numpy-1.13.1+mkl-cp36-cp36m-win_amd64.whl` i `matplotlib-2.0.2-cp36-cp36m-win_amd64.whl`. Następnie otwieramy terminal w katalogu z pobranymi pakietami i instalujemy:

```
pip install numpy-1.13.1+mkl-cp36-cp36m-win_amd64.whl
pip matplotlib-2.0.2-cp36-cp36m-win_amd64.whl
```

Informacja: Oficjalne kompilacje **matplotlib** dla Windows dostępne są w serwisie Sourceforge matplotlib.


Python 2.7 pygame-1.9.1 Setup
Select Python Installations
Select the Python locations where pygame-1.9.1 should be installed.
Python 2.7 from registry
Python from another location
< Back
Next >
Cancel


C:\Windows\system32\cmd.exe
C:\Users\petrus\Downloads>pip install "numpy-1.13.1+mkl-cp36-cp36m-win_amd64.whl
"
Processing c:\users\petrus\downloads\numpy-1.13.1+mkl-cp36-cp36m-win_amd64.whl
Installing collected packages: numpy
Successfully installed numpy-1.13.1+mkl
C:\Users\petrus\Downloads>pip install matplotlib-2.0.2-cp36-cp36m-win_amd64.whl
Processing c:\users\petrus\downloads\matplotlib-2.0.2-cp36-cp36m-win_amd64.whl
Collecting six>=1.10 (from matplotlib==2.0.2)
Downloading six-1.11.0-py2.py3-none-any.whl
Collecting python-dateutil (from matplotlib==2.0.2)
Downloading python_dateutil-2.6.1-py2.py3-none-any.whl (194kB)
100% | 194kB 1.6MB/s
Collecting cycler>=0.10 (from matplotlib==2.0.2)
Downloading cycler-0.10.0-py2.py3-none-any.whl
Collecting pytz (from matplotlib==2.0.2)
Downloading pytz-2017.2-py2.py3-none-any.whl (484kB)
100% | 491kB 1.1MB/s
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=1.5.6 (from matplotlib==2.0.2)
Downloading pyparsing-2.2.0-py2.py3-none-any.whl (56kB)
100% | 61kB 1.3MB/s
Requirement already satisfied: numpy>=1.7.1 in c:\python36\lib\site-packages (fr
om matplotlib==2.0.2)
Installing collected packages: six, python-dateutil, cycler, pytz, pyparsing, ma
tplotlib
Successfully installed cycler-0.10.0 matplotlib-2.0.2 pyparsing-2.2.0 python-dat
eutil-2.6.1 pytz-2017.2 six-1.11.0
C:\Users\petrus\Downloads>

Frameworki WWW

Instalacja bibliotek wymaganych do scenariuszy *Aplikacje WWW*:

```
pip install flask flask-wtf peewee sqlalchemy flask-sqlalchemy django
```

Brak Pythona?

Jeżeli nie możemy wywołać interpretera lub instalatora `pip` w wierszu poleceń, oznacza to zazwyczaj, że zapomnieliśmy zaznaczyć opcji "Add Python.exe to Path" podczas instalacji interpretera. Najprościej zainstalować go jeszcze raz z zaznaczoną opcją.

Można też samemu rozszerzyć zmienną systemową `PATH` swojego użytkownika o ścieżkę do `python.exe`. Najwygodniej wykorzystać konsolę PowerShell:

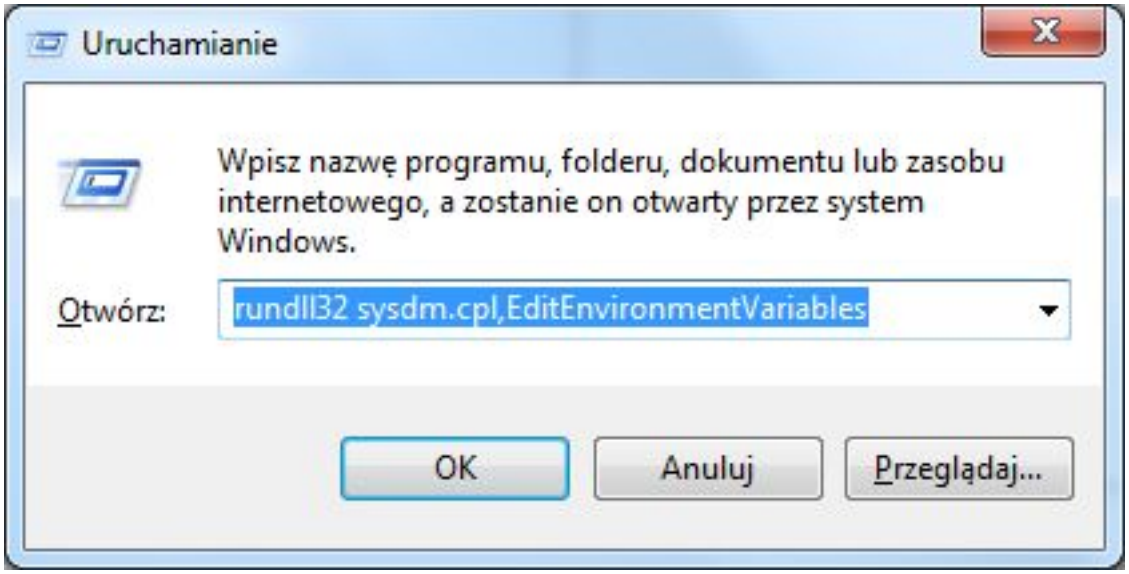
```
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python36\;
↪C:\Python36\Scripts\", "User")
```

Ewentualnie, jeśli posiadamy uprawnienia administracyjne, możemy zmienić zmienną `PATH` wszystkim użytkownikom:

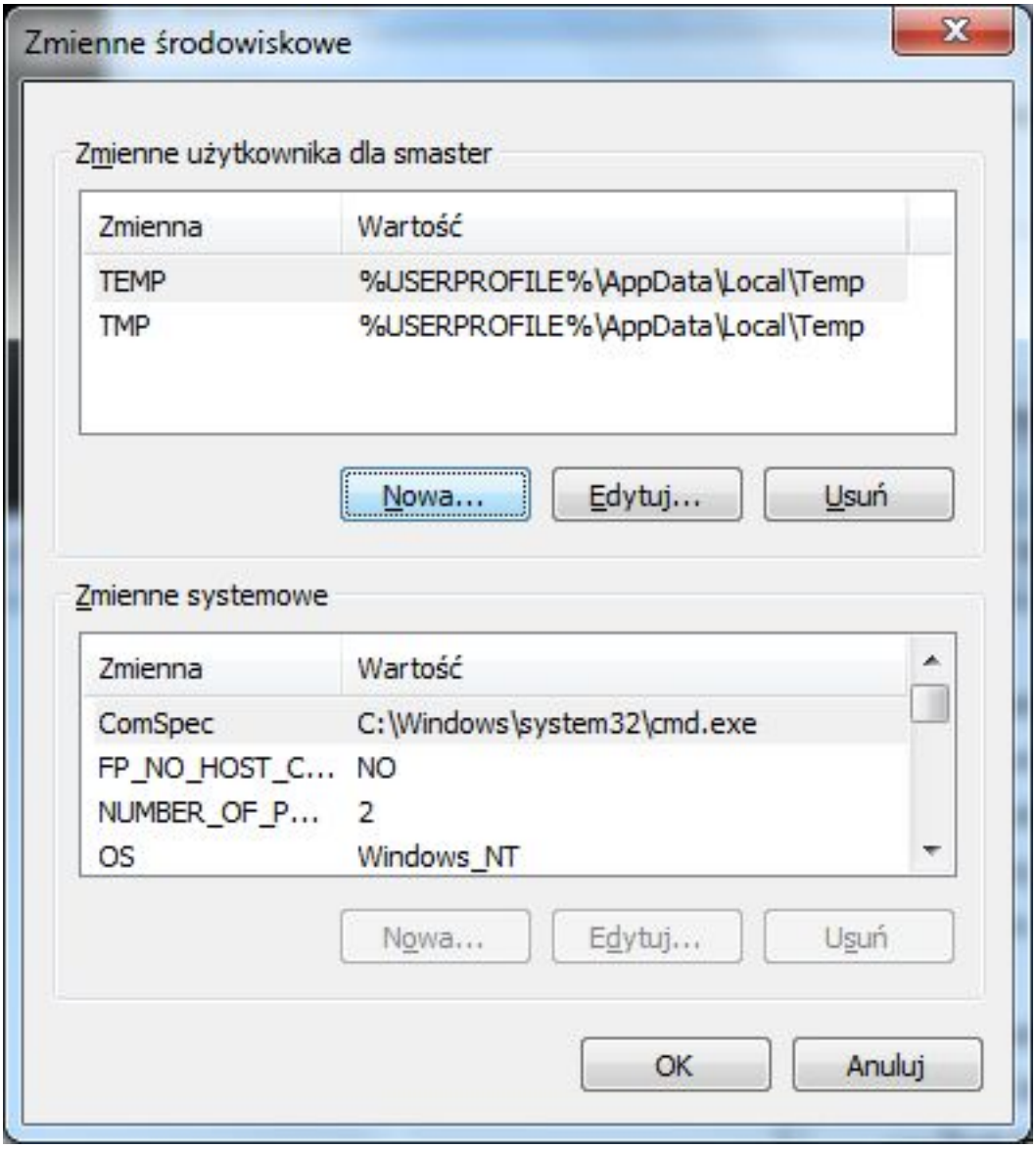
```
$CurrentPath=[Environment]::GetEnvironmentVariable("Path", "Machine")
[Environment]::SetEnvironmentVariable("Path", "$CurrentPath;C:\Python36\;
↪C:\Python36\Scripts\", "Machine")
```

Jeżeli nie mamy dostępu do konsoli PowerShell, w oknie "Uruchamianie" (`WIN+R`) wpisujemy polecenie wywołujące okno "Zmienne środowiskowe" – można je również uruchomić z okna właściwości komputera:

```
rundll32 sysdm.cpl,EditEnvironmentVariables
```



Następnie klikamy przycisk "Nowa" i dopisujemy ścieżkę do katalogu z Pythonem, np.: `PATH=%PATH%;C:\Python36\;C:\Python36\Scripts\`; w przypadku zmiennej systemowej klikamy "Edytuj", a ścieżki `C:\Python36\;C:\Python36\Scripts\` dopisujemy po średniku. Dla pojedynczej sesji (do momentu przelogowania się) możemy użyć polecenia w konsoli tekstowej:


Zmienne środowiskowe
Zmienne użytkownika dla smaster
Zmienna
Wartość
TEMP
%USERPROFILE%\AppData\Local\Temp
TMP
%USERPROFILE%\AppData\Local\Temp
Nowa...
Edytuj...
Usuń
Zmienne systemowe
Zmienna
Wartość
ComSpec
C:\Windows\system32\cmd.exe
FP_NO_HOST_C...
NO
NUMBER_OF_P...
2
OS
Windows_NT
Nowa...
Edytuj...
Usuń
OK
Anuluj

```
set PATH=%PATH%;c:\Python36\;c:\Python36\Scripts\
```

Ostrzeżenie: W powyższych przykładach założono, że Python zainstalowany został w katalogu `C:Python36`, co nie jest opcją domyślną.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.1.4 Podstawy Pythona

Python to język interpretowany. Kod źródłowy Pythona zapisujemy w plikach tekstowych z rozszerzeniem `.py`. Skrypty Pythona **uruchamiamy** w terminalu przy użyciu interpretera:

```
~$ python3 nazwa_skryptu.py
```

– lub z poziomu edytora kodu, który oferuje taką możliwość.

Ze względów praktycznych warto korzystać z programów ułatwiających pisanie kodu (obsługa wcięć, podświetlenia itd.), tzw. IDE, czyli Integrated Development Environment. Zobacz Edytory kodu.

Tryb interaktywny interpretera Pythona jest podstawowym narzędziem nauki i testowania kodu. Uruchamiamy go, wydając w terminalu używanego systemu polecenie:

```
~$ python3
```

Po uruchomieniu interpreter wyświetla swoją wersję, opcjonalnie wersję kompilatora C++, a także znak zachęty `>>>`. Przydatne polecenia:

```
>>> help()          # uruchomienie interaktywnej pomocy
help> quit          # wyjście z trybu interaktywnej pomocy
>>> help(obiekt)    # wyświetla pomoc dotyczącą dowolnego obiektu
>>> import math     # zaimportowanie przykładowego modułu math
>>> dir(math)       # przegląd dostępnych w module stałych i funkcji
>>> help(math.pow)  # wyświetla pomoc nt. stałej lub funkcji dostępnej w module
>>> exit()          # wyjście z trybu iteraktywnego interpretera
```

Znaki `...` oznaczają, że testujemy instrukcję złożoną, np. warunkową lub pętlę, i dalszy kod wymaga wcięć.

Środowisko wirtualne

Wirtualne środowisko Pythona (ang. *Python virtual environment*) pozwala instalować dodatkowe pakiety w wybranych wersjach, a także dodatkowe narzędzia bez uprawnień administratora. W praktyce to katalog zawierający niezbędne pliki potrzebne do działania interpretera oraz menedżer *pip*. Po utworzeniu środowiska przed każdym użyciem należy go aktywować.

Utworzenie i korzystanie ze środowiska:

```
~$ python3 -m venv pve           # utworzenie środowiska katalogu pve
~$ source pve/bin/activate       # aktywacja w Linuksie
> pve\\Scripts\\activate.bat     # aktywacja w Windowsie
(pve) ~$ python skrypt.py        # uruchamianie skryptu w wirtualnym środowisku
(pve) ~$ deactivate              # deaktywacja
```

Przydatne polecenia instalatora pakietów *pip*:

```
(pve) ~$ pip install biblioteka==1.4   # instalacja biblioteki we wskazanej wersji
(pve) ~$ pip -V                        # wersja narzędzia pip
(pve) ~$ pip list                      # lista zainstalowanych pakietów
(pve) ~$ pip install nazwa_pakietu     # instalacja pakietu
(pve) ~$ pip install nazwa_pakietu -U  # aktualizacja pakietu
(pve) ~$ pip uninstall nazwa_pakietu   # usunięcie pakietu
```

Przykład instalacji pakietów wykorzystywanych w materiałach:

```
~$ sudo pip3 install matplotlib
~$ sudo pip3 install pygame
~$ sudo pip3 install flask flask-wtf peewee sqlalchemy flask-sqlalchemy django
```

Rozszerzone powłoki

Zchęcamy do korzystania z powłok rozszerzonych *IPython* i/lub *Jupyter QtConsole*, oferujących podpowiedzi, dopełnianie, formatowanie kodu itp. ułatwienia. Najłatwiej zainstalować je w środowisku wirtualnym:

```
(pve) ~$ pip install ipython3
(pve) ~$ pip install qtconsole pyqt5
```

Uruchamiamy je poleceniami:

```
~$ ipython3
~$ jupyter-qtconsole
```

Wskazówka: Do terminala skopiowane polecenia wklejamy bez znaku zachęty `$` i poprzedzającego tekstu za pomocą środkowego klawisza myszki lub skrótów `CTRL+SHIFT+V`, `CTRL+SHIFT+Insert`.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.2 Realizacja scenariuszy

2.2.1 Katalog użytkownika

Jeżeli w scenariuszu mowa o **katalogu domowym** użytkownika, w systemie Linux należy przez to rozumieć podfolder katalogu `/home` o nazwie zalogowanego użytkownika, np. `/home/uczen`. W poleceniach wydawanych w terminalu (zob. terminal) ścieżkę do tego katalogu symbolizuje znak `~`.

Zapis typu `~/quiz2$` oznacza więc, że dane polecenie należy wykonać w podkatalogu `quiz2` katalogu domowego użytkownika.

Znak `$` oznacza, że komendy wydajemy jako zwykły użytkownik, natomiast `#` – jako root, czyli administrator.

Informacja: W przygotowanym przez nas systemie *MX Linux Live* pracujesz jako użytkownik z loginem i hasłem *demo*.

2.2.2 W systemie Windows

Jeżeli scenariusze będziemy wykonywać w MS Windows, musimy pamiętać o różnicach:

- Katalog domowy użytkownika w Windows nie nadaje się do przechowywania w nim kodów programów lub repozytoriów, najlepiej utworzyć jakiś katalog na partycji innej niż systemowa (oznaczana literą *C:*), np. `D:python` i w nim tworzyć foldery dla poszczególnych scenariuszy.
- Domyślnym terminalem jest program `cmd`, czyli wiersz poleceń; jest on jednak ograniczony i niewygodny, warto używać konsoli PowerShell lub jeszcze lepiej konsoli instalowanych razem z Pythonem i klientem Git.
- W systemie Windows znaki `/` (slash) w ścieżkach zmieniamy na `\` (backslash).
- Zamieniamy również komendy systemu Linux na odpowiedniki wiersza poleceń Windows, np. `mkdir` na `md`.
- Pamiętajmy, żeby skrypty zapisywać w plikach kodowanych jako UTF-8.

2.2.3 Kod źródłowy

W materiałach znajdziesz przykłady kodu źródłowego, które pokazują, jak rozwija się program. Warto je wpisywać w wybranym edytorze samodzielnie, aby nauczyć się składni języka i lepiej poznać środowisko programistyczne. Podczas przepisywania można pominąć komentarze, czyli teksty zaczynające się od znaku `#` lub zamknięte pomiędzy potrójnymi cudzysłowami `"`.

W przypadku braku czasu kod można zaznaczać, kopiować i wklejać, pamiętając o zachowaniu wcięć.

Większość fragmentów kodu jest numerowana, ale jeśli Twój kod różni się nieznacznie numeracją linii, nie musi to oznaczać błędu.

Dla przykładu kod poniżej powinien zostać wklejony w linii `51` omawianego pliku:

```
51 def run(self):
52     """
53     Główna pętla programu
54     """
55     while not self.handle_events():
56         self.ball.move(self.board)
57         self.board.draw(
58             self.ball,
59         )
60         self.fps_clock.tick(30)
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

2.3 Git – wersjonowanie kodu

TYMCZASOWO

2.3.1 Wykorzystanie Git-a

Materiały szkoleniowe zostały umieszczone w repozytorium w serwisie GitHub dzięki temu każdy może w łatwy sposób pobrać, zmieniać, a także zsynchronizować swoją lokalną kopię.

W katalogu domowym użytkownika uruchamiamy komendę:

```
~$ git clone --recursive https://github.com/koduj-z-klasa/python101.git
```

W efekcie otrzymamy katalog `python101` z kodami źródłowymi materiałów.

Synchronizacja kodu

Informacja: Poniższe instrukcje nie są wymagane w ramach przygotowania, ale warto się z nimi zapoznać w przypadku gdybyśmy chcieli skorzystać z możliwości pozbycia się lokalnych zmian wprowadzonych podczas ćwiczeń i przywrócenia stanu do punktu wyjścia.

Materiały zostały podzielone w repozytorium na części, które w kolejnych krokach są rozbudowywane. Dzięki temu na początku szkolenia mamy niewielki zbiór plików, natomiast w kolejnych krokach szkolenia możemy aktualizować wersję roboczą o nowe treści.

Uczestnicy mogą spokojnie edytować i zmieniać materiały bez obaw o późniejsze różnice względem reszty grupy.

Zmiany możemy szybko wyczyścić i powrócić do stanu z początku ćwiczenia:

```
$ git reset --hard
```

Możemy także skakać pomiędzy punktami kontrolnymi np. skoczyć do następnego lub skoczyć do następnego punktu kontrolnego i zsynchronizować kody źródłowe grupy bez zachowania zmian poszczególnych uczestników:

```
$ git checkout -f pong/z1
```

Jeśli uczestnicy chcą wcześniej zachować swoje modyfikacje, mogą je zapisać w swoim lokalnym repozytorium (wykonują tzw. commit).

TYMCZASOWO Pokażemy tutaj, jak nauczyciele mogą wykorzystać profesjonalne i bezpłatne narzędzia do wersjonowania kodów źródłowych i wszystkich innych plików.

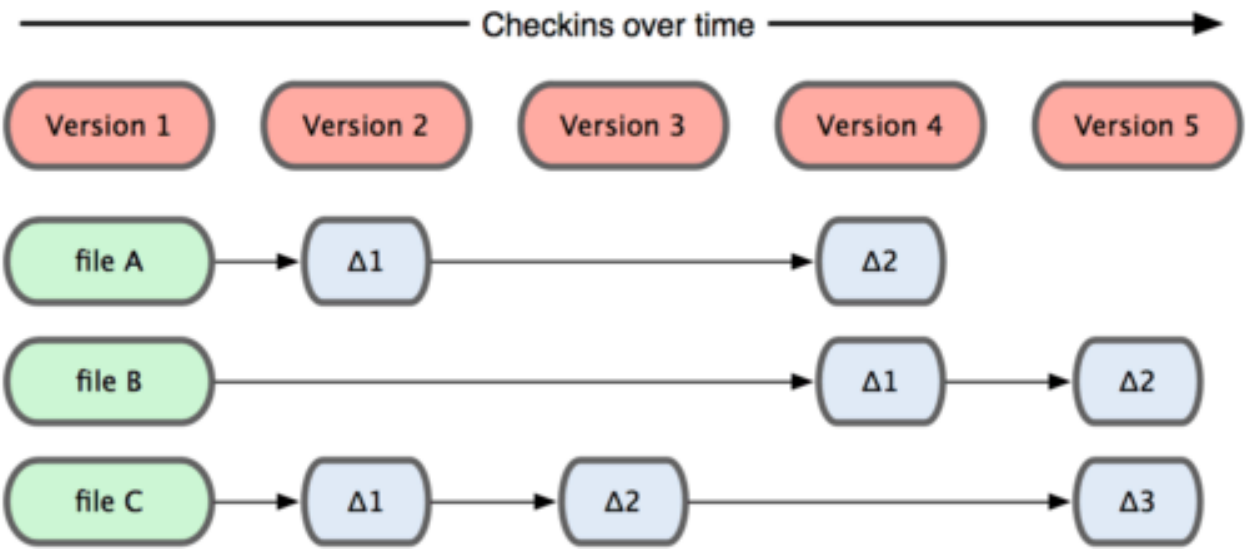
Przybliżamy tutaj jak GIT jest wykorzystywany w naszych materiałach i pokażemy jak go wykorzystać go podczas zajęć w szkole.

Poniżej przeprowadzimy szybkie wprowadzenie po więcej informacji oraz pełne szczegółowe wprowadzenie i przykłady użycia znajdziecie w dostępnej online i do pobrania polskiej wersji książki Pro Git. Polecamy także cheat sheet z podręcznymi komendami.

2.3.2 Co to jest GIT?

GIT to system kontroli wersji, pozwala zapamiętać i synchronizować pomiędzy użytkownikami zmiany dokonywane na plikach. Umożliwia przywołanie dowolnej wcześniejszej wersji, a co najważniejsze, automatycznie łączy zmiany, które ze sobą nie kolidują, np. dokonane w różnych miejscach w pliku.

Nauczyciele pracujący z plikami, które zmieniają się z przykładu na przykład, z ćwiczenia na ćwiczenie mogą skorzystać z systemu kontroli wersji do synchronizacji przykładów z uczniami na poszczególnych etapach swojej pracy.



Dzięki takim narzędziom możemy porzucić przesyłanie i rozpakowywanie archiwów oraz kopiowanie plików na rzecz komend, które szybko ujednolicą stan plików na komputerach naszych uczniów.

Lokalne repozytoria z historią zmian

Każdy z uczniów może mieć lokalną kopię całej historii zmian w plikach, będzie mógł modyfikować swoje przykłady, ale w kluczowym momencie nauczyciel może poprosić, by wszyscy zsynchronizowali swoje kopie z jedną sprawdzoną wersją, tak by dalej prowadzić zajęcia na jednolitym fundamencie.

Okresowa synchronizacja przykładów, które uczniowie z założenia zmieniają podczas zajęć, pozwala wykluczyć pomyłki i wyeliminować problemy wynikające z różnic we wprowadzonych zmianach.

Poniżej mamy przykład komendy która otworzy pliki w *wersji 5* dla *zadania 2*. Nazwy `zadanie2` oraz `wersja5` są tylko przykładem, mogą być dowolnie wybrane przez autora.

```
$ git checkout -f zadanie2/wersja5
```

Przed porzuceniem zmian uczeń może zapisać kopię swojej pracy w repozytorium.

```
$ git commit -a -m "Moje zmiany w przykładzie 5"
```

2.3.3 Instalujemy narzędzie GIT

Do korzystania z naszego repozytorium lokalnie na naszym komputerze musimy doinstalować niezbędne oprogramowanie.

W Linuksie

Do instalacji użyjemy menadżera pakietów, np. *apt-get*:

```
$ sudo apt-get install git
```

W Windows

Zaczynamy od instalacji narzędzia GIT dla konsoli:

```
> @powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.
↪webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH
↪%;%ALLUSERSPROFILE%\chocolatey\bin
> choco install git
```

Pod Windowsem polecamy zainstalować SourceTree, aplikację okienkową i narzędzia konsolowe:

```
@powershell -NoProfile -ExecutionPolicy unrestricted -Command "iex ((new-object net.
↪webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH
↪%;%ALLUSERSPROFILE%\chocolatey\bin
choco install sourcetree
```

Jeśli nie mamy PowerShell'a, możemy ściągnąć i zainstalować narzędzie ręcznie.

Jeśli korzystamy z narzędzia KeePass do przechowywania haseł i kluczy SSH, to dobrze jest połączyć je z GITem za pomocą programu Plink.

Do tego celu musimy dodać zmienną systemową podmieniającą domyślne narzędzie SSH. Uruchamiamy konsole PowerShell z uprawnieniami administracyjnymi:

```
[Environment]::SetEnvironmentVariable("GIT_SSH", "d:\usr\tools\PuTTY\plink.exe", "User
↪")
```

Konfiguracja i pierwsze uruchomienie

Przed pierwszym użyciem warto jeszcze skonfigurować dwie informacje identyfikujące Ciebie jako autora zmian. W komendach poniżej wstaw swoje dane.

```
$ git config --global user.name "Jan Nowak"
$ git config --global user.email jannowak@example.com
```

Więcej o konfiguracji przeczytacie tutaj.

2.3.4 Pierwsze kroki i podstawy GIT

Na początek utwórzmy sobie piaskownicę do zabawy z GITem. Naszą piaskownicą będzie zwyczajny katalog, dla ułatwienia pracy z ćwiczeniami zalecamy nazwać go tak samo jak my, ale ostatecznie jego nazwa i lokalizacja nie ma znaczenia.

```
~$ mkdir git101
~$ cd git101/
```

Tworzymy lokalną historię zmian

Przed rozpoczęciem pracy z wersjami plików w nowym lub istniejącym projekcie (takim który jeszcze nie ma historii zmian), inicjalizujemy GITa w katalogu tego projektu. Tworzymy lokalne repozytorium poleceniem:

```
~/git101$ git init
Initialized empty Git repository in ~/git101/.git/
```

W katalogu projektu (na razie pustym) pojawi się katalog `.git`, w którym narzędzie będzie miało swój schowek.

Zaczynamy śledzić pliki

W każdym momencie możemy sprawdzić status naszego repozytorium:

```
~/git101$ git status
On branch master

Initial commit

nothing to commit (create/copy files and use "git add" to track)
```

Kluczowe jest `nothing to commit`, oznacza to, że narzędzie nie wykryło zmian w stosunku do tego co jest zapisane w repozytorium. Słusznie, bo katalog jest pusty. Dodajmy jakieś pliki:

```
~/git101$ touch README hello.py
~/git101$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README
    hello.py

nothing added to commit but untracked files present (use "git add" to track)
```

W powyższym komunikacie najważniejsze jest `untracked files present`: narzędzie wykryło pliki, które jeszcze nie są śledzone. Możemy rozpocząć ich śledzenie wykonując polecenie podane we wskazówce:

```
~/git101$ git add hello.py README
~/git101$ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   README
    new file:   hello.py
```

W efekcie wyraźnie zaznaczyliśmy, które pliki GIT ma śledzić. Działa to także w drugą stronę, jeśli jakieś pliki mają zostać zignorowane, to trzeba to wyraźnie zaznaczyć, narzędzie nie decyduje o tym za nas.

Informacja: Operacji dodawania nie musimy powtarzać za każdym razem, gdy plik się zmieni, musimy ją wykonać tylko raz, kiedy pojawiają się nowe pliki.

Zapamiętujemy wersję plików

Zamiany w plikach zapisujemy wykonując komendę `git commit`:

```
~/git101$ git commit -m "Moja pierwsza wersja plików"
[master (root-commit) e9cffa4] Moja pierwsza wersja plików
```

```
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README
 create mode 100644 hello.py
```

Parametr `-m` pozwala wprowadzić komentarz, który pojawi się w historii zmian.

Informacja: Komentarz jest wymagany, bo to dobra praktyka. Jeśli jesteśmy leniwi, możemy podać jedno słowo albo nawet literę, wtedy nie jest potrzebny cudzysłów.

Sprawdźmy status, a następnie zmodyfikujmy jeden z plików:

```
~/git101$ git status
On branch master
nothing to commit, working directory clean
~/git101$ echo "To jest piaskownica Git101." > README
~/git101$ touch tanie_dranie.py
~/git101$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    tanie_dranie.py

no changes added to commit (use "git add" and/or "git commit -a")
```

GIT poprawnie wskazał, że nie ma zmian, następnie wykrył zmianę w pliki `README` oraz pojawienie się nowego jeszcze nie śledzonego pliku.

Informacja: Wskazówka zawiera tekst: `no changes added to commit (use "git addąnd/or "git commit -a")`, sugerując użycie komendy `git add`. Wcześniej mówiliśmy, że nie trzeba operacji dodawania powtarzać za każdym razem – otóż nie trzeba, ale można.

Dzięki temu możemy wybierać pliki, których wersje nie zostaną zapisane, tworząc tzw. poczekalnię (ang. *staging*). W niej przygotowujemy zestaw plików, który zostanie zapisany w historii zmian w monecie wykonania `git commit`.

Na razie nie zawracajmy sobie tym głowy, a po więcej informacji zapraszamy do rozdziału o poczekalni.

Zapamiętajmy zmiany pliku `README` w repozytorium przy pomocy wskazanej komendy `git commit -a`:

```
~/git101$ git commit -a -m zmiana1
[master c22799b] zmiana1
 1 file changed, 1 insertion(+)
~/git101$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    tanie_dranie.py
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

GIT pokazuje nam, że plik `tanie_dranie.py` wciąż nie jest śledzony. To nowy plik w naszym katalogu, a my zapomnieliśmy go wcześniej *dodać*:

```
~/git101$ git add tanie_dranie.py
~/git101$ git commit -am nowy1
[master 226e556] nowy1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 tanie_dranie.py
~/git101$ git status
On branch master
nothing to commit, working directory clean
```

Podgląd historii zmian i wyciąganie wersji archiwalnych

W każdym momencie możemy wyciągnąć wersję archiwalną z repozytorium. Sprawdźmy, co sobie zapisaliśmy w repozytorium.

```
~/git101$ git log
commit 226e556d93ab9df6f21574ecdd29ba6b38f6aaab
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:43:28 2015 +0200

    nowy1

commit 1e2678f4190cbf78f3e67aafb0b896128298de03
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:29:37 2015 +0200

    zmiana1

commit e9cffa4b65487f9c5291fa1b9607b1e75e394bc1
Author: Janusz Skonieczny <js@br..labs.pl>
Date:   Thu Jul 16 19:00:04 2015 +0200

    Moja pierwsza wersja plików
```

Teraz sprawdźmy, co się kryje w naszym pliku `README` i wyciągnijmy jego pierwsza wersję:

```
~/git101$ cat README
To jest piaskownica Git101.
~/git101$ git checkout e9cffa
Note: checking out 'e9cffa'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at e9cffa4... Moja pierwsza wersja plików
```

```
~/git101$ cat README
~/git101$ git checkout master
Previous HEAD position was e9cffa4... Moja pierwsza wersja plików
Switched to branch 'master'
~/git101$ cat README
To jest piaskownica Git101.
```

Działo się! Zwróćmy uwagę, jak wskazaliśmy wersję z historii zmian, podaliśmy początek skrótu `e9cffa4b65487f9c5291fa1b9607b1e75e394bc1`, czyli tego opisanego komentarzem `Moja pierwsza wersja plików` do komendy `git checkout`.

Następnie przywróciliśmy najnowsze wersje plików z gałęzi `master`. Wyjaśnienia co to są gałęzie, zostawmy na później, tymczasem wystarczy nam to, że komenda `git checkout master` zapisze nasze pliki w najnowszych wersjach zapamiętanych w repozytorium.

Na razie nie przejmujemy się także ostrzeżeniem `You are in 'detached HEAD' state.`, to także zostawiamy na później.

Spróbujcie teraz poćwiczyć wprowadzanie zmian i zapisywanie ich w repozytorium.

2.3.5 Centrale repozytoria dostępne przez internet

Posługując się repozytoriami plików często mówimy o nich jako o „projektach". Projekty mogą mieć swoje centralne repozytoria dostępne publicznie lub dla wybranych użytkowników.

W szczególności polecamy serwisy:

1. GitHub - https://github.com/ - bezpłatne repozytoria dla projektów widocznych publicznie
2. Bitbucket - https://bitbucket.org/ - bezpłatne repozytoria dla projektów bez wymogu ich upubliczniania

W każdym z nich możemy ograniczyć możliwość modyfikacji kodu do wybranych osób, a wymienione serwisy różnią się tym, że GitHub jest większy i bardziej popularny w środowisku open source, natomiast Bitbucket bezpłatnie umożliwia całkowite ukrycie projektów.

Dodatkowo te serwisy oferują rozszerzony bezpłatnych dostęp dla uczniów i nauczycieli, a także oferują rozbudowane płatne funkcje.

Nowe konto GitHub

Zakładamy, że nauczyciele nie muszą korzystać z prywatnych repozytoriów, a dostęp do większej liczby projektów pomoże w nauce, dlatego początkującym proponujemy założenie konta w serwisie GitHub.

Dodatkowo dla dalszej pracy z tymi przykładami warto jest skonfigurować sobie uwierzytelnianie przy pomocy kluczy SSH.

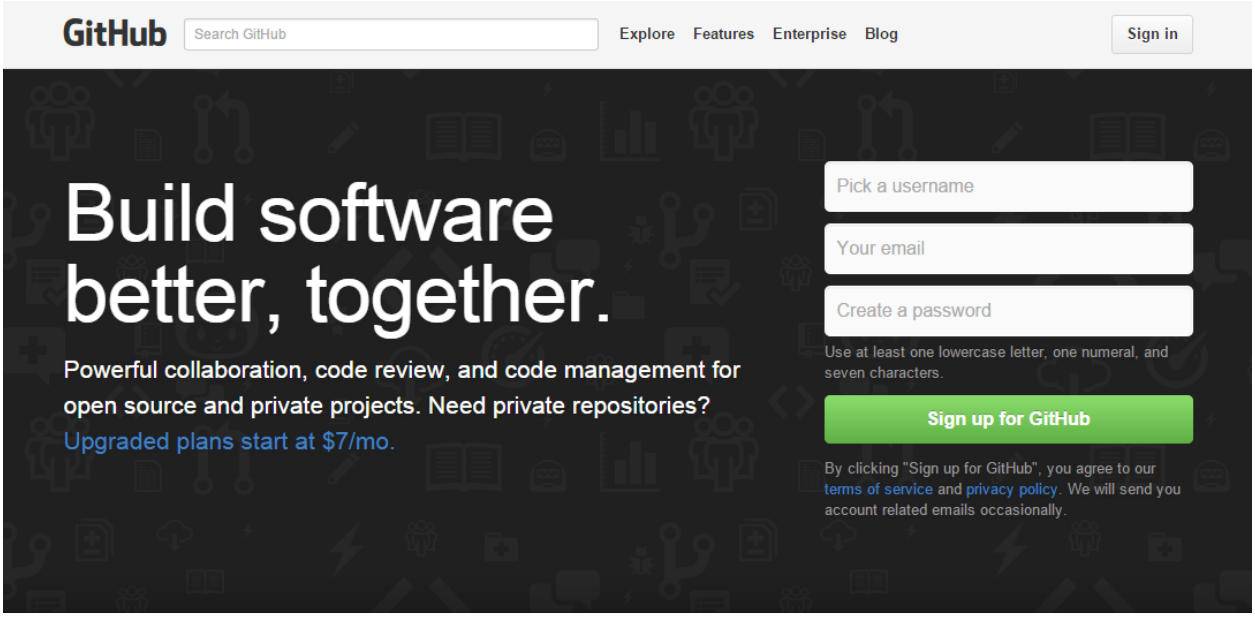
Forkujemy pierwszy projekt

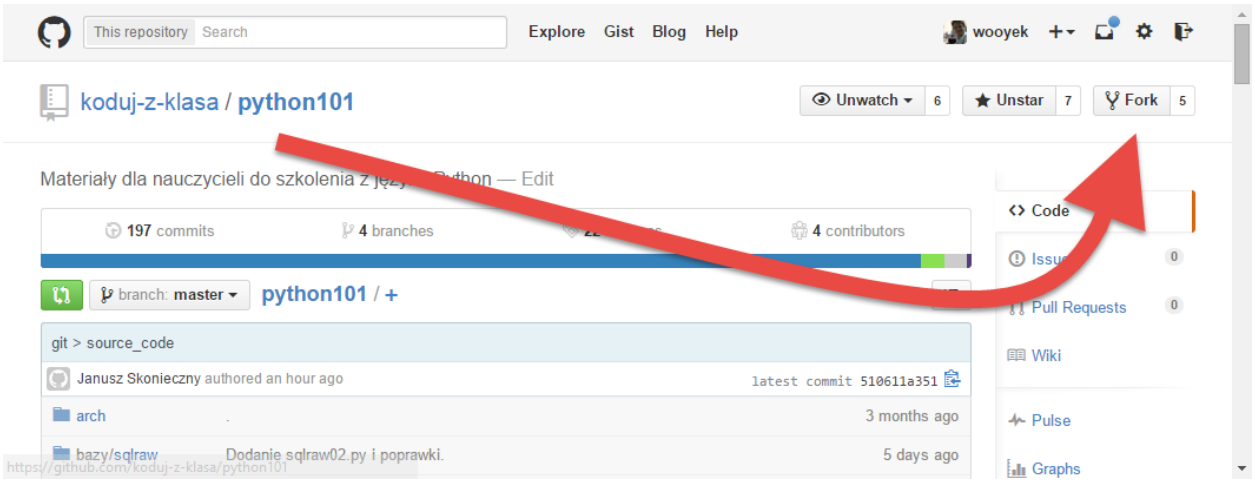
Każdy może sobie skopiować (do własnego repozytorium) i modyfikować projekty publicznie dostępne w GitHub. Dzięki temu każdy może wykonać — na swojej kopii — poprawki i zaprezentować te poprawki światu i autorom projektu :)

Wykonajmy teraz forka naszego projektu z przykładami i tą dokumentacją (tą którą czytasz).

https://github.com/koduj-z-klasa/python101

Oczywiście możemy sobie założyć nowy pusty projekt, ale łatwiej będzie nam się pobawić narzędziami na istniejącym projekcie.


GitHub
Search GitHub
Explore
Features
Enterprise
Blog
Sign in
Build software better, together.
Powerful collaboration, code review, and code management for open source and private projects. Need private repositories?
Upgraded plans start at $7/mo.
Pick a username
Your email
Create a password
Use at least one lowercase letter, one numeral, and seven characters.
Sign up for GitHub
By clicking "Sign up for GitHub", you agree to our terms of service and privacy policy. We will send you account related emails occasionally.

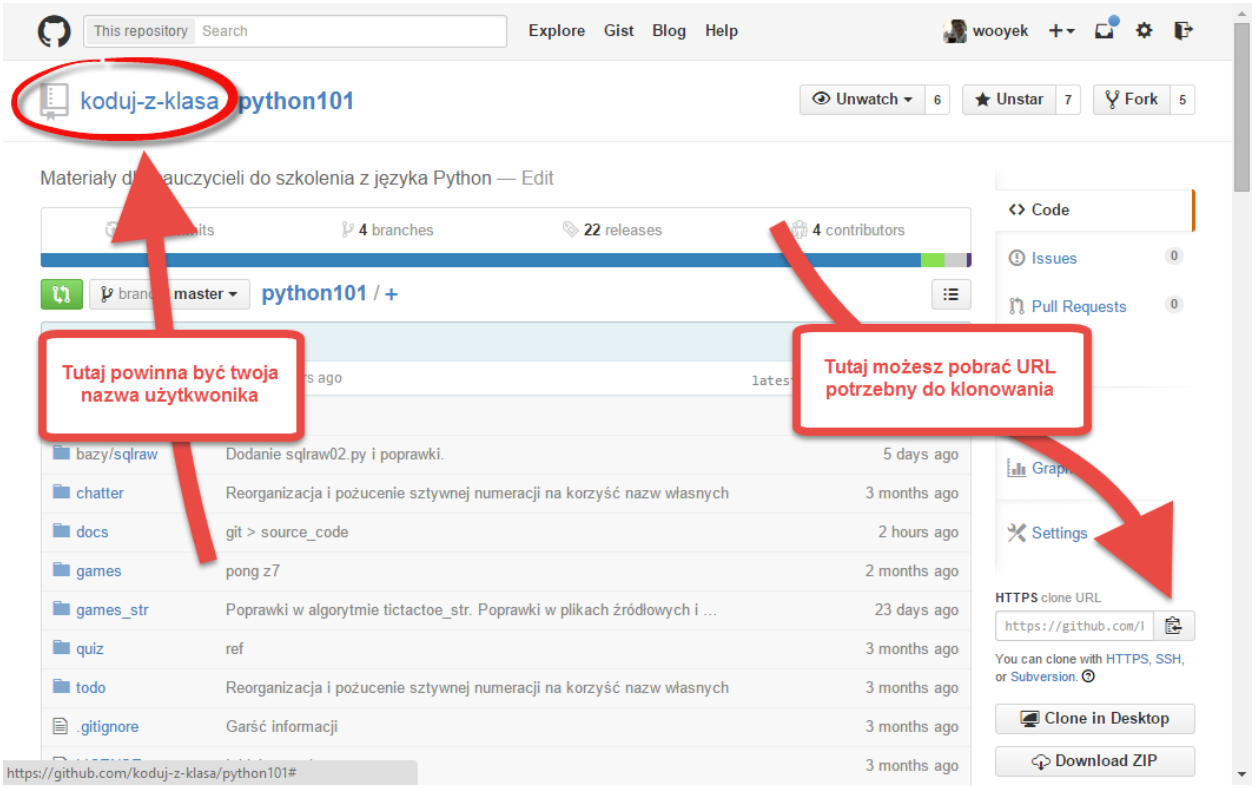

This repository
Search
Explore
Gist
Blog
Help
wooyek
koduj-z-klasa / python101
Unwatch
6
Unstar
7
Fork
5
197 commits
4 branches
4 contributors
branch: master
python101 / +
git > source_code
Janusz Skonieczny authored an hour ago
latest commit 510611a351
arch
3 months ago
bazy/sqlraw
Dodanie sqlraw02.py i poprawki.
5 days ago
Code
Pull Requests
Wiki
Pulse
Graphs

Informacja: Forkując, klonujemy historię zmian w projekcie (więcej o klonowaniu za chwilę).

Forkiem często określamy kopię projektu, która będzie rozwijana niezależnie od oryginału. Np. jeśli chcemy wprowadzić modyfikacje, które nam są potrzebne, ale które nie zostaną przekazane do oryginalnego repozytorium.

Klonujemy nasz projekt lokalnie

Klonowanie to proces tworzenia lokalnej kopii historii zmian. Dzięki temu możemy wprowadzić zmiany i zapisać je lokalnej kopii historii zmian, a następnie synchronizować historie zmian pomiędzy repozytoriami.




```
~$ git clone https://github.com/<MOJA-NAZWA-UŻYTKOWNIKA>/python101.git
```

W efekcie uzyskamy katalog `python101` zawierający kopie plików, które będziemy zmieniać.

Informacja: W podobny sposób uczniowie mogą wykonać lokalną kopię naszych materiałów. Dyskusję czy to jest fork czy klon zostawmy na później ;)

Skok do wybranej wersji z historii zmian

Klon repozytorium zawiera całą historię zmian projektu:

```
~$ cd python101
~/python101$ git log
```

```
commit 510611a351c7c3ff60e2506d8704e3f786fcedb7
Author: Janusz Skonieczny <...>
Date:   Thu Dec 11 15:37:46 2014 +0100

    git > source_code

commit f7019bc1f433eb4a6c2c88f8f48337c77e5e415e
Author: Janusz Skonieczny <...>
Date:   Thu Dec 11 15:26:16 2014 +0100

    req

commit 302fb3a974954ad936a825ba37519e145c148290
Author: wilku-ceo <...>
Date:   Thu Dec 11 11:05:43 2014 +0100

    poprawiona nazwa CEO
```

Możemy skoczyć do dowolnej z nich ustawiając wersje plików w kopii roboczej według jednej z wersji zapamiętanej w historii zmian.

```
~/python101$ git checkout 302fb3

Previous HEAD position was 510611a... git > source_code
HEAD is now at 302fb3a... poprawiona nazwa CEO
```

Zmiany można też oznaczyć czytelnym tagiem tak by łatwiej było zapamiętać miejsca docelowe. W przykładzie poniżej `pong/z1` jest przykładową etykietą wersji plików potrzebnej podczas pracy z pierwszym zadaniem ćwiczenia z grą pong.

```
~/python101$ git checkout pong/z1
```

Tyle tytułem wprowadzenia. Wróćmy do ostatniej wersji i wprowadź jakieś zmiany.

```
~/python101$ git checkout master
```

Zmieniamy i zapisujemy zmiany w lokalnym repozytorium

Dopiszmy coś co pliku `README` i zapiszmy go na dysku. A następnie sprawdźmy pzy pomocy komendy `git status` czy nasza zmiana zostanie wykryta.

```
~/python101$ git status

On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Następnie dodajmy zmiany do repozytorium. Normalnie nie zajmuje to tylu operacji, ale chcemy zobaczyć co się dzieje na każdym etapie.

```
~/python101$ git add README.md
~/python101$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.md


~/python101$ git commit -m "Moja pierwsza zmiana!"
[master 87ec5f4] Moja pierwsza zmiana!
1 file changed, 1 insertion(+), 1 deletion(-)

~/python101$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working directory clean
```

Zazwyczaj wszystkie operacje zapisania zmian w historii zawrzemy w jednej komendzie:

```
~/python101$ git commit -a -m "Moja pierwsza zmiana!"`
```

Wysyłamy zmiany do centralnego repozytorium

Na razie historia naszych zmian została zapisana lokalnie. Możemy w ten sposób pracować nad projektami jednak gdy chcemy podzielić swoim geniuszem ze światem, musimy go wysłać do repozytorium dostępnego przez innych.

```
~/python101$ git push origin master
```

Komenda `push` przyjmuje dwa parametry alias zdalnego repozytorium `origin` oraz nazwę gałęzi zmian `master`.

Wskazówka: Dla uproszczenia wystarczy, że zapamiętasz tą komendę tak jak jest, bez wnikania w znaczenie wartości parametrów. W większości przypadków jest ona wystarczająca do osiągnięcia celu.

Sprawdź teraz czy w twoim repozytorium w serwisie GitHub pojawiły się zmiany.

Przypisujemy tagi do konkretnych wersji w historii zmian

Możemy etykietę przypisać do aktualnej wersji zmian:

```
~/python101$ git tag moja_zmiana
```

Lub wybrać i przypisać ją do wybranej wersji historycznej.

```
~/python101$ git log --pretty=oneline
87ec5f4d8e639365f360bc11b9b51629b909ee9d Moja pierwsza zmiana!
510611a351c7c3ff60e2506d8704e3f786fcedb7 git > source_code
f7019bc1f433eb4a6c2c88f8f48337c77e5e415e req
302fb3a974954ad936a825ba37519e145c148290 poprawiona nazwa CEO
```

```
~/python101$ git tag zmiana_ceo 302fb3a

~/python101$ git show zmiana_ceo
commit 302fb3a974954ad936a825ba37519e145c148290
Author: wilku-ceo <grzegorz.wilczek@ceo.org.pl>
Date:   Thu Dec 11 11:05:43 2014 +0100

    poprawiona nazwa CEO

diff --git a/docs/copyright.rst b/docs/copyright.rst
index 85feb38..431eb81 100644
--- a/docs/copyright.rst
+++ b/docs/copyright.rst
@@ -5,7 +5,7 @@
              <img alt="Licencja Creative Commons" style="border-width:0" src="ht
         Materiały <span xmlns:dct="http://purl.org/dc/terms/" href="http://purl
         udostępniane przez <a xmlns:cc="http://creativecommons.org/ns#" href="h
-        Centrum Edudkacji Europejsci</a> na licencji <a rel="license" href="htt
+        Centrum Edukacji Obywatelskiej</a> na licencji <a rel="license" href="h
         Creative Commons Uznanie autorstwa-Na tych samych warunkach 4.0 Międzyn
     </p>
```

Wysyłamy tagi do centralnego repozytorium

Etykiety które przypiszemy do wersji w historii zmian muszą zostać wypchnięte do centralnego repozytorium przy pomocy specjalnej wersji komendy push.

```
~/python101$ git push origin --tags --force
```

Parametr `--tags` mówi komendzie by wypchnęła nasze etykiety, natomiast `--force` wymusi zmiany w ew. istniejących etykietach — bez `--force` serwer może odrzucić nasze zmiany jeśli takie same etykiety już istnieją w centralnym repozytorium i są przypisane do innych wersji zmian.

Pobieramy zmiany z centralnego repozytorium

Jeśli już mamy klona repozytorium i chcemy upewnić się że mamy lokalnie najnowsze wersje plików (np. gdy nauczyciel zaktualizował przykłady lub dodał nowe pliki), to ciągniemy zmiany z centralnego repozytorium:

```
~/python101$ git pull
```

Ta komenda ściągnie historię zmian z centralnego repozytorium i zaktualizuje naszą kopię roboczą plików.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4 Zaczynamy!

2.4.1 Podstawy Pythona

Python jest dynamicznie typowanym językiem interpretowanym (zob. *język interpretowany*) wysokiego poziomu. Cechuje się czytelnością i zwięzłością kodu. Stworzony został w latach 90. przez Guido van Rossuma, nazwa zaś pochodzi od tytułu serialu komediowego emitowanego w BBC pt. "Latający cyrk Monty Pythona".

Według zestawień serwisu TIOBE Python jest w czołówce popularności języków programowania – 4 miejsce na koniec 2015 r.

W systemach opartych na Linuksie *interpreter* Pythona jest standardowo zainstalowany. W systemach Microsoft Windows należy go *doinstalować*. Interpreter Pythona może i powinien być używany w trybie interaktywnym do nauki i testowania kodu.

Funkcjonalność Pythona może być dowolnie rozszerzana dzięki licznym bibliotekom, które pozwalają tworzyć aplikacje matematyczne (Matplotlib), okienkowe (np. PyQt, PyGTK, wxPython), internetowe (Flask, Django) czy multimedialne i gry (Pygame).

Istnieją również kompleksowe projekty oparte na Pythonie wspomagające naukową analizę, obliczenia i przetwarzanie danych, np.: Anaconda, Enthought Deployment Manager czy Enthought Tool Suite.

Mały Lotek

W *Toto Lotku* trzeba zgadywać liczby. Napiszmy prosty program, w którym będziemy mieli podobne zadanie. Użyjemy języka Python.

Szablon

Zaczynamy od utworzenia pliku o nazwie `toto.py` w dowolnym katalogu za pomocą dowolnego edytora. Zapis `~$` poniżej oznacza katalog domowy użytkownika. Obowiązkowa zawartość pliku:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
```

Pierwsza linia to ścieżka do interpretera Pythona (zob. *interpreter*), druga linia deklaruje sposób kodowania znaków, dzięki czemu możemy używać polskich znaków.

Wartości i zmienne

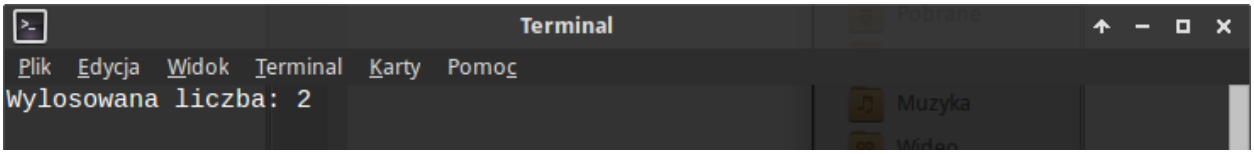
Zaczniemy od wylosowania jednej liczby. Potrzebujemy funkcji `randint(a, b)` z modułu `random`. Zwróci nam ona liczbę całkowitą z zakresu <a; b>. Do naszego pliku dopisujemy:

```
4 import random
5
6 liczba = random.randint(1, 10)
7 print("Wylosowana liczba:", liczba)
```

Wylosowana liczba zostanie zapamiętana w **zmiennej** `liczba` (zob. *zmienna*). Funkcja `print()` wydrukuje ją razem z komunikatem na ekranie. Program możemy już uruchomić w terminalu (zob. terminal), wydając w katalogu z plikiem polecenie:

```
~$ python3 toto.py
```

Efekt działania naszego skryptu:



Wskazówka: Skrypty Pythona możemy też uruchamiać z poziomu edytora, o ile oferuje on taką możliwość.

Wejście – wyjście

Liczbę mamy, niech gracz, czyli użytkownik ją zgadnie. Pytanie tylko, na ile prób mu pozwolimy. Zacznijmy od jednej! Dopisujemy zatem:

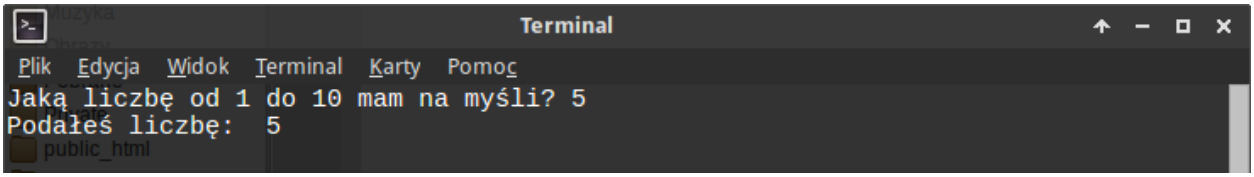
```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 import random
5
6 liczba = random.randint(1, 10)
7 print("Wylosowana liczba:", liczba)
8
9 odp = input("Jaką liczbę od 1 do 10 mam na myśli? ")
```

Liczbę podaną przez użytkownika pobieramy za pomocą funkcji `input()` i zapamiętujemy w zmiennej `odp`.

Uwaga: Zakładamy na razie, że gracz wprowadza poprawne dane, czyli liczby całkowite!

Ćwiczenie 1

- Zgadywanie, gdy losowana liczba jest drukowana, nie jest zabawne. Zakomentuj więc instrukcję drukowania: `# print("Wylosowana liczba:", liczba)` – będzie pomijana przez interpreter.
- Dopisz odpowiednie polecenie, które wyświetli liczbę podaną przez gracza. Przetestuj jego działanie.



Instrukcja warunkowa

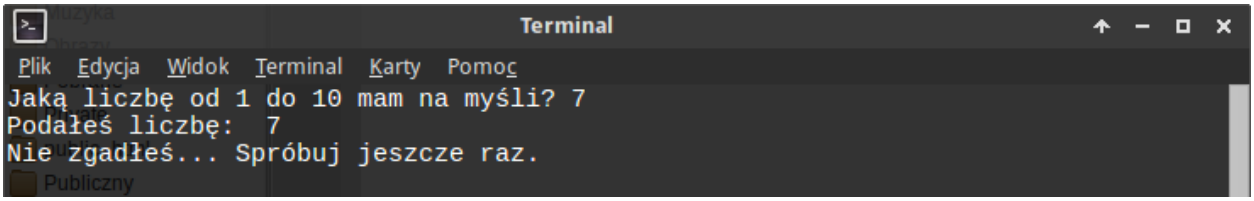
Mamy wylosowaną liczbę i typ gracza, musimy sprawdzić, czy trafił. Uzupełniamy nasz program:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

liczba = random.randint(1, 10)
# print("Wylosowana liczba:", liczba)

odp = input("Jaką liczbę od 1 do 10 mam na myśli? ")
# print("Podałeś liczbę: ", odp)

if liczba == int(odp):
    print("Zgadłeś! Dostajesz długopis!")
else:
    print("Nie zgadłeś. Spróbuj jeszcze raz.")
```

Używamy **instrukcji warunkowej** `if`, która sprawdza prawdziwość warunku `liczba == int(odp)` (zob. *instrukcja warunkowa*). Jeżeli wylosowana i podana liczba są sobie równe (==), wyświetlamy informację o wygranej, w przeciwnym razie (`else:`) zachętę do ponownej próby.

Informacja: Instrukcja `input()` wszystkie pobrane dane zwraca jako napisy (typ *string*). Do przekształcenia napisu na liczbę całkowitą (typ *integer*) wykorzystujemy funkcję `int()`, która w przypadku niepowodzenia zgłasza wyjątek `ValueError`. Obsługę wyjątków omówimy później.

Przetestuj kilkukrotnie działanie programu.



Pętla for

Trafienie za pierwszym razem wylosowanej liczby jest bardzo trudne, damy graczowi 3 szanse. Zmieniamy i uzupełniamy kod:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

liczba = random.randint(1, 10)
# print("Wylosowana liczba:", liczba)

for i in range(3):
    odp = input("Jaką liczbę od 1 do 10 mam na myśli? ")
    # print("Podałeś liczbę: ", odp)
```

```
12
13      if liczba == int(odp):
14          print("Zgadłeś! Dostajesz długopis!")
15          break
16      else:
17          print("Nie zgadłeś. Spróbuj jeszcze raz.")
18          print()
```

Pobieranie i sprawdzanie kolejnych liczb wymaga powtórzeń, czyli **pętli** (zob. *pętla*). Blok powtarzających się operacji umieszczamy więc w instrukcji `for`. Ilość powtórzeń określa wyrażenie `i in range(3)`. **Zmienna iteracyjna** `i` to "licznik" powtórzeń. Przyjmuje on kolejne wartości wygenerowane przez konstruktor `range(n)`. Funkcja ta tworzy sekwencję liczb całkowitych od *0* do *n-1*.

A więc polecenia naszego skryptu, które umieściliśmy w pętli, wykonają się 3 razy, chyba że... użytkownik trafi za 1 lub 2 razem. Wtedy warunek w instrukcji `if` stanie się prawdziwy, wyświetli się informacja o nagrodzie, a polecenie `break` przerwie działanie pętli.

Uwaga: Uwaga na WCIĘCIA!

Podporządkowane bloki kodu wyodrębniamy za pomocą wcięć (zob. *formatowanie kodu*). Standardem są 4 spacje i ich wielokrotności. Przyjęty rozmiar wcięć obowiązuje w całym pliku. Błędy wcięć sygnalizowane są komunikatem `IndentationError`.

W naszym kodzie linie 10, 13, 16 wcięte są na 4 spacje, zaś 14-15, 17-18 na 8.

Ćwiczenia

Sprawdźmy działanie konstruktora `range()` w trybie interaktywnym interpretera Pythona. W terminalu wpisz polecenia:

```
~$ python3
>>> list(range(30))
>>> for i in range(0, 100, 2)
...   print i
...
>>> exit()
```

Funkcja `range()` może przyjmować opcjonalne parametry określające początek, koniec oraz krok generowanej listy wartości.

Uzupełnij kod naszego programu, tak aby wyświetlane były komunikaty: "Próba 1", "Próba 2" itd. przed podaniem liczby.

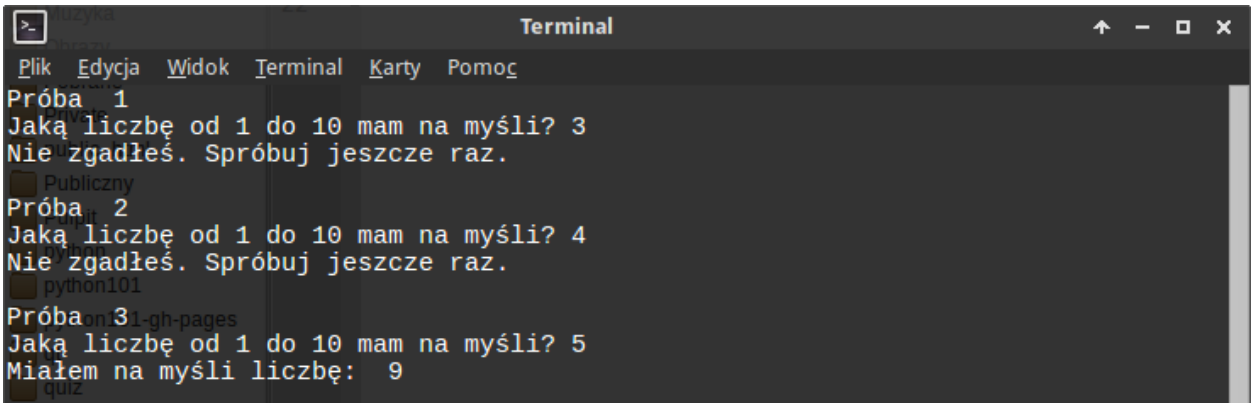
Instrukcja if...elif

Po 3 błędnej próbie program ponownie wyświetla komunikat: "Nie zgadłeś..." Za pomocą członu `elif` możemy wychwycić ten przypadek i właściwie go obsłużyć. Kod przyjmie następującą postać:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  import random
5
6  liczba = random.randint(1, 10)
```

```
# print("Wylosowana liczba:", liczba)

for i in range(3):
    print("Próba ", i + 1)
    odp = input("Jaką liczbę od 1 do 10 mam na myśli? ")
    # print("Podałeś liczbę: ", odp)

    if liczba == int(odp):
        print("Zgadłeś! Dostajesz długopis!")
        break
    elif i == 2:
        print("Miałem na myśli liczbę: ", liczba)
    else:
        print("Nie zgadłeś. Spróbuj jeszcze raz.")
    print()
```

Ostateczny wynik działania naszego programu prezentuje się tak:



Materiały

Źródła:

- Mały Lotek

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Duży Lotek

Zakładamy, że znasz już podstawy podstaw :-) Pythona, czyli scenariusz *Mały Lotek*.

Jedna liczba to za mało, wylosujmy ich więcej! Zasady dużego lotka to typowanie 6 liczb z 49. Ponieważ trafienie jest tu bardzo trudne, napiszemy program w taki sposób, aby można było łatwo dostosować poziom jego trudności.

Na początek

1. Utwórz nowy plik `toto2.py` i uzupełnij go wymaganymi liniami wskazującymi interpreter pythona i użyte kodowanie.
2. Wykorzystując funkcje `input()` oraz `int()` pobierz od użytkownika ilość liczb, które chce odgadnąć i zapisz wartość w zmiennej `ileliczb`.

3. Podobnie jak wyżej pobierz od użytkownika i zapisz maksymalną losowaną liczbę w zmiennej `maksliczba`.
4. Na koniec wyświetl w konsoli komunikat "Wytypuj *ileliczb* z *maksliczba* liczb: ".

Wskazówka: Do wyświetlenia komunikatu można użyć konstrukcji: `print("Wytypuj", ileliczb, ź", maksliczba, łiczb: ")`. Jednak wygodniej korzystać z operatora `%`. Wtedy instrukcja przyjmie postać: `print("Wytypuj %s z %s liczb: "% (ileliczb, maksliczba))`. Symbole zastępcze `%s` zostaną zastąpione kolejnymi wartościami z listy podanej po operatorze `%`. Najczęściej używamy symboli: `%s` – wartość zostaje zamieniona na napis przez funkcję `str()`; `%d` – wartość ma być dziesiętną liczbą całkowitą; `%f` – oczekujemy liczby zmiennoprzecinkowej.

Listy

Ćwiczenie

Jedną wylosowaną liczbę zapamiętywaliśmy w jednej zmiennej, ale przechowywanie wielu wartości w osobnych zmiennych nie jest dobrym pomysłem. Najwygodniej byłoby mieć jedną zmienną, w której można zapisać wiele wartości. W Pythonie takim złożonym typem danych jest *lista*.

Przetestuj w interpreterze następujące polecenia:

```
~$ python3
>>> liczby = []
>>> liczby
>>> liczby.append(1)
>>> liczby.append(2)
>>> liczby.append(4)
>>> liczby.append(4)
>>> liczby
>>> liczby.count(1)
>>> liczby.count(4)
>>> liczby.count(0)
```

Wskazówka: Klawisze kursora (góra, dół) służą w terminalu do przywoływania poprzednich poleceń. Każde przywołane polecenie możesz przed zatwierdzeniem zmienić używając klawiszy lewo, prawo, del i backspace.

Jak widać po zadeklarowaniu pustej listy (`liczby = []`), metoda `.append()` pozwala dodawać do niej wartości, a metoda `.count()` podaje, ile razy dana wartość wystąpiła w liście. To się nam przyda ;-)

Wróćmy do programu i pliku `toto2.py`, który powinien w tym momencie wyglądać tak:

```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 import random
5
6 ileliczb = int(input("Podaj ilość typowanych liczb: "))
7 maksliczba = int(input("Podaj maksymalną losowaną liczbę: "))
8 # print("Wytypuj", ileliczb, "z", maksliczba, " liczb: ")
9 print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))
```

Kodujemy dalej. Użyj pętli:

- dodaj instrukcję `for`, aby wylosować `ileliczb` z zakresu ograniczonego przez `maksliczba`;

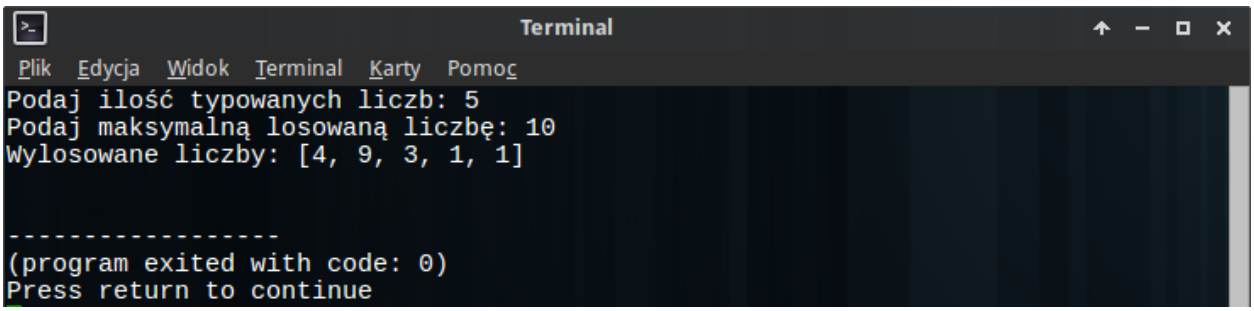
- kolejne losowane wartości drukuj w terminalu;
- sprawdź działanie kodu.

Trzeba zapamiętać losowane wartości:

- przed pętlą zadeklaruj pustą listę;
- wewnątrz pętli umieść polecenie dodające wylosowane liczby do listy;
- na końcu programu (uwaga na wcięcia) wydrukuj zawartość listy;
- kilkukrotnie przetestuj program.

Pętla while

Czy lista zawsze zawiera akceptowalne wartości?



Pętla `for` nie nadaje się do losowania unikalnych liczb, ponieważ wykonuje się określoną ilość razy, a nie możemy zagwarantować, że losowane liczby będą za każdym razem inne. Do wylosowania podanej ilości liczb wykorzystamy więc pętlę `while wyrażenie_logiczne:`, która powtarza kod dopóki podane wyrażenie jest prawdziwe. Uzupełniamy kod w pliku `toto2.py`:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

ileliczb = int(input("Podaj ilość typowanych liczb: "))
maksliczba = int(input("Podaj maksymalną losowaną liczbę: "))
# print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))

liczby = []
# for i in range(ileliczb):
i = 0
while i < ileliczb:
    liczba = random.randint(1, maksliczba)
    if liczby.count(liczba) == 0:
        liczby.append(liczba)
        i = i + 1

print("Wylosowane liczby:", liczby)
```

Losowane liczby zapamiętujemy w **liście** `liczby`. Zmienna `i` to licznik unikalnych wylosowanych liczb, korzystamy z niej w wyrażeniu warunkowym `i < ileliczb`, które kontroluje powtórzenia pętli. W instrukcji warunkowej wykorzystujemy funkcję zliczającą wystąpienia wylosowanej wartości w liście (`liczby.count(liczba)`), aby dodawać (`liczby.append(liczba)`) do listy tylko liczby wcześniej niedodane.

Zbiory

Przy pobieraniu typów użytkownika użyjemy podobnie jak przed chwilą pętli `while`, ale typy zapisywać będziemy w zbiorze, który z założenia nie może zawierać duplikatów (zob. *zbiór*).

Ćwiczenie

W interpreterze Pythona przetestuj następujące polecenia:

```
~$ python3
>>> typy = set()
>>> typy.add(1)
>>> typy.add(2)
>>> typy
>>> typy.add(2)
>>> typy
>>> typy.add(0)
>>> typy.add(9)
>>> typy
```

Pierwsza instrukcja deklaruje pusty zbiór (`typy = set()`). Metoda `.add()` dodaje do zbioru elementy, ale nie da się dodać dwóch takich samych elementów. Drugą cechą zbiorów jest to, że ich elementy nie są w żaden sposób uporządkowane.

Wykorzystajmy zbiór, aby pobrać od użytkownika typy liczb. W pliku `toto2.py` dopisujemy:

```
20 print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))
21 typy = set()
22 i = 0
23 while i < ileliczb:
24     typ = input("Podaj liczbę %s: " % (i + 1))
25     if typ not in typy:
26         typy.add(typ)
27         i = i + 1
```

Zauważ, że operator `in` pozwala sprawdzić, czy podana liczba jest (`if typ in typy`) lub nie (`if typ not in typy:`) w zbiorze. Przetestuj program.

Operacje na zbiorach

Określenie ilości trafień w większości języków programowania wymagałoby przeszukiwania listy wylosowanych liczb dla każdego podanego typu. W Pythonie możemy użyć arytmetyki zbiorów: wyznaczymy część wspólną.

Ćwiczenie

W interpreterze przetestuj poniższe instrukcje:

```
~$ python3
>>> liczby = [1,3,5,7,9]
>>> typy = set([2,3,4,5,6])
>>> set(liczby) | typy
>>> set(liczby) - typy
>>> trafione = set(liczby) & typy
```

```
>>> trafione
>>> len(trafione)
```

Polecenie `set(liczby)` przekształca listę na zbiór. Kolejne operatory zwracają sumę (`|`), różnicę (`-`) i iloczyn (`&`), czyli część wspólną zbiorów. Ta ostania operacja bardzo dobrze nadaje się do sprawdzenia, ile liczb trafił użytkownik. Funkcja `len()` zwraca ilość elementów każdej sekwencji, czyli np. napisu, listy czy zbioru.

Do pliku `toto2.py` dopisujemy:

```
trafione = set(liczby) & typy
if trafione:
    print("\nIlość trafień: %s" % len(trafione))
    print("Trafione liczby: ", trafione)
else:
    print("Brak trafień. Spróbuj jeszcze raz!")
```

Instrukcja `if trafione:` sprawdza, czy część wspólna zawiera jakiekolwiek elementy. Jeśli tak, drukujemy liczbę trafień i trafione liczby.

Przetestuj program dla 5 typów z 10 liczb. Działa? Jeśli masz wątpliwości, wpisz wylosowane i wytypowane liczby w interpreterze, np.:

```
>>> liczby = [1,4,2,6,7]
>>> typy = set([1,2,3,4,5])
>>> trafione = set(liczby) & typy
>>> if trafione:
...   print(len(trafione))
...
>>> print(trafione)
```

Wnioski? Logika kodu jest poprawna, czego dowodzi test w terminalu, ale program nie działa. Dlaczego?

Wskazówka: Przypomnij sobie, jakiego typu wartości zwraca funkcja `input()` i użyj we właściwym miejscu funkcji `int()`.

Wynik działania programu powinien wyglądać następująco:

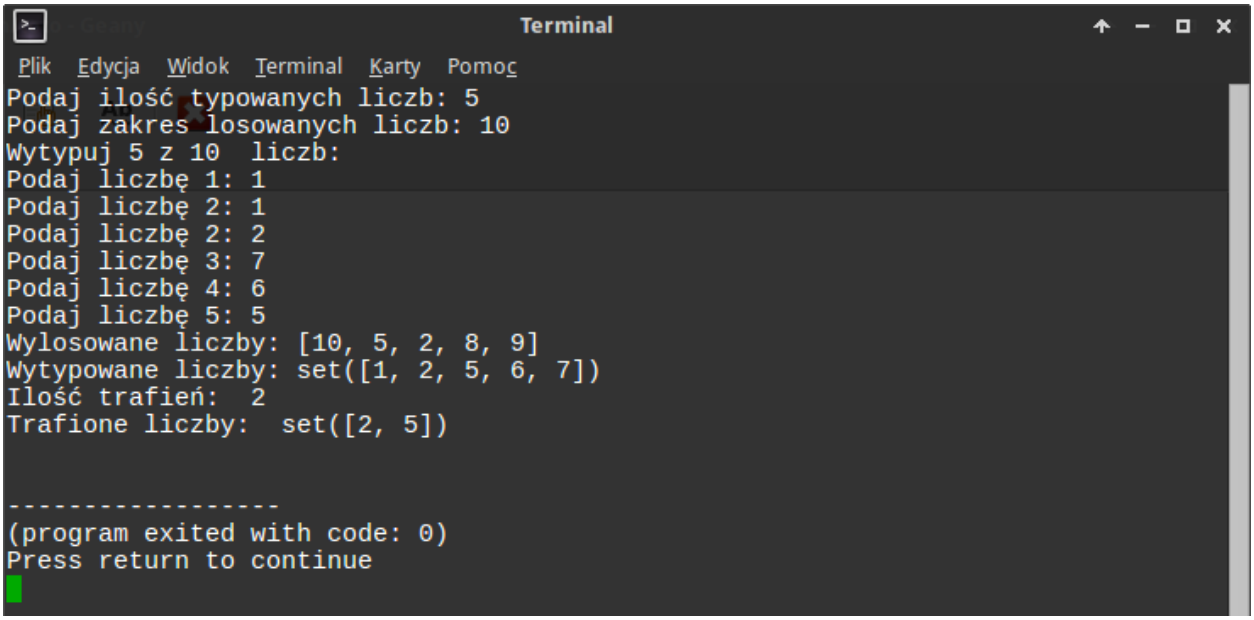
Do 3 razy sztuka

Zastosuj pętlę `for` tak, aby użytkownik mógł 3 razy typować liczby z tej samej serii liczb wylosowanych. Wynik działania programu powinien przypominać poniższy zrzut:

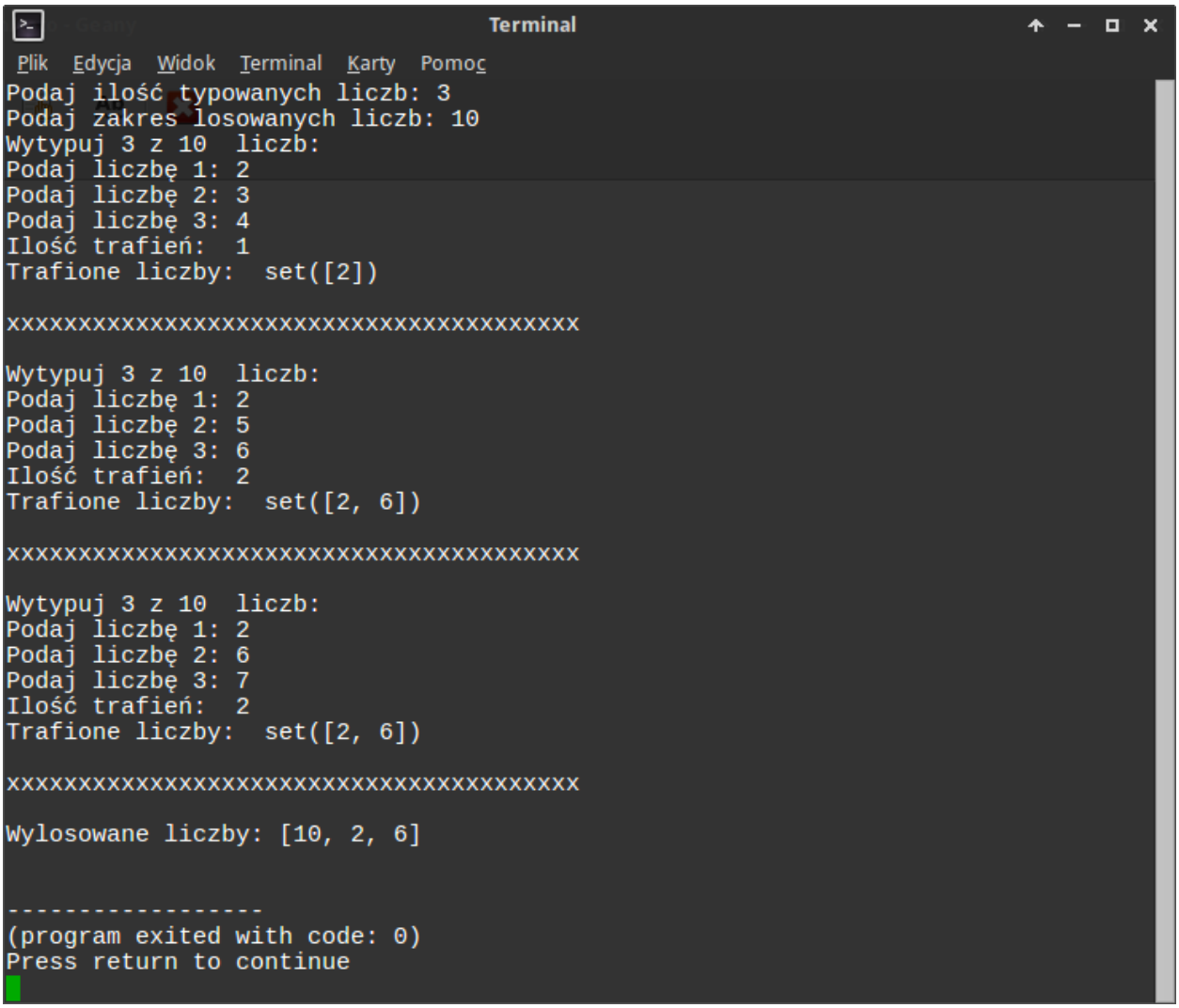
Błędy i wyjątki

Kod naszego programu do tej pory przedstawia się mniej więcej tak:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

ileliczb = int(input("Podaj ilość typowanych liczb: "))
maksliczba = int(input("Podaj maksymalną losowaną liczbę: "))

```


Terminal
Plik Edycja Widok Terminal Karty Pomoc
Podaj ilość typowanych liczb: 5
Podaj zakres losowanych liczb: 10
Wytypuj 5 z 10 liczb:
Podaj liczbę 1: 1
Podaj liczbę 2: 1
Podaj liczbę 2: 2
Podaj liczbę 3: 7
Podaj liczbę 4: 6
Podaj liczbę 5: 5
Wylosowane liczby: [10, 5, 2, 8, 9]
Wytypowane liczby: set([1, 2, 5, 6, 7])
Ilość trafień: 2
Trafione liczby: set([2, 5])

(program exited with code: 0)
Press return to continue


Terminal
Plik Edycja Widok Terminal Karty Pomoc
Podaj ilość typowanych liczb: 3
Podaj zakres losowanych liczb: 10
Wytypuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 3
Podaj liczbę 3: 4
Ilość trafień: 1
Trafione liczby: set([2])
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Wytypuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 5
Podaj liczbę 3: 6
Ilość trafień: 2
Trafione liczby: set([2, 6])
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Wytypuj 3 z 10 liczb:
Podaj liczbę 1: 2
Podaj liczbę 2: 6
Podaj liczbę 3: 7
Ilość trafień: 2
Trafione liczby: set([2, 6])
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Wylosowane liczby: [10, 2, 6]

(program exited with code: 0)
Press return to continue

```
liczby = []
i = 0
while i < ileliczb:
    liczba = random.randint(1, maksliczba)
    # print("Wylosowane liczba: %s " % liczba)
    if liczby.count(liczba) == 0:
        liczby.append(liczba)
        i = i + 1

for i in range(3):
    print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))
    typy = set()
    i = 0
    while i < ileliczb:
        typ = int(input("Podaj liczbę %s: " % (i + 1)))
        if typ not in typy:
            typy.add(typ)
            i = i + 1

    trafione = set(liczby) & typy
    if trafione:
        print("\nIlość trafień: %s" % len(trafione))
        print("Trafione liczby: ", trafione)
    else:
        print("Brak trafień. Spróbuj jeszcze raz!")

    print("\n" + "x" * 40 + "\n")  # wydrukuj 40 znaków x

print("Wylosowane liczby:", liczby)
```

Uruchom powyższy program i podaj ilość losowanych liczb większą od maksymalnej losowanej liczby. Program wpada w nieskończoną pętlę! Po chwili zastanowienia dojdziemy do wniosku, że nie da się wylosować np. 6 unikalnych liczb z zakresu 1-5.

Ćwiczenie

- Użyj w kodzie instrukcji warunkowej, która w przypadku gdy użytkownik chciałby wylosować więcej liczb niż podany zakres maksymalny, wyświetli komunikat "Błędne dane!" i przerwie wykonywanie programu za pomocą funkcji `exit()`.

Testujemy dalej. Uruchom program i zamiast liczby podaj tekst. Co się dzieje? Uruchom jeszcze raz, ale tym razem jako typy podaj wartości spoza zakresu <0;maksliczba>. Da się to zrobić?

Jak pewnie zauważyłeś, w pierwszym wypadku zgłoszony zostaje wyjątek `ValuError` (zob.: *wyjątki*) i komunikat `invalid literal for int() with base 10`, który informuje, że funkcja `int()` nie jest w stanie przekształcić podanego ciągu znaków na liczbę całkowitą. W drugim wypadku podanie nielogicznych typów jest możliwe.

Uzupełnijmy program tak, aby był nieco odporniejszy na niepoprawne dane:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

try:
    ileliczb = int(input("Podaj ilość typowanych liczb: "))
    maksliczba = int(input("Podaj maksymalną losowaną liczbę: "))
```

```
    if ileliczb > maksliczba:
        print("Błędne dane!")
        exit()
except ValueError:
    print("Błędne dane!")
    exit()

liczby = []
i = 0
while i < ileliczb:
    liczba = random.randint(1, maksliczba)
    if liczby.count(liczba) == 0:
        liczby.append(liczba)
        i = i + 1

for i in range(3):
    print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))
    typy = set()
    i = 0
    while i < ileliczb:
        try:
            typ = int(input("Podaj liczbę %s: " % (i + 1)))
        except ValueError:
            print("Błędne dane!")
            continue

        if 0 < typ <= maksliczba and typ not in typy:
            typy.add(typ)
            i = i + 1

    trafione = set(liczby) & typy
    if trafione:
        print("\nIlość trafień: %s" % len(trafione))
        print("Trafione liczby: ", trafione)
    else:
        print("Brak trafień. Spróbuj jeszcze raz!")

    print("\n" + "x" * 40 + "\n")  # wydrukuj 40 znaków x

print("Wylosowane liczby:", liczby)
```

Do przechwytywania wyjątków używamy konstrukcji `try: ... except wyjątek: ...`, czyli: spróbuj wykonać kod w bloku `try`, a w razie błędów przechwyć `wyjątek` i wykonaj podporządkowane instrukcje. W powyższych przypadkach przechwytujemy wyjątek `ValueError`, wyświetlamy odpowiedni komunikat i kończymy działanie programu (`exit()`) lub wymuszamy ponowne wykonanie pętli (`continue`) zamiast ją przerywać (`break`).

Poza tym sprawdzamy, czy użytkownik podaje sensowne typy. Warunek `if 0 < typ <= maksliczba:` to skrócony zapis wyrażenia logicznego z użyciem operatora koniunkcji: `typ > 0 and typ <= maksliczba`. Sprawdzamy w ten sposób, czy wartość zmiennej `typ` jest większa od zera i mniejsza lub równa wartości zmiennej `maksliczba`.

Materiały

Źródła:

- `Duży Lotek`

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Extra Lotek

Kod Toto Lotka wypracowany w dwóch poprzednich częściach wprowadził podstawy programowania w Pythonie: podstawowe typy danych (napisy, liczby, listy, zbiory), instrukcje sterujące (warunkową i pętlę) oraz operacje wejścia-wyjścia w konsoli. Uzyskany skrypt wygląda następująco:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random

try:
    ileliczb = int(input("Podaj ilość typowanych liczb: "))
    maksliczba = int(input("Podaj maksymalną losowaną liczbę: "))
    if ileliczb > maksliczba:
        print("Błędne dane!")
        exit()
except ValueError:
    print("Błędne dane!")
    exit()

liczby = []
i = 0
while i < ileliczb:
    liczba = random.randint(1, maksliczba)
    if liczby.count(liczba) == 0:
        liczby.append(liczba)
        i = i + 1

for i in range(3):
    print("Wytypuj %s z %s liczb: " % (ileliczb, maksliczba))
    typy = set()
    i = 0
    while i < ileliczb:
        try:
            typ = int(input("Podaj liczbę %s: " % (i + 1)))
        except ValueError:
            print("Błędne dane!")
            continue

        if 0 < typ <= maksliczba and typ not in typy:
            typy.add(typ)
            i = i + 1

    trafione = set(liczby) & typy
    if trafione:
        print("\nIlość trafień: %s" % len(trafione))
        print("Trafione liczby: ", trafione)
    else:
        print("Brak trafień. Spróbuj jeszcze raz!")

    print("\n" + "x" * 40 + "\n")  # wydrukuj 40 znaków x

print("Wylosowane liczby:", liczby)
```

Funkcje i moduły

Tam, gdzie w programie występują powtarzające się operacje lub zestaw poleceń realizujący wyodrębnione zadanie, wskazane jest używanie funkcji. Są to nazwane bloki kodu, które można grupować w ramach modułów (zob. *funkcja*, *moduł*). Funkcje zawarte w modułach można importować do różnych programów. Do tej pory korzystaliśmy np. z funkcji `randit()` zawartej w module `random`.

Wyodrębnienie funkcji ułatwia sprawdzanie i poprawianie kodu, ponieważ wymusza podział programu na logicznie uporządkowane kroki. Jeżeli program korzysta z niewielu funkcji, można umieszczać je na początku pliku programu głównego.

Tworzymy więc nowy plik `totomodul.py` i umieszczamy w nim następujący kod:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import random


def ustawienia():
    """Funkcja pobiera ilość losowanych liczb, maksymalną losowaną wartość
    oraz ilość prób. Pozwala określić stopień trudności gry."""
    while True:
        try:
            ile = int(input("Podaj ilość typowanych liczb: "))
            maks = int(input("Podaj maksymalną losowaną liczbę: "))
            if ile > maks:
                print("Błędne dane!")
                continue
            ilelos = int(input("Ile losowań: "))
            return (ile, maks, ilelos)
        except ValueError:
            print("Błędne dane!")
            continue


def losujliczby(ile, maks):
    """Funkcja losuje ile unikalnych liczb całkowitych od 1 do maks"""
    liczby = []
    i = 0
    while i < ile:
        liczba = random.randint(1, maks)
        if liczby.count(liczba) == 0:
            liczby.append(liczba)
            i = i + 1
    return liczby


def pobierztypy(ile, maks):
    """Funkcja pobiera od użytkownika jego typy wylosowanych liczb"""
    print("Wytypuj %s z %s liczb: " % (ile, maks))
    typy = set()
    i = 0
    while i < ile:
        try:
```

```
            typ = int(input("Podaj liczbę %s: " % (i + 1)))
        except ValueError:
            print("Błędne dane!")
            continue

        if 0 < typ <= maks and typ not in typy:
            typy.add(typ)
            i = i + 1
    return typy
```

Funkcja w Pythonie składa się ze słowa kluczowego `def`, nazwy, obowiązkowych nawiasów okrągłych i opcjonalnych parametrów. Na końcu umieszczamy dwukropek. Funkcje zazwyczaj zwracają jakieś dane za pomocą instrukcji `return`.

Zmienne lokalne w funkcjach są niezależne od zmiennych w programie głównym, ponieważ definiowane są w różnych zasięgach, a więc w różnych przestrzeniach nazw. Możliwe jest modyfikowanie zmiennych globalnych dostępnych w całym programie, o ile wskażemy je w funkcji instrukcją typu: `global nazwa_zmiennej`.

Program główny po zmianach przedstawia się następująco:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

from totomodul import ustawienia, losujliczby, pobierztypy


def main(args):
    # ustawienia gry
    ileliczb, maksliczba, ilerazy = ustawienia()

    # losujemy liczby
    liczby = losujliczby(ileliczb, maksliczba)

    # pobieramy typy użytkownika i sprawdzamy, ile liczb trafił
    for i in range(ilerazy):
        typy = pobierztypy(ileliczb, maksliczba)
        trafione = set(liczby) & typy
        if trafione:
            print("\nIlość trafień: %s" % len(trafione))
            print("Trafione liczby: %s" % trafione)
        else:
            print("Brak trafień. Spróbuj jeszcze raz!")

        print("\n" + "x" * 40 + "\n")  # wydrukuj 40 znaków x

    print("Wylosowane liczby:", liczby)
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Na początku z modułu `totomodul`, którego nazwa jest taka sama jak nazwa pliku, importujemy potrzebne funkcje. Następnie w funkcji głównej `main()` wywołujemy je podając nazwę i ewentualne argumenty. Zwracane przez nie wartości zostają przypisane podanym zmiennym.

Warto zauważyć, że funkcja może zwracać więcej niż jedną wartość naraz, np. w postaci tupli `return (ile,`

`maks, ilelos)`. **Tupla** to rodzaj listy, w której nie możemy zmieniać wartości (zob. *tupla*). Jest często stosowana do przechowywania i przekazywania danych, których nie należy modyfikować.

Wiele wartości zwracanych w tupli można jednocześnie przypisać kilku zmiennym dzięki operacji tzw. **rozpakowania tupli**: `ileliczb, maksliczba, ilerazy = ustawienia()`. Należy jednak pamiętać, aby ilość zmiennych z lewej strony wyrażenia odpowiadała ilości elementów w tupli.

Konstrukcja `while True` oznacza nieskończoną pętlę. Stosujemy ją w funkcji `ustawienia()`, aby wymusić na użytkowniku podanie poprawnych danych.

Cały program zawarty został w funkcji głównej `main()`. O tym, czy zostanie ona wykonana decyduje warunek `if __name__ == '__main__':`, który będzie prawdziwy, kiedy nasz skrypt zostanie uruchomiony jako główny. Wtedy nazwa specjalna `__name__` ustawiana jest na `__main__`. Jeżeli korzystamy ze skryptu jako modułu, importując go, `__main__` ustawiane jest na nazwę pliku, dzięki czemu kod się nie wykonuje.

Informacja: Komentarze: w rozbudowanych programach dobrą praktyką ułatwiającą późniejsze przeglądanie i poprawianie kodu jest opatrywanie jego fragmentów **komentarzami**. Zazwyczaj umieszczamy je po znaku `#`. Z kolei funkcje opatruje się krótkim opisem działania i/lub wymaganych argumentów, ograniczanym **potrójnymi cudzysłowami**. Notacja `"..."` lub `'''...'''` pozwala zamieszczać teksty wielowierszowe.

Ćwiczenie

- Przenieś kod powtarzany w pętli `for` (linie 17-24) do funkcji zapisanej w module programu.Wywołanie funkcji: `iletraf = wyniki(set(liczby), typy)` umieść w linii 17 programu głównego. Wykorzystaj szkielet funkcji:

```
def wyniki(liczby, typy):
    """Funkcja porównuje wylosowane i wytypowane liczby,
    zwraca ilość trafień"""
    ...

    return len(trafione)
```

- Popraw wyświetlanie listy trafionych liczb. W funkcji `wyniki()` przed instrukcją `print("Trafione liczby:  %s"% trafione)` wstaw: `trafione = ", ".join(map(str, trafione))`.

  Funkcja `map()` (zob. *mapowanie funkcji*) pozwala na zastosowanie jakiejś innej funkcji, w tym wypadku `str` (czyli konwersji na napis), do każdego elementu sekwencji, w tym wypadku zbioru `trafione`.

  Metoda napisów `join()` pozwala połączyć elementy listy (muszą być typu *string*) podanymi znakami, np. przecinkami (`", "`).

Zapis/odczyt plików

Uruchamiając wielokrotnie program, musimy podawać wiele danych, aby zadziałał. Dodamy więc możliwość zapamiętywania ustawień i ich zmiany. Dane zapisywać będziemy w zwykłym pliku tekstowym. W pliku `toto2.py` dodajemy tylko jedną zmienną `nick`:

```
8     # ustawienia gry
9     nick, ileliczb, maksliczba, ilerazy = ustawienia()
```

W pliku `totomodul.py` zmieniamy funkcję `ustawienia()` oraz dodajemy dwie nowe: `czytaj_ust()` i `zapisz_ust()`.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import random
import os


def ustawienia():
    """Funkcja pobiera nick użytkownika, ilość losowanych liczb, maksymalną
    losowaną wartość oraz ilość typowań. Ustawienia zapisuje."""

    nick = input("Podaj nick: ")
    nazwapliku = nick + ".ini"
    gracz = czytaj_ust(nazwapliku)
    odp = None
    if gracz:
        print("Twoje ustawienia:\nLiczb: %s\nZ Maks: %s\nLosowań: %s" %
              (gracz[1], gracz[2], gracz[3]))
        odp = input("Zmieniasz (t/n)? ")

    if not gracz or odp.lower() == "t":
        while True:
            try:
                ile = int(input("Podaj ilość typowanych liczb: "))
                maks = int(input("Podaj maksymalną losowaną liczbę: "))
                if ile > maks:
                    print("Błędne dane!")
                    continue
                ilelos = int(input("Ile losowań: "))
                break
            except ValueError:
                print("Błędne dane!")
                continue
        gracz = [nick, str(ile), str(maks), str(ilelos)]
        zapisz_ust(nazwapliku, gracz)

    return gracz[0:1] + [int(x) for x in gracz[1:4]]


def czytaj_ust(nazwapliku):
    if os.path.isfile(nazwapliku):
        plik = open(nazwapliku, "r")
        linia = plik.readline()
        plik.close()
        if linia:
            return linia.split(";")
    return False


def zapisz_ust(nazwapliku, gracz):
    plik = open(nazwapliku, "w")
    plik.write(";".join(gracz))
    plik.close()


```

Operacje na plikach:

- `plik = open(nazwapliku, tryb)` – otwarcie pliku w trybie `"w"` (zapis), "r" (odczyt) lub "a" (dopisywanie);
- `plik.readline()` – odczytanie pojedynczej linii z pliku;
- `plik.write(napis)` – zapisanie podanego napisu do pliku;
- `plik.close()` – zamknięcie pliku.

Operacje na tekstach:

- operator +: konkatenacja, czyli łączenie tekstów,
- `linia.split(";")` – rozbijanie tekstu wg podanego znaku na elementy listy,
- `";".join(gracz)` – wspomniane już złączanie elementów listy za pomocą podanego znaku,
- `odp.lower()` – zmiana wszystkich znaków na małe litery,
- `str(arg)` – przekształcanie podanego argumentu na typ tekstowy.

W funkcji `ustawienia()` pobieramy nick użytkownika i tworzymy nazwę pliku z ustawieniami, następnie próbujemy je odczytać wywołując funkcję `czytaj_ust()`. Funkcja ta sprawdza, czy podany plik istnieje na dysku i otwiera go do odczytu. Plik powinien zawierać 1 linię, która przechowuje ustawienia w formacie: `nick;ile_liczb;maks_liczba;ile_prób`. Po jej odczytaniu i rozbiciu na elementy (`linia.split(";")`) zwracamy ją jako listę `gracz`.

Jeżeli uda się odczytać zapisane ustawienia, pytamy użytkownika, czy chce je zmienić. Jeżeli brak ustawień lub użytkownik chce je zmienić, pobieramy informacje, tworzymy z nich listę i przekazujemy do zapisania: `zapisz_ust(nazwapliku, gracz)`.

Ponieważ w programie głównym oczekujemy, że funkcja `ustawienia()` zwróci dane typu *napis, liczba, liczba, liczba* – używamy konstrukcji: `return gracz[0:1] + [int(x) for x in gracz[1:4]]`.

Na początku za pomocą notacji wycinkowej (ang. *slice*, *notacja wycinkowa*) tworzymy 1-elementową listę zawierającą nick użytkownika (`gracz[0:1]`). Pozostałe elementy z listy `gracz` (`gracz[1:4]`) umieszczamy w wyrażeniu listowym (*wyrażenie listowe*). Przy użyciu pętli przekształca ono każdy element na liczbę całkowitą i umieszcza w nowej liście.

Na końcu operator + ponownie tworzy nową listę, która zawiera wartości oczekiwanych typów.

Ćwiczenie

Przećwicz w konsoli notację wycinkową, wyrażenia listowe i łączenie list:

```
~$ python3
>>> dane = ['a', 'b', 'c', '1', '2', '3']
>>> dane[0:3]
>>> dane[3:6]
>>> duze = [x.upper() for x in dane[0:3]]
>>> kwadraty = [int(x)**2 for x in dane[3:6]]
>>> duze + kwadraty
```

Słowniki

Skoro umiemy już zapamiętywać wstępne ustawienia programu, możemy również zapamiętywać losowania użytkownika, tworząc rejestr do celów informacyjnych i/lub statystycznych. Zadanie wymaga po pierwsze zdefiniowania jakieś struktury, w której będziemy przechowywali dane, po drugie zapisu danych albo w plikach, albo w bazie danych.

Na początku dopiszemy kod w programie głównym `toto2.py`:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

from totomodul import ustawienia, losujliczby, pobierztypy, wyniki
from totomodul import czytaj_json, zapisz_json
import time


def main(args):
    # ustawienia gry
    nick, ileliczb, maksliczba, ilerazy = ustawienia()

    # losujemy liczby
    liczby = losujliczby(ileliczb, maksliczba)

    # pobieramy typy użytkownika i sprawdzamy, ile liczb trafił
    for i in range(ilerazy):
        typy = pobierztypy(ileliczb, maksliczba)
        iletraf = wyniki(set(liczby), typy)

    nazwapliku = nick + ".json"  # nazwa pliku z historią losowań
    losowania = czytaj_json(nazwapliku)

    losowania.append({
        "czas": time.time(),
        "dane": (ileliczb, maksliczba),
        "wylosowane": liczby,
        "ile": iletraf
    })

    zapisz_json(nazwapliku, losowania)

    print("\nLosowania:", liczby)
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Dane graczy zapisywać będziemy w plikach nazwanych nickiem użytkownika z rozszerzeniem ".json": `nazwapliku = nick + ".json"`. Informacje o grach umieścimy w liście `losowania`, którą na początku zainicjujemy danymi o grach zapisanymi wcześniej: `losowania = czytaj(nazwapliku)`.

Każda gra w liście `losowania` to *słownik*. Struktura ta pozwala przechowywać dane w parach "klucz: wartość", przy czym indeksami mogą być napisy:

- `ćzas"` – będzie indeksem daty gry (potrzebny import modułu `time`!),
- `"dane"` – będzie wskazywał tuplę z ustawieniami,
- `"wylosowane"` – listę wylosowanych liczb,
- `"ile"` – ilość trafień.

Na koniec dane ostatniej gry dopiszemy do listy (`losowania.append()`), a całą listę zapiszemy do pliku: `zapisz(nazwapliku, losowania)`.

Teraz zobaczmy, jak wyglądają funkcje `czytaj_json()` i `zapisz_json()` w module `totomodul.py`:

```
102 def czytaj_json(nazwapliku):
103     """Funkcja odczytuje dane w formacie json z pliku"""
104     dane = []
105     if os.path.isfile(nazwapliku):
106         with open(nazwapliku, "r") as plik:
107             dane = json.load(plik)
108     return dane
109
110
111 def zapisz_json(nazwapliku, dane):
112     """Funkcja zapisuje dane w formacie json do pliku"""
113     with open(nazwapliku, "w") as plik:
114         json.dump(dane, plik)
```

Kiedy czytamy i zapisujemy dane, ważną sprawą staje się ich format. Najprościej zapisywać dane jako znaki, tak jak zrobiliśmy to z ustawieniami, jednak często programy użytkowe potrzebują zapisywać złożone struktury danych, np. listy, zbiory czy słowniki. Znakowy zapis wymagałby wtedy wielu dodatkowych manipulacji, aby możliwe było poprawne odtworzenie informacji. Prościej jest skorzystać z *serializacji*, czyli zapisu danych obiektowych (zob. *serializacja*). Często stosowany jest prosty format tekstowy JSON.

W funkcji `czytaj()` zawartość podanego pliki dekodujemy do listy: `dane = json.load(plik)`. Funkcja `zapisz()` oprócz nazwy pliku wymaga listy danych. Po otwarciu pliku w trybie zapisu `"w"`, co powoduje wyczyszczenie jego zawartości, dane są serializowane i zapisywane formacie JSON: `json.dump(dane, plik)`.

Dobrą praktyką jest zwalnianie uchwytu do otwartego pliku i przydzielonych mu zasobów poprzez jego zamknięcie: `plik.close()`. Tak robiliśmy w funkcjach czytających i zapisujących ustawienia. Teraz jednak pliki otworzyliśmy przy użyciu konstrukcji typu `with open(nazwapliku, ż") as plik:`, która zadba o ich właściwe zamknięcie.

Przetestuj, przynajmniej kilkukrotnie, działanie programu.

Ćwiczenie

Załóżmy, że jednak chcielibyśmy zapisywać historię losowań w pliku tekstowym, którego poszczególne linie zawierałyby dane jednego losowania, np.: `wylosowane:[4, 5, 7];dane:(3, 10);ile:0;czas:1434482711.67`

Funkcja zapisująca dane mogłaby wyglądać np. tak:

```
def zapisz_str(nazwapliku, dane):
    """Funkcja zapisuje dane w formacie txt do pliku"""
    with open(nazwapliku, "w") as plik:
        for slownik in dane:
            linia = [k + ":" + str(w) for k, w in slownik.iteritems()]
            linia = ";".join(linia)
            # plik.write(linia+"\n") - zamiast tak, można:
            print >>plik, linia
```

Napisz funkcję `czytaj_str()` odczytującą tak zapisane dane. Funkcja powinna zwrócić listę słowników.

Materiały

Źródła:

- `Extra Lotek`

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Python w przykładach

Poznawanie Pythona zrealizujemy poprzez rozwiązywanie prostych zadań, które pozwolą zaprezentować elastyczność i łatwość tego języka. Sugerujemy używanie konsoli Pythona do testowania poznawanych funkcji, konstrukcji i fragmentów kodu.

Mów mi Python!

ZADANIE: Pobierz od użytkownika *imię*, *wiek* i powitaj go komunikatem: "Mów mi Python, mam *x* lat. Witaj w moim świecie *imie*. Jesteś starszy(młodszy) ode mnie."

POJĘCIA: *zmienna, wartość, wyrażenie, wejście i wyjście danych, instrukcja warunkowa, komentarz.*

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

# deklarujemy i inicjalizujemy zmienne
aktRok = 2014
pythonRok = 1989
# obliczamy wiek Pythona
wiekPythona = aktRok - pythonRok

# pobieramy dane
imie = input('Jak się nazywasz? ')
wiek = int(input('Ile masz lat? '))

# wyświetlamy komunikaty
print("Witaj", imie)
print("Mów mi Python, mam", wiekPythona, "lat.")

# instrukcja warunkowa
if wiek > wiekPythona:
    print('Jesteś starszy ode mnie.')
else:
    print('Jesteś młodszy ode mnie.')
```

Deklaracja zmiennej w Pythonie nie jest wymagana, wystarczy podanej nazwie przypisać jakąś wartość za pomocą operatora przypisania "=". Zmiennym często przypisujemy wartości za pomocą wyrażeń, czyli działań arytmetycznych lub logicznych.

Informacja: Niekiedy mówi się, że w Pythonie zmiennych nie ma, są natomiast wartości określonego typu.

Wejście i wyjście danych:

- `input()` zwraca pobrane z klawiatury znaki jako napis, czyli typ **string**.
- `print()` drukuje podane argumenty oddzielone przecinkami.

Napisy ujmujemy w cudzysłowy podwójne lub pojedyncze.

Instrukcja `if wyrażenie` (jeżeli) steruje warunkowym wykonaniem kodu. Jeżeli podane wyrażenie jest prawdziwe (przyjmuje wartość `True`), wykonywana jest pierwsza instrukcja, w przeciwnym wypadku (`else`), kiedy wyrażenie

jest fałszywe (wartość `False`), wykonywana jest instrukcja druga. Części instrukcji warunkowej kończymy dwukropkiem.

Charakterystyczną cechą Pythona jest używanie wcięć do zaznaczania bloków kodu. Standardem są 4 spacje. Komentarze wprowadzamy po znaku `#`.

Funkcja `int()` umożliwia konwersję napisu na liczbę całkowitą, czyli typ **integer**.

Zadania

Zmień program tak, aby zmienna *aktRok* (aktualny rok) była podawana przez użytkownika na początku programu.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Instrukcje warunkowe

Trzy liczby

ZADANIE: Pobierz od użytkownika trzy liczby, sprawdź, która jest najmniejsza i wydrukuj ją na ekranie.

POJĘCIA: *pętla while, obiekt, typ danych, metoda, instrukcja warunkowa zagnieżdżona*.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

op = "t"
while op == "t":
    a, b, c = input("Podaj trzy liczby oddzielone spacjami: ").split(" ")

    print("Wprowadzono liczby:", a, b, c)
    print("\nNajmniejsza:")

    if a < b:
        if a < c:
            najmniejsza = a
        else:
            najmniejsza = c
    elif b < c:
        najmniejsza = b
    else:
        najmniejsza = c

    print(najmniejsza)

    op = input("Jeszcze raz (t/n)? ")

print("Koniec.")
```

Pętla `while warunek` umożliwia powtarzanie bloku operacji, dopóki warunek jest prawdziwy. W tym wypadku dopóki zmienna *op* ma wartość "t". Zwróć uwagę na operator porównania: ==.

W Pythonie wszystko jest obiektem. Każdy obiekt przynależy do jakiegoś typu i ma jakąś wartość. Typ determinuje, jakie operacje można wykonać na wartości danego obiektu. Funkcja `input()` zwraca pobrane dane jako napis (typ *string*). Metoda `split(separator)` pozwala rozbić napis na składowe (w tym wypadku liczby).

Instrukcje warunkowe (`if`), jak i pętle, można zagnieżdżać stosując wcięcia. Instrukcje o takich samych wcięciach tworzą bloki kodu. W jednej złożonej instrukcji warunkowej można sprawdzać wiele warunków (`elif:`).

Zadania

Sprawdź, co się stanie, jeśli podasz liczby oddzielone przecinkiem lub podasz za mało liczb. Zmień program tak, aby poprawnie interpretował dane oddzielane przecinkami.

Trójkąt

ZADANIE: Napisz program, który na podstawie danych pobranych od użytkownika, czyli długości boków, sprawdza, czy da się zbudować trójkąt i czy jest to trójkąt prostokątny. Jeżeli da się zbudować trójkąt, należy wydrukować jego obwód i pole, w przeciwnym wypadku komunikat, że nie da się utworzyć trójkąta.

POJĘCIA: *pętla for, obiekt, typ danych, metoda, lista, instrukcja warunkowa zagnieżdżona*.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import math  # dołączamy bibliotekę matematyczną

op = "t"  # deklarujemy i inicjujemy zmienną pomocniczą
while op != "n":  # dopóki wartość zmiennej op jest inna niż znak "n"
    dane = input("Podaj 3 boki trójkąta (oddzielone przecinkami): ")

    lista = []  # definicja pustej listy
    for x in dane.split(","):
        lista.append(int(x))  # dodanie lementu do listy
    a, b, c = lista  # rozpakowanie listy
    # wyrażenie listowe, które zastępuje kod 10-13:
    # a, b, c = [int(x) for x in dane.split(",")]

    print("Podano boki: ", a, b, c)

    if a + b > c and a + c > b and b + c > a:  # warunek złożony
        print("Z podanych boków można zbudować trójkąt.")
        # czy boki spełniają warunki trójkąta prostokątnego?
        if (a**2 + b**2 == c**2 or
                a**2 + c**2 == b**2 or
                b**2 + c**2 == a**2):
            print("Do tego prostokątny!")

        # na wyjściu możemy wyprowadzać wyrażenia
        print("Obwód wynosi:", (a + b + c))
        p = 0.5 * (a + b + c)  # obliczmy współczynnik wzoru Herona
        # liczymy pole ze wzoru Herona
        P = math.sqrt(p * (p - a) * (p - b) * (p - c))
        print("Pole wynosi:", P)
        op = "n"  # ustawiamy zmienną na "n", aby wyjść z pętli while
    else:
        print("Z podanych odcinków nie można utworzyć trójkąta prostokątnego.")
        op = input("Spróbujesz jeszcze raz (t/n): ")

print("Do zobaczenia...")
```

Pętla `while` działa podobnie jak w poprzednim przykładzie, ale wykorzystuje warunek sformułowany przy wykorzystaniu operatora "różne od": `!=`.

Metoda `split(",")` zwraca listę napisów wyodrębnionych z podanego ciągu. Lista (zob. *lista*) to sekwencja uporządkowanych danych, np. ['3', '4', '5']. Do przeglądania takich sekwencji używa się pętli `for`.

Pętla `for zmienna in sekwencja` odczytuje kolejne elementy *sekwencji* i udostępnia je w *zmiennej*. W ciele pętli zmienną skonwertowaną na liczbę całkowitą dodajemy do nowej listy za pomocą metody `append()`.

Zapis `a, b, c = lista` jest przykładem rozpakowania listy, co polega na przypisaniu zmiennym z lewej strony kolejnych wartości z listy.

Informacja: Pętle, które wykonują jakieś operacje na sekwencjach i zapisują je w listach zastępuje się w Pythonie tzw. wyrażeniami listowymi. Zostaną one omówione w kolejnych przykładach.

Operatory logiczne:

- `and` – koniunkcja ("i"), wskazuje, że obydwa warunki muszą być prawdziwe;
- `or` – alternatywa ("lub"), przynajmniej jeden z podanych warunków powinien być prawdziwy.

Działania matematyczne:

- `x**y` – podnoszenie podstawy `x` do potęgi `y`;
- `sqrt()` – funkcja z modułu `math`, oblicza pierwiastek kwadratowy.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Wydrukuj alfabet

ZADANIE: Wydrukuj alfabet w porządku naturalnym, a następnie odwróconym w formacie: "mała => duża litera". W jednym wierszu trzeba wydrukować po pięć takich grup.

POJĘCIA: *iteracja, pętla, kod ASCII, lista, inkrementacja, operatory arytmetyczne, logiczne, przypisania i zawieranie.*

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

print("Alfabet w porządku naturalnym:")
x = 0
for i in range(65, 91):
    litera = chr(i)
    x += 1
    tmp = litera + " => " + litera.lower()
    if i > 65 and x % 5 == 0:
        x = 0
        tmp += "\n"
    print(tmp, end=" ")

x = -1
print("\nAlfabet w porządku odwróconym:")
for i in range(122, 96, -1):
    litera = chr(i)
    x += 1
    if x == 5:
```

```
        x = 0
        print("\n", end=" ")
    print(litera.upper(), "=>", litera, end=" ")
```

Pętla `for` wykorzystuje zmienną iteracyjną `i`, która przybiera kolejne wartości zwracane przez funkcję `range()`. Parametry tej funkcji określają wartość początkową i końcową, przy czym wartość końcowa nie jest zwracana. Kod `range(122,96,-1)` generuje wartości malejące od 122 do 97(!) z krokiem -1. Sprawdź w interpreterze:

```
>>> list(range(0, 100))
>>> list(range(122,96,-1))
```

Operacje na znakach:

- `chr(kod_ascii)` – zwraca znak odpowiadający podanemu kodowi ASCII;
- `lower()` – zwraca napis zamieniony na małe litery;
- `upper()` – zwraca napis zamieniony na duże litery;
- `+` – operator łączenia (konkatenacji) naspisów.

Operatory arytmetyczne i logiczne:

- `x += 1` – dodanie do zmiennej *x* wartości po prawej stronie – *1*;
- `%` – dzielenie modulo, zwraca resztę z dzielenia;
- == – operator porównania, nie mylić z operatorem przypisania (=);
- `and` – operator logicznej koniunkcji, obydwa warunki muszą być prawdziwe.

Zob.: *operatory* dostępne w Pythonie.

Zadania

- Uprość warunek w pierwszej pętli `for` drukującej alfabet w porządku naturalnym tak, aby nie używać operatora modulo.
- Wydrukuj co n-tą grupę liter alfabetu, przy czym wartość *n* podaje użytkownik. Wskazówka: użyj opcjonalnego, trzeciego argumentu funkcji `range()`.
- Sprawdź działanie różnych operatorów Pythona w konsoli.

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Pobierz *n* liczb

ZADANIE: Pobierz od użytkownika *n* liczb i zapisz je w liście. Wydrukuj: elementy listy i ich indeksy, elementy w odwrotnej kolejności, posortowane elementy. Usuń z listy pierwsze wystąpienie elementu podanego przez użytkownika. Usuń z listy element o podanym indeksie. Podaj ilość wystąpień oraz indeks pierwszego wystąpienia podanego elementu. Wybierz z listy elementy od indeksu *i* do *j*.

POJĘCIA: *lista, metoda, notacja wycinkowa, tupla*.

Wszystkie poniższe przykłady warto wykonać w konsoli Pythona. Treść komunikatów w funkcjach `print()` można skrócić. Można również wpisywać kolejne polecenia do pliku i sukcesywnie go uruchomiać.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

from random import randint

ile = int(input("Ile liczb wylosować? "))
lista = []  # pusta lista
for i in range(0, ile):
    lista.append(randint(0, 100))
print(lista)

print("Dodawanie elementów na końcu listy")
for i in range(0, 3):
    liczba = int(input("Podaj liczbę: "))
    lista.append(liczba)
print(lista)

print("Zawartość listy ([indeks] wartość):")
for i, v in enumerate(lista):
    print("[", i, "]", v)

print("Elementy w odwróconym porządku:")
for e in reversed(lista):
    print(e, end=" ")

print()
print("Elementy posortowane rosnąco:")
for e in sorted(lista):
    print(e, end=" ")

print()
e = int(input("Którą liczbę usunąć? "))
lista.remove(e)
print(lista)

print("Wstawianie elementów do listy")
a, i = eval(input("Podaj element i indeks oddzielone przecinkiem: "))
lista.insert(i, a)
print(lista)

print("Wyszukiwanie i zliczanie elementu w liście")
e = int(input("Podaj liczbę: "))
print("Liczba wystąpień: ")
print(lista.count(e))
print("Indeks pierwszego wystąpienia: ")
if lista.count(e):
    print(lista.index(e))
else:
    print("Brak elementu w liście")

print("Pobieramy ostatni element z listy: ")
print(lista.pop())
print(lista)

print("Część listy (notacja wycinkowa):")
i, j = eval(input("Podaj indeks początkowy i końcowy oddzielone przecinkiem: "))
print(lista[i:j])

```

```
59 print()
60 print("Tupla to lista niemodyfikowalna.")
61 print("Próba zmiany tupli generuje błąd:")
62 tupla = tuple(lista)
63 tupla[0] = 100
```

Na początku z modułu `random` importujemy funkcję `randint(a, b)`, która służy do generowania liczb z przedziału [a, b]. Wylosowane liczby dodajemy do listy.

Lista (zob. *lista*) to sekwencja indeksowanych danych, zazwyczaj tego samego typu. Listę tworzymy ujmując wartości oddzielone przecinkami w nawiasy kwadratowe, np. `lista = [1, 'a']`. Dostęp do elementów sekwencji uzyskujemy podając nazwę i indeks, np. `lista[0]`. Elementy indeksowane są od 0 (zera!). Z każdej sekwencji możemy wydobywać fragmenty dzięki notacji wycinkowej (ang. *slice*, zob. *notacja wycinkowa*), np.: `lista[1:4]`.

Informacja: Sekwencjami w Pythonie są również napisy i tuple.

Funkcje działające na sekwencjach:

- `len()` – zwraca ilość elementów;
- `enumerate()` – zwraca obiekt zawierający indeksy i elementy sekwencji;
- `reversed()` – zwraca obiekt zawierający odwróconą sekwencję;
- `sorted(lista)` – zwraca kopię listy posortowanej rosnąco;
- `sorted(lista, reverse=True)` – zwraca kopię listy w odwrotnym porządku;

Lista ma wiele użytecznych metod:

- `.append(x)` – dodaje x do listy;
- `.remove(x)` – usuwa pierwszy x z listy;
- `.insert(i, x)` – wstawia x przed indeksem i;
- `.count(x)` – zwraca ilość wystąpień x;
- `.index(x)` – zwraca indeks pierwszego wystąpienia x;
- `.pop()` – usuwa i zwraca ostatni element listy;
- `.sort()` – sortuje listę rosnąco;
- `.reverse()` – sortuje listę w odwróconym porządku.

Tupla to niemodyfikowalna lista. Wykorzystywana jest do zapamiętywania i przekazywania wartości, których nie powinno się zmieniać. Tuple tworzymy podając wartości w nawiasach okrągłych, np. `tupla = (1, 'a')` lub z listy za pomocą funkcji: `tuple(lista)`. Tupla może powstać również poprzez spakowanie wartości oddzielonych przecinkami, np. `tupla = 1, 'a'`. Próba zmiany wartości w tupli generuje błąd.

Funkcja `eval()` interpretuje swój argument jako kod Pythona. W instrukcji `a, i = eval(input("Podaj element i indeks oddzielone przecinkiem: "))` podane przez użytkownika liczby oddzielone przecinkiem interpretowane są jako tupla, która następnie zostaje rozpakowana, czyli jej elementy zostają przypisane do zmiennych z lewej strony. Przetestuj w konsoli Pythona:

```
>>> tupla = 2, 6
>>> a, b = tupla
>>> print(a, b)
```

Zadania dodatkowe

Utwórz w konsoli Pythona dowolną listę i przećwicz notację wycinkową. Sprawdź działanie indeksów pustych i ujemnych, np. `lista[2:], lista[:4], lista[-2], lista[-2:]`. Posortuj trwale dowolną listę malejąco. Utwórz kopię listy posortowaną rosnąco.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Ciąg Fibonacciego

ZADANIE: Wypisz ciąg Fibonacciego aż do *n*-tego wyrazu podanego przez użytkownika. Ciąg Fibonacciego to ciąg liczb naturalnych, którego każdy wyraz poza dwoma pierwszymi jest sumą dwóch wyrazów poprzednich. Początkowe wyrazy tego ciągu to: 0 1 1 2 3 5 8 13 21. Przyjmujemy, że 0 wchodzi w skład ciągu.

POJĘCIA: *funkcja, zwracanie wartości, tupla, rozpakowanie tupli, przypisanie wielokrotne*.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-


def fib_iter1(n):  # definicja funkcji
    """
        Funkcja drukuje kolejne wyrazy ciągu Fibonacciego
        aż do wyrazu n-tego, który zwraca.
        Wersja iteracyjna z pętlą while.
    """
    pwyrazy = (0, 1)  # dwa pierwsze wyrazy ciągu zapisane w tupli
    a, b = pwyrazy  # przypisanie wielokrotne, rozpakowanie tupli
    print(a, end=" ")
    while n > 1:
        print(b, end=" ")
        a, b = b, a + b  # przypisanie wielokrotne
        n -= 1


def fib_iter2(n):
    """
        Funkcja drukuje kolejne wyrazy ciągu Fibonacciego
        aż do wyrazu n-tego, który zwraca.
        Wersja iteracyjna z pętlą for.
    """
    a, b = 0, 1
    print("wyraz", 1, a)
    print("wyraz", 2, b)
    for i in range(1, n - 1):
        # wynik = a + b
        a, b = b, a + b
        print("wyraz", i + 2, b)

    print()  # wiersz odstępu
    return b


def fib_rek(n):
    """
```

```
        Funkcja zwraca n-ty wyraz ciągu Fibonacciego.
        Wersja rekurencyjna.
    """
    if n < 1:
        return 0
    if n < 2:
        return 1
    return fib_rek(n - 1) + fib_rek(n - 2)


def main(args):
    n = int(input("Podaj nr wyrazu: "))
    fib_iter1(n)
    print()
    print("=" * 40)
    fib_iter2(n)
    print("=" * 40)
    print(fib_rek(n - 1))
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Instrukcje realizujące jedno zadanie zazwyczaj grupujemy w funkcje, które można później wielokrotnie wywoływać. Funkcję definiujemy za pomocą słowa kluczowego `def` wg schematu `def nazwa_funkcji(parametry):`. Przy czym parametry są opcjonalne. Po dwukropku od nowego wiersza umieszczamy odpowiednio wcięte instrukcje, które tworzą ciało funkcji. Funkcja może zwrócać jakąś wartość za pomocą polecenia `return wartość`.

Zapis `a, b = pwyrazy` jest przykładem rozpakowania tupli, tzn. zmienne *a* i *b* przyjmują wartości kolejnych elementów tupli `pwyrazy`. Zapis równoważny, w którym nie definiujemy tupli tylko wprost podajemy wartości, to `a, b = 0, 1`; ten sposób przypisania wielokrotnego stosujemy w kodzie `a, b = b, b + a`. Jak widać, ilość zmiennych z lewej strony musi odpowiadać liczbie wartości rozpakowywanych z tupli lub liczbie wartości podawanych wprost z prawej strony.

Podane przykłady pokazują, że algorytmy iteracyjne można implementować za pomocą różnych instrukcji sterujących, w tym wypadku pętli `while` i `for`, a także z wykorzystaniem podejścia rekurencyjnego. W tym ostatnim wypadku zwróć uwagę na argument wywołania funkcji.

Zadania dodatkowe

Zmień funkcje tak, aby zwracały poprawne wartości przy założeniu, że dwa pierwsze wyrazy ciągu równe są 1 (bez zera).

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Oceny z przedmiotów

ZADANIE: Napisz program, który umożliwi wprowadzanie ocen z podanego przedmiotu ścisłego (np. fizyki), następnie policzy i wyświetla średnią, medianę i odchylenie standardowe wprowadzonych ocen. Funkcje pomocnicze i statystyczne umieść w osobnym module.

POJĘCIA: *import, moduł, zbiór, przechwytywanie wyjątków, formatowanie napisów i danych na wyjściu, argumenty funkcji, zwracanie wartości.*

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

# importujemy funkcje z modułu ocenyfun zapisanego w pliku ocenyfun.py
from ocenyfun import drukuj, srednia, mediana, odchylenie


def main(args):
    przedmioty = set(['polski', 'angielski'])  # definicja zbioru
    drukuj(przedmioty, "Lista przedmiotów zawiera: ")

    print("\nAby przerwać wprowadzanie przedmiotów, naciśnij Enter.")
    while True:
        przedmiot = input("Podaj nazwę przedmiotu: ")
        if len(przedmiot):
            if przedmiot in przedmioty:  # czy przedmiot jest w zbiorze?
                print("Ten przedmiot już mamy :-)")
            przedmioty.add(przedmiot)  # dodaj przedmiot do zbioru
        else:
            drukuj(przedmioty, "\nTwoje przedmioty: ")
            przedmiot = input("\nZ którego przedmiotu wprowadzisz oceny? ")
            # jeżeli przedmiotu nie ma w zbiorze
            if przedmiot not in przedmioty:
                print("Brak takiego przedmiotu, możesz go dodać.")
            else:
                break  # wyjście z pętli

    oceny = []  # pusta lista ocen
    ocena = None  # zmienna sterująca pętlą i do pobierania ocen
    print("\nAby przerwać wprowadzanie ocen, podaj 0 (zero).")

    while not ocena:
        try:
            ocena = int(input("Podaj ocenę (1-6): "))
            if (ocena > 0 and ocena < 7):
                oceny.append(float(ocena))
            elif ocena == 0:
                break
            else:
                print("Błędna ocena.")
            ocena = None
        except ValueError:
            print("Błędne dane!")

    drukuj(oceny, przedmiot.capitalize() + " - wprowadzone oceny: ")
    s = srednia(oceny)  # wywołanie funkcji z modułu ocenyfun
    m = mediana(oceny)  # wywołanie funkcji z modułu ocenyfun
    o = odchylenie(oceny, s)  # wywołanie funkcji z modułu ocenyfun
    print("\nŚrednia: {0:5.2f}".format(s))
    print("Mediana: {0:5.2f}\nOdchylenie: {1:5.2f}".format(m, o))
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Jak to działa

Klauza `from moduł import funkcja` umożliwia wykorzystanie w programie funkcji zdefiniowanych w innych modułach i zapisanych w osobnych plikach. Dzięki temu utrzymujemy przejrzystość programu głównego, a jednocześnie możemy funkcje z modułów wykorzystywać, importując je w innych programach. Nazwa modułu to nazwa pliku z kodem pozbawiona jednak rozszerzenia *.py*. Moduł musi być dostępny w ścieżce przeszukiwania, aby można go było poprawnie dołączyć.

Informacja: W przypadku prostych programów zapisuj moduły w tym samym katalogu co program główny.

Operacje na zbiorach:

- `set()` – tworzy *zbiór*, czyli nieuporządkowany zestaw niepowtarzalnych (!) elementów;
- `.add(x)` – pozwala dodać element x do zbioru, o ile nie jest już w zbiorze;
- `.remove(x)` – usuwa element x ze zbioru;
- `element (not) in zbior` – operator zawierania `(not) in` sprawdza, czy podany element jest lub nie w zbiorze.

Oceny z wybranego przedmiotu pobieramy w pętli dopóty, dopóki użytkownik nie wprowadzi 0 (zera). Blok `try...except` pozwala przechwycić wyjątki, czyli w tym przypadku błąd przekształcenia wartości na liczbę całkowitą. Jeżeli funkcja `int()` zwróci wyjątek, wykonywane są instrukcje w bloku `except ValueError:`, w przeciwnym razie po sprawdzeniu poprawności oceny dodajemy ją jako liczbę zmiennoprzecinkową (typ *float*) do listy: `oceny.append(float(ocena))`.

Metoda `.capitalize()` pozwala wydrukować podany napis dużą literą.

W funkcji `print("Mediana: {0:5.2f}\nOdchylenie: {1:5.2f}".format(m, o))` zastosowano formatowanie wyświetlanych wartości. Nawiasy `{}` oznaczają pola zastępowane przez wartości podane jako argumenty metody `format()`. W ciągu `{0:5.2f}` pierwsza cyfra wskazuje, który argument (numerowane od zera) metody `format()`, wydrukować. Po dwukropku podajemy szerokość pola przeznaczonego na wydruk. Po kropce – ilość miejsc po przecinku. Symbol *f* oznacza liczbę zmiennoprzecinkową stałej precyzji.

Więcej informacji nt. formatowania danych wyjściowych: PyFormat.

Funkcje wykorzystywane w programie **oceny**, umieszczamy w osobnym pliku `ocenyfun.py`.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

"""
    Moduł ocenyfun zawiera funkcje wykorzystywane w pliku 05_oceny_03.py
"""

import math  # zaimportuj moduł matematyczny


def drukuj(co, kom="Sekwencja zawiera: "):
    print(kom)
    for i in co:
        print(i, end=" ")


def srednia(oceny):
```

```
18      suma = sum(oceny)
19      return suma / float(len(oceny))
20
21
22  def mediana(oceny):
23      """
24      Jeżeli ilość ocen jest parzysta, medianą jest średnia arytmetyczna
25      dwóch środkowych ocen. Jesli ilość  jest nieparzysta mediana równa
26      się elementowi środkowemu ouporządkowanej rosnąco listy ocen.
27      """
28      oceny.sort()
29      if len(oceny) % 2 == 0:  # parzysta ilość ocen
30          half = int(len(oceny) / 2)
31          # można tak:
32          # return float(oceny[half-1]+oceny[half]) / 2.0
33          # albo tak:
34          return float(sum(oceny[half - 1:half + 1])) / 2.0
35      else:  # nieparzysta ilość ocen
36          return oceny[len(oceny) / 2]
37
38
39  def wariancja(oceny, srednia):
40      """
41      Wariancja to suma kwadratów różnicy każdej oceny i średniej
42      podzielona przez ilość ocen:
43      sigma = (o1-s)+(o2-s)+...+(on-s) / n, gdzie:
44      o1, o2, ..., on - kolejne oceny,
45      s - średnia ocen,
46      n - liczba ocen.
47      """
48      sigma = 0.0
49      for ocena in oceny:
50          sigma += (ocena - srednia)**2
51      return sigma / len(oceny)
52
53
54  def odchylenie(oceny, srednia):  # pierwiastek kwadratowy z wariancji
55      w = wariancja(oceny, srednia)
56      return math.sqrt(w)
```

Klauzula `import math` udostępnia w pliku wszystkie metody z modułu matematycznego, dlatego musimy odwoływać się do nich za pomocą notacji *moduł.funkcja*, np.: `math.sqrt()` – zwraca pierwiastek kwadratowy.

Funkcja `drukuj(co, kom="...")` przyjmuje dwa argumenty, *co* – listę lub zbiór, który drukujemy w pętli for, oraz *kom* – komunikat, który wyświetlamy przed wydrukiem. Argument *kom* jest opcjonalny, przypisano mu bowiem wartość domyślną, która zostanie użyta, jeżeli użytkownik nie poda innej w wywołaniu funkcji.

Funkcja `srednia()` do zsumowania wartości ocen wykorzystuje funkcję `sum()`.

Funkcja `mediana()` sortuje otrzymaną listę "w miejscu" (`oceny.sort()`), tzn. trwale zmienia porządek elementów. W zależności od długości listy zwraca wartość środkową (długość nieparzysta) lub średnią arytmetyczną dwóch środkowych wartości (długość). Zapis `oceny[half-1:half+1]` wycina i zwraca dwa środkowe elementyz listy, przy czym wyrażenie `half = int(len(oceny) / 2)` wylicza nam indeks drugiego ze środkowych elementów.

Informacja: Przypomnijmy: alternatywna funkcja `sorted(lista)` zwraca uporządkowaną rosnąco kopię listy.

W funkcji `wariancja()` pętla *for* odczytuje kolejne oceny i w kodzie `sigma += (ocena-srednia)**2` ko-

rzysta z operatorów skróconego dodawania (+=) i potęgowania (**), aby wyliczyć sumę kwadratów różnic kolejnych ocen i średniej.

Zadania dodatkowe

- W konsoli Pythona utwórz listę `wyrazy` zawierającą elementy: *abrakadabra* i *kordoba*. Utwórz zbiór *w1* poleceniem `set(wyrazy[0])`. Oraz zbiór *w2* poleceniem `set(wyrazy[1])`. Wykonaj kolejno polecenia ilustrujące użycie klasycznych operatorów na zbiorach, czyli: różnica (-) , suma (|), przecięcie (część wspólna, &) i elementy unikalne (^):

```
>>> print(w1 - w2)
>>> print(w1 | w2)
>>> print(w1 & w2)
>>> print(w1 ^ w2)
```

- W pliku `ocenyfun.py` dopisz funkcję, która wyświetli wszystkie oceny oraz ich odchylenia od wartości średniej.

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Słownik słówek

ZADANIE: Przygotuj słownik zawierający obce wyrazy oraz ich możliwe znaczenia. Pobierz od użytkownika dane w formacie: *wyraz obcy: znaczenie1, znaczenie2, ...* itd. Pobieranie danych kończy wpisanie słowa "koniec". Podane dane zapisz w pliku. Użytkownik powinien mieć możliwość dodawania nowych i zmieniania zapisanych danych.

POJĘCIA: *słownik, odczyt i zapis plików, formatowanie napisów, format csv.*

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import os  # moduł udostępniający funkcję isfile()

slownik = {}  # pusty słownik
sPlik = "slownik.txt"  # nazwa pliku zawierającego wyrazy i ich tłumaczenia


def otworz(plik):
    if os.path.isfile(sPlik):  # czy istnieje plik słownika?
        with open(sPlik, "r") as pliktxt:  # otwórz plik do odczytu
            for line in pliktxt:  # przeglądamy kolejne linie
                # rozbijamy linię na wyraz obcy i tłumaczenia
                t = line.split(":")
                wobcy = t[0]
                # usuwamy znaki nowych linii
                znaczenia = t[1].replace("\n", "")
                znaczenia = znaczenia.split(",")  # tworzymy listę znaczeń
                # dodajemy do słownika wyrazy obce i ich znaczenia
                slownik[wobcy] = znaczenia
    return len(slownik)  # zwracamy ilość elementów w słowniku


def zapisz(slownik):
    # otwieramy plik do zapisu, istniejący plik zostanie nadpisany(!)
```

```
    pliktxt = open(sPlik, "w")
    for wobcy in slownik:
        # "sklejamy" znaczenia przecinkami w jeden napis
        znaczenia = ",".join(slownik[wobcy])
        # wyraz_obcy:znaczenie1,znaczenie2,...
        linia = ":".join([wobcy, znaczenia])
        pliktxt.write(linia)  # zapisujemy w pliku kolejne linie
        # można też tak:
        # print(linia, file=pliktxt)
    pliktxt.close()  # zamykamy plik


def oczysc(str):
    str = str.strip()  # usuń początkowe lub końcowe białe znaki
    str = str.lower()  # zmień na małe litery
    return str


def main(args):
    print("""Podaj dane w formacie:
    wyraz obcy: znaczenie1, znaczenie2
    Aby zakończyć wprowadzanie danych, podaj 0.
    """)

    # wobce = set() # pusty zbiór wyrazów obcych
    # zmienna oznaczająca, że użytkownik uzupełnił lub zmienił słownik
    nowy = False
    ileWyrazow = otworz(sPlik)
    print("Wpisów w bazie:", ileWyrazow)

    # główna pętla programu
    while True:
        dane = input("Podaj dane: ")
        t = dane.split(":")
        wobcy = t[0].strip().lower()  # robimy to samo, co funkcja oczysc()
        if wobcy == 'koniec':
            break
        elif dane.count(":") == 1:  # sprawdzamy poprawność danych
            if wobcy in slownik:
                print("Wyraz", wobcy, " i jego znaczenia są już w słowniku.")
                op = input("Zastąpić wpis (t/n)? ")
            # czy wyrazu nie ma w słowniku? a może chcemy go zastąpić?
            if wobcy not in slownik or op == "t":
                znaczenia = t[1].split(",")  # znaczenia zapisujemy w liście
                znaczenia = list(map(oczysc, znaczenia))  # oczyszczamy listę
                slownik[wobcy] = znaczenia
                nowy = True
        else:
            print("Błędny format!")

    if nowy:
        zapisz(slownik)

    print(slownik)

    print("=" * 50)
    print("{0: <15}{1: <40}".format("Wyraz obcy", "Znaczenia"))
    print("=" * 50)
```

```
85      for wobcy in slownik:
86          print("{0: <15}{1: <40}".format(wobcy, ",".join(slownik[wobcy])))
87      return 0
88
89
90  if __name__ == '__main__':
91      import sys
92      sys.exit(main(sys.argv))
```

Słownik (zob. *słownik*) to struktura nieuporządkowanych danych w formacie *klucz:wartość*. Kluczami są najczęściej napisy, które wskazują na wartości dowolnego typu, np. inne napisy, liczby, listy, tuple itd. W programie wykorzystujemy słownik, którego kluczami są wyrazy obce, natomiast wartościami są listy możliwych znaczeń.

Operacje na słowniku:

- `slownik = { 'go':['iść','pojechać'] }` – utworzenie 1-elementowego słownika;
- `slownik['make'] = ['robić','marka']` – dodanie nowego elementu;
- `slownik['go']` – odczyt elementu.

Aby zilustrować niektóre operacje na napisach i listach, elementy słownika zapisywać będziemy do pliku w formacie *wyraz_obcy:znaczenie1,znaczeni2,...*. Funkcja `otworz()` przekształca format pliku na słownik, a funkcja `zapisz()` słownik na format pliku.

Operacje na plikach:

- `os.path.isfile(plik)` – sprawdzenie, czy istnieje podany plik;
- `open(plik, "w")` – otwarcie pliku w podanym trybie: *"r"* – odczyt(domyślny), *"w"* – zapis, *"a"* – dopisywanie;
- `with open(plik) as zmienna:` – otwarcie pliku w instrukcji `with ... as ...` zapewnia obsługę błędów, dba o zamknięcie pliku i udostępnia jego zawartość w *zmiennej*;
- `for linia in zmienna:` – pętla, która odczytuje kolejne linie pliku;
- `plik.write(tresc)` – zapisuje do pliku podaną treść;
- `plik.close()` – zamyka plik.

Operacje na napisach:

- `.split(":")` – zwraca listę części napisu wydzielone według podanego znaku;
- `",".join(lista)` – zwraca elementy listy połączone podanym znakiem (w tym wypadku przecinkiem);
- `.lower()` – zamienia znaki na małe litery;
- `.strip()` – usuwa początkowe i końcowe białe znaki (spacje, tabulatory);
- `.replace(ćo",ćzym")` – zastępuje w ciągu wszystkie wystąpienia *co* – *czym*;
- `.count(znak)` – zwraca ilość wystąpień znaku w napisie.

W pętli głównej programu dane pobrane w formacie *wyraz_obcy:znaczenie1,znaczeni2,...* rozbijamy na wyraz obcy i jego znaczenia, które zapisujemy w liście *t*. Wszystkie elementy oczyszczamy, tj. zamieniamy na małe litery i usuwamy białe znaki. Funkcja `map(oczysc, znaczenia)` pozwala zastosować podaną jako pierwszy argument funkcję *oczysc* do wszystkich elementów listy *znaczenia* podanej jako argument drugi. Instrukcja `list()` przekształca zwrócony przez funkcję `map()` obiekt z powrotem na listę.

Formatowanie napisów

Metoda napisów `format()` pozwala na drukowanie przekazanych jej jako argumentów danych zgodnie z ciągami formatującymi umieszczanymi w nawiasach klamrowych w napisie, np. `{0: <15}{1: <40}`. Pierwsza cyfra

wskazuje, do którego z kolejnych argumentów metody `format()` odnosi się ciąg formatujący. Po dwukropku podajemy znak wypełnienia (" " – spacja), symbol "<" oznacza wyrównanie do lewej, a ostatnia cyfra ("15") to szerokość pola. Zob. dokumentację Format String Syntax.

Zapis w pliku csv

Dane można też wygodnie zapisywać do pliku w formacie csv. Jest to rozwiązanie wygodniejsze, ponieważ zwalnia nas od konieczności ręcznego przekształcania odczytywanych z pliku linii na struktury danych.

Na początku pliku dodajemy import modułu: `import csv`. Następnie zmieniamy funkcje `otworz()` i `zapisz()` na podane niżej:

```
11 def otworz(plik):
12     if os.path.isfile(sFile):  # czy istnieje plik słownika?
13         with open(sFile, newline='') as plikcsv:  # otwórz plik do odczytu
14             tresc = csv.reader(plikcsv)
15             for linia in tresc:  # przeglądamy kolejne linie
16                 slownik[linia[0]] = linia[1:]
17     return len(slownik)  # zwracamy ilość elementów w słowniku
18
19
20 def zapisz(slownik):
21     # otwieramy plik do zapisu, istniejący plik zostanie nadpisany(!)
22     with open(sFile, "w", newline='') as plikcsv:
23         tresc = csv.writer(plikcsv)
24         for wobcy in slownik:
25             lista = slownik[wobcy]
26             lista.insert(0, wobcy)
27             tresc.writerow(lista)
```

Format csv polega na zapisywaniu wartości oddzielonych separatorem, czyli domyślnie przecinkiem. Jeżeli wartość zawiera znak separatora, jest cytowana domyślnie za pomocą cudzysłowu. W naszym wypadku przykładowa linia pliku przyjmie postać: *wyraz obcy,znaczenie1,znaczenie2,...*

W powyższym kodzie używamy metody `csv.reader(plik)`, która interpretuje podany plik jako zapisany w formacie csv i każdą linię zwraca w postaci listy elementów. Instrukcja `slownik[linia[0]] = linia[1:]` zapisuje dane w słowniku, kluczem jest wyraz obcy (1 ellement listy), wartościami – lista znaczeń.

W funkcji zapisującej dane w formacie csv, na początku tworzymy obiekt `tresc` zwrócony przez metodę `csv.writer(plik)`. Po przygotowaniu listy zawierającej wyraz obcy i jego znaczenia zapisujemy ją za pomocą metody `writerow(lista)`.

Zadania dodatkowe

- Kod drukujący słownik zamień w funkcję. Wykorzystaj ją do wydrukowania słownika odczytanego z dysku i słownika uzupełnionego przez użytkownika.
- Spróbuj zmienić program tak, aby umożliwiał usuwanie wpisów.
- Dodaj do programu możliwość uczenia się zapisanych w słowniku słówek. Niech program wyświetla kolejne słowa obce i pobiera od użytkownika możliwe znaczenia. Następnie powinien wyświetlać, które z nich są poprawne.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Szyfr Cezara

ZADANIE: Napisz program, który podany przez użytkownika ciąg znaków szyfruje przy użyciu szyfru Cezara i wyświetla zaszyfrowany tekst.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

KLUCZ = 3


def szyfruj(txt):
    zaszyfrowny = ""
    for i in range(len(txt)):
        if ord(txt[i]) > 122 - KLUCZ:
            zaszyfrowny += chr(ord(txt[i]) + KLUCZ - 26)
        else:
            zaszyfrowny += chr(ord(txt[i]) + KLUCZ)
    return zaszyfrowny


def main(args):
    tekst = input("Podaj ciąg do zaszyfrowania:\n")
    print("Ciąg zaszyfrowany:\n", szyfruj(tekst))
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

W programie możemy wykorzystywać zmienne globalne, np. `KLUCZ`. `def nazwa_funkcji(argumenty)` – tak definiujemy funkcje, które mogą lub nie zwracać jakieś wartości. `nazwa_funkcji(argumenty)` – tak wywołujemy funkcje. Napisy są indeksowane (od 0), co daje dostęp do pojedynczych znaków. Funkcja `len(str)` zwraca długość napisu, wykorzystana jako argument funkcji `range()` pozwala iterować po znakach napisu. Operator `+=` oznacza dodanie argumentu z prawej strony do wartości z lewej.

Zadania dodatkowe

- Podany **kod można uprościć**, ponieważ napisy w Pythonie są sekwencjami. Zatem pętlę odczytującą kolejne znaki można zapisać jako `for znak in tekst:`, a wszystkie wystąpienia notacji indeksowej `txt[i]` zastąpić zmienną `znak`.
- Napisz funkcję deszyfrującą `deszyfruj(txt)`.
- Dodaj do funkcji `szyfruj() i deszyfruj()` drugi parametr w postaci długości klucza podawanej przez użytkownika.
- Dodaj poprawne szyfrowanie dużych liter, obsługę białych znaków i znaków interpunkcyjnych.

Przykład funkcji deszyfrującej:

```
def deszyfruj(tekst):
    odszyfrowany = ""
    KLUCZM = KLUCZ % 26
    for znak in tekst:
        if (ord(tekst) - KLUCZM < 97):
```

```
            odszyfrowany += chr(ord(tekst) - KLUCZM + 26)
        else:
            odszyfrowany += chr(ord(tekst) - KLUCZM)
    return odszyfrowany
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Pythonizmy

Python jest językiem wydajnym i zwartym dzięki wbudowanym mechanizmom ułatwiającym wykonywanie typowych i częstych zadań programistycznych. Podane niżej **przykłady należy przećwiczyć w konsoli Pythona**, którą uruchamiamy poleceniem w terminalu:

```
~$ python
```

Operatory * i **

Operator $\star$ służy rozpakowaniu listy zawierającej wiele argumentów, które chcemy przekazać do funkcji:

```
# wygeneruj liczby parzyste od 2 do 10
lista = [2,11,2]
list(range(*lista))
```

Operator $\star\star$ potrafi z kolei rozpakować słownik, dostarczając funkcji nazwanych argumentów (ang. *keyword argument*):

```
def kalendarz(data, wydarzenie):
    print("Data:", data,"\nWydarzenie:", wydarzenie)

slownik = {"data" : "10.02.2015", "wydarzenie" : "szkolenie"}
kalendarz(**slownik)
```

Pętle

Pętla to podstawowa konstrukcja wykorzystywana w językach programowania. Python oferuje różne sposoby powtarzania wykonywania określonych operacji, niekiedy wygodniejsze lub zwięźlejsze niż pętle. Są to przede wszystkim generatory wyrażeń i wyrażenia listowe, a także funkcje `map()` i `filter()`.

```
kwadraty = []
for x in range(10):
    kwadraty.append(x**2)
print(kwadraty)
```

Iteratory

Obiekty, z których pętle odczytują kolejne dane to *iteratory* (ang. *iterators*) Są to strumienie danych zwracanych po jednej wartości na raz za pomocą metody `__next()__`. Jeżeli w strumieniu nie ma więcej danych, wywoływany jest wyjątek `StopIteration`.

Wbudowana funkcja `iter()` zwraca iterator utworzony z dowolnego iterowalnego obiektu. Iteratory wykorzystujemy do przeglądania **list**,** tupli**, **słowników** i **plików** używając instrukcji `for x in y`, w której *y* jest obiektem iterowalnym równoważnym wyrażeniu `iter(y)`. Np.:

```
lista = [2, 4, 6]
for x in lista:
    print(x)

slownik = {'Adam':1, 'Bogdan':2 , 'Cezary':3}
for x in slownik:
    print(x, slownik[x])
```

Listy można łączyć ze sobą i przekształcać w inne iterowalne obiekty. Z dwóch list lub z jednej zawierającej tuple (klucz, wartość) można utworzyć słownik:

```
panstwa = ['Polska', 'Niemcy', 'Francja']  # lista państw
stolice = ['Warszawa', 'Berlin', 'Paryż']  # lista stolic
panstwa_stolice = zip(panstwa, stolice)  # utworzenie iteratora
lista_tupli = list(panstwa_stolice)  # utworzenie listy tupli (państwo, stolica)
print(lista_tupli)
slownik = dict(lista_tupli)  # utworzenie słownika z listy tupli
print(slownik)

slownik.items()  # zwraca tuple (klucz, wartość)
slownik.keys()  # zwraca klucze
slownik.values()  # zwraca wartości

for klucz, wartosc in slownik.items():
    print(klucz, wartosc)
```

Generatory wyrażeń

Jeżeli chcemy wykonać jakąś operację na każdym elemencie sekwencji lub wybrać podzespół elementów spełniający określone warunki, stosujemy *generatory wyrażeń* (ang. *generator expressions*), które zwracają iteratory. Poniższy przykład wydrukuje wszystkie imiona z dużej litery:

```
wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
imiona = (imie.capitalize() for imie in wyrazy)
for imie in imiona:
    print(imie)
```

Schemat składniowy generatora jest następujący: `( wyrażenie for x in sekwencja if warunek )` – przy czym:

- `wyrażenie` – powinno zawierać zmienną `x` z pętli `for`
- `if warunek` – opcjonalna klauzula filtrująca wartości nie spełniające warunku

Gdybyśmy chcieli wybrać tylko imiona 3-literowe w wyrażeniu, użyjemy:

```
imiona = (imie.capitalize() for imie in wyrazy if len(imie) == 3)
list(imiona)
```

Omawiane wyrażenia można zagnieżdżać. Przykłady podajemy niżej.

Wyrażenia listowe

Jeżeli nawiasy okrągłe w generatorze wyrażeń zamienimy na kwadratowe, dostaniemy *wyrażenie listowe* (ang. *list comprehensions*), które – jak wskazuje nazwa – zwraca listę:

```
# wszystkie poniższe wyrażenia listowe możemy przypisać do zmiennych,
# aby móc później korzystać z utworzonych list

# lista kwadratów liczb od 0 do 9
[x**2 for x in range(10)]

# lista dwuwymiarowa [20,40] o wartościach a
a = int(input("Podaj liczbę całkowtią: "))
[[a for y in range(20)] for x in range(40)]

# lista krotek (x, y), przy czym x != y
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]

# utworzenie listy 3-literowych imion i ich pierwszych liter
wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
[ [imie, imie[0]] for imie in wyrazy if len(imie) == 3 ]

# zagnieżdzone wyrażenie listowe tworzące listę współrzędnych
# opisujących tabelę
[ (x,y) for x in range(5) for y in range(3) ]

# zagnieżdzone wyrażenie listowe wykorzystujące filtrowanie danych
# lista kwadratów z zakresu {5;50}
[ y for y in [ x**2 for x in range(10) ] if y > 5 and y < 50 ]
```

Wyrażenia listowe w elegancki i wydajny sposób zastępują takie rozwiązania, jak:

- *pętla*
- *mapowanie funkcji*
- *wyrażenia lambda*
- *filtrowanie danych*

Mapowanie funkcji

Funkcja `map()` funkcję podaną jako pierwszy argument stosuje do każdego elementu sekwencji podanej jako argument drugi:

```
def kwadrat(x):
    return x**2

kwadraty = map(kwadrat, range(10))
list(kwadraty)
```

Wyrażenia lambda

Słowo kluczowe `lambda` pozwala utworzyć zwięzły odpowiednik prostej, jednowyrażeniowej funkcji. Poniższy przykład należy rozumieć następująco: do każdej liczby wygenerowanej przez funkcję `range()` zastosuj funkcję w postaci wyrażenia lambda podnoszącą argument do kwadratu, a uzyskane wartości zapisz w liście `kwadraty`.

```
kwadraty = map(lambda x: x**2, range(10))
list(kwadraty)
```

Funkcje *lambda* często stosowane są w poleceniach sortowania jako wyrażenie zwracające klucz (wartość), wg którego mają zostać posortowane elementy. Jeżeli np. mamy listę tupli opisującą uczniów:

```
uczniowie = [
    ('jan','Nowak','1A',15),
    ('ola','Kujawiak','3B',17),
    ('andrzej','bilski','2F',16),
    ('kamil','czuja','1B',14)
]
```

- `sorted(uczniowie)` – posortuje listę wg pierwszego elementu każdej tupli, czyli imienia;
- `sorted(uczniowie, key=lambda x: x[1])` – posortuje listę wg klucza zwróconego przez jednoargumentową funkcję lambda, w tym wypadku będzie to nazwisko;
- `max(uczniowie, key=lambda x: x[3])` – zwróci najstarszego ucznia;
- `min(uczniowie, key=lambda x: x[3])` – zwróci najmłodszego ucznia.

Filtrowanie danych

Funkcja `filter()` jako pierwszy argument pobiera funkcję zwracającą `True` lub `False`, stosuje ją do każdego elementu sekwencji podanej jako argument drugi i zwraca tylko te, które spełniają założony warunek:

```
wyrazy = ['anna', 'ala', 'ela', 'wiola', 'ola']
imiona = filter(lambda imie: len(imie) == 3, wyrazy)
list(imiona)
```

Generatory

Generatory (ang. *generators*) to funkcje ułatwiające tworzenie iteratorów. Od zwykłych funkcji różnią się tym, że:

- zwracają iterator za pomocą słowa kluczowego `yield`,

- zapamiętują swój stan z momentu ostatniego wywołania, są więc wznawialne (ang. *resumable*),
- zwracają następną wartość ze strumienia danych podczas kolejnych wywołań metody `next()`.

Najprostszy przykład generatora zwracającego kolejne liczby parzyste:

```
def gen_parzyste(N):
    for i in range(N):
        if i % 2 == 0:
            yield i

gen = gen_parzyste(10)
next(gen)
next(gen)
list(gen)
```

Pliki

Czytanie plików tekstowych:

```
with open("test.txt", "r") as f:  # odczytywanie linia po linii
    for linia in f:
        print(linia.strip())

f = open('test.txt', 'r')
for linia in f:  # odczytywanie linia po linii
    print(linia.strip())
f.close()

f = open('test.txt', 'r')
tresc = f.read()  # odczytanie zawartości całego pliku
for znak in tresc:  # odczytaywanie znak po znaku
    print(znak)
f.close()
```

Pierwsza metoda używająca instrukcji `with ... as ...` jest preferowana, ponieważ zapewnia obsługę błędów i dba o zamknięcie pliku.

Zapisywanie danych do pliku tekstowego:

```
dane = ['pierwsza linia', 'druga linia']
with open('output.txt', 'w') as f:
    for linia in dane:
        f.write(linia + '\n')
```

Użycie formatu csv:

```
import csv  # moduł do obsługi formatu csv

dane = ([1, 'jan', 'kowalski'], [2, 'anna', 'nowak'])
plik = "test.csv"

with open(plik, 'w', newline='') as plikcsv:
    tresc = csv.writer(plikcsv, delimiter=';')
    for lista in dane:
        tresc.writerow(lista)

with open(plik, newline='') as plikcsv:  # otwórz plik do odczytu
```

```
    tresc = csv.reader(plikcsv, delimiter=';')
    for linia in tresc:  # przeglądamy kolejne linie
        print(linia)
```

Użycie formatu json:

```
import os
import json

dane = {'uczen1':[1, 'jan', 'kowalski'], 'uczen2':[2, 'anna', 'nowak']}
plik = "test.json"

with open(plik, 'w') as plikjson:
    json.dump(dane, plikjson)

if os.path.isfile(plik):  # sprawdzenie, czy plik istnieje
    with open(plik, 'r') as plikjson:
        dane = json.load(plikjson)
    print(dane)
```

Materiały

1. http://pl.wikibooks.org/wiki/Zanurkuj_w_Pythonie
2. http://brain.fuw.edu.pl/edu/TI:Programowanie_z_Pythonem
3. http://pl.python.org/docs/tut/
4. http://en.wikibooks.org/wiki/Python_Programming/Input_and_Output
5. https://wiki.python.org/moin/HandlingExceptions
6. http://learnpython.org/pl
7. http://www.checkio.org
8. http://www.codecademy.com
9. https://www.coursera.org

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Słownik Pythona

język interpretowany język, który jest tłumaczony i wykonywany "w locie", np. Python lub PHP. Tłumaczeniem i wykonywaniem programu zajmuje się specjalny program nazwany interpreterem języka.

interpreter program, który analizuje kod źródłowy, a następnie go wykonuje. Interpretery są podstawowym składnikiem języków wykorzystywanych do pisania skryptów wykonywanych po stronie klienta WWW (JavaScript) lub serwera (np. Python, PHP).

Interpreter Pythona jest interaktywny, tzn. można w nim wydawać polecenia i obserwować ich działanie, co pozwala wygodnie uczyć się i testować oprogramowanie. Uruchamiany jest w terminalu, zazwyczaj za pomocą polecenia `python`.

formatowanie kodu Python wymaga formatowania kodu za pomocą wcięć, podstawowym wymogiem jest stosowanie takich samych wcięć w obrębie pliku, np. 4 spacji i ich wielokrotności. Wcięcia odpowiadają nawiasom

w innych językach, służą grupowaniu instrukcji i wydzielaniu bloków kodu. Błędy wcięć zgłaszane są jako wyjątki `IndentationError`.

zmienna nazwa określająca jakąś zapamiętywaną i wykorzystywaną w programie wartość lub strukturę danych. Zmienna może przechowywać pojedyncze wartości określonego typu, np.: `imie = Anna"`, jak i rozbudowane struktury danych, np.: `imiona = ('Ala', 'Ola', 'Ela')`. W nazwach zmiennych nie używamy znaków narodowych, nie rozpoczynamy ich od cyfr.

typy danych Wszystkie dane w Pythonie są obiektami i jako takie przynależą do określonego typu, który determinuje możliwe na nich operacje. W pewnym uproszczeniu podstawowe typy danych to: *string* – napis (łańcuch znaków), podtyp sekwencji; *integer* – dodatnie i ujemne liczby całkowite; *float* – liczba zmiennoprzecinkowa (separatorem jest kropka); *boolean* – wartość logiczna True (prawda, 1) lub False (fałsz, 0), podtyp typu całkowitego.

operatory **Arytmetyczne**: +, -, *, /, //, %, ** (potęgowanie); znak + znak (konkatenacja napisów); znak * 10 (powielenie znaków); **Przypisania**: =, +=, -=, *=, /=, %=, **=, //=; **Logiczne**: and, or, not; Fałszem logicznym są: liczby zero (0, 0.0), False, None (null), puste kolekcje ([], (), {}, set()), puste napisy. Wszystko inne jest prawdą logiczną. **Zawierania**: in, not in; **Porównania**: ==, >, <, <>, <=, >= != (jest różne).

Operator * rozpakowuję listę paramterów przekazaną funkcji. Operator ** rozpakuje słownik.

lista jedna z podstawowych struktur danych, indeksowana sekwencja takich samych lub różnych elementów, które można zmieniać. Przypomina tabele z innych języków programowania. Np. `imiona = ['Ala', 'Ola', 'Ela']`. Deklaracja pustej listy: `lista = []`.

tupla podobnie jak lista, zawiera indeksowaną sekwencję takich samych lub różnych elementów, ale nie można ich zmieniać. Często służy do przechowywania lub przekazywania ustawień, stałych wartości itp. Np. `imiona = ('Ala', 'Ola', 'Ela')`. 1-elementową tuplę należy zapisywać z dodatkowym przecinkiem: `tupla1 = (1,)`.

zbiór nieuporządkowany, nieindeksowany zestaw elementów tego samego lub różnych typów, nie może zawierać duplikatów, obsługuje charakterystyczne dla zbiorów operacje: sumę, iloczyn oraz różnicę. Np. `imiona = set(['Ala', 'Ola', 'Ela'])`. Deklaracja pustego zbioru: `zbior = set()`.

słownik typ mapowania, zestaw par elementów w postaci "klucz: wartość". Kluczami mogą być liczby, ciągi znaków czy tuple. Wartości mogą być tego samego lub różnych typów. Np. `osoby = {'Ala': 'Lipiec' , 'Ola': 'Maj', 'Ela': 'Styczeń'}`. Dane ze słownika łatwo wydobyć: `slownik['klucz']`, lub zmienić: `slownik['klucz'] = wartosc`. Deklaracja pustego słownika: `slownik = dict()`.

notacja wycinkowa (ang. *slice notation*) pojedyncze elementy wszystkich sekwencji takich jak napisy, listy, tuple są indeksowane zaczynając od 0, odczytujemy je za pomocą indeksu, np.: `napis[0]`; możliwe jest również odczytanie kilku elementów sekwencji naraz, w najprostszej postacji trzeba określić indeks pierwszego i ostatniego (niewliczanego) elementu, np. `napis[1:5]`.

instrukcja warunkowa podstawowa konstrukcja w programowaniu, wykorzystuje wyrażenie logiczne przyjmujące wartość True (prawda) lub False (fałsz) do wyboru odpowiedniego działania. Umożliwia rozgałezianie kodu. Np.:

```
if wiek < 18:
    print "Treść zabroniona"
else:
    print "Zapraszamy"
```

pętla podstawowa konstrukcja w programowaniu, umożliwia powtarzanie fragmentów kodu zadaną ilość razy (pętla `for`) lub dopóki podane wyrażenie logiczne jest prawdziwe (pętla `while`). Należy zadbać, aby pętla była skończona za pomocą odpowiedniego warunku lub instrukcji przeywającej powtarzanie. Np.:

```
for i in range(11):
    print i
```

zmienna iteracyjna zmienna występująca w pętli, której wartość zmienia się, najczęściej jest zwiększana (inkremntacja) o 1, w każdym wykonaniu pętli. Może pełnić rolę "licznika" powtórzeń lub być elementem wyrażenia logicznego wyznaczającego koniec działania pętli.

iteratory (ang. *iterators*) – obiekt reprezentujący sekwencję danych, zwracający z niej po jednym elemencie na raz przy użyciu metody `next()`; jeżeli nie ma następnego elementu, zwracany jest wyjątek `StopIteration`. Funkcja `iter()` potrafi zwrócić iterator z podanego obiektu.

generatory wyrażeń (ang. *generator expressions*) – zwięzły w notacji sposób tworzenia iteratorów według składni: `( wyrażenie for wyraz in sekwencja if warunek )`

wyrażenie listowe (ang. *list comprehensions*) – efektywny sposób tworzenia list na podstawie elementów dowolnych sekwencji, na których wykonywane są te same operacje i które opcjonalnie spełniają określone warunki. Składnia: `[ wyrażenie for wyraz in sekwencja if warunek ]`

mapowanie funkcji w kontekście funkcji `map()` oznacza zastosowanie danej funkcji do wszystkich dostarczonych wartości

wyrażenia lambda zwane czasem *funkcjami lambda*, mechanizm pozwalający zwięźle zapisywać proste funkcje w postaci pojedynczych wyrażeń

filtrowanie danych selekcja danych na podstawie jakichś kryteriów

wyjątki to komunikaty zgłaszane przez interpreter Pythona, pozwalające ustalić przyczyny błędnego działania kodu.

funkcja blok często wykonywanego kodu wydzielony słowem kluczowym `def`, opatrzony unikalną w danym zasięgu nazwą; może przyjmować dane i zwracać wartości za pomocą słowa kluczowego `return`.

moduł plik zawierający wiele zazwyczaj często używanych w wielu programach funkcji lub klas; zanim skorzystamy z zawartych w nim fragmentów kodu, trzeba je lub cały moduł zaimportować za pomocą słowa kluczowego `import`.

serializacja proces przekształcania obiektów w strumień znaków lub bajtów, który można zapisać w pliku (bazie) lub przekazać do innego programu.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Materiały

1. Python (dokumentacja)
2. `Python 2 - przewodnik (pl)`

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.2 Matplotlib

Jedną z potężniejszych bibliotek Pythona jest matplotlib, która służy do tworzenia różnego rodzaju wykresów. *Pylab* to *API* ułatwiające korzystanie z omawianej biblioteki na wzór środowiska Matlab. Poniżej pokazujemy, jak łatwo przy użyciu Pythona wizualizować wykresy różnych funkcji.

Zobacz, jak zainstalować matplotlib w systemie *Linux* lub *Windows*.

Informacja: W systemach **Linux** *matplotlib* wymaga pakietu `python-tk` (systemy oparte na Debianie) lub `tk` (systemy oparte na Arch Linux).

Informacja: Bibliotekę *matplotlib* można importować na kilka sposobów. Najprostszym jest użycie instrukcji `import pylab`, która udostępnia moduł *pyplot* (do tworzenia wykresów) oraz bibliotekę *numpy* (funkcje matematyczne) w jednej przestrzeni nazw. Tak będziemy robić w konsoli i początkowych przykładach.

Oficjalna dokumentacja sugeruje jednak, aby w bardziej złożonych projektach stosować jawne importy podane niżej. Tak zrobimy w przykładach korzystających z funkcji matematycznych.

```
import numpy as np
import matplotlib.pyplot as plt
```

Wskazówka: Jeżeli konsolę rozszerzoną uruchomimy poleceniem `ipython --pylab`, nie trzeba będzie podawać przedrostka `pylab` przy korzystaniu z funkcji rysowania.

Funkcja liniowa

Zabawę zacznijmy w konsoli Pythona:

```
import pylab
x = [1,2,3]
y = [4,6,5]
pylab.plot(x,y)
pylab.show()
```

Tworzenie wykresów jest proste. Musimy mieć zbiór wartości x i odpowiadający im zbiór wartości y. Obie listy przekazujemy jako argumenty funkcji `plot()`, a następnie rysujemy funkcją `show()`.

Spróbujmy zrealizować bardziej złożone zadanie.

ZADANIE: wykonaj wykres funkcji $f(x) = a*x + b$, gdzie $x = <-10;10>$ z krokiem 1, $a = 1$, $b = 2$.

W pliku `pylab01.py` umieszczamy poniższy kod:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import pylab

a = 1
b = 2
x = range(-10, 11)  # lista argumentów x

y = []  # lista wartości
for i in x:
```

```
    y.append(a * i + b)

pylab.plot(x, y)
pylab.title('Wykres f(x) = a*x - b')
pylab.grid(True)
pylab.show()
```

Na początku dla ułatwienia importujemy interfejs `pylab`. Następnie postępujemy wg omówionego schematu: zdefiniuj dziedzinę argumentów funkcji, a następnie zbiór wyliczonych wartości. W powyższym przypadku generujemy listę wartości *x* za pomocą funkcji `range()` – co warto przetestować w interaktywnej konsoli Pythona. Wartości *y* wyliczamy w pętli i zapisujemy w liście.

Dodatkowe metody: `title()` ustawia tytuł wykresu, `grid()` włącza wyświetlanie pomocniczej siatki. Uruchom program.

Ćwiczenie 1

Zmodyfikuj kod tak, aby współczynniki *a* i *b* mógł podawać użytkownik. Nie zapomnij przekonwertować danych tekstowych na liczby całkowite.

Ćwiczenie 2

W konsoli Pythona wydajemy następujące polecenia:

```
>>> a = 2
>>> x = range(11)
>>> for i in x:
...   print(a + i)
>>> y = [a + i for i in range(11)]
>>> print(y)
```

Powyższy przykład wykorzystuje tzw. *wyrażenie listowe*, które zwięźle zastępuje pętlę i zwraca listę wartości. Jego działanie należy rozumieć następująco: dla każdej wartości `i` (nazwa zmiennej dowolna) w liście `x` wylicz wyrażenie `a + i` i umieść w liście `y`.

Użyj wyrażenia listowego w naszym programie:

```
a = int(input('Podaj współczynnik a: '))
b = int(input('Podaj współczynnik b: '))
x = range(-10, 11)  # lista argumentów x

# wyrażenie listowe wylicza dziedzinę y
y = [a * i + b for i in x]  # lista wartości

```

Dwie funkcje

ZADANIE: wykonaj wykres funkcji:

- *f(x) = x/(-3) + a* dla *x* <= 0,
- *f(x) = x*x/3* dla *x* >= 0,

– gdzie *x* = <-10;10> z krokiem 0.5. Współczynnik *a* podaje użytkownik.

Wykonanie zadania wymaga umieszczenia na wykresie dwóch funkcji. Wykorzystamy funkcję `arange()`, która zwraca listę wartości zmiennoprzecinkowych (zob. *typy danych*) z zakresu określonego przez dwa pierwsze argumenty i z krokiem wyznaczonym przez argument trzeci. Drugą przydatną konstrukcją będzie wyrażenie listowe uzupełnione o instrukcję warunkową, która ogranicza wartości, dla których obliczane jest podane wyrażenie.

Ćwiczenie 3

Zanim zrealizujemy zadanie przećwiczmy w konsoli Pythona następujący kod:

```
>>> import pylab
>>> x = pylab.arange(-10, 10.5, 0.5)
>>> print(x)
>>> len(x)
>>> a = 3
>>> y1 = [i / -3 + a for i in x if i <= 0]
>>> len(y1)
```

Uwaga: nie zamykaj tej sesji konsoli, zaraz się nam jeszcze przyda.

W pliku `pylab02.py` umieszczamy poniższy kod:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

# ZADANIE: wykonaj wykres funkcji f(x), gdzie x = <-10;10> z krokiem 0.5
# f(x) = x/-3 + a dla x <= 0
# f(x) = x*x/3 dla x >= 0

import pylab

x = pylab.arange(-10, 10.5, 0.5)  # lista argumentów x
a = int(input("Podaj współczynnik a: "))
y1 = [i / -3 + a for i in x if i <= 0]

print(x, len(x))
print(y1, len(y1))

pylab.plot(x, y1)
pylab.title('Wykres f(x)')
pylab.grid(True)
pylab.show()
```

Uruchom program. Nie działa, dostajemy komunikat: *ValueError: x and y must have same first dimension*, czyli listy wartości *x* i *y1* nie zawierają tyle samo elementów.

Co należy z tym zrobić? Jak wynika z warunków zadania, wartości *y1* obliczane są tylko dla argumentów mniejszych od zera. Zatem trzeba ograniczyć listę *x*, tak aby zawierała tylko wartości z odpowiedniego przedziału. Wróćmy do konsoli Pythona:

Ćwiczenie 4

```
>>> x
>>> x[0]
>>> x[0:5]
```

```
>>> x[:5]
>>> x[:len(y1)]
>>> len(x[:len(y1)])
```

Uwaga: nie zamykaj tej sesji konsoli, zaraz się nam jeszcze przyda.

Z pomocą przychodzi nam wydobywanie z listy wartości wskazywanych przez indeksy liczone od 0. Jednak prawdziwym ułatwieniem jest **notacja wycinania** (ang. *slice*), która pozwala podać pierwszy i ostatni indeks interesującego nas zakresu. Zmieniamy więc wywołanie funkcji `plot()`:

```
pylab.plot(x[:len(y1)], y1)
```

Uruchom i przetestuj działanie programu.

Udało się nam zrealizować pierwszą część zadania. Spróbujmy zakodować część drugą. Dopisujemy:

```
14 y2 = [i**2 / 3 for i in x if i >= 0]
15
16 pylab.plot(x[:len(y1)], y1, x, y2)
```

Wyrażenie listowe wylicza nam drugą dziedzinę wartości. Następnie do argumentów funkcji `plot()` dodajemy drugą parę list. Spróbuj uruchomić program. Nie działa, znowu dostajemy komunikat: *ValueError: x and y must have same first dimension*. Teraz jednak wiemy już dlaczego...

Ćwiczenie 5

Przetestujmy kod w konsoli Pythona:

```
>>> len(x)
>>> x[-10]
>>> x[-10:]
>>> len(y2)
>>> x[-len(y2):]
```

Jak widać, w **notacji wycinania** możemy używać indeksów ujemnych wskazujących elementy od końca listy. Jeżeli taki indeks umieścimy jako pierwszy przed dwukropkiem, czyli separatorem przedziału, dostaniemy resztę elementów listy.

Na koniec musimy więc zmodyfikować funkcję `plot()`:

```
pylab.plot(x[:len(y1)], y1, x[-len(y2):], y2)
```

Ćwiczenie 6

Spróbuj dziedziny wartości *x* dla funkcji *y1* i *y2* wyznaczyć nie za pomocą notacji wycinkowej, ale przy użyciu wyrażeń listowych, których wynik przypisz do zmiennych *x1* i *x2*. Użyj ich jako argumentów funkcji `plot()` i przetestuj program.

Ruchy Browna

Napiszemy program, który symuluje ruchy Browna. Jak wiadomo są to chaotyczne ruchy cząsteczek, które będziemy mogli zwizualizować w płaszczyźnie dwuwymiarowej. Na początku przyjmujemy następujące założenia:

- cząsteczka, której ruch będziemy śledzić, znajduje się w początku układu współrzędnych (0, 0);

- w każdym ruchu cząsteczka przemieszcza się o stały wektor o wartości 1;
- kierunek ruchu wyznaczać będziemy losując kąt z zakresu <0; 2Pi>;
- współrzędne kolejnego położenia cząsteczki wyliczać będziemy ze wzorów:

$$x_n = x_{n-1} + r * cos(\phi)$$
$$y_n = y_{n-1} + r * sin(\phi)$$

– gdzie: *r* – długość jednego kroku, ϕ – kąt wskazujący kierunek ruchu w odniesieniu do osi *OX*.

- końcowy wektor przesunięcia obliczymy ze wzoru: $|s| = \sqrt{(x^2 + y^2)}$

Zacznijmy od wyliczenia współrzędnych opisujących ruch cząsteczki. Do pustego pliku o nazwie `rbrowna.py` wpisujemy:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import random

n = int(input("Ile ruchów? "))
x = y = 0

for i in range(0, n):
    # wylosuj kąt i zamień go na radiany
    rad = float(random.randint(0, 360)) * np.pi / 180
    x = x + np.cos(rad)  # wylicz współrzędną x
    y = y + np.sin(rad)  # wylicz współrzędną y
    print(x, y)

# oblicz wektor końcowego przesunięcia
s = np.sqrt(x**2 + y**2)
print("Wektor przesunięcia:", s)
```

Funkcje trygonometryczne zawarte w module `math` wymagają kąta podanego w radianach, dlatego wylosowany kąt po zamianie na liczbę zmiennoprzecinkową mnożymy przez wyrażenie `math.pi / 180`. Uruchom i przetestuj kod.

Ćwiczenie 6

Do przygotowania wykresu ilustrującego ruch cząsteczki generowane współrzędne musimy zapisać w listach. Wstaw w odpowiednich miejscach pliku poniższe instrukcje:

```
lx = [0]
ly = [0]

lx.append(x)
ly.append(y)
```

Na końcu skryptu dopisz instrukcje wyliczającą końcowy wektor przesunięcia ($|s| = \sqrt{(x^2 + y^2)}$) i drukującą go na ekranie. Przetestuj program.

Pozostaje dopisanie importu biblioteki *matplotlib* oraz instrukcji generujących wykres. Poniższy kod ilustruje również użycie opcji wzbogacających wykres o legendę, etykiety czy tytuł.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import numpy as np
import random
import matplotlib.pyplot as plt

n = int(input("Ile ruchów? "))
x = y = 0
lx = [0]
ly = [0]

for i in range(0, n):
    # wylosuj kąt i zamień go na radiany
    rad = float(random.randint(0, 360)) * np.pi / 180
    x = x + np.cos(rad)  # wylicz współrzędną x
    y = y + np.sin(rad)  # wylicz współrzędną y
    # print(x, y)
    lx.append(x)
    ly.append(y)

print(lx, ly)

# oblicz wektor końcowego przesunięcia
s = np.fabs(np.sqrt(x**2 + y**2))
print("Wektor przesunięcia:", s)

plt.plot(lx, ly, "o:", color="green", linewidth=2, alpha=0.5)
plt.legend(["Dane x, y\nPrzemieszczenie: " + str(s)], loc="upper left")
plt.xlabel("lx")
plt.ylabel("ly")
plt.title("Ruchy Browna")
plt.grid(True)
plt.show()
```

Warto zwrócić uwagę na dodatkowe opcje formatujące wykres w poleceniu `p.plot(lx, ly, ó:", color="green", linewidth=2, alpha=0.5)`. Trzeci parametr określa styl linii, możesz sprawdzić inne wartości, np: `r:.`, `r:+`, `r.`, `r+`. Można też określać kolor (`color`), grubość linii (`linewidth`) i przezroczystość (`alpha`). Poeksperymentuj.

Ćwiczenie 7

Spróbuj uzupełnić kod tak, aby na wykresie zaznaczyć prostą linią w kolorze niebieskim wektor przesunięcia. Efekt końcowy może wyglądać następująco:

Zadania dodatkowe

Przygotuj wykres funkcji kwadratowej: *f(x) = a*x^2 + b*x + c*, gdzie *x* = <-10;10> z krokiem 1, przyjmij następujące wartości współczynników: *a = 1, b = -3, c = 1*.

Uzyskany wykres powinien wyglądać następująco:

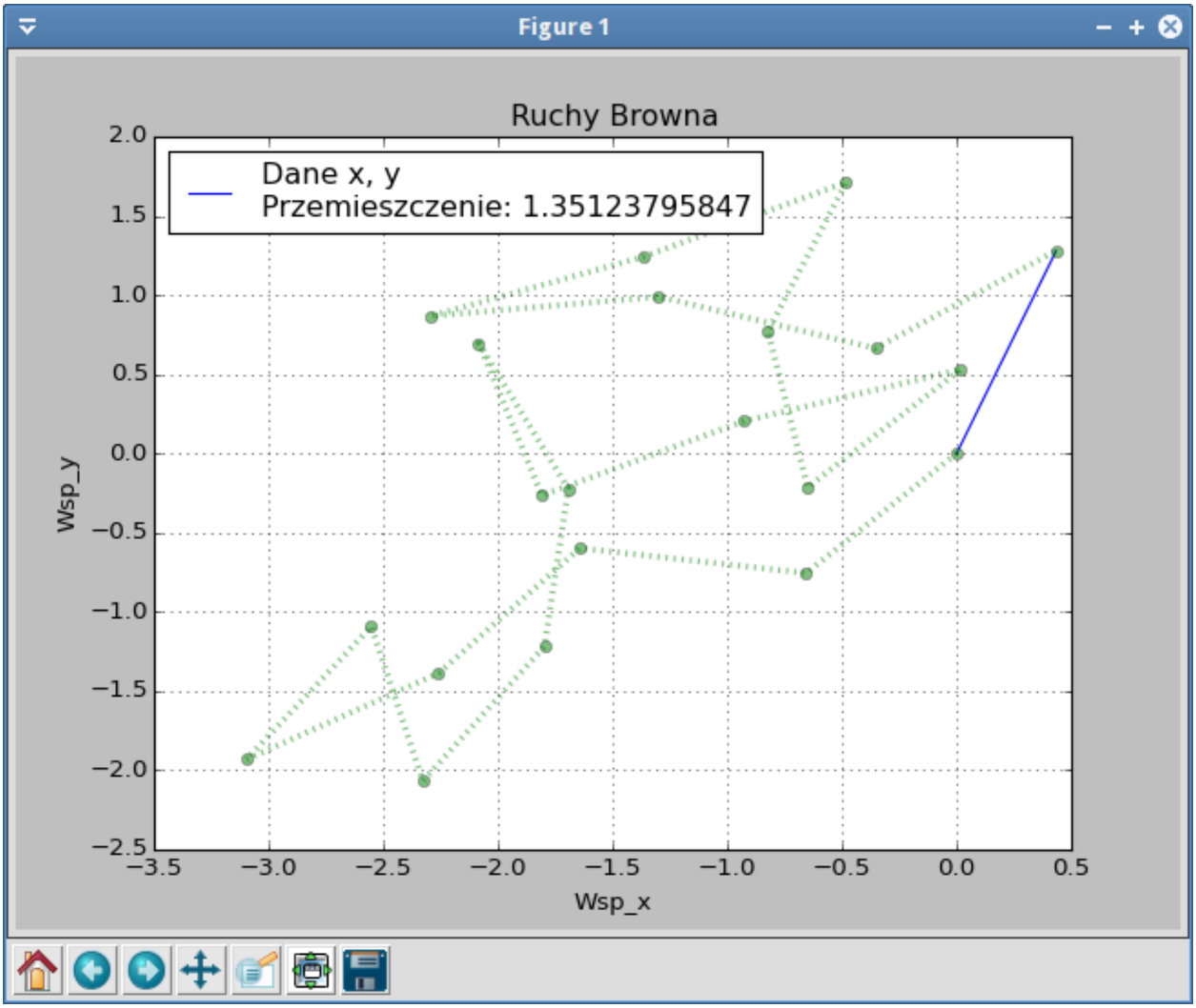


Figure 1
Ruchy Browna
Dane x, y
Przemieszczenie: 1.35123795847
Wsp_y
Wsp_x
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−3.5
−3.0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5

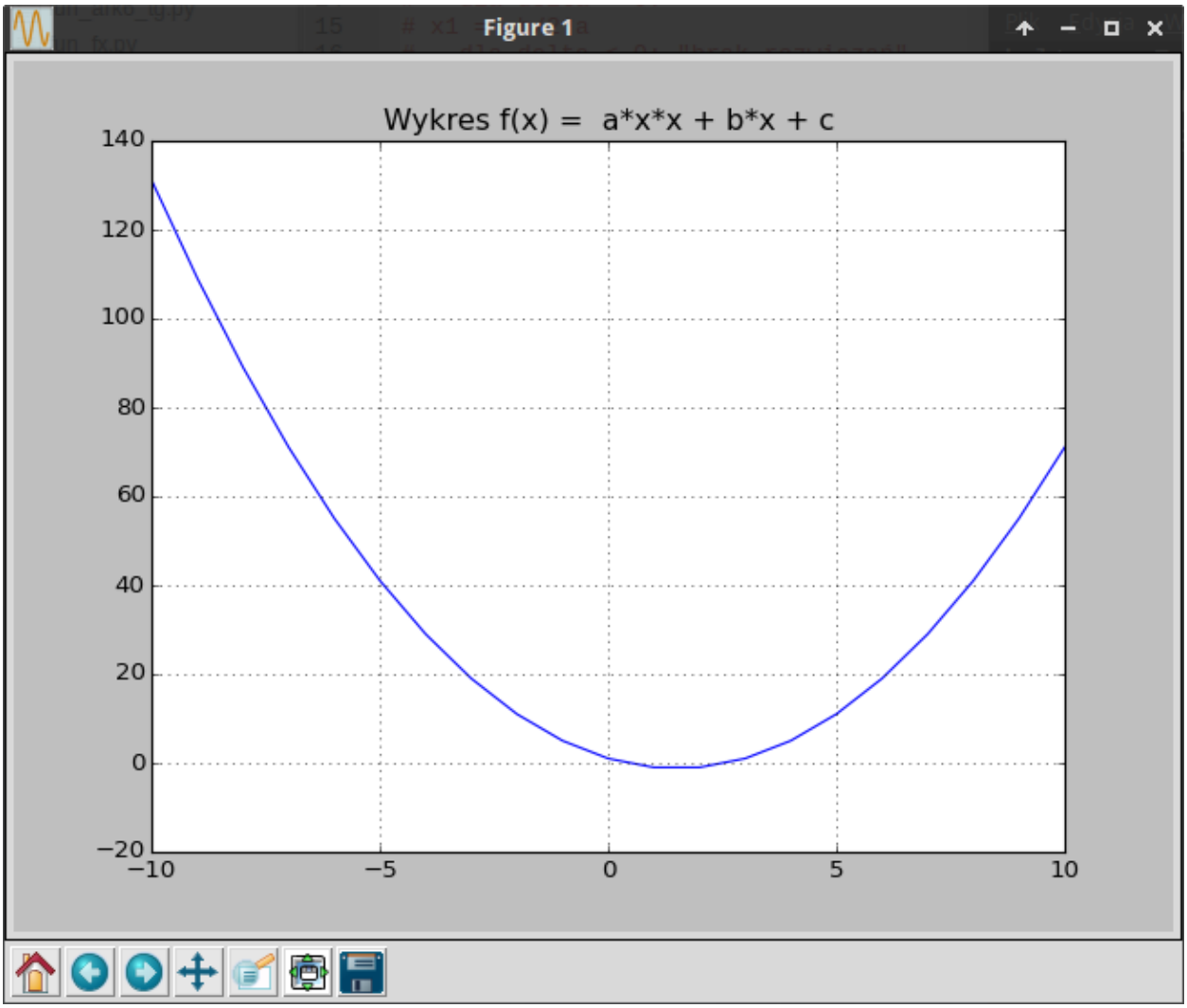


Figure 1
Wykres f(x) = a*x*x + b*x + c
140
120
100
80
60
40
20
0
−20
−10
−5
0
5
10

Źródła

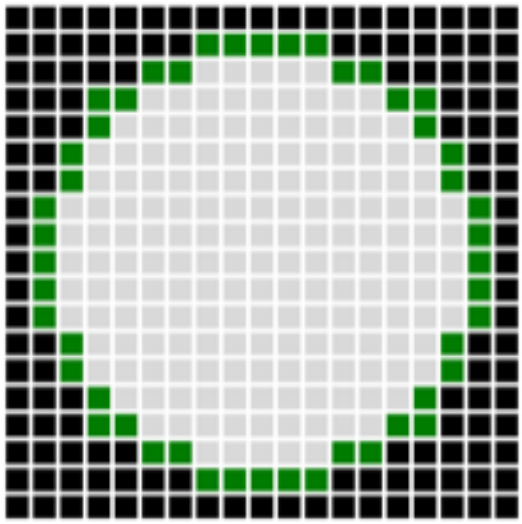
- `pylab.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

2.4.3 Gra robotów

RobotGame to gra, w której walczą ze sobą programy – roboty na planszy o wymiarach 19x19 pól. Celem gry jest umieszczenie na niej jak największej ilości robotów w ciągu 100 rund rozgrywki.



Czarne pola (ang. *obstacle*) wyznaczają granicę areny walk, zielone pola (ang. *spawn points*) to punkty wejścia, w których co **10** rund pojawia się po **5** robotów, każdy z 50 punktami HP (ang. *health points*) na starcie.

W każdej rundzie każdy robot musi wybrać jedno z następujących działań:

- **Ruch** (ang. *move*) na przyległe pole w pionie (góra, dół) lub poziomie (lewo, prawo). W przypadku, kiedy w polu docelowym znajduje się lub znajdzie się inny robot następuje *kolizja* i utrata po 5 punktów HP.
- **Atak** (ang. *attack*) na przyległe pole, wrogi robot na tym polu traci 8-10 punktów HP.
- **Samobójstwo** (ang. *suicide*) – robot ginie pod koniec rundy zabierając wszystkim wrogim robotom obok po 15 punktów HP.
- **Obrona** (ang. *guard*) – robot pozostaje w miejscu, tracąc połowę punktów HP w wyniku ataku lub samobójstwa.

W grze nie można uszkodzić własnych robotów.

Sztuczna inteligencja

Zadaniem gracza jest stworzenie sztucznej inteligencji robota, która pozwoli mu w określonych sytuacjach na arenie wybrać odpowiednie działanie. Trzeba więc: określić daną sytuację, ustalić działanie robota, zakodować je i przetestować, np.:

1. Gdzie ma iść robot po po wejściu na arenę?
2. Działanie: "Idź do środka".
3. Jaki kod umożliwi robotowi realizowanie tej reguły?

4. Czy to działa?

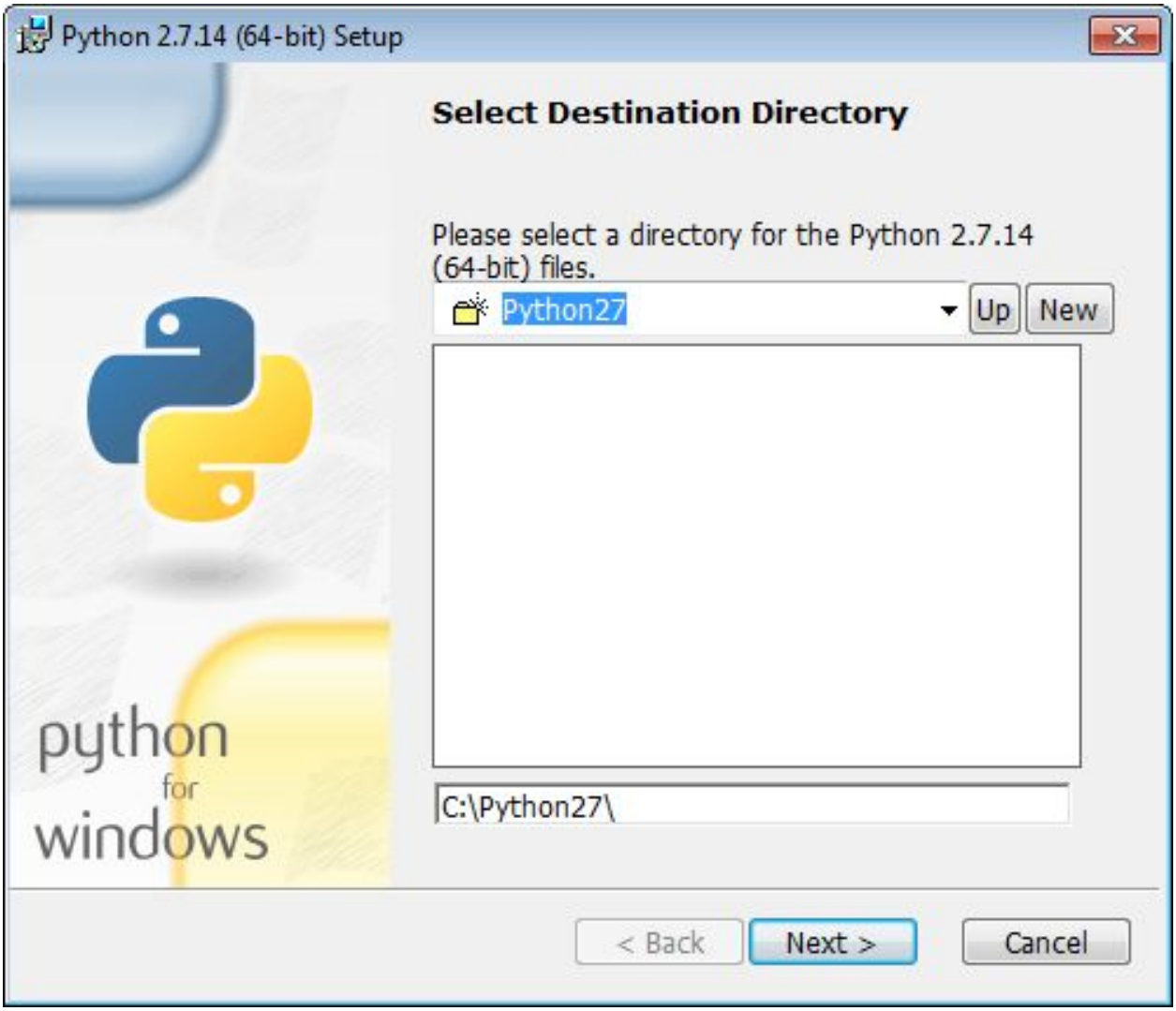
Aby ułatwić budowanie robota, przedstawiamy kilka przykładowych reguł i "klocków", z których można zacząć składać swojego robota. Pokazujemy również, jak testować swoje roboty. Nie podajemy jednak "przepisu" na robota najlepszego. Do tego musisz dojść sam.

Środowisko testowe

Do budowania i testowania robotów używamy pakietu *rgkit*. Działa on pod Pythonem 2 i 3, ale symulator, który jest nieocenionym narzędziem testowania robotów, działa tylko w Pythonie 2. W Linuksie Pythona 2 trzeba doinstalować:

```
~$ sudo apt install python2-minimal
```

W MS Windows na stronie Python Releases klikamy link *Latest Python2 Release* i pobieramy instalator *Windows x86-64 MSI installer* (wersja 64-bitowa). Podczas instalacji zaznaczamy opcję "Add python.exe to path".



Środowisko deweloperskie przygotujemy w katalogu `robot`.


Python 2.7.14 (64-bit) Setup
Customize Python 2.7.14 (64-bit)
Select the way you want features to be installed.
Click on the icons in the tree below to change the
way features will be installed.
Register Extensions
Tcl/Tk
Documentation
Utility Scripts
pip
Test suite
Add python.exe to Path
Prepend C:\Python27\ to the system Path
variable. This allows you to type 'python' into a
command prompt without needing the full path.
This feature requires 0KB on your hard drive.
python
for
windows
Disk Usage
Advanced
< Back
Next >
Cancel

Informacja: W polecanych przez nas dystrybucjach *Linux Live* środowisko testowe jest już przygotowane.

W terminalu wydajemy polecenia:

```
~$ mkdir robot; cd robot
~robot$ virtualenv -p python2.7 env
~robot$ source env/bin/activate
(env):~/robot$ pip install git+https://github.com/outkine/rgkit.git
```

Informacja: W systemie Windows:

- po instalacji Pythona 2, trzeba doinstalować narzędzie do tworzenia wirtualnego środowiska poleceniem w terminalu: `pip2 install virtualenv`,
- polecenie aktywujące środowisko wirtualne będzie miało postać `env\Scripts\activate.bat`.

Dodatkowo instalujemy pakiet zawierający roboty open source, następnie symulator ułatwiający testowanie, a na koniec tworzymy skrót do jego uruchamiania:

```
(env):~/robot$ git clone https://github.com/mpeterv/robotgame-bots bots
(env):~/robot$ git clone https://github.com/mpeterv/rgsimulator.git
(env):~/robot$ ln -s rgsimulator/rgsimulator.py symuluj
```

Po wykonaniu wszystkich powyższych poleceń i komendy `ls -l` powinniśmy zobaczyć:

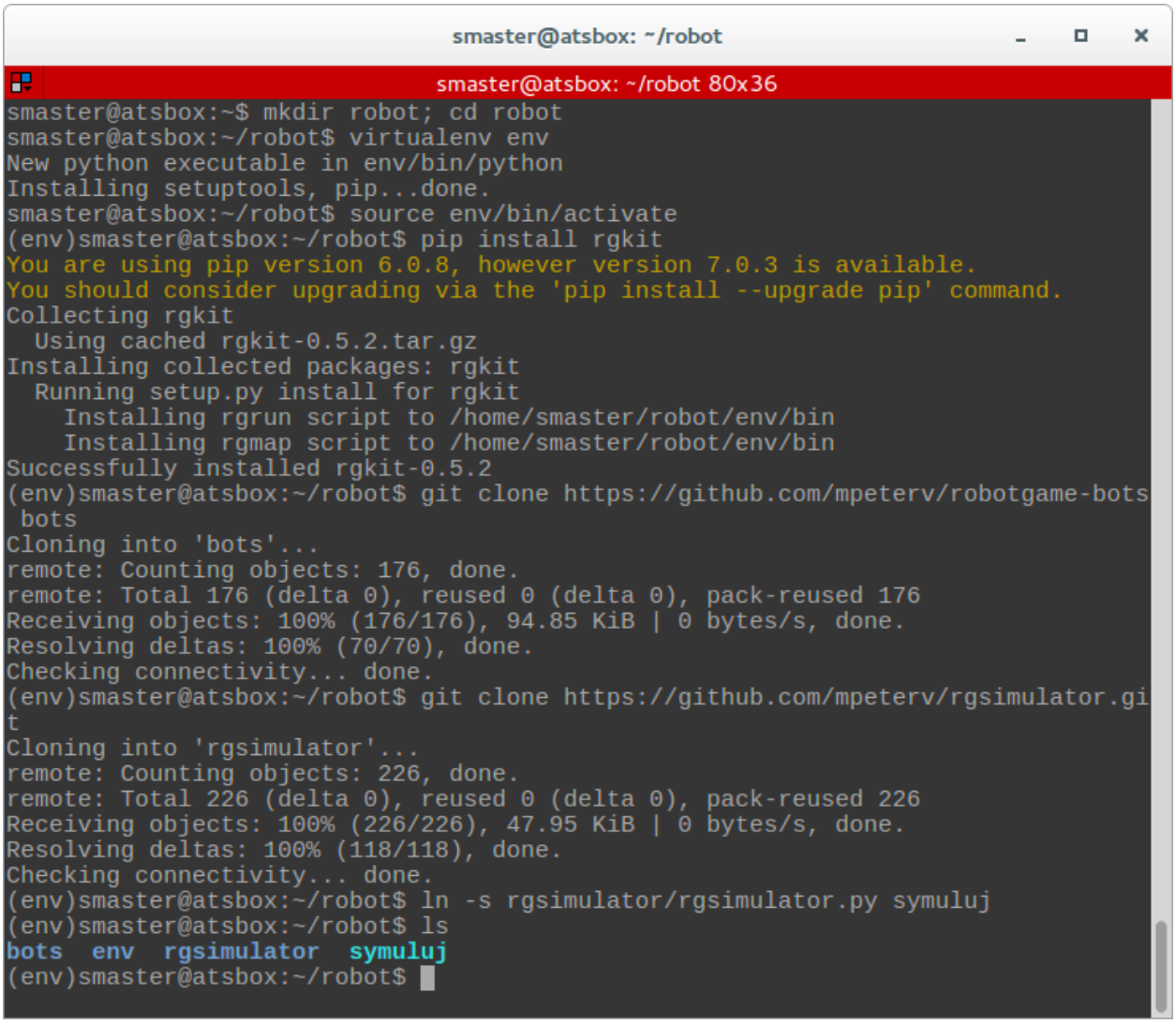
Kolejne wersje robota proponujemy zapisywać w plikach *robot01.py*, *robot02.py* itd. Będziemy mogli je uruchamiać lub testować za pomocą poleceń:

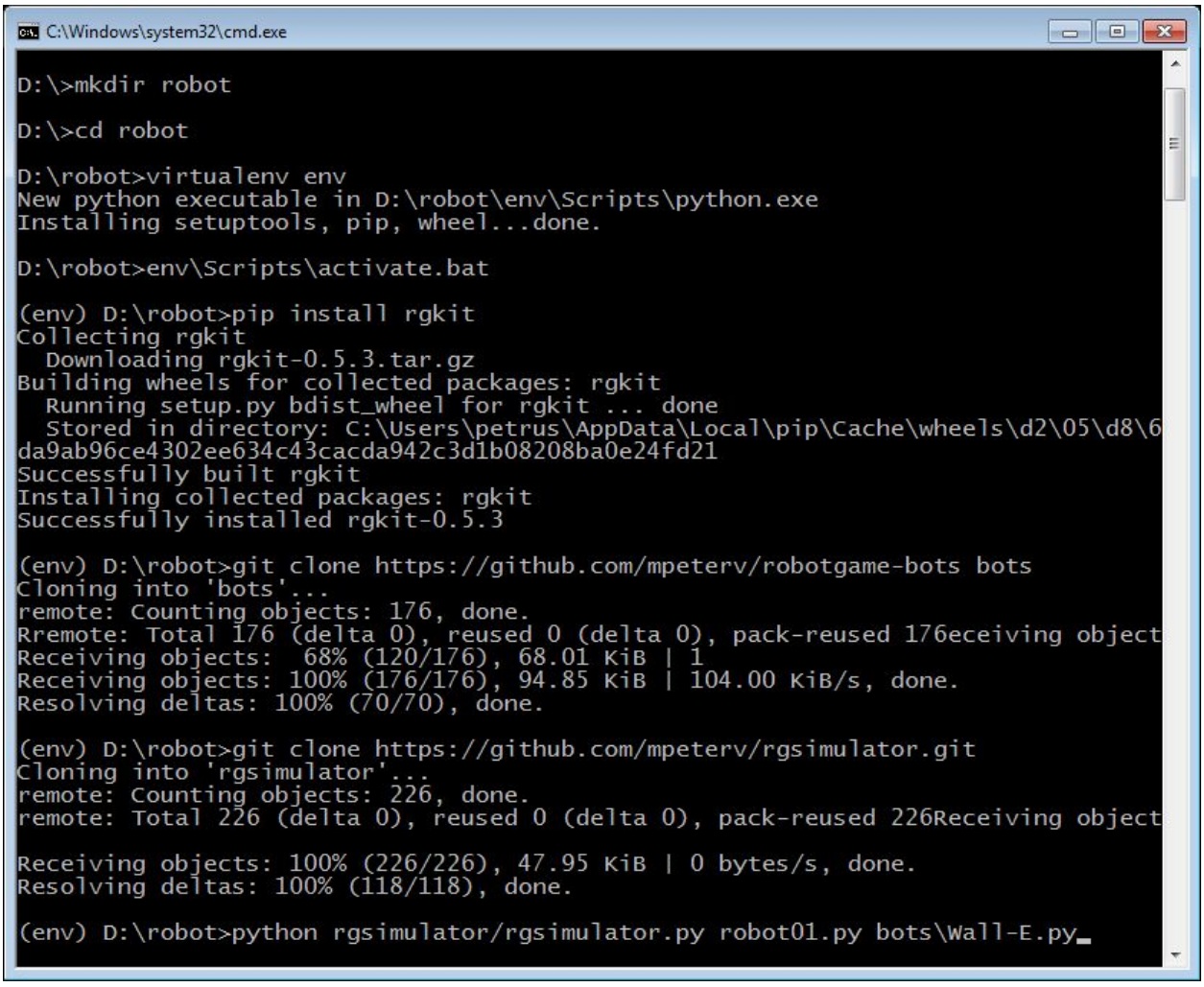
```
(env)~/robot$ rgrun robot01.py robot02.py
(env)~/robot$ rgrun bots/stupid26.py robot01.py
(env)~/robot$ python ./symuluj robot01.py
(env)~/robot$ python ./symuluj robot01.py robot02.py
```

Obsługa symulatora

- Klawisz **F**: utworzenie robota-przyjaciela w zaznaczonym polu.
- Klawisz **E**: utworzenie robota-wroga w zaznaczonym polu.
- Klawisze **Delete** or **Backspace**: usunięcie robota z zaznaczonego pola.
- Klawisz **H**: zmiana punktów HP robota.
- Klawisz **C**: wyczyszczenie planszy gry.
- Klawisz **Spacja**: pokazuje planowane ruchy robotów.
- Klawisz **Enter**: uruchomienie rundy.
- Klawisz **G**: tworzy i usuwa roboty w punktach wejścia (ang. *spawn locations*), "generowanie robotów".

Uwaga: Opisana instalacja zakłada użycie środowiska wirtualnego, które przed uruchomieniem rozgrywki lub symulacji trzeba aktywować w katalogu `robot` poleceniem `source env/bin/activate` (Linux) lub `env\\Scripts\\activate.bat` (Windows).


smaster@atsbox: ~/robot
smaster@atsbox: ~/robot 80x36
smaster@atsbox:~$ mkdir robot; cd robot
smaster@atsbox:~/robot$ virtualenv env
New python executable in env/bin/python
Installing setuptools, pip...done.
smaster@atsbox:~/robot$ source env/bin/activate
(env)smaster@atsbox:~/robot$ pip install rgkit
You are using pip version 6.0.8, however version 7.0.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
Collecting rgkit
Using cached rgkit-0.5.2.tar.gz
Installing collected packages: rgkit
Running setup.py install for rgkit
Installing rgrun script to /home/smaster/robot/env/bin
Installing rgmap script to /home/smaster/robot/env/bin
Successfully installed rgkit-0.5.2
(env)smaster@atsbox:~/robot$ git clone https://github.com/mpeterv/robotgame-bots
bots
Cloning into 'bots'...
remote: Counting objects: 176, done.
remote: Total 176 (delta 0), reused 0 (delta 0), pack-reused 176
Receiving objects: 100% (176/176), 94.85 KiB | 0 bytes/s, done.
Resolving deltas: 100% (70/70), done.
Checking connectivity... done.
(env)smaster@atsbox:~/robot$ git clone https://github.com/mpeterv/rgsimulator.gi
t
Cloning into 'rgsimulator'...
remote: Counting objects: 226, done.
remote: Total 226 (delta 0), reused 0 (delta 0), pack-reused 226
Receiving objects: 100% (226/226), 47.95 KiB | 0 bytes/s, done.
Resolving deltas: 100% (118/118), done.
Checking connectivity... done.
(env)smaster@atsbox:~/robot$ ln -s rgsimulator/rgsimulator.py symuluj
(env)smaster@atsbox:~/robot$ ls
bots env rgsimulator symuluj
(env)smaster@atsbox:~/robot$


C:\Windows\system32\cmd.exe
D:\>mkdir robot
D:\>cd robot
D:\robot>virtualenv env
New python executable in D:\robot\env\Scripts\python.exe
Installing setuptools, pip, wheel...done.
D:\robot>env\Scripts\activate.bat
(env) D:\robot>pip install rgkit
Collecting rgkit
Downloading rgkit-0.5.3.tar.gz
Building wheels for collected packages: rgkit
Running setup.py bdist_wheel for rgkit ... done
Stored in directory: C:\Users\petrus\AppData\Local\pip\Cache\wheels\d2\05\d8\6
da9ab96ce4302ee634c43cacda942c3d1b08208ba0e24fd21
Successfully built rgkit
Installing collected packages: rgkit
Successfully installed rgkit-0.5.3
(env) D:\robot>git clone https://github.com/mpeterv/robotgame-bots bots
Cloning into 'bots'...
remote: Counting objects: 176, done.
Rremote: Total 176 (delta 0), reused 0 (delta 0), pack-reused 176eceiving object
Receiving objects: 68% (120/176), 68.01 KiB | 1
Receiving objects: 100% (176/176), 94.85 KiB | 104.00 KiB/s, done.
Resolving deltas: 100% (70/70), done.
(env) D:\robot>git clone https://github.com/mpeterv/rgsimulator.git
Cloning into 'rgsimulator'...
remote: Counting objects: 226, done.
remote: Total 226 (delta 0), reused 0 (delta 0), pack-reused 226Receiving object
Receiving objects: 100% (226/226), 47.95 KiB | 0 bytes/s, done.
Resolving deltas: 100% (118/118), done.
(env) D:\robot>python rgsimulator/rgsimulator.py robot01.py bots\Wall-E.py

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

RG – klocki 1

Wskazówka:

- Każdy "klocek" można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!

Idź do środka

To będzie nasza domyślna reguła. Umieszczamy ją w pliku `robot01.py` zawierającym niezbędne minimum działającego bota:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg


class Robot:

    def act(self, game):

        # idź do środka planszy, ruch domyślny
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Metody i właściwości biblioteki *rg*:

- `rg.toward(poz_wyj, poz_cel)` – zwraca następne położenie na drodze z bieżącego miejsca do podanego.
- `self.location` – pozycja robota, który podejmuje działanie (`self`).
- `rg.CENTER_POINT` – środek areny.

W środku broń się lub giń

Co powinien robić robot, kiedy dojdzie do środka? Może się bronić lub popełnić samobójstwo:

```
# jeżeli jesteś w środku, broń się
if self.location == rg.CENTER_POINT:
    return ['guard']

# LUB

# jeżeli jesteś w środku, popełnij samobójstwo
if self.location == rg.CENTER_POINT:
    return ['suicide']
```

Atakuj wrogów obok

Wersja wykorzystująca pętlę.

```
# jeżeli obok są przeciwnicy, atakuj
# wersja z pętlą przeglądającą wszystkie pola zajęte przez roboty
for poz, robot in game.robots.iteritems():
    if robot.player_id != self.player_id:
        if rg.dist(poz, self.location) <= 1:
            return ['attack', poz]
```

Metody i właściwości biblioteki *rg*:

- Słownik `game.robots` zawiera dane wszystkich robotów na planszy. Metoda `.iteritems()` zwraca indeks `poz`, czyli położenie *(x,y)* robota, oraz słownik `robot` opisujący jego właściwości, czyli:
  - *player_id* – identyfikator gracza, do którego należy robot;
  - *hp* – ilość punktów HP robota;
  - *location* – tupla (x, y) oznaczająca położenie robota na planszy;
  - *robot_id* – identyfikator robota w Twojej drużynie.
- `rg.dist(poz1, poz1)` – zwraca matematyczną odległość między dwoma położeniami.

Robot podstawowy

Łącząc omówione wyżej trzy podstawowe reguły, otrzymujemy robota podstawowego:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg


class Robot:

    def act(self, game):
        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            return ['guard']

        # jeżeli wokół są przeciwnicy, atakuj
        for poz, robot in game.robots.iteritems():
            if robot.player_id != self.player_id:
                if rg.dist(poz, self.location) <= 1:
                    return ['attack', poz]

        # idź do środka planszy
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Wybrane działanie robota zwracamy za pomocą instrukcji `return`. Zwróć uwagę, jak ważna jest w tej wersji kodu **kolejność umieszczenia reguł**, pierwszy spełniony warunek powoduje wyjście z funkcji, więc pozostałe możliwości nie są już sprawdzane!

Powyższy kod można przekształcić wykorzystując zmienną pomocniczą `ruch`, inicjowaną działaniem domyślnym, które może zostać zmienione przez kolejne reguły. Dopiero na końcu zwracamy ustaloną akcję:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg


class Robot:

    def act(self, game):
        # działanie domyślne:
        ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]

        if self.location == rg.CENTER_POINT:
            ruch = ['guard']

        for poz, robot in game.robots.iteritems():
            if robot.player_id != self.player_id:
                if rg.dist(poz, self.location) <= 1:
                    ruch = ['attack', poz]

        return ruch
```

Ćwiczenie 1

Przetestuj działanie robota podstawowego wystawiając go do gry z samym sobą w symulatorze. Zaobserwuj zachowanie się robotów tworząc różne układy początkowe:

```
(env)~/robot$ python ./symuluj robot04a.py robot04b.py
```

Możliwe ulepszenia

Robota podstawowego można rozbudowywać na różne sposoby przy użyciu różnych technik kodowania. Proponujemy więc *wersję **A*** opartą na funkcjach i listach oraz *wersję **B*** opartą na zbiorach. Obie wersje implementują te same reguły, jednak efekt końcowy wcale nie musi być identyczny. Przetestuj i przekonaj się sam.

Wskazówka: Przydatną rzeczą byłaby możliwość dokładniejszego śledzenia decyzji podejmowanych przez robota. Najprościej można to osiągnąć używając polecenia `print` w kluczowych miejscach algorytmu. Podany niżej **Kod nr 6** wyświetla w terminalu pozycję aktualnego i atakowanego robota. **Kod nr 7**, który nadaje się zwłaszcza do wersji robota wykorzystującej pomocniczą zmienną *ruch*, umieszczony przed instrukcją `return` pozwoli zobaczyć w terminalu kolejne ruchy naszego robota.

```
for poz, robot in game.robots.iteritems():
    if robot.player_id != self.player_id:
        if rg.dist(poz, self.location) <= 1:
            print "Atak", self.location, "=>", poz
            return ['attack', poz]
```

```
print ruch[0], self.location, "=>",
if (len(ruch) > 1):
    print ruch[1]
else:
    print
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

RG – klocki 2A

Wersja **A** oparta jest na funkcjach, czyli metodach klasy `Robot`.

Wskazówka:

- Każdy "klocek" można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!

Typy pól

Zobaczmy, w jaki sposób dowiedzieć się, w jakim miejscu się znajdujemy, gdzie wokół mamy wrogów lub pola, na które można wejść. Dysponując takimi informacjami, będziemy mogli podejmować bardziej przemyślane działania. Wykorzystamy kilka pomocniczych funkcji.

Czy to wejście?

```
# funkcja zwróci prawdę, jeżeli "poz" wskazuje punkt wejścia
def czy_wejscie(poz):
    if 'spawn' in rg.loc_types(poz):
        return True
    return False
```

Metody i właściwości biblioteki *rg*:

- `gr.loc_types(poz)` – zwraca typ pola wskazywanego przez `poz`:
  - `invalid` – poza granicami planszy(np. (-1, -5) lub (23, 66));
  - `normal` – w ramach planszy;
  - `spawn` – punkt wejścia robotów;
  - `obstacle` – pola zablokowane ograniczające arenę.

Czy obok jest wróg?

```
# funkcja zwróci prawdę, jeżeli "poz" wskazuje wroga
def czy_wrog(poz):
    if game.robots.get(poz) != None:
        if game.robots[poz].player_id != self.player_id:
            return True
    return False

# lista wrogów obok
wrogowie_obok = []
for poz in rg.locs_around(self.location):
```

```
    if czy_wrog(poz):
        wrogowie_obok.append(poz)

# warunek sprawdzający, czy obok są wrogowie
if len(wrogowie_obok):
    pass
```

W powyższym kodzie metoda `.get(poz)` pozwala pobrać dane robota, którego kluczem w słowniku jest `poz`.

Metody i właściwości biblioteki *rg*:

- `rg.locs_around(poz, filter_out=None)` – zwraca listę położeń sąsiadujących z `poz`. Jako `filter_out` można podać typy położeń do wyeliminowania, np.: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))`.

Wskazówka: Definicje funkcji i list należy wstawić na początku metody `Robot.act()` – przed pierwszym użyciem.

Wykorzystując powyższe "klocki" możemy napisać robota realizującego następujące reguły:

1. Opuść jak najszybciej wejście;
2. Atakuj wrogów obok;
3. W środku broń się;
4. W ostateczności idź do środka.

Implementacja

Przykładowa implementacja może wyglądać następująco:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

    def act(self, game):

        def czy_wejscie(poz):
            if 'spawn' in rg.loc_types(poz):
                return True
            return False

        def czy_wrog(poz):
            if game.robots.get(poz) != None:
                if game.robots[poz].player_id != self.player_id:
                    return True
            return False

        # lista wrogów obok
        wrogowie_obok = []
        for poz in rg.locs_around(self.location):
            if czy_wrog(poz):
                wrogowie_obok.append(poz)
```

```
26
27          # jeżeli jesteś w punkcie wejścia, opuść go
28          if czy_wejscie(self.location):
29              return ['move', rg.toward(self.location, rg.CENTER_POINT)]
30
31          # jeżeli obok są przeciwnicy, atakuj
32          if len(wrogowie_obok):
33              return ['attack', wrogowie_obok.pop()]
34
35          # jeżeli jesteś w środku, broń się
36          if self.location == rg.CENTER_POINT:
37              return ['guard']
38
39          # idź do środka planszy
40          return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Metoda `.pop()` zastosowana do listy zwraca jej ostatni element.

Ćwiczenie 1

Zapisz powyższą implementację w katalogu `robot` i przetestuj ją w symulatorze, a następnie wystaw ją do walki z robotem podstawowym. Poeksperymentuj z kolejnością reguł, która określa ich priorytety!

Atakuj, jeśli nie umrzesz

Warto atakować, ale nie wtedy, gdy grozi nam śmierć. Można przyjąć zasadę, że atakujemy tylko wtedy, kiedy suma potencjalnych uszkodzeń będzie mniejsza niż zdrowie naszego robota. Zmień więc dotychczasowe reguły ataku wroga korzystając z poniższych "klocków":

```
# WERSJA A
# jeżeli suma potencjalnych uszkodzeń jest mniejsza od naszego zdrowia
# funkcja zwróci prawdę
def czy_atak():
    if 9*len(wrogowie_obok) < self.hp:
        return True
    return False
```

Metody i właściwości biblioteki *rg*:

- `self.hp` – ilość punktów HP robota.

Ćwiczenie 2

Dodaj powyższą regułę do poprzedniej wersji robota.

Ruszaj się bezpiecznie

Zamiast iść na oślep lepiej wchodzić czy uciekać na bezpieczne pola. Za "bezpieczne" przyjmiemy na razie pole puste, niezablokowane i nie będące punktem wejścia.

```
# WERSJA A
# funkcja zwróci prawdę jeżeli pole poz będzie puste
def czy_puste(poz):
    if ('normal' in rg.loc_types(poz)) and not ('obstacle' in rg.loc_types(poz)):
        if game.robots.get(poz) == None:
            return True
    return False

puste = [] # lista pustych pól obok
bezpieczne = [] # lista bezpiecznych pól obok

for poz in rg.locs_around(self.location):
    if czy_puste(poz):
        puste.append(poz)
    if czy_puste(poz) and not czy_wejscie(poz):
        bezpieczne.append(poz)
```

Atakuj 2 kroki obok

Jeżeli w odległości 2 kroków jest przeciwnik, zamiast iść w jego kierunku i narażać się na szkody, lepiej go zaatakuj, aby nie mógł bezkarnie się do nas zbliżyć.

```
# funkcja zwróci prawdę, jeżeli w odległości 2 kroków z przodu jest wróg
def zprzodu(l1, l2):
    if rg.wdist(l1, l2) == 2:
        if abs(l1[0] - l2[0]) == 1:
            return False
        else:
            return True
    return False

# funkcja zwróci współrzędne pola środkowego między dwoma innymi
# oddalonymi o 2 kroki
def miedzypole(p1, p2):
    return (int((p1[0]+p2[0]) / 2), int((p1[1]+p2[1]) / 2))

for poz, robot in game.get('robots').items():
    if czy_wrog(poz):
        if rg.wdist(poz, self.location) == 2:
            if zprzodu(poz, self.location):
                return ['attack', miedzypole(poz, self.location)]
            if rg.wdist(rg.toward(loc, rg.CENTER_POINT), self.location) == 1:
                return ['attack', rg.toward(poz, rg.CENTER_POINT)]
            else:
                return ['attack', (self.location[0], poz[1])]
```

Składamy reguły

Ćwiczenie 3

Jeżeli czujesz się na siłach, spróbuj dokładać do robota w wersji **A** (opartego na funkcjach) po jednej z przedstawionych reguł, czyli: 1) Atakuj, jeśli nie umrzesz; 2) Ruszaj się bezpiecznie; 3) Atakuj na 2 kroki. Przetestuj w symulatorze każdą zmianę.

Omówione reguły można poskładać w różny sposób, np. tak:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

    def act(self, game):

        def czy_wejscie(poz):
            if 'spawn' in rg.loc_types(poz):
                return True
            return False

        def czy_wrog(poz):
            if game.robots.get(poz) != None:
                if game.robots[poz].player_id != self.player_id:
                    return True
            return False

        def czy_atak():
            if 9*len(wrogowie_obok) < self.hp:
                return True
            return False

        def czy_puste(poz):
            if ('normal' in rg.loc_types(poz)) and not ('obstacle' in rg.loc_
↪types(poz)):
                if game.robots.get(poz) == None:
                    return True
            return False

        puste = [] # lista pustych pól obok
        bezpieczne = [] # lista bezpiecznych pól obok

        for poz in rg.locs_around(self.location):
            if czy_puste(poz):
                puste.append(poz)
            if czy_puste(poz) and not czy_wejscie(poz):
                bezpieczne.append(poz)

        # funkcja zwróci prawdę, jeżeli w odległości 2 kroków z przodu jest wróg
        def zprzodu(l1, l2):
            if rg.wdist(l1, l2) == 2:
                if abs(l1[0] - l2[0]) == 1:
                    return False
                else:
                    return True
            return False

        # funkcja zwróci współrzędne pola środkowego między dwoma innymi
        # oddalonymi o 2 kroki
        def miedzypole(p1, p2):
            return (int((p1[0]+p2[0]) / 2), int((p1[1]+p2[1]) / 2))

        # lista wrogów obok
        wrogowie_obok = []
```

```
        for poz in rg.locs_around(self.location):
            if czy_wrog(poz):
                wrogowie_obok.append(poz)

        # jeżeli jesteś w punkcie wejścia, opuść go
        if czy_wejscie(self.location):
            return ['move', rg.toward(self.location, rg.CENTER_POINT)]

        # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne
        if len(wrogowie_obok):
            if czy_atak():
                return ['attack', wrogowie_obok.pop()]
            elif bezpieczne:
                return ['move', bezpieczne.pop()]

        # jeżeli wróg jest o dwa kroki, atakuj
        for poz, robot in game.get('robots').items():
            if czy_wrog(poz) and rg.wdist(poz, self.location) == 2:
                if zprzodu(poz, self.location):
                    return ['attack', miedzypole(poz, self.location)]
                if rg.wdist(rg.toward(poz, rg.CENTER_POINT), self.location) == 1:
                    return ['attack', rg.toward(poz, rg.CENTER_POINT)]
                else:
                    return ['attack', (self.location[0], poz[1])]

        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            return ['guard']

        # idź do środka planszy
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Możliwe ulepszenia

Poniżej pokazujemy “klocki”, których możesz użyć, aby zoptymalizować robota. Zamieszczamy również listę pytań do przemyślenia, aby zachęcić cię do samodzielnego konstruowania najlepszego robota :-)

Atakuj najsłabszego

```
# funkcja zwracająca atak na najsłabszego wroga obok
def atakuj():
    r = wrogowie_obok[0]
    for poz in wrogowie_obok:
        if game.robots[poz]['hp'] > game.robots[r]['hp']:
            r = poz
    return ['attack', r]
```

Inne

- Czy warto atakować, jeśli obok jest więcej niż 1 wróg?
- Czy warto atakować 1 wroga obok, ale mocniejszego od nas?

- Jeżeli nie można bezpiecznie się ruszyć, może lepiej się bronić?
- Jeśli jesteśmy otoczeni przez wrogów, może lepiej popełnić samobójstwo...

Proponujemy, żebyś sam zaczął wprowadzać i testować zasugerowane ulepszenia. Możesz też zajrzeć do drugiego *drugiego* i *trzeciego* zestawu klocków opartych na zbiorach.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

RG – klocki 2B

Wersja **B** oparta jest na zbiorach i operacjach na nich.

Wskazówka:

- Każdy "klocek" można testować osobno, a później w połączeniu z innymi. Warto i trzeba zmieniać kolejność stosowanych reguł!

Typy pól

Zobaczmy, w jaki sposób dowiedzieć się, w jakim miejscu się znajdujemy, gdzie wokół mamy wrogów lub pola, na które można wejść. Dysponując takimi informacjami, będziemy mogli podejmować bardziej przemyślane działania. Wykorzystamy wyrażenia zbiorów (ang. *set comprehensions*) (zob. *wyrażenie listowe*) i operacje na zbiorach (zob. *zbiór*).

Czy to wejście?

```
# wszystkie pola na planszy jako współrzędne (x, y)
wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}

# punkty wejścia (spawn)
wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}

# warunek sprawdzający, czy "poz" jest w punkcie wejścia
if poz in wejscia:
    pass
```

Metody i właściwości biblioteki *rg*:

- `gr.loc_types(poz)` – zwraca typ pola wskazywanego przez `poz`:
  - `invalid` – poza granicami planszy(np. (-1, -5) lub (23, 66));
  - `normal` – w ramach planszy;
  - `spawn` – punkt wejścia robotów;
  - `obstacle` – pola zablokowane ograniczające arenę.

Czy obok jest wróg?

Wersja oparta na zbiorach wykorzystuje różnicę i cześć wspólną zbiorów.

```
# pola zablokowane
zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}

# pola zajęte przez nasze roboty
przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id == self.
↪player_id}

# pola zajęte przez wrogów
wrogowie = set(game.robots) - przyjaciele

# pola sąsiednie
sasiednie = set(rg.locs_around(self.location)) - zablokowane

# pola obok zajęte przez wrogów
wrogowie_obok = sasiednie & wrogowie

# warunek sprawdzający, czy obok są wrogowie
if wrogowie_obok:
    pass
```

Metody i właściwości biblioteki *rg*:

- `rg.locs_around(poz, filter_out=None)` – zwraca listę położeń sąsiadujących z `poz`. Jako `filter_out` można podać typy położeń do wyeliminowania, np.: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))`.

Wskazówka: Definicje zbiorów należy wstawić na początku metody `Robot.act()` – przed pierwszym użyciem.

Wykorzystując powyższe "klocki" możemy napisać robota realizującego następujące reguły:

1. Opuść jak najszybciej wejście;
2. Atakuj wrogów obok;
3. W środku broń się;
4. W ostateczności idź do środka.

Implementacja

Przykładowa implementacja może wyglądać następująco:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

    def act(self, game):

        # wszystkie pola
        wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
```

```
        # punkty wejścia
        wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
        # pola zablokowane
        zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
        # pola zajęte przez nasze roboty
        przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==␣
↪self.player_id}
        # pola zajęte przez wrogów
        wrogowie = set(game.robots) - przyjaciele
        # pola sąsiednie
        sasiednie = set(rg.locs_around(self.location)) - zablokowane
        # pola sąsiednie zajęte przez wrogów
        wrogowie_obok = sasiednie & wrogowie

        # działanie domyślne:
        ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]

        # jeżeli jesteś w punkcie wejścia, opuść go
        if self.location in wejscia:
            ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]

        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            ruch = ['guard']

        # jeżeli obok są przeciwnicy, atakuj
        if wrogowie_obok:
            ruch = ['attack', wrogowie_obok.pop()]

        return ruch
```

Metoda `.pop()` zastosowana do zbioru zwraca element losowy.

Ćwiczenie 1

Zapisz powyższą implementację w katalogu `robot` i przetestuj ją w symulatorze, a następnie wystaw ją do walki z robotem podstawowym. Poeksperymentuj z kolejnością reguł, która określa ich priorytety!

Wskazówka: Do kontrolowania logiki działania robota zamiast rozłącznych instrukcji warunkowych: `if war1: ... if war2: ...` itd. można użyć instrukcji złożonej: `if war1: ... elif war2: ... [elif war3: ...] else: ...`.

Atakuj, jeśli nie umrzesz

Warto atakować, ale nie wtedy, gdy grozi nam śmierć. Można przyjąć zasadę, że atakujemy tylko wtedy, kiedy suma potencjalnych uszkodzeń będzie mniejsza niż zdrowie naszego robota. Zmień więc dotychczasowe reguły ataku wroga korzystając z poniższych "klocków":

```
# WERSJA B
# jeżeli obok są przeciwnicy, atakuj
if wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        pass
```

```
    else:
        ruch = ['attack', wrogowie_obok.pop()]
```

Metody i właściwości biblioteki *rg*:

- `self.hp` – ilość punktów HP robota.

Ćwiczenie 2

Dodaj powyższą regułę do poprzedniej wersji robota.

Ruszaj się bezpiecznie

Zamiast iść na oślep lepiej wchodzić czy uciekać na bezpieczne pola. Za "bezpieczne" przyjmiemy na razie pole puste, niezablokowane i nie będące punktem wejścia.

```
# WERSJA B
# zbiór bezpiecznych pól
bezpieczne = sasiednie - wrogowie_obok - wejscia - przyjaciele
```

Atakuj 2 kroki obok

Jeżeli w odległości 2 kroków jest przeciwnik, zamiast iść w jego kierunku i narażać się na szkody, lepiej go zaatakuj, aby nie mógł bezkarnie się do nas zbliżyć.

```
# WERSJA B
wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) & wrogowie)} -
↪ przyjaciele

if wrogowie_obok2:
    ruch = ['attack', wrogowie_obok2.pop()]
```

Składamy reguły

Ćwiczenie 3

Jeżeli czujesz się na siłach, spróbuj dokładać do robota w wersji **B** (opartego na zbiorach) po jednej z przedstawionych reguł, czyli: 1) Atakuj, jeśli nie umrzesz; 2) Ruszaj się bezpiecznie; 3) Atakuj na 2 kroki. Przetestuj w symulatorze każdą zmianę.

Omówione reguły można poskładać w różny sposób, np. tak:

W wersji **B** opartej na zbiorach:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

```

```
    def act(self, game):

        # wszystkie pola
        wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
        # punkty wejścia
        wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
        # pola zablokowane
        zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
        # pola zajęte przez nasze roboty
        przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==␣
↪self.player_id}
        # pola zajęte przez wrogów
        wrogowie = set(game.robots) - przyjaciele
        # pola sąsiednie
        sasiednie = set(rg.locs_around(self.location)) - zablokowane
        # pola sąsiednie zajęte przez wrogów
        wrogowie_obok = sasiednie & wrogowie
        wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &␣
↪wrogowie)} - przyjaciele
        # pola bezpieczne
        bezpieczne = sasiednie - wrogowie_obok - wejscia - przyjaciele

        # działanie domyślne:
        ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]

        # jeżeli jesteś w punkcie wejścia, opuść go
        if self.location in wejscia:
            ruch = ['move', rg.toward(self.location, rg.CENTER_POINT)]

        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            ruch = ['guard']

        # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne
        if wrogowie_obok:
            if 9*len(wrogowie_obok) >= self.hp:
                if bezpieczne:
                    ruch = ['move', bezpieczne.pop()]
            else:
                ruch = ['attack', wrogowie_obok.pop()]

        if wrogowie_obok2:
            ruch = ['attack', wrogowie_obok2.pop()]

        return ruch
```

Możliwe ulepszenia

Poniżej pokazujemy “klocki”, których możesz użyć, aby zoptymalizować robota. Zamieszczamy również listę pytań do przemyślenia, aby zachęcić cię do samodzielnego konstruowania najlepszego robota :-)

Atakuj najsłabszego

```
# wersja B
# funkcja znajdująca najsłabszego wroga obok z podanego zbioru (bots)
def minhp(bots):
    return min(bots, key=lambda x: game.robots[x].hp)

if wrogowie_obok:
    ...
    else:
        ruch = ['attack', minhp(wrogowie_obok)]
```

Najkrócej do celu

Funkcji `mindist()` można użyć do znalezienia najbliższego wroga, aby iść w jego kierunku, kiedy opuścimy punkt wejścia:

```
# WERSJA B
# funkcja zwraca ze zbioru pól (bots) pole najbliższe podanego celu (poz)
def mindist(bots, poz):
    return min(bots, key=lambda x: rg.dist(x, poz))

najblizszy_wrog = mindist(wrogowie,self.location)
```

Inne

- Czy warto atakować, jeśli obok jest więcej niż 1 wróg?
- Czy warto atakować 1 wroga obok, ale mocniejszego od nas?
- Jeżeli nie można bezpiecznie się ruszyć, może lepiej się bronić?
- Jeśli jesteśmy otoczeni przez wrogów, może lepiej popełnić samobójstwo...
- Spróbuj zmienić akcję domyślną.
- Spróbuj użyć jednej złożonej instrukcji warunkowej!

Proponujemy, żebyś sam zaczął wprowadzać i testować zasugerowane ulepszenia. Możesz też zajrzeć do *trzeciego* zestawu klocków.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

RG – klocki 3B

Robot dotychczasowy

Na podstawie reguł i klocków z części pierwszej mogliśmy stworzyć następującego robota:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg
```

```
5
6  class Robot:
7
8      def act(self, game):
9
10         wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
11         wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
12         zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
13         przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==␣
   ↪self.player_id}
14         wrogowie = set(game.robots) - przyjaciele
15
16         sasiednie = set(rg.locs_around(self.location)) - zablokowane
17         wrogowie_obok = sasiednie & wrogowie
18         wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &␣
   ↪wrogowie)} - przyjaciele
19         bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia -␣
   ↪przyjaciele
20
21         def mindist(bots, poz):
22             return min(bots, key=lambda x: rg.dist(x, poz))
23
24         if wrogowie:
25             najblizszy_wrog = mindist(wrogowie,self.location)
26         else:
27             najblizszy_wrog = rg.CENTER_POINT
28
29         # działanie domyślne:
30         ruch = ['guard']
31
32         if self.location in wejscia:
33             if bezpieczne:
34                 ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
35         elif wrogowie_obok:
36             if 9*len(wrogowie_obok) >= self.hp:
37                 if bezpieczne:
38                     ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
39             else:
40                 ruch = ['attack', wrogowie_obok.pop()]
41         elif wrogowie_obok2:
42             ruch = ['attack', wrogowie_obok2.pop()]
43         elif bezpieczne:
44             ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
45
46         return ruch
```

Jego działanie opiera się na wyznaczeniu zbiorów pól określonego typu zastosowaniu następujących reguł:

1. jeżeli nie ma nic lepszego, broń się,
2. z punktu wejścia idź bezpiecznie do środka;
3. atakuj wrogów wokół siebie, o ile to bezpieczne, jeżeli nie, idź bezpiecznie do środka;
4. atakuj wrogów dwa pola obok;
5. idź bezpiecznie na najbliższego wroga.

Spróbujemy go ulepszyć dodając, ale i prezycując reguły.

Śledź wybrane miejsca

Aby unikać niepotrzebnych kolizji, nie należy wchodzić na wybrane wcześniej pola. Trzeba więc zapamiętywać pola wybrane w danej rundzie.

Przed klasą `Robot` definiujemy dwie zmienne globalne, następnie na początku metody `.act()` inicjujemy dane:

```
# zmienne globalne
runda_numer = 0 # numer rundy
wybrane_pola = set() # zbiór wybranych w rundzie pól

# inicjacja danych
# wyzeruj zbiór wybrane_pola przy pierwszym robocie w rundzie
global runda_numer, wybrane_pola
if game.turn != runda_numer:
    runda_numer = game.turn
    wybrane_pola = set()
```

Do zapamiętywania wybranych w rundzie pól posłużą funkcje `ruszaj()` i `stoj()`:

```
# jeżeli się ruszamy, zapisujemy docelowe pole
def ruszaj(poz):
    wybrane_pola.add(poz)
    return ['move', poz]

# jeżeli stoimy, zapisujemy zajmowane miejsce
def stoj(act, poz=None):
    wybrane_pola.add(self.location)
    return [act, poz]
```

Ze zbioru `bezpieczne` wyłączamy wybrane pola i stosujemy nowe funkcje:

```
# ze zbioru bezpieczne wyłączamy wybrane_pola
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - \
             wejscia - przyjaciele - wybrane_pola

# stosujemy nowy kod w regule "atakuj wroga dwa pola obok"
elif wrogowie_obok2 and self.location not in wybrane_pola:

# stosujemy funkcje "ruszaj()" i "stoj()"

# zamiast: ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))

# zamiast: ruch = ['attack', wrogowie_obok.pop()]
ruch = stoj('attack', wrogowie_obok.pop())

# zamiast: ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
ruch = ruszaj(mindist(bezpieczne, najblizszy_wrog))
```

Wskazówka: Można zapamiętywać wszystkie wybrane ruchy lub tylko niektóre. Przetestuj, czy ma to wpływ na skuteczność AI.

Atakuj najsłabszego

Do tej pory atakowaliśmy przypadkowego robota wokół nas, lepiej wybrać najsłabszego.

```
# funkcja znajdująca najsłabszego wroga obok
def minhp(bots):
    return min(bots, key=lambda x: game.robots[x].hp)

elif wrogowie_obok:
    ...
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Funkcja `minhp()` poda nam położenie najsłabszego wroga. Argument parametru `key`, czyli wyrażenie *lambda* zwraca właściwość robotów, czyli punkty HP, wg której są porównywane.

Samobójstwo lepsze niż śmierć?

Jeżeli grozi nam śmierć, a nie ma bezpiecznego miejsca, aby uciec, lepiej popełnić samobójstwo:

```
# samobójstwo lepsze niż śmierć
elif wrogowie_obok:
    if bezpieczne:
        ...
    else:
        ruch = stoj('suicide')
```

Unikaj nierównych starć

Nie warto walczyć z przeważającą liczbą wrogów.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        ...
    elif len(wrogowie_obok) > 1:
        if bezpieczne:
            ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Goń najsłabszych

Zamiast atakować słabego uciekającego robota, lepiej go gonić, może trafi w gorsze miejsce...

```
elif wrogowie_obok:
    ...
    else:
        cel = minhp(wrogowie_obok)
        if game.robots[cel].hp <= 5:
            ruch = ruszaj(cel)
        else:
            ruch = stoj('attack', minhp(wrogowie_obok))
```

Robot zaawansowany

Po dodaniu/zmodyfikowaniu omwionych powyej reguł kod naszego robota może wyglądać tak:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

runda_numer = 0 # numer rundy
wybrane_pola = set() # zbiór wybranych w rundzie pól

class Robot:

    def act(self, game):

        global runda_numer, wybrane_pola
        if game.turn != runda_numer:
            runda_numer = game.turn
            wybrane_pola = set()

        # jeżeli się ruszamy, zapisujemy docelowe pole
        def ruszaj(loc):
            wybrane_pola.add(loc)
            return ['move', loc]

        # jeżeli stoimy, zapisujemy zajmowane miejsce
        def stoj(act, loc=None):
            wybrane_pola.add(self.location)
            return [act, loc]

        # wszystkie pola
        wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
        # punkty wejścia
        wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
        # pola zablokowane
        zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
        # pola zajęte przez nasze roboty
        przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==␣
↪self.player_id}
        # pola zajęte przez wrogów
        wrogowie = set(game.robots) - przyjaciele
        # pola sąsiednie
        sasiednie = set(rg.locs_around(self.location)) - zablokowane
        # pola sąsiednie zajęte przez wrogów
        wrogowie_obok = sasiednie & wrogowie
        wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &␣
↪wrogowie)} - przyjaciele
        # pola bezpieczne
        bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia -␣
↪przyjaciele - wybrane_pola

        # funkcja znajdująca najsłabszego wroga obok z podanego zbioru (bots)
        def mindist(bots, loc):
            return min(bots, key=lambda x: rg.dist(x, loc))

        if wrogowie:
            najblizszy_wrog = mindist(wrogowie,self.location)
```

```
        else:
            najblizszy_wrog = rg.CENTER_POINT

        # działanie domyślne:
        ruch = ['guard']

        # jeżeli jesteś w punkcie wejścia, opuść go
        if self.location in wejscia:
            ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))

        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            ruch = ['guard']

        # jeżeli obok są przeciwnicy, atakuj, o ile to bezpieczne,
        # najsłabszego wroga
        if wrogowie_obok:
            if 9*len(wrogowie_obok) >= self.hp:
                if bezpieczne:
                    ruch = ruszaj(mindist(bezpieczne, rg.CENTER_POINT))
            else:
                ruch = ['attack', minhp(wrogowie_obok)]

        if wrogowie_obok2 and self.location not in wybrane_pola:
            ruch = ['attack', wrogowie_obok2.pop()]

        return ruch
```

Na koniec trzeba przetestować robota. Czy rzeczywiście jest lepszy od poprzednich wersji?

Podsumowanie

Nie myśl, że zastosowanie wszystkich powyższych reguł automatycznie ulepszy robota. Weź pod uwagę fakt, że roboty pojawiają się w losowych punktach, oraz to, że strategia przeciwnika może być inna od zakładanej. Zaproponowane połączenie klocków nie musi być optymalne. Przetestuj kolejne wersje robotów, ustal ich zalety i wady, eksperymentuj, aby znaleźć lepsze rozwiązania.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

RG – dokumentacja

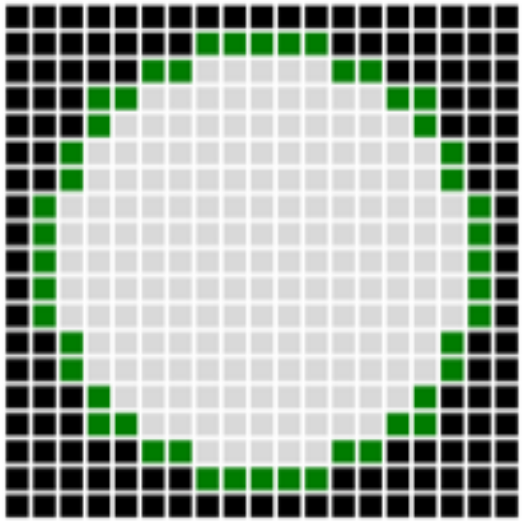
RobotGame to gra, w której walczą ze sobą programy – boty. Poniżej nieautoryzowane tłumaczenie oryginalnej dokumentacji oraz materiałów dodatkowych:

Zasady gry

W *Grze robotów* piszesz programy kierujące walczącymi dla Ciebie robotami. Planszą gry jest siatka o wymiarach 19x19 pól. Celem gry jest umieszczenie na niej jak największej ilości robotów w ciągu 100 rund rozgrywki.

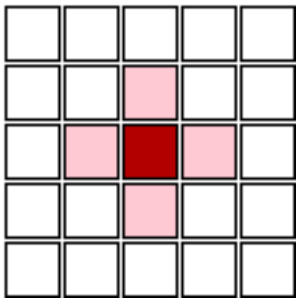
Czarne kwadraty to pola, na które nie ma wstępu. Wyznaczają kolistą arenę dla walk robotów.

Zielone kwadraty oznaczają punkty wejścia do gry. Co **10** rund po **5** robotów każdego gracza rozpoczyna walkę w losowych punktach wejścia (ang. *spawn points*). Roboty z poprzednich tur pozostające w tych punktach giną.



Każdy robot rozpoczyna grę z 50 punktami HP (ang. *health points*).

Roboty mogą działać (przemieszczać się, atakować itd.) na przyległych kwdratach w pionie (góra, dół) i w poziomie (lewo, prawo).



W każdej rundzie możliwe są następujące działania robota:

- **Ruch** na przyległy kwadrat. Jeżeli znajduje się tam już robot lub inny robot próbuje zająć to samo miejsce, obydwa tracą 5 punktów HP z powodu kolizji, a ruch(y) nie dochodzi(ą) do skutku. Jeżeli jednak robot chce przejść na pole zajęte przez innego, a ten drugi opuszcza zajmowane pole, ruch jest udany.

  Minimum cztery roboty w kwadracie, przemieszczające się zgodnie ze wskazówkami zegara, będą mogły się poruszać, podobnie dowolna ilość robotów w kole. (Roboty nie mogą bezpośrednio zamieniać się miejscami!)

- **Atak** na przyległy kwadrat. Jeżeli w atakowanym kwadracie znajdzie się robot pod koniec rundy, np. robot pozostał w miejscu lub przeszedł na nie, robot ten traci od 8 do 10 punktów HP w następstwie uszkodzeń.

- **Samobójstwo** – robot ginie pod koniec rundy, zabierając 15 punktów HP wszystkim robotom w sąsiedztwie.

- **Obrona** – robot pozostaje w miejscu, tracąc połowę punktów HP wskutek ataku lub samobójstwa, nie odnosi uszkodzeń z powodu kolizji.

W grze nie można uszkodzić własnych robotów. Kolizje, ataki i samobójstwa wyrządzają szkody tylko przeciwnikom.

Wygrawa gracz, który po 100 rundach ma największą liczbę robotów na planszy.

Zadaniem gracza jest zakodowanie sztucznej inteligencji (ang. AI – *artificial itelligance*), dla wszystkie swoich robotów. Aby wygrać, roboty gracza muszą ze sobą współpracować, np. żeby otoczyć przeciwnika.

Informacja: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stonie RobotGame.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Rozpoczynamy

Spis treści

Tworzenie robota

Podstawowa struktura klasy reprezentującej każdego robota jest następująca:

```
class Robot:

    def act(self, game):
        return [<some action>, <params>]
```

Na początku gry powstaje jedna instanacja klasy `Robot`. Oznacza to, że właściwości klasy oraz globalne zmienne modułu są współdzielone między wywołaniami. W każdej rundzie system wywołuje metodę `act` tej instancji dla każdego robota, aby określić jego działanie. (Uwaga: na początku przeczytaj reguły.)

Metoda `act` musi zwrócić jedną z następujących odpowiedzi:

```
['move', (x, y)]
['attack', (x, y)]
['guard']
['suicide']
```

Jeżeli metoda `act` zwróci wyjątek lub błędne polecenie, robot pozostaje w obronie, ale jeżeli powtórzy się to zbyt wiele razy, gracz zostanie zmuszony do kapitulacji. Szczegóły omówiono w dziale *Zabezbieczenia*.

Odczytywanie właściwości robota

Każdy robot, przy użyciu wskaźnika `self`, udostępnia następujące właściwości:

- `location` – położenie robota w formie tupli (x, y);
- `hp` – punkty zdrowia wyrażone liczbą całkowitą
- `player_id` – identyfikator gracza, do którego należy robot (czyli oznaczenie "drużyny")
- `robot_id` – unikalny identyfikator robota, ale tylko w obrębie "drużyny"

Dla przykładu: kod `self.hp` – zwróci aktualny stan zdrowia robota.

W każdej rundzie system wywołując metodę `act` udostępnia jej stan gry w następującej strukturze `game`:

```
{
    # słownik wszystkich robotów na polach wyznaczonych
    # przez {location: robot}
    'robots': {
        (x1, y1): {
            'location': (x1, y1),
            'hp': hp,
            'player_id': player_id,

            # jeżeli robot jest w twojej drużynie
            'robot_id': robot_id
        },

        # ...i pozostałe roboty
    },

    # ilość odbytych rund (wartość początkowa 0)
    'turn': turn
}
```

Wszystkie roboty w strukturze `game['robots']` są instancjami specjalnego słownika udostępniającego ich właściwości, co przyśpiesza kodowanie. Tak więc następujące konstrukcje są tożsame:

```
game['robots'][location]['hp']
game['robots'][location].hp
game.robots[location].hp
```

Poniżej zwięzły przykład drukujący położenie wszystkich robotów z danej drużyny:

```
class Robot:
    def act(self, game):
        for loc, robot in game.robots.items():
            if robot.player_id == self.player_id:
                print loc
```

Przykładowy robot

Poniżej mamy kod prostego robota, który można potraktować jako punkt wyjścia. Robot, jeżeli znajdzie wokół siebie przeciwnka, atakuje go, w przeciwnym razie przemieszcza się do środka planszy (`rg.CENTER_POINT`).

```
import rg

class Robot:
    def act(self, game):
        # if we're in the center, stay put
        if self.location == rg.CENTER_POINT:
            return ['guard']

        # if there are enemies around, attack them
        for loc, bot in game.robots.iteritems():
            if bot.player_id != self.player_id:
                if rg.dist(loc, self.location) <= 1:
                    return ['attack', loc]
```

```
        # move toward the center
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Użyliśmy, jak widać modułu `rg`, który zostanie omówiony dalej.

Informacja: Podczas gry tworzona jest tylko jedna instancja robota, w której można zapisywać trwałe dane.

Informacja: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stonie RobotGame.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Biblioteka rg

Gra robotów udostępnia bibliotekę ułatwiającą programowanie. Zawarta jest w module `rg`, który importujemy na początku pliku instrukcją `import rg`.

Uwaga: Położenie robota (`loc`) reprezentowane jest przez tuplę (x, y).

rg.dist(loc1, loc2)

Zwraca matematyczną odległość między dwoma położeniami.

rg.wdist(loc1, loc2)

Zwraca różnicę w ruchach między dwoma położeniami. Ponieważ robot nie może poruszać się na ukos, jest to suma `dx + dy`.

rg.loc_types(loc)

Zwraca listę typów położeń wskazywanych przez `loc`. Możliwe wartości to:

- `invalid` – poza granicami planszy(np. (-1, -5) lub (23, 66));
- `normal` – w ramach planszy;
- `spawn` – punkt wejścia robotów;
- `obstacle` – pola, na które nie można się ruszyć (szare kwadraty).

Metoda nie ma dostępu do kontekstu gry, np. wartość `obstacle` nie oznacza, że na sprawdzanym kwadracie nie ma wrogiego robota; wiemy tylko, że dany kwadrat jest przeszkodą na mapie.

Zwrócona lista może zawierać kombinacje wartości typu: `['normal', 'obstacle']`.

rg.locs_around(loc, filter_out=None)

Zwraca listę położeń sąsiadujących z `loc`. Jako drugi argument `filter_out` można podać listę typów położeń do wyeliminowania. Dla przykładu: `rg.locs_around(self.location, filter_out=('invalid', 'obstacle'))` – poda listę kwadratów, na które można wejść.

rg.toward(current_loc, dest_loc)

Zwraca następne położenie na drodze z bieżącego miejsca do podanego. Np. poniższy kod:

```
import rg

class Robot:
    def act(self, game):
        if self.location == rg.CENTER_POINT:
            return ['suicide']
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

– skieruje robota do środka planszy, gdzie popełni on samobójstwo.

rg.CENTER_POINT

Stała (ang. *constant*) definiująca położenie środkowego punktu planszy.

rg.settings

Specjalny typ słownika (AttrDict) zawierający ustawienia gry.

- `rg.settings.spawn_every` – ilość rozegranych rund od wejścia robota do gry;
- `rg.settings.spawn_per_player` - ilość robotów wprowadzonych przez gracza;
- `rg.settings.robot_hp` – domyślna ilość punktów HP robota;
- `rg.settings.attack_range` – tupla (minimum, maksimum) przechowująca zakres uszkodzeń wyrządzonych przez atak;
- `rg.settings.collision_damage` – uszkodzenia wyrządzone przez kolizję;
- `rg.settings.suicide_damage` – uszkodzenia wyrządzone przez samobójstwo;
- `rg.settings.max_turns` – liczba rund w grze.

Czy w danym położeniu jest robot

Ponieważ struktura `game.robots` jest słownikiem robotów, w którym kluczami są położenia, a wartościami roboty, można użyć testu `(x, y) in game.robots`, który zwróci `True`, jeśli w danym położeniu jest robot, lub `Flase` w przeciwnym razie.

Informacja: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stonie RobotGame.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Testowanie robotów

Pakiet rgkit

Do budowania i testowania robotów używamy pakietu *rgkit*. W tym celu przygotowujemy środowisko deweloperskie, zawierające bibliotekę `rg`:

```
~$ mkdir robot; cd robot
~robot/$ virtualenv env
~robot/$ source env/bin/activate
(env):~robot$ pip install rgkit
```

Po wykonaniu powyższych poleceń i zapisaniu implementacji klasy `Robot` np. w pliku `~/robot/robot01.py` możemy uruchamiać grę przeciwko samemu sobie:

```
(env)~/robot$ rgrun robot01.py robot01.py
```

Jeżeli utworzymy inne implementacje robotów, np. w pliku `~/robot/robot02.py` skonfrontujemy je poleceniem:

```
(env)~/robot$ rgrun robot01.py robot02.py
```

Przydatne opcje polecenia `rgrun`:

- `-H` – symulacja bez UI
- `-r` – roboty wprowadzane losowo zamiast symetrycznie.

Uwaga: Pokazana powyżej instalacja zakłada użycie środowiska wirtualnego tworzonego przez pakiet *virtualenv*, dlatego przed uruchomieniem symulacji, a także przed użyciem omówionego niżej pakietu *robotgame-bots* trzeba pamiętać o wydaniu w katalogu `robot` polecenia:

```
~/robot$ source env/bin/activate
```

Roboty open-source

Swoje roboty warto wystawić do gry przeciwko przykładowym robotom dostarczanym przez projekt robotgame-bots: Instalacja sprowadza się do wykonania polecenia w utworzonym wcześniej katalogu `robot`:

```
~/robot$ git clone https://github.com/mpeterv/robotgame-bots bots
```

Wynikiem polecenia będzie utworzenia podkatalogu `~/robot/bots` zawierającego kod przykładowych robotów.

Listę dostępnych robotów najłatwiej użyskać wydając polecenie:

```
(env)~/robot$ ls bots
```

Aby zmierzyć się z wybranym robotem – na początek sugerujemy *stupid26.py* – wydajemy polecenie:

```
(env)~/robot$ rgrun mojrobot.py bots/stupid26.py
```

Od czasu do czasu można zaktualizować dostępne roboty poleceniem:

```
~/robot/bots$ git pull --rebase origin master
```

Symulator rg

Bardzo przydatny jest symulator zachowania robotów. Instalacja w katalogu `robot`:

```
~/robot$ git clone https://github.com/mpeterv/rgsimulator.git
```

Następnie uruchamiamy symulator podając jako parametr nazwę przynajmniej jednego robota (można dwóch):

```
(env)~/robot$ rgsimulator/rgsimulator.py robot01.py [robot02.py]
```

Symulatorem sterujemy za pomocą klawiszy:

- Klawisze kursora lub WASD do zaznaczania pól.
- Klawisz F: utworzenie robota-przyjaciela w zaznaczonym polu.
- Klawisz E: utworzenie robota-wroga w zaznaczonym polu.
- Klawisze Delete or Backspace: usunięcie robota z zaznaczonego pola.
- Klawisz H: zmiana punktów HP robota.
- Klawisz T: zmiana rundy.
- Klawisz C: wyczyszczenie planszy gry.
- Klawisz Spacja: pokazuje planowane ruchy robotów.
- Klawisz Enter: uruchomienie rundy.
- Klawisz L: załadowanie meczu z robotgame.net. Należy podać tylko numer meczu.
- Klawisz K: załadowanie podanej rundy z załadowanego meczu. Also updates the simulator turn counter.
- Klawisz P: zamienia kod robotów gracza 1 z 2.
- Klawisz O: ponowne załadowanie kodu obydwu robotów.
- Klawisz N: zmienia działanie robota, wyznacza "następne działanie".
- Klawisz G: tworzy i usuwa roboty w punktach wejścia (ang. *spawn locations*), "generowanie robotów".

Wskazówka: W Linuksie warto utworzyć sobie przyjazny link do wywoływania symulatora. W katalogu `robot` wydajemy polecenia:

```
(env)~/robot$ ln -s rgsimulator/rgsimulator.py symuluj
(env)~/robot$ symuluj robot01.py [robot02.py]
```

Informacja: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stonie RobotGame, a także RobotGame – rgkit. Opis działania symulatora robotów przetłumaczono na podstawie strony projektu Rgsimulator.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Strategia podstawowa

Przykład robota

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

    def act(self, game):
        # jeżeli jesteś w środku, broń się
        if self.location == rg.CENTER_POINT:
            return ['guard']

        # jeżeli wokół są przeciwnicy, atakuj
        for poz, robot in game.robots.iteritems():
            if robot.player_id != self.player_id:
                if rg.dist(poz, self.location) <= 1:
                    return ['attack', poz]

        # idź do środka planszy
        return ['move', rg.toward(self.location, rg.CENTER_POINT)]
```

Z powyższego kodu wynikają trzy zasady:

- broń się, jeżeli jesteś w środku planszy;
- atakuj przeciwnika, jeżeli jest obok;
- idź do środka.

To pozwala nam rozpocząć grę, ale wiele możemy ulepszyć. Większość usprawnień (ang. *feature*), które zostaną omówione, to rozszerzenia wersji podstawowej. Konstruując robota, można je stosować wybiórczo.

Kolejne reguły

Rozbudujemy przykład podstawowy. Oto lista reguł, które warto rozważyć:

- **Reguła 1: Opuść punkt wejścia.**

Pozostawanie w punkcie wejścia nie jest dobre. Sprawdźmy, czy jesteśmy w punkcie wejścia i czy powinniśmy z niego wyjść. Nawet wtedy, gdy jest ktoś do zaatakowania, ponieważ nie chcemy zostać zamknięci w pułapce wejścia.

- **Reguła 2: Uciekaj, jeśli masz zginąć.**

Przykładowy robot atakuje aż do śmierci. Ponieważ jednak wygrana zależy od liczby pozostałych robotów, a nie ich zdrowia, bardziej opłaca się zachować robota niż poświęcać go, żeby zadał dodakowe obrażenia przeciwnikowi. Jeżeli więc jesteśmy zagrożeni śmiercią, uciekamy, a nie giniemy na próżno.

- **Reguła 3: Atakuje przeciwnika o dwa kroki od ciebie.**

Przyjrzyj się grającemu wg reguł robotowi, zauważysz, że kiedy wchodzi na pole atakowane przez przeciwnika, odnosi obrażenia. Dlatego, jeśli prawdopodobne jest, że przeciwnik może znaleźć się w naszym sąsiedztwie, trzeba go zatakować. Dzięki temu nit się do nas bezkarnie nie zbliży.

Informacja: Połączenie ucieczki i ataku w kierunku przeciwnika naprawdę jest skuteczne. Każdy agresywny wróg zanim nas zaatakuje, sam spotyka się z atakiem. Jeżeli w porę odskoczysz, zanim się zbliży, działanie takie możesz powtórzyć. Technika ta nazywana jest w grach kiting, a jej działanie ilustruje poniższa animacja:

Zwróć uwagę na słabego robota ze zdrowiem 8 HP, który podchodzi do mocnego robota z 50 HP, a następnie ucieka. Zbliżając się atakuje pole, na które wchodzi przeciwnik, ucieka i ponawia działanie. Trwa to do momentu, kiedy silniejszy robot popełni samobójstwo (co w tym wypadku jest mało przydatne). Wszystko bez uszczerbku na zdrowiu słabszego robota.

- **Reguła 4: Wchodź tylko na wolne pola.**

Przykładowy robot idzie do środka planszy, ale w wielu wypadkach lepiej zrobić coś innego. Np. iść tam, gdzie jest bezpiecznie, zamiast narażać się na bezużyteczne niebezpieczeństwo. Co jest bowiem ryzykowne? Po wejściu na planszę ruch na pole przeciwnika lub wchodzenie w jego sąsiedztwo. Wiadomo też, że nie możemy wchodzić na zajęte pola i że możemy zmniejszyć ilość kolizji, nie wchodząc na pola zajęte przez naszą drużynę.

- **Reguła 5: Idź na wroga, jeżeli go nie ma w zasięgu dwóch kroków.**

Po co iść do środka, skoro mamy inne bezpieczne możliwości? Wprawdzie stanie w punkcie wejścia jest złe, ale to nie znaczy, że środek planszy jest dobry. Lepszym wyborem jest ruch w kierunku, ale nie na pole, przeciwnika. W połączeniu z atakiem daje nam to lepszą kontrolę nad planszą. Później przekonamy się jeszcze, że są sytuacje, kiedy wejście na potencjalnie niebezpieczne pole warte jest ryzyka, ale na razie poprzestańmy na tym, co ustaliliśmy.

Łączenie ulepszeń

Zapiszmy wszystkie reguły w pseudokodzie. Możemy użyć do tego jednej rozbudowanej instrukcji warunkowej if/else.

```
jeżeli jesteś w punkcie wejścia:
    rusz się bezpiecznie (np. poza wejście)
jeżeli jeddnak mamy przeciwnika o krok dalej:
    jeżeli możemy umrzeć:
        ruszamy się w bezpieczne miejsce
    w przeciwnym razie:
        atakujemy przeciwnika
jeżeli jednak mamy przeciwnika o dwa kroki dalej:
    atakujemy w jego kierunku
jeżeli mamy bezpieczny ruch (i nikogo wokół siebie):
    ruszamy się bezpiecznie, ale w kierunku przeciwnika
w przeciwnym razie:
    bronimy się w miejscu, bo nie ma gdzie ruszyć się lub atakować
```

Implementacja

Do zakodowania omówionej logiki potrzebujemy struktury danych gry z jej ustawieniami i kilku funkcji. Pamiętajmy, że jest wiele sobosobów na zapisanie kodu w Pythonie. Poniższy w żdanym razie nie jest optymalny, ale działa jako przykład.

Zbiory zamiast list

Dla ułatwienia użyjemy pythonowych zbiorów razem z funkcją `set()` i wyrażeniami zbiorów (ang. *set comprehensions*).

Informacja: Zbiory i operacje na nich omówiono w dokumentacji zbiorów, podobnie przykłady wyrażeń listowych i odpowiadających im pętli.

Podstawowe operacje na zbiorach, których użyjemy to:

- `|` lub suma – zwraca zbiór wszystkich elementów zbiorów;
- `-` lub różnica – zbiór elementów obecnych tylko w pierwszym zbiorze;
- `&` lub iloczyn – zwraca zbiór elementów występujących w obydwu zbiorach.

Załóżmy, że zaczniemy od wygenerowania następujących list: `drużyna` – członkowie drużyny, `wrogowie` – przeciwnicy, `wejścia` – punkty wejścia oraz `przeszkody` – położenia zablokowane, tzn. szare kwadraty.

Zbiory pól

Aby ułatwić implementację omówionych ulepszeń, przygotujemy kilka zbiorów reprezentujących pola różnych kategorii na planszy gry. W tym celu używamy wyrażeń listowych (ang. *list comprehensions*).

```
# zbiory pól na planszy

# wszystkie pola
wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}

# punkty wejścia (spawn)
wejscia = {loc for loc in wszystkie if 'spawn' in rg.loc_types(loc)}

# pola zablokowane (obstacle)
zablokowane = {loc for loc in wszystkie if 'obstacle' in rg.loc_types(loc)}

# pola zajęte przez nasze roboty
przyjaciele = {loc for loc in game.robots if game.robots[loc].player_id == self.
↪player_id}

# pola zajęte przez wrogów
wrogowie = set(game.robots) - przyjaciele
```

Warto zauważyć, że zbiór wrogich robotów otrzymujemy jako różnicę zbioru wszystkich robotów i tych z naszej drużyny.

Wykorzystanie zbiorów

Przy poruszaniu się i atakowaniu mamy tylko cztery możliwe kierunki, które zwraca funkcja `rg.locs_around`. Możemy wykluczyć położenia zablokowane (ang. *obstacle*), ponieważ nigdy ich nie zajmujemy i nie atakujemy. Iloczyn zbiorów `sasiednie & wrogowie` da nam zbiór przeciwników w sąsiedztwie:

```
# pola sąsiednie
sasiednie = set(rg.locs_around(self.location)) - zablokowane
```

```
# pola sąsiednie zajęte przez wrogów
wrogowie_obok = sasiednie & wrogowie
```

Aby odnaleźć wrogów oddalonych o dwa kroki, szukamy przyległych kwadratów, obok których są przeciwnicy. Wyłączamy sąsiednie pola zajęte przez członków drużyny.

```
# pola zajęte przez wrogów w odległości 2 kroków
wrogowie_obok2 = {loc for loc in sasiednie if (set(rg.locs_around(loc)) & wrogowie)} -
↪ przyjaciele
```

Teraz musimy sprawdzić, które z położeń są bezpieczne. Usuwamy pola zajmowane przez przeciwników w odległości 1 i 2 kroków. Pozbywamy się także punktów wejścia, nie chcemy na nie wracać. Podobnie, aby zmniejszyć możliwość kolizji, wyrzucamy pola zajmowane przez drużynę. W miarę komplikowania logiki będzie można zastąpić to ograniczenie dodatkowym warunkiem, ale na razie to najlepsze, co możemy zrobić.

```
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia - przyjaciele
```

Potrzebujemy funkcji, która wybierze ze zbioru położeń pole najbliższe podanego. Możemy użyć tej funkcji do znajdowania najbliższego wroga, jak również do wyboru pola z bezpiecznej listy. Możemy więc wybrać ruch najbardziej przybliżający nas do założonego celu.

```
def mindist(bots, loc):
    return min(bots, key=lambda x: rg.dist(x, loc))
```

Możemy użyć metody `pop()` zbioru, aby pobrać jego dowolny element, np. przeciwnika, którego zaatakujemy. Żeby dowiedzieć się, czy jesteśmy zagrożeni śmiercią, możemy pomnożyć liczbę sąsiadujących przeciwników przez średni poziom uszkodzeń (9 punktów HP) i sprawdzić, czy mamy więcej siły.

Ze względu na sposób napisania funkcji `minidist()` trzeba pamiętać o przekazywaniu jej niepustych zbiorów. Jeśli np. zbiór przeciwników będzie pusty, funkcja zwróci błąd.

Składamy wszystko razem

Po złożeniu wszystkich kawałków kodu razem otrzymujemy przykładową implemetację robota wyposażonego we wszystkie założone wyżej właściwości:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import rg

class Robot:

    def act(self, game):

        wszystkie = {(x, y) for x in xrange(19) for y in xrange(19)}
        wejscia = {poz for poz in wszystkie if 'spawn' in rg.loc_types(poz)}
        zablokowane = {poz for poz in wszystkie if 'obstacle' in rg.loc_types(poz)}
        przyjaciele = {poz for poz in game.robots if game.robots[poz].player_id ==␣
↪self.player_id}
        wrogowie = set(game.robots) - przyjaciele

        sasiednie = set(rg.locs_around(self.location)) - zablokowane
        wrogowie_obok = sasiednie & wrogowie
        wrogowie_obok2 = {poz for poz in sasiednie if (set(rg.locs_around(poz)) &␣
↪wrogowie)} - przyjaciele
```

```
        bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 - wejscia -␣
↪przyjaciele

        def mindist(bots, poz):
            return min(bots, key=lambda x: rg.dist(x, poz))

        if wrogowie:
            najblizszy_wrog = mindist(wrogowie,self.location)
        else:
            najblizszy_wrog = rg.CENTER_POINT

        # działanie domyślne:
        ruch = ['guard']

        if self.location in wejscia:
            if bezpieczne:
                ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
        elif wrogowie_obok:
            if 9*len(wrogowie_obok) >= self.hp:
                if bezpieczne:
                    ruch = ['move', mindist(bezpieczne, rg.CENTER_POINT)]
            else:
                ruch = ['attack', wrogowie_obok.pop()]
        elif wrogowie_obok2:
            ruch = ['attack', wrogowie_obok2.pop()]
        elif bezpieczne:
            ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]

        return ruch
```

Informacja: Niniejsza dokumentacja jest swobodnym i nieautoryzowanym tłumaczeniem materiałów dostępnych na stonie Robotgame basic strategy.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Strategia pośrednia

Zacznijmy od znanego

W poprzednim poradniku (*Strategia podstawowa*) zaczęliśmy od bota realizującego następujące zasady:

- Broń się w środku planszy
- Atakuj wrogów obok
- Idź do środka

Zmieniliśmy lub dodaliśmy następujące reguły:

- Opuść wejście
- Uciekaj, jeśli masz zginąć
- Atakuj wrogów dwa kroki obok

- Wchodź na bezpieczne, niezajęte pola
- Idź na wroga, jeśli w pobliżu go nie ma

Do powyższych dodamy kolejne reguły w postaci fragmentów kodu, które trzeba zintergrować z dotychczasową implementacją bota, aby go ulepszyć.

Śledź wybierane miejsca

To raczej złożona funkcja, ale jest potrzebna, aby zmniejszyć ilość kolizji. Dotychczasowe boty drużyny próbują wejść na to samo miejsce i atakują się nawzajem. Co prawda nie tracimy wtedy pukntów życia, ale (prawie) zawsze mamy lepszy wybór. Jeżeli będziemy śledzić wszystkie wybrane przez nas ruchy w ramach rundy, możemy uniknąć niepotrzebnych kolizji. Niestety, to wymaga wielu fragementów kodu.

Na początku dodamy zmienną, która posłuży do sprawdzenia, czy dany robot jest pierwszym wywoływanym w rundzie. Jeżeli tak, musimy wyczyścić listę poprzednich ruchów i zaktualizować licznik rund. Odpowiedni kod trzeba umieścić na początku metody `Robot.act`:

Uwaga: Trzeba zainicjować zmienną globalną `runda_numer`.

```
global runda_numer, wybrane_pola
if game.turn != runda_numer:
    runda_numer = game.turn
    wybrane_pola = set()
```

Kolejne fragmenty odpowiadać będą za zapamiętywanie wykonywanych ruchów. Kod najwygodniej umieścić w pojedynczych funkcjach, które zanim zwrócą wybrany ruch, zapiszą go na liście. Warto zauważyć, że zapisywane będą współrzędne pól, na które wchodzimy lub na których pozostajemy (obrona, atak, samobójstwo). Funkcje muszą znaleźć się w metodzie `Robot.act`, aby współdzieliły jej przestrzeń nazw.

```
# Jeżeli się ruszamy, zapisujemy docelowe pole

def ruszaj(loc):
    wybrane_pola.add(loc)
    return ['move', loc]

# Jeżeli pozostajemy w miejscu, zapisujemy aktualne położenie
# przy użyciu self.location

def stoj(act, loc=None):
    wybrane_pola.add(self.location)
    return [act, loc]
```

Następnym krokiem jest usunięcie listy `wybrane_pola` ze zbioru bezpiecznych pól, które są podstawą dalszych wyborów:

```
bezpieczne = sasiednie - wrogowie_obok - wrogowie_obok2 \
             - wejscia - przyjaciele - wybrane_pola
```

Roboty atakujące przeciwnika o dwa kroki obok często go otaczają (to dobrze), ale jednocześnie blokują innych członków drużyny. Dlatego możemy wykluczać ataki na pola *wrogowie_obok2*, jeśli znajdują się na liście wykonanych ruchów.

[Robots that attack two moves away often form a perimeter around the enemy (a good thing) but it prevents your own bots from run across the line. For that reason we can choose to not let a robot do an an adjacent_enemy2 attack if they

are sitting in a taken spot.]

```
elif wrogowie_obok2 and self.location not in wybrane_pola:
```

Na koniec podmieniamy kod zwracający ruchy:

```
ruch = ['move', mindist(bezpieczne, najblizszy_wrog)]
ruch = ['attack', wrogowie_obok.pop()]
```

– tak aby wykorzystywał nowe funkcje:

```
ruch = ruszaj(mindist(bezpieczne, najblizszy_wrog))
ruch = stoj('attack', wrogowie_obok.pop())
```

Warto pamiętać, że roboty nie mogą zamieniać się miejscami. Wprawdzie jest możliwe zakodowanie tego, ale zamiana nie dojdzie do skutku.

Atakuj najsłabszego wroga

Każdy udany atak zmniejsza punkty HP wrogów tak samo, ale wynik gry zależy od liczby pozostałych przy życiu robotów, a nie od ich żywotności. Dlatego korzystniejsze jest wyeliminowanie słabego bota niż atakowanie/osłabienie silnego. Odpowiednią funkcję umieścimy w funkcji `Robot.act` i użyjemy do wyboru robota z listy zamiast dotychczasowej funkcji `.pop()`, która zwracała losowe roboty.

```
# funkcja znajdująca najsłabszego robota

def minhp(bots):
    return min(bots, key=lambda x: game.robots[x].hp)

elif wrogowie_obok:
    ...
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Samobójstwo lepsze niż śmierć

Na razie usiłujemy uciec, jeżeli grozi nam śmierć, ale czasami może się nam nie udać, bo natkniemy się na atakującego wroga. Jeżeli brak bezpiecznego ruchu, a grozi nam śmierć, o ile pozostaniemy w miejscu, możemy popełnić samobójstwo, co osłabi wrogów bardziej niż atak.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        if bezpieczne:
            ruch = ruszaj(mindist(safe, rg.CENTER_POINT))
        else:
            ruch = stoj('suicide')
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Unikaj nierównych starć

W walce jeden na jednego nikt nie ma przewagi, ponieważ wróg może odpowiadać atakiem na każdy nasz atak, jeżeli jesteśmy obok. Ale gdy wróg ma liczebną przewagę, atakując dwoma robotami naszego jednego, dostaniemy

podwójnie za każdy wyprowadzony atak. Dlatego należy uciekać, jeśli wrogów jest więcej. Warto zauważyć, że jest to kluczowa zasada w dążeniu do zwycięstwa w *Grze robotów*, nawet w rozgrywkach na najwyższym poziomie. Walka z wykorzystaniem przewagi jest zresztą warunkiem wygranej w większości pojedynków.

```
elif wrogowie_obok:
    if 9*len(wrogowie_obok) >= self.hp:
        ...
    elif len(wrogowie_obok) > 1:
        if bezpieczne:
            ruch = ruszaj(mindist(safe, rg.CENTER_POINT))
    else:
        ruch = stoj('attack', minhp(wrogowie_obok))
```

Goń słabe roboty

Możemy założyć, że słabe roboty będą uciekać. Zamiast atakować podczas ucieczki, powinniśmy je gonić. W ten sposób możemy wymusić kolejny ruch w następnej turze, dzięki czemu trafią być może w gorsze miejsce. Bierzemy pod uwagę roboty, które mają maksymalnie 5 punktów HP, nawet gdy zaatakują zamiast uciekać, zginą w wyniku uszkodzeń z powodu kolizji.

```
elif wrogowie_obok:
    ...
    else:
        cel = minhp(wrogowie_obok)
        if game.robots[cel].hp <= 5:
            ruch = ruszaj(cel)
        else:
            ruch = stoj('attack', minhp(wrogowie_obok))
```

Trzeba pamiętać, że startegia gonienia słabego robota ma jedną oczywistą wadę. Jeżeli słaby robot wybierzez obronę, goniący odniesie uszkodzenia z powodu kolizji, broniący nie. Można temu przeciwdziałać wybierając atak, a nie pogoń – koło się zamyka.

Podsumowanie

Poniżej zestawienie reguł, które dodaliśmy:

- Śledź wybierane miejsca
- Atakuj najsłabszego wroga
- Samobójstwo lepsze niż śmierć
- Unikaj nierównych starć
- Goń słabe roboty

Dodanie powyższych zmian umożliwi stworzenie robota podobnego do *simplebot* z pakietu open-source. Sprawdź jego kod, aby ulepszyć swojego. Do tej pory tworzyliśmy robota walczącego według zbioru kilku reguł, ale w następnym materiale poznamy roboty inaczej decydujące o ruchach, dodatkowo wykorzystujące kilka opartych na zasadach sztuczek.

Jeśli jesteś gotów, sprawdź “Zaawansowane strategie” (już wkrótce...)

Informacja: Niniejsza dokumentacja jest swobodnym i nieautoryzowanym tłumaczeniem materiałów dostępnych na stonie Robotgame Intermediate Strategy.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Informacja: Niniejsza dokumentacja jest nieautoryzowanym tłumaczeniem oficjalnej dokumentacji dostępnej na stronie *RobotGame* oraz materiałów dodatkowych dostępnych na stronie robotgame robots and scripts.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.4 Gry w Pythonie

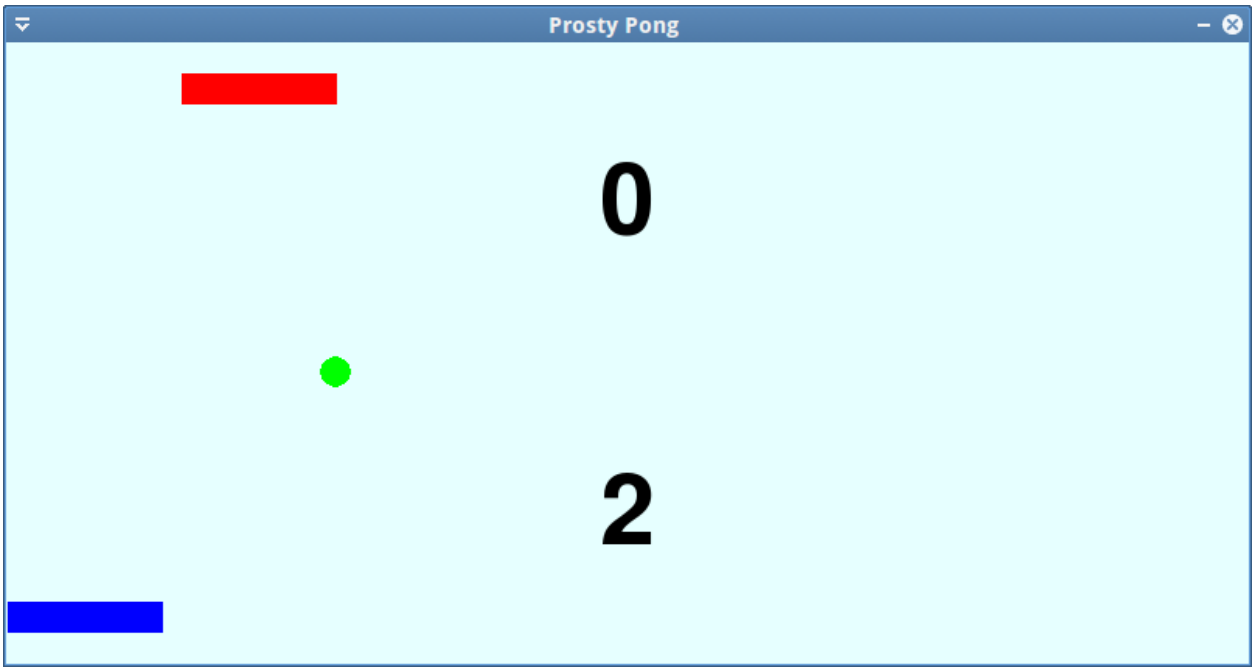
Pygame to zbiór modułów w języku Python wpomagających tworzenie aplikacji multimedialnych, zwłaszcza gier. Wykorzystuje możliwości biblioteki *SDL (Simple DirectMedia Layer)*, jest darmowy i rozpowszechniany na licencji GNU General Public Licence. Działa na wielu platformach i systemach operacyjnych.

Zobacz, jak **zainstalować** *PyGame* w systemie *Windows* i *Linuks*.

Informacja: Poniżej prezentujemy trzy gry zaimplementowane strukturalnie (str) i obiektowo (obj). Być może warto zacząć od wersji strukturalnych, następnie polecamy porównanie z wersjami obiektowymi.

Pong (str)

Wersja strukturalna klasycznej gry w odbijanie piłeczki zrealizowana z użyciem biblioteki PyGame.



Pole gry

Tworzymy plik `pong_str.py` w terminalu lub w wybranym edytorze, zapisujemy na dysku i wprowadzamy poniższy kod:

```
#! /usr/bin/env python2
# -*- coding: utf-8 -*-

import pygame
import sys
from pygame.locals import *

# inicjacja modułu pygame
pygame.init()

# szerokość i wysokość okna gry
OKNOGRY_SZER = 800
OKNOGRY_WYS = 400
# kolor okna gry, składowe RGB zapisane w tupli
LT_BLUE = (230, 255, 255)

# powierzchnia do rysowania, czyli inicjacja pola gry
oknogry = pygame.display.set_mode((OKNOGRY_SZER, OKNOGRY_WYS), 0, 32)
# tytuł okna gry
pygame.display.set_caption('Prosty Pong')

# pętla główna programu
while True:
    # obsługa zdarzeń generowanych przez gracza
    for event in pygame.event.get():
        # przechwyć zamknięcie okna
        if event.type == QUIT:
            pygame.quit()
            sys.exit()
```

```
30
31      # rysowanie obiektów
32      oknogry.fill(LT_BLUE)  # kolor okna gry
33
34      # zaktualizuj okno i wyświetl
35      pygame.display.update()
36
37  # KONIEC
```

Na początku importujemy wymagane biblioteki i inicjujemy moduł `pygame`. Dużymi literami zapisujemy nazwy zmiennych określające właściwości pola gry, które inicjalizujemy w instrukcji `pygame.display.set_mode()`. Tworzy ona powierzchnię o wymiarach 800x400 pikseli i 32 bitowej głębi kolorów, na której umieszczać będziemy pozostałe obiekty. W kolejnej instrukcji ustawiamy tytuł okna gry.

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w tzw. **głównej pętli**, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (np. obliczanie przesunięć elementów) i rysowanie go.

Zadanie z punktu *a)* realizuje pętla `for`, która odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Za pomocą instrukcji warunkowych możemy przechwytywać zdarzenia, które chcemy obsłużyć, np. naciśnięcie przycisku zamknięcia okna: `if event.type == QUIT`.

Instrukcja `oknogry.fill(BLUE)` wypełnia okno zdefiniowanym kolorem. Jego wyświetlenie następuje w poleceniu `pygame.display.update()`.

Uruchom aplikację, wydając w terminalu polecenie:

```
~$ python pong_str.py
```

Paletka gracza

Planszę gry już mamy, pora umieścić na niej paletkę gracza. Poniższy kod wstawiamy **przed pętlą główną** programu:

```
22  # paletka gracza ########################################################
23  PALETKA_SZER = 100  # szerokość
24  PALETKA_WYS = 20  # wysokość
25  BLUE = (0, 0, 255)  # kolor wypełnienia
26  PALETKA_1_POZ = (350, 360)  # początkowa pozycja zapisana w tupli
27  # utworzenie powierzchni paletki, wypełnienie jej kolorem,
28  paletka1 = pygame.Surface([PALETKA_SZER, PALETKA_WYS])
29  paletka1.fill(BLUE)
30  # ustawienie prostokąta zawierającego paletkę w początkowej pozycji
31  paletka1_prost = paletka1.get_rect()
32  paletka1_prost.x = PALETKA_1_POZ[0]
33  paletka1_prost.y = PALETKA_1_POZ[1]
```

Elementy graficzne tworzymy za pomocą polecenia `pygame.Surface((szerokosc, wysokosc), flagi, głębia)`. Utworzony obiekt możemy wypełnić kolorem: `.fill(kolor)`. Położenie obiektu określimy pobierając na początku prostokątny obszar (*Rect*), który go reprezentuje, metodą `get_rect()`. Następnie podajemy współrzędne `x` i `y` wyznaczające położenie w poziomie i pionie.

Informacja:

- Początek układu współrzędnych w *Pygame* to lewy górny róg okna głównego.
- Położenie obiektu można ustawić również podając nazwane argumenty: `obiekt_prost = obiekt.get_rect(x = 350, y =350)`.
- Położenie obiektów klasy `Rect` (prostokątów) możemy odczytywać wykorzystując właściwości, takie jak: `.x, .y, .centerx, .right, .left, .top, .bottom`.

Omówiony kod utworzy obiekt reprezentujący paletkę gracza, ale trzeba ją jeszcze umieścić na planszy gry. W tym celu użyjemy metody `.blit()`, która służy rysowaniu jednego obrazka na drugim. Poniższy kod musimy wstawić w pętli głównej przed instrukcją wyświetlającą okno.

```
47      # narysuj w oknie gry paletki
48      oknogry.blit(paletka1, paletka1_prost)
```

Pozostaje uruchomienie kodu.

Ruch paletki

W pętli przechwytującej zdarzenia dopisujemy zaznaczony poniżej kod:

```
35  # pętla główna programu
36  while True:
37      # obsługa zdarzeń generowanych przez gracza
38      for event in pygame.event.get():
39          # przechwyć zamknięcie okna
40          if event.type == QUIT:
41              pygame.quit()
42              sys.exit()
43
44          # przechwyć ruch myszy
45          if event.type == MOUSEMOTION:
46              myszaX, myszaY = event.pos  # współrzędne x, y kursora myszy
47
48              # oblicz przesunięcie paletki gracza
49              przesuniecie = myszaX - (PALETKA_SZER / 2)
50
51              # jeżeli wykraczamy poza okno gry w prawo
52              if przesuniecie > OKNOGRY_SZER - PALETKA_SZER:
53                  przesuniecie = OKNOGRY_SZER - PALETKA_SZER
54              # jeżeli wykraczamy poza okno gry w lewo
55              if przesuniecie < 0:
56                  przesuniecie = 0
57              # zaktualizuj położenie paletki w poziomie
58              paletka1_prost.x = przesuniecie
59
60      # rysowanie obiektów
61      oknogry.fill(LT_BLUE)  # kolor okna gry
62
63      # narysuj w oknie gry paletki
64      oknogry.blit(paletka1, paletka1_prost)
65
66      # zaktualizuj okno i wyświetl
67      pygame.display.update()
```

Chcemy sterować paletką za pomocą myszy. Zadaniem powyższego kodu jest przechwycenie jej ruchu (`MOUSEMOTION`), odczytanie współrzędnych kursora z tupli `event.pos` i obliczenie przesunięcia określającego

nowe położenie paletki. Kolejne instrukcje warunkowe korygują nową pozycję paletki, jeśli wykraczamy poza granice pola gry.

Przetestuj kod.

Piłka w grze

Piłkę tworzymy podobnie jak paletkę. Przed pętlą główną programu wstawiamy poniższy kod:

```
35 # piłka ######################################################################
36 P_SZER = 20  # szerokość
37 P_WYS = 20  # wysokość
38 P_PREDKOSC_X = 6  # prędkość pozioma x
39 P_PREDKOSC_Y = 6  # prędkość pionowa y
40 GREEN = (0, 255, 0)  # kolor piłki
41 # utworzenie powierzchni piłki, narysowanie piłki i wypełnienie kolorem
42 pilka = pygame.Surface([P_SZER, P_WYS], pygame.SRCALPHA, 32).convert_alpha()
43 pygame.draw.ellipse(pilka, GREEN, [0, 0, P_SZER, P_WYS])
44 # ustawienie prostokąta zawierającego piłkę w początkowej pozycji
45 pilka_prost = pilka.get_rect()
46 pilka_prost.x = OKNOGRY_SZER / 2
47 pilka_prost.y = OKNOGRY_WYS / 2
48
49 # ustawienia animacji #############################################
50 FPS = 30  # liczba klatek na sekundę
51 fpsClock = pygame.time.Clock()  # zegar śledzący czas
```

Przy tworzeniu powierzchni dla piłki używamy flagi `SRCALPHA`, co oznacza, że obiekt graficzny będzie zawierał przezroczyste piksele. Samą piłkę rysujemy za pomocą instrukcji `pygame.draw.ellipse(powierzchnia, kolor, prostokąt)`. Ostatni argument to lista zawierająca współrzędne lewego górnego i prawego dolnego rogu prostokąta, w który wpisujemy piłkę.

Ruch piłki, aby był płynny, wymaga użycia animacji. Ustawiamy więc liczbę generowanych klatek na sekundę (`FPS = 30`) i przygotowujemy obiekt zegara, który będzie kontrolował czas.

Teraz **pod pętlą** (nie w pętli!) `for`, która przechwytuje zdarzenia, umieszczamy kod:

```
78     # ruch piłki ###############################################
79     # przesuń piłkę po obsłużeniu zdarzeń
80     pilka_prost.move_ip(P_PREDKOSC_X, P_PREDKOSC_Y)
81
82     # jeżeli piłka wykracza poza pole gry
83     # z lewej/prawej - odwracamy kierunek ruchu poziomego piłki
84     if pilka_prost.right >= OKNOGRY_SZER:
85         P_PREDKOSC_X *= -1
86     if pilka_prost.left <= 0:
87         P_PREDKOSC_X *= -1
88
89     if pilka_prost.top <= 0:  # piłka uciekła górą
90         P_PREDKOSC_Y *= -1  # odwracamy kierunek ruchu pionowego piłki
91
92     if pilka_prost.bottom >= OKNOGRY_WYS:  # piłka uciekła dołem
93         pilka_prost.x = OKNOGRY_SZER / 2  # więc startuję ze środka
94         pilka_prost.y = OKNOGRY_WYS / 2
95
96     # jeżeli piłka dotknie paletki gracza, skieruj ją w przeciwną stronę
97     if pilka_prost.colliderect(paletka1_prost):
98         P_PREDKOSC_Y *= -1
```

```
99          # zapobiegaj przysłanianiu paletki przez piłkę
100         pilka_prost.bottom = paletka1_prost.top
```

Na uwagę zasługuje metoda `.move_ip(offset, offset)`, która przesuwa prostokąt zawierający piłkę o podane jako `offset` wartości. Dalej decydujemy, co ma się dziać, kiedy piłka wyjdzie poza pole gry. Metoda `.colliderect(prostokąt)` pozwala sprawdzić, czy dwa obiekty nachodzą na siebie. Dzięki temu możemy odwrócić bieg piłeczki po jej zetknięciu się z paletką gracza.

Piłkę trzeba umieścić na polu gry. Podaną niżej instrukcję umieszczamy poniżej polecenia rysującego paletkę gracza:

```
108     # narysuj w oknie piłkę
109     oknogry.blit(pilka, pilka_prost)
```

Na koniec ograniczamy prędkość animacji wywołując metodę `.tick(fps)`, która wstrzymuje wykonywanie programu na podaną jako argument liczbę klatek na sekundę. Podany niżej kod trzeba dopisać na końcu w pętli głównej:

```
114     # zaktualizuj zegar po narysowaniu obiektów
115     fpsClock.tick(FPS)
```

Teraz możesz już zagrać sam ze sobą! Przetestuj działanie programu.

AI – przeciwnik

Dodamy do gry przeciwnika AI (ang. *artificial inteligence*), czyli paletkę sterowaną programowo.

Przed główną pętlą programu dopisujemy kod tworzący paletkę AI:

```
53  # paletka ai ###############################################################
54  RED = (255, 0, 0)
55  PALETKA_AI_POZ = (350, 20)  # początkowa pozycja zapisana w tupli
56  # utworzenie powierzchni paletki, wypełnienie jej kolorem,
57  paletkaAI = pygame.Surface([PALETKA_SZER, PALETKA_WYS])
58  paletkaAI.fill(RED)
59  # ustawienie prostokąta zawierającego paletkę w początkowej pozycji
60  paletkaAI_prost = paletkaAI.get_rect()
61  paletkaAI_prost.x = PALETKA_AI_POZ[0]
62  paletkaAI_prost.y = PALETKA_AI_POZ[1]
63  # szybkość paletki AI
64  PREDKOSC_AI = 5
```

Tu nie ma nic nowego, więc od razu przed instrukcją wykrywającą kolizję piłki z paletką gracza (`if pilka_prost.colliderect(paletka1_prost)`) dopisujemy kod sterujący ruchem paletki AI:

```
111     # AI (jak gra komputer) ######################################################
112     # jeżeli piłka ucieka na prawo, przesuń za nią paletkę
113     if pilka_prost.centerx > paletkaAI_prost.centerx:
114         paletkaAI_prost.x += PREDKOSC_AI
115     # w przeciwnym wypadku przesuń w lewo
116     elif pilka_prost.centerx < paletkaAI_prost.centerx:
117         paletkaAI_prost.x -= PREDKOSC_AI
118
119     # jeżeli piłka dotknie paletki AI, skieruj ją w przeciwną stronę
120     if pilka_prost.colliderect(paletkaAI_prost):
121         P_PREDKOSC_Y *= -1
122         # uwzględnij nachodzenie paletki na piłkę (przysłonięcie)
123         pilka_prost.top = paletkaAI_prost.bottom
```

Samą paletkę AI trzeba umieścić na planszy, po instrukcji rysującej paletkę gracza dopisujemy więc:

```
```

```
    # narysuj w oknie gry paletki
    oknogry.blit(paletka1, paletka1_prost)
    oknogry.blit(paletkaAI, paletkaAI_prost)
```

Pozostaje zmienić kod odpowiedzialny za odbijanie piłki od górnej krawędzi planszy (`if pilka_prost.top <= 0`), żeby przeciwnik AI mógł przegrywać. W tym celu dokonujemy zmian wg poniższego kodu:

```
    if pilka_prost.top <= 0:  # piłka uciekła górą
        #  P_PREDKOSC_Y *= -1  # odwracamy kierunek ruchu pionowego piłki
        pilka_prost.x = OKNOGRY_SZER / 2  # więc startuję ze środka
        pilka_prost.y = OKNOGRY_WYS / 2
```

Teraz można już zagrać z komputerem :-).

Liczymy punkty

Co to za gra, w której nie wiadomo, kto wygrywa... Dodamy kod zliczający i wyświetlający punkty. Przed główną pętlą programu wstawiamy poniższy kod:

```
# komunikaty tekstowe ###################################################
# zmienne przechowujące punkty i funkcje wyświetlające punkty
PKT_1 = '0'
PKT_AI = '0'
fontObj = pygame.font.Font('freesansbold.ttf', 64)  # czcionka komunikatów


def drukuj_punkty1():
    tekst1 = fontObj.render(PKT_1, True, (0, 0, 0))
    tekst_prost1 = tekst1.get_rect()
    tekst_prost1.center = (OKNOGRY_SZER / 2, OKNOGRY_WYS * 0.75)
    oknogry.blit(tekst1, tekst_prost1)


def drukuj_punktyAI():
    tekstAI = fontObj.render(PKT_AI, True, (0, 0, 0))
    tekst_prostAI = tekstAI.get_rect()
    tekst_prostAI.center = (OKNOGRY_SZER / 2, OKNOGRY_WYS / 4)
    oknogry.blit(tekstAI, tekst_prostAI)
```

Po zdefiniowaniu zmiennych przechowujących punkty graczy, tworzymy obiekt czcionki z podanego pliku (`pygame.font.Font()`). Następnie definiujemy funkcje, których zadaniem jest rysowanie punktacji graczy. Na początku tworzą one nowe obrazki z punktacją gracza (`.render()`), pobierają ich prostokąty (`.get_rect()`), pozycjonują je (`.center()`) i rysują na głównej powierzchni gry (`.blit()`).

Informacja: Plik wykorzystywany do wyświetlania tekstu (`freesansbold.ttf`) musi znaleźć się w katalogu ze skryptem.

W pętli głównej programu musimy umieścić wyrażenia zliczające punkty. Jeżeli piłka ucieknie górą, punkty dostaje gracz, w przeciwnym wypadku AI. Dopisz podświetlone instrukcje:

```
122     if pilka_prost.top <= 0:  # piłka uciekła górą
123         # P_PREDKOSC_Y *= -1  # odwracamy kierunek ruchu pionowego piłki
124         pilka_prost.x = OKNOGRY_SZER / 2  # więc startuję ze środka
125         pilka_prost.y = OKNOGRY_WYS / 2
126         PKT_1 = str(int(PKT_1) + 1)
127 
128     if pilka_prost.bottom >= OKNOGRY_WYS:  # piłka uciekła dołem
129         pilka_prost.x = OKNOGRY_SZER / 2  # więc startuję ze środka
130         pilka_prost.y = OKNOGRY_WYS / 2
131         PKT_AI = str(int(PKT_AI) + 1)
```

Obie funkcje wyświetlające punkty również trzeba wywołać z pętli głównej, a więc po instrukcji wypełniającej okno gry kolorem (`oknogry.fill(LT_BLUE)`) dopisujemy:

```
153     # rysowanie obiektów ################################################
154     oknogry.fill(LT_BLUE)  # wypełnienie okna gry kolorem
155 
156     drukuj_punkty1()  # wyświetl punkty gracza
157     drukuj_punktyAI()  # wyświetl punkty AI
```

Sterowanie klawiszami

Skoro możemy przechwytywać ruch myszy, nic nie stoi na przeszkodzie, aby umożliwić poruszanie paletką za pomocą klawiszy. W pętli `for` odczytującej zdarzenia dopisujemy:

```
114         # przechwyć naciśnięcia klawiszy kursora
115         if event.type == pygame.KEYDOWN:
116             if event.key == pygame.K_LEFT:
117                 paletka1_prost.x -= 5
118                 if paletka1_prost.x < 0:
119                     paletka1_prost.x = 0
120             if event.key == pygame.K_RIGHT:
121                 paletka1_prost.x += 5
122                 if paletka1_prost.x > OKNOGRY_SZER - PALETKA_SZER:
123                     paletka1_prost.x = OKNOGRY_SZER - PALETKA_SZER
```

Naciśnięcie klawisza generuje zdarzenie `pygame.KEYDOWN`. Dalej w instrukcji warunkowej sprawdzamy, czy naciśnięto klawisz kursora lewy lub prawy i przesuwamy paletkę o 5 pikseli.

Wskazówka: Kody klawiszy możemy sprawdzić w dokumentacji *Pygame*.

Uruchom program i sprawdź, jak działa. Szybko zauważysz, że wciśnięcie strzałki porusza paletką, ale żeby poruszyła się znowu, trzeba naciskanie powtarzać. To niewygodne, paletka powinna ruszać się dopóki klawisz jest wciśnięty. Przed pętlą główną dodamy więc poniższy kod:

```
86 # powtarzalność klawiszy (delay, interval)
87 pygame.key.set_repeat(50, 25)
```

Dzięki tej instrukcji włączyliśmy powtarzalność wciśnięć klawiszy. Przetestuj, czy działa.

Zadania dodatkowe

- Zmodyfikuj właściwości obiektów (paletek, piłki) takie jak rozmiar, kolor, początkowa pozycja.

- Zmień położenie paletek tak, aby znalazły się przy lewej i prawej krawędzi okna, wprowadź potrzebne zmiany w kodzie, aby poruszały się w pionie.
- Dodaj trzecią paletkę, która co jakiś czas będzie "przelatywać" przez środek planszy i zmieniać w przypadku kolizji tor i kolor piłki.

Materiały

Źródła:

- `pong_str.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Pong (obj)

Klasyczna gra w odbijanie piłeczki zrealizowana z użyciem biblioteki PyGame. Wersja obiektowa. Biblioteka PyGame ułatwia tworzenie aplikacji multimedialnych, w tym gier.

Przygotowanie

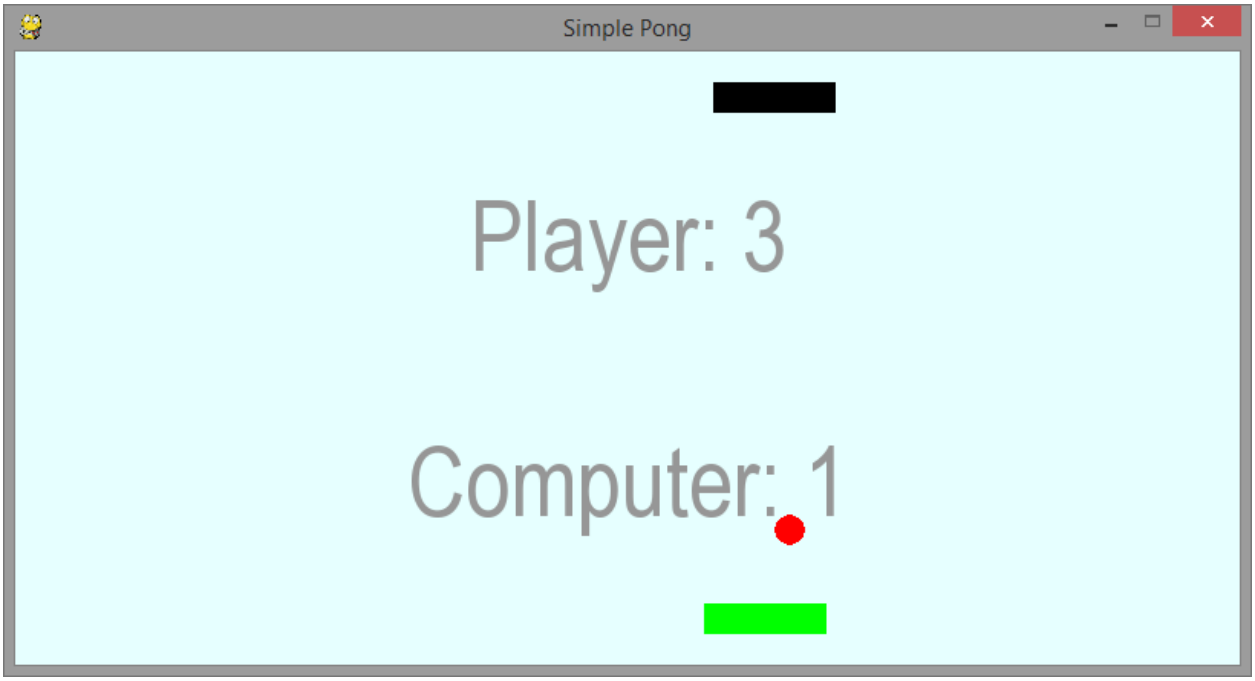
Do rozpoczęcia pracy z przykładem pobieramy szczątkowy kod źródłowy:

```
~/python101$ git checkout -f pong/z1
```

Okienko gry

Na wstępie w pliku `~/python101/games/pong.py` otrzymujemy kod który przygotuje okienko naszej gry:

```
1  # coding=utf-8
2
3  import pygame
4  import pygame.locals
5
6
7  class Board(object):
```




```
    """
    Plansza do gry. Odpowiada za rysowanie okna gry.
    """

    def __init__(self, width, height):
        """
        Konstruktor planszy do gry. Przygotowuje okienko gry.

        :param width:
        :param height:
        """
        self.surface = pygame.display.set_mode((width, height), 0, 32)
        pygame.display.set_caption('Simple Pong')

    def draw(self, *args):
        """
        Rysuje okno gry

        :param args: lista obiektów do narysowania
        """
        background = (230, 255, 255)
        self.surface.fill(background)
        for drawable in args:
            drawable.draw_on(self.surface)

        # dopiero w tym miejscu następuje fatyczne rysowanie
        # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
        pygame.display.update()


board = Board(800, 400)
board.draw()
```

W powyższym kodzie zdefiniowaliśmy klasę `Board` z dwiema metodami:

1. konstruktorem `__init__`, oraz
2. metodą `draw` posługującą się biblioteką `PyGame` do rysowania w oknie.

Na końcu utworzyliśmy instancję klasy `Board` i wywołaliśmy jej metodę `draw` na razie bez żadnych elementów wymagających narysowania.

Informacja: Każdy plik skryptu *Python* jest uruchamiany w momencie importu — plik/moduł główny jest importowany jako pierwszy.

Deklaracje klas są faktycznie instrukcjami sterującymi mówiącymi, by w aktualnym module utworzyć typy zawierające wskazane definicje.

Możemy mieszać deklaracje klas ze zwykłymi instrukcjami sterującymi takimi jak `print`, czy przypisaniem wartości zmiennej `board = Board(800, 400)` i następnie wywołaniem metody na obiekcie `board.draw()`.

Nasz program możemy uruchomić komendą:

```
~/python101$ python games/pong.py
```

Mrugnęło? Program się wykonał i zakończył działanie :). Żeby zobaczyć efekt na dłużej, możemy na końcu chwilkę uśpić nasz program:

```
39 import time
40 time.sleep(5)
```

Jednak zamiast tego, dla lepszej kontroli powinniśmy zadeklarować klasę kontrolera gry, usuńmy kod od linii 37 do końca i dodajmy klasę kontrolera:

```
38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height):
44         pygame.init()
45         self.board = Board(width, height)
46         # zegar którego użyjemy do kontrolowania szybkości rysowania
47         # kolejnych klatek gry
48         self.fps_clock = pygame.time.Clock()
49
50     def run(self):
51         """
52         Główna pętla programu
53         """
54         while not self.handle_events():
55             # działaj w pętli do momentu otrzymania sygnału do wyjścia
56             self.board.draw()
57             self.fps_clock.tick(30)
58
59     def handle_events(self):
60         """
61         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
62
63         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
64         """
```

```
        for event in pygame.event.get():
            if event.type == pygame.locals.QUIT:
                pygame.quit()
                return True


# Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
# chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
if __name__ == "__main__":
    game = PongGame(800, 400)
    game.run()
```

Informacja: Prócz dodania kontrolera zmieniliśmy także sposób, w jaki gra jest uruchamiana — nie mylić z uruchomieniem programu.

Na końcu dodaliśmy instrukcję warunkową `if __name__ == "__main__":`, w niej sprawdzamy, czy nasz moduł jest modułem głównym programu, jeśli nim jest, gra zostanie uruchomiona.

Dzięki temu, jeśli nasz moduł zostałby zaimportowany gdzieś indziej instrukcją `import pong`, deklaracje klas wykonałyby się, ale sama gra nie zostanie uruchomiona.

Gotowy kod możemy pobrać komendą:

```
~/python101$ git checkout -f pong/z2
```

Piłeczka

Czas dodać piłkę do gry. *Piłeczką* będzie kolorowe kółko, które z każdym przejściem naszej pętli przesuniemy o kilka punktów w osi X i Y, zgodnie wektorem prędkości.

Wcześniej jednak zdefiniujemy wspólną klasę bazową dla obiektów, które będziemy rysować w oknie naszej gry:

```
class Drawable(object):
    """
    Klasa bazowa dla rysowanych obiektów
    """

    def __init__(self, width, height, x, y, color=(0, 255, 0)):
        self.width = width
        self.height = height
        self.color = color
        self.surface = pygame.Surface([width, height], pygame.SRCALPHA, 32).convert_
↪alpha()
        self.rect = self.surface.get_rect(x=x, y=y)

    def draw_on(self, surface):
        surface.blit(self.surface, self.rect)
```

Następnie dodajmy klasę samej piłeczki dziedzicząc z `Drawable`:

```
class Ball(Drawable):
    """
    Piłeczka, sama kontroluje swoją prędkość i kierunek poruszania się.
    """
    def __init__(self, width, height, x, y, color=(255, 0, 0), x_speed=3, y_speed=3):
```

```
92          super(Ball, self).__init__(width, height, x, y, color)
93          pygame.draw.ellipse(self.surface, self.color, [0, 0, self.width, self.height])
94          self.x_speed = x_speed
95          self.y_speed = y_speed
96          self.start_x = x
97          self.start_y = y
98
99      def bounce_y(self):
100         """
101         Odwraca wektor prędkości w osi Y
102         """
103         self.y_speed *= -1
104
105     def bounce_x(self):
106         """
107         Odwraca wektor prędkości w osi X
108         """
109         self.x_speed *= -1
110
111     def reset(self):
112         """
113         Ustawia piłeczkę w położeniu początkowym i odwraca wektor prędkości w osi Y
114         """
115         self.rect.move(self.start_x, self.start_y)
116         self.bounce_y()
117
118     def move(self):
119         """
120         Przesuwa piłeczkę o wektor prędkości
121         """
122         self.rect.x += self.x_speed
123         self.rect.y += self.y_speed
```

W przykładzie powyżej wykonaliśmy *dziedziczenie* oraz *przesłanianie* konstruktora, ponieważ rozszerzamy `Drawable` i chcemy zachować efekt działania konstruktora na początku konstruktora `Ball` wywołujemy konstruktor klasy bazowej:

```
super(Ball, self).__init__(width, height, x, y, color)
```

Teraz musimy naszą piłeczkę zintegrować z resztą gry:

```
38  class PongGame(object):
39      """
40      Łączy wszystkie elementy gry w całość.
41      """
42
43      def __init__(self, width, height):
44          pygame.init()
45          self.board = Board(width, height)
46          # zegar którego użyjemy do kontrolowania szybkości rysowania
47          # kolejnych klatek gry
48          self.fps_clock = pygame.time.Clock()
49          self.ball = Ball(20, 20, width/2, height/2)
50
51      def run(self):
52          """
53          Główna pętla programu
54          """
```

```
55          while not self.handle_events():
56              # działaj w pętli do momentu otrzymania sygnału do wyjścia
57              self.ball.move()
58              self.board.draw(
59                  self.ball,
60              )
61              self.fps_clock.tick(30)
```

Informacja: Metoda `Board.draw` oczekuje wielu opcjonalnych argumentów, choć na razie przekazujemy tylko jeden. By zwiększyć czytelność potencjalnie dużej listy argumentów — kto wie co jeszcze dodamy :) — podajemy każdy argument w osobnej linii zakończonej przecinkiem `,`.

Python nie traktuje takich osieroconych przecinków jako błąd, jest to ukłon w stronę programistów, którzy często zmieniają kod, kopiują i wklejają kawałki.

Dzięki temu możemy wstawiać nowe i zmieniać kolejność bez zwracania uwagi, czy na końcu jest przecinek, czy go brakuje, czy go należy usunąć. Zgodnie z konwencją powinien być tam zawsze.

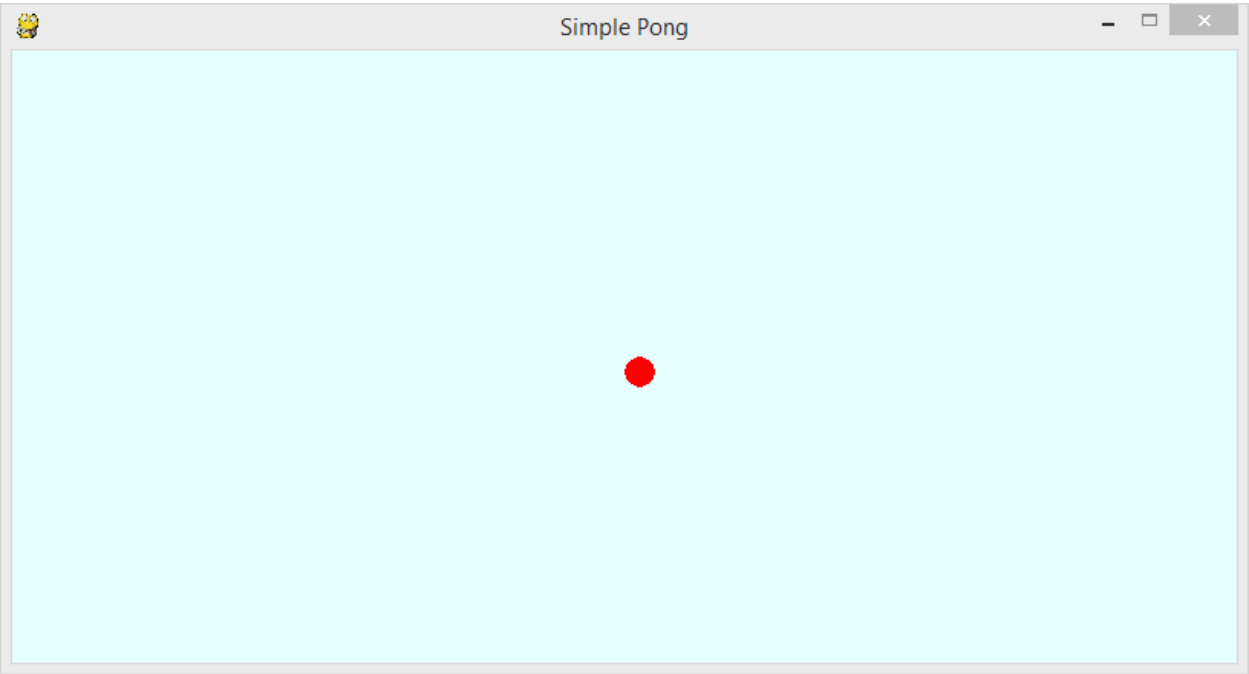
Gotowy kod możemy pobrać komendą:

```
~/python101$ git checkout -f pong/z3
```

Odbijanie piłeczki

Uruchommy naszą "grę" ;)

```
~/python101$ python games/pong.py
```



Efekt nie jest powalający, ale mamy już jakiś ruch na planszy. Szkoda, że piłka spada z planszy. Może mogła by się odbijać od krawędzi okienka? Możemy wykorzystać wcześniej przygotowane metody do zmiany kierunku wektora

prędkości, musimy tylko wykryć moment w którym piłeczka będzie dotykać krawędzi.

W tym celu piłeczka musi być świadoma istnienia planszy i pozycji krawędzi, dlatego zmodyfikujemy metodę `Ball.move` tak by przyjmowała `board` jako argument i na jego podstawie sprawdzimy, czy piłeczka powinna się odbijać:

```
122 def move(self, board):
123     """
124     Przesuwa piłeczkę o wektor prędkości
125     """
126     self.rect.x += self.x_speed
127     self.rect.y += self.y_speed
128
129     if self.rect.x < 0 or self.rect.x > board.surface.get_width():
130         self.bounce_x()
131
132     if self.rect.y < 0 or self.rect.y > board.surface.get_height():
133         self.bounce_y()
```

Jeszcze zmodyfikujmy wywołanie metody `move` w naszej pętli głównej:

```
51 def run(self):
52     """
53     Główna pętla programu
54     """
55     while not self.handle_events():
56         self.ball.move(self.board)
57         self.board.draw(
58             self.ball,
59         )
60         self.fps_clock.tick(30)
```

Ostrzeżenie: Powyższe przykłady mają o jedno wcięcie za mało. Poprawnie wcięte przykłady straciłyby kolorowanie w tej formie materiałów. Ze względu na czytelność kodu zdecydowaliśmy się na taki drobny błąd. Kod po ewentualnym wklejeniu należy poprawić dodając jedno wcięcie (4 spacje).

Sprawdzamy, czy piłka się odbija, uruchamiamy nasz program:

```
~/python101$ python games/pong.py
```

Gotowy kod możemy pobrać komendą:

```
~/python101$ git checkout -f pong/z4
```

Odbijamy piłeczkę rakietką

Dodajmy "rakietkę", przy pomocy której będziemy mogli odbijać piłeczkę. Będzie to zwykły prostokąt, który będziemy przesuwać za pomocą myszki.

```
136 class Racket(Drawable):
137     """
138     Rakietka, porusza się w osi X z ograniczeniem prędkości.
139     """
140
141     def __init__(self, width, height, x, y, color=(0, 255, 0), max_speed=10):
```

```
142         super(Racket, self).__init__(width, height, x, y, color)
143         self.max_speed = max_speed
144         self.surface.fill(color)
145
146     def move(self, x):
147         """
148         Przesuwa rakietkę w wyznaczone miejsce.
149         """
150         delta = x - self.rect.x
151         if abs(delta) > self.max_speed:
152             delta = self.max_speed if delta > 0 else -self.max_speed
153         self.rect.x += delta
```

Informacja: W tym przykładzie zastosowaliśmy operator warunkowy, który ogranicza prędkość poruszania się rakietki:

```
delta = self.max_speed if delta > 0 else -self.max_speed
```

Zmienna `delta` otrzyma wartość `max_speed` ze znakiem + lub − w zależności od znaku jaki ma aktualnie.

Następnie "pokażemy" rakietkę piłeczce, tak by mogła się od niej odbijać. Wiemy że rakietek będzie więcej, dlatego od razu tak zmodyfikujemy metodę `Ball.move`, by przyjmowała kolekcję rakietek:

```
122 def move(self, board, *args):
123     """
124     Przesuwa piłeczkę o wektor prędkości
125     """
126     self.rect.x += self.x_speed
127     self.rect.y += self.y_speed
128
129     if self.rect.x < 0 or self.rect.x > board.surface.get_width():
130         self.bounce_x()
131
132     if self.rect.y < 0 or self.rect.y > board.surface.get_height():
133         self.bounce_y()
134
135     for racket in args:
136         if self.rect.colliderect(racket.rect):
137             self.bounce_y()
```

Tak jak w przypadku dodawania piłeczki, rakietkę też trzeba dodać do "gry", dodatkowo musimy ją pokazać piłeczce:

```
38 class PongGame(object):
39     """
40     Łączy wszystkie elementy gry w całość.
41     """
42
43     def __init__(self, width, height):
44         pygame.init()
45         self.board = Board(width, height)
46         # zegar którego użyjemy do kontrolowania szybkości rysowania
47         # kolejnych klatek gry
48         self.fps_clock = pygame.time.Clock()
49         self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
50         self.player1 = Racket(width=80, height=20, x=width/2, y=height/2)
51
```

```
    def run(self):
        """
        Główna pętla programu
        """
        while not self.handle_events():
            # działaj w pętli do momentu otrzymania sygnału do wyjścia
            self.ball.move(self.board, self.player1)
            self.board.draw(
                self.ball,
                self.player1,
            )
            self.fps_clock.tick(30)

    def handle_events(self):
        """
        Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką

        :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
        """
        for event in pygame.event.get():
            if event.type == pygame.locals.QUIT:
                pygame.quit()
                return True

            if event.type == pygame.locals.MOUSEMOTION:
                # myszka steruje ruchem pierwszego gracza
                x, y = event.pos
                self.player1.move(x)
```

Gotowy kod możemy pobrać komendą:

```
~/python101$ git checkout -f pong/z5
```

Informacja: W tym miejscu można się pobawić naszą grą. Zmodyfikuj ją według uznania i podziel się rezultatem z innymi. Jeśli kod przestanie działać, można szybko cofnąć zmiany poniższą komendą.

```
~/python101$ git reset --hard
```

Gramy przeciwko komputerowi

Dodajemy przeciwnika, nasz przeciwnik będzie mistrzem, będzie dokładnie śledził piłeczkę i zawsze starał się utrzymać rakietkę gotową do odbicia piłeczki.

```

class Ai(object):
    """
    Przeciwnik, steruje swoją rakietką na podstawie obserwacji piłeczki.
    """
    def __init__(self, racket, ball):
        self.ball = ball
        self.racket = racket

    def move(self):
```

```
177         x = self.ball.rect.centerx
178         self.racket.move(x)
```

Tak jak w przypadku piłeczki i rakietki dodajemy nasze `Ai` do gry, a wraz nią drugą rakietkę. Rakietki ustawiamy na przeciwległych brzegach planszy.

Trzeba pamiętać, by pokazać drugą rakietkę piłeczce, tak by mogła się od niej odbijać.

```
38  class PongGame(object):
39      """
40      Łączy wszystkie elementy gry w całość.
41      """
42
43      def __init__(self, width, height):
44          pygame.init()
45          self.board = Board(width, height)
46          # zegar którego użyjemy do kontrolowania szybkości rysowania
47          # kolejnych klatek gry
48          self.fps_clock = pygame.time.Clock()
49          self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
50          self.player1 = Racket(width=80, height=20, x=width/2 - 40, y=height - 40)
51          self.player2 = Racket(width=80, height=20, x=width/2 - 40, y=20, color=(0, 0,␣
    ↪0))
52          self.ai = Ai(self.player2, self.ball)
53
54      def run(self):
55          """
56          Główna pętla programu
57          """
58          while not self.handle_events():
59              # działaj w pętli do momentu otrzymania sygnału do wyjścia
60              self.ball.move(self.board, self.player1, self.player2)
61              self.board.draw(
62                  self.ball,
63                  self.player1,
64                  self.player2,
65              )
66              self.ai.move()
67              self.fps_clock.tick(30)
```

Pokazujemy punkty

Dodajmy klasę sędziego, który patrząc na poszczególne elementy gry będzie decydował, czy graczom należą się punkty i będzie ustawiał piłkę w początkowym położeniu.

```
184
185
186 class Judge(object):
187     """
188     Sędzia gry
189     """
190
191     def __init__(self, board, ball, *args):
192         self.ball = ball
193         self.board = board
194         self.rackets = args
```

```
        self.score = [0, 0]

        # Przed pisaniem tekstów, musimy zainicjować mechanizmy wyboru fontów PyGame
        pygame.font.init()
        font_path = pygame.font.match_font('arial')
        self.font = pygame.font.Font(font_path, 64)

    def update_score(self, board_height):
        """
        Jeśli trzeba przydziela punkty i ustawia piłeczkę w początkowym położeniu.
        """
        if self.ball.rect.y < 0:
            self.score[0] += 1
            self.ball.reset()
        elif self.ball.rect.y > board_height:
            self.score[1] += 1
            self.ball.reset()

    def draw_text(self, surface,  text, x, y):
        """
        Rysuje wskazany tekst we wskazanym miejscu
        """
        text = self.font.render(text, True, (150, 150, 150))
        rect = text.get_rect()
        rect.center = x, y
        surface.blit(text, rect)

    def draw_on(self, surface):
        """
        Aktualizuje i rysuje wyniki
        """
        height = self.board.surface.get_height()
        self.update_score(height)

        width = self.board.surface.get_width()
        self.draw_text(surface, "Player: {}".format(self.score[0]), width/2, height *␣
↪0.3)
        self.draw_text(surface, "Computer: {}".format(self.score[1]), width/2, height␣
↪* 0.7)
```

Jak zwykle dodajemy instancję nowej klasy do gry:

```
class PongGame(object):
    """
    Łączy wszystkie elementy gry w całość.
    """

    def __init__(self, width, height):
        pygame.init()
        self.board = Board(width, height)
        # zegar którego użyjemy do kontrolowania szybkości rysowania
        # kolejnych klatek gry
        self.fps_clock = pygame.time.Clock()
        self.ball = Ball(width=20, height=20, x=width/2, y=height/2)
        self.player1 = Racket(width=80, height=20, x=width/2 - 40, y=height - 40)
        self.player2 = Racket(width=80, height=20, x=width/2 - 40, y=20, color=(0, 0,␣
↪0))
        self.ai = Ai(self.player2, self.ball)
```

```
53          self.judge = Judge(self.board, self.ball, self.player2, self.ball)
54
55      def run(self):
56          """
57          Główna pętla programu
58          """
59          while not self.handle_events():
60              # działaj w pętli do momentu otrzymania sygnału do wyjścia
61              self.ball.move(self.board, self.player1, self.player2)
62              self.board.draw(
63                  self.ball,
64                  self.player1,
65                  self.player2,
66                  self.judge,
67              )
68              self.ai.move()
69              self.fps_clock.tick(30)
70
```

Zadania dodatkowe

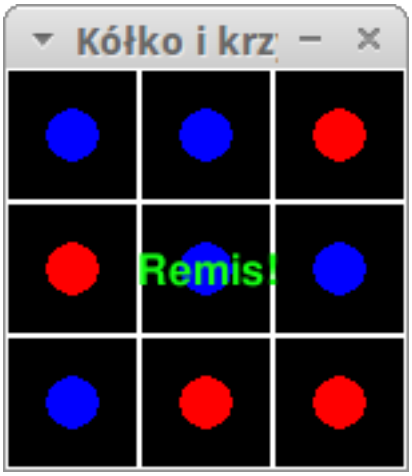
1. Piłeczka "odbija się" po zewnętrznej prawej i dolnej krawędzi. Można to poprawić.
2. Metoda `Ball.move` otrzymuje w argumentach planszę i rakietki. Te elementy można piłeczce przekazać tylko raz w konstruktorze.
3. Komputer nie odbija piłeczkę rogiem rakietki.
4. Rakietka gracza rusza się tylko, gdy gracz rusza myszką, ruch w stronę myszki powinien być kontynuowany także, gdy myszka jest bezczynna.
5. Gdy piłeczka odbija się od boków rakietki, powinna odbijać się w osi X.
6. Gra dwuosobowa z użyciem komunikacji po sieci.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Kółko i krzyżyk (str)

Klasyczna gra w kółko i krzyżyk zrealizowana przy pomocy PyGame.



Zmienne i plansza gry

Tworzymy plik `tictactoe.py` w terminalu lub w wybranym edytorze i zaczynamy od zdefiniowania zmiennych określających właściwości obiektów w naszej grze.

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import pygame
import sys
import random
from pygame.locals import *  # udostępnienie nazw metod z locals

# inicjacja modułu pygame
pygame.init()

# przygotowanie powierzchni do rysowania, czyli inicjacja okna gry
OKNOGRY = pygame.display.set_mode((150, 150), 0, 32)
# tytuł okna gry
pygame.display.set_caption('Kółko i krzyżyk')

# lista opisująca stan pola gry, 0 - pole puste, 1 - gracz, 2 - komputer
POLE_GRY = [0, 0, 0,
            0, 0, 0,
            0, 0, 0]

RUCH = 1  # do kogo należy ruch: 1 - gracz, 2 - komputer
WYGRANY = 0  # wynik gry: 0 - nikt, 1 - gracz, 2 - komputer, 3 - remis
WYGRANA = False
```

W instrukcji `pygame.display.set_mode()` inicjalizujemy okno gry o rozmiarach 150x150 pikseli i 32 bitowej głębi kolorów. Tworzymy w ten sposób powierzchnię główną do rysowania zapisaną w zmiennej `OKNOGRY`. `POLE_GRY` to lista elementów reprezentujących pola planszy, które mogą być puste (wartość 0), zawierać kółka gracza (wartość 1) lub komputera (wartość 2). Pozostałe zmienne określają, do kogo należy następny ruch, kto wygrał i czy nastąpił koniec gry.

Rysuj planszę gry

Planszę można narysować na wiele sposobów, np. tak:

```
WYGRANA = False

# rysowanie planszy gry, czyli linii oddzielających pola


def rysuj_plansze():
    for i in range(0, 3):  # x
        for j in range(0, 3):  # y
            # argumenty: powierzchnia, kolor, x,y, w,h, grubość linii
            pygame.draw.rect(OKNOGRY, (255, 255, 255),
                             Rect((j * 50, i * 50), (50, 50)), 1)

# narysuj kółka


def rysuj_pole_gry():
    for i in range(0, 3):
        for j in range(0, 3):
            pole = i * 3 + j  # zmienna pole przyjmuje wartości od 0-8
            # x i y określają środki kolejnych pól,
            # a więc wartości: 25,25, 25,75 25,125 75,25 itd.
            x = j * 50 + 25
            y = i * 50 + 25

            if POLE_GRY[pole] == 1:
                # rysuj kółko gracza
                pygame.draw.circle(OKNOGRY, (0, 0, 255), (x, y), 10)
            elif POLE_GRY[pole] == 2:
                # rysuj kółko komputera
                pygame.draw.circle(OKNOGRY, (255, 0, 0), (x, y), 10)
```

Pierwsza funkcja, `rysuj_plansze()`, wykorzystując zagnieżdżone pętle, rysuje nam 9 kwadratów o białym obramowaniu i szerokości 50 pikseli (formalnie są to obiekty *Rect* zwracane przez metodę `pygame.draw.rect()`). Zadaniem funkcji `rysuj_pole_gry()` jest narysowanie w zależności od stanu planszy gry zapisanego w liście `POLE_GRY` kółek o niebieskim (gracz) lub czerwonym (komputer) kolorze za pomocą metody `pygame.draw.circle()`.

Sztuczna inteligencja

Decydującą rolę w grze odgrywa komputer, od którego inteligencji zależy, czy rozgrywka przyniesie jakąś satysfakcję. Dopisujemy więc funkcje obsługujące sztuczną inteligencję:

```
            y = i * 50 + 25

            if POLE_GRY[pole] == 1:
                # rysuj kółko gracza
                pygame.draw.circle(OKNOGRY, (0, 0, 255), (x, y), 10)
            elif POLE_GRY[pole] == 2:
                # rysuj kółko komputera
                pygame.draw.circle(OKNOGRY, (255, 0, 0), (x, y), 10)

# postaw kółko lub krzyżyk (w tej wersji też kółko, ale w innym kolorze :-))


def postaw_znak(pole, RUCH):
    if POLE_GRY[pole] == 0:
```

```
60          if RUCH == 1:  # ruch gracza
61              POLE_GRY[pole] = 1
62              return 2
63          elif RUCH == 2:  # ruch komputera
64              POLE_GRY[pole] = 2
65              return 1
66
67      return RUCH
68
69  # funkcja pomocnicza sprawdzająca, czy komputer może wygrać, czy powinien
70  # blokować gracza, czy może wygrał komputer lub gracz
71
72
73  def sprawdz_pola(uklad, wygrany=None):
74      wartosc = None
75      # lista wielowymiarowa, której elementami są inne listy zagnieżdżone
76      POLA_INDEKSY = [  # trójki pól planszy do sprawdzania
77          [0, 1, 2], [3, 4, 5], [6, 7, 8],  # indeksy pól w poziomie (wiersze)
78          [0, 3, 6], [1, 4, 7], [2, 5, 8],  # indeksy pól w pionie (kolumny)
79          [0, 4, 8], [2, 4, 6]  # indeksy pól na skos (przekątne)
80      ]
81
82      for lista in POLA_INDEKSY:
83          kol = []  # lista pomocnicza
84          for ind in lista:
85              kol.append(POLE_GRY[ind])  # zapisz wartość odczytaną z POLE_GRY
86          if (kol in uklad):  # jeżeli znalazłeś układ wygrywający lub blokujący
87              # zwróć wygranego (1,2) lub indeks pola do zaznaczenia
88              wartosc = wygrany if wygrany else lista[kol.index(0)]
89
90      return wartosc
91
92  # ruchy komputera
93
94
95  def ai_ruch(RUCH):
96      pole = None  # które pole powinien zaznaczyć komputer
97
98      # listy wielowymiarowe, których elementami są inne listy zagnieżdżone
99      uklady_wygrywam = [[2, 2, 0], [2, 0, 2], [0, 2, 2]]
100     uklady_blokuje = [[1, 1, 0], [1, 0, 1], [0, 1, 1]]
101
102     # sprawdź, czy komputer może wygrać
103     pole = sprawdz_pola(uklady_wygrywam)
104     if pole is not None:
105         return postaw_znak(pole, RUCH)
106
107     # jeżeli komputer nie może wygrać, blokuj gracza
108     pole = sprawdz_pola(uklady_blokuje)
109     if pole is not None:
110         return postaw_znak(pole, RUCH)
111
112     # jeżeli nie można wygrać i gracza nie trzeba blokować, wylosuj pole
113     while pole is None:
114         pos = random.randrange(0, 9)  # wylosuj wartość od 0 do 8
115         if POLE_GRY[pos] == 0:
116             pole = pos
117
```

```
118     return postaw_znak(pole, RUCH)
```

Za sposób gry komputera odpowiada funkcja `ai_ruch()` (*ai* – ang. *artificial intelligence*, sztuczna inteligencja). Na początku zawiera ona definicje dwóch list (`uklady_wygrywam, uklady_blokuje`), zawierających układy wartości, dla których komputer wygrywa oraz które powinien zablokować, aby nie wygrał gracz. O tym, które pole należy zaznaczyć, decyduje funkcja `sprawdz_pola()` przyjmująca jako argument najpierw układy wygrywające, później blokujące. Podstawą działania funkcji `sprawdz_pola()` jest lista `POLA_INDEKSY` zawierająca jako elementy listy indeksów pól tworzących wiersze, kolumny i przekątne `POLA_GRY` (czyli planszy). Pętla `for lista in POLA_INDEKSY:` pobiera kolejne listy, tworzy w liście pomocniczej kol trójkę wartości odczytanych z `POLA_GRY` i próbuje ją dopasować do przekazanego jako argument układu wygrywającego lub blokującego. Jeżeli znajdzie dopasowanie zwraca liczbę oznaczającą gracza lub komputer, o ile opcjonalny argument `WYGRANY` ma wartość inną niż `None`, w przeciwnym razie zwracany jest indeks `POLA_GRY`, na którym komputer powinien postawić swój znak. Jeżeli indeks zwrócony przez funkcję `sprawdz_pola()` jest inny niż `None`, przekazywany jest do funkcji `postaw_znak()`, której zadaniem jest zapisanie w `POLU_GRY` pod otrzymanym indeksem wartości symbolizującej znak komputera (czyli 2) oraz nadanie i zwrócenie zmiennej RUCH wskazującej na gracza (wartość 1). O ile na planszy nie ma układu wygrywającego lub nie ma konieczności blokowania gracza, komputer w pętli losuje przypadkowe pole (`random.randrange(0,9)`), dopóki nie znajdzie pustego, i przekazuje jego indeks do funkcji `postaw_znak()`.

Główna pętla programu

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w pętli, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (przesunięcia elementów, aktualizacja planszy),
3. aktualizacja wyświetlanego okna (narysowanie nowego stanu gry).

Dopisujemy więc do kodu główną pętlę wraz z obsługą zdarzeń oraz dwie funkcje pomocnicze w niej wywoływane:

```
105         return postaw_znak(pole, RUCH)
106
107     # jeżeli komputer nie może wygrać, blokuj gracza
108     pole = sprawdz_pola(uklady_blokuje)
109     if pole is not None:
110         return postaw_znak(pole, RUCH)
111
112     # jeżeli nie można wygrać i gracza nie trzeba blokować, wylosuj pole
113     while pole is None:
114         pos = random.randrange(0, 9)  # wylosuj wartość od 0 do 8
115         if POLE_GRY[pos] == 0:
116             pole = pos
117
118     return postaw_znak(pole, RUCH)
119
120 # sprawdź, kto wygrał, a może jest remis?
121
122
123 def kto_wygral():
124     # układy wygrywające dla gracza i komputera
125     uklad_gracz = [[1, 1, 1]]
126     uklad_komp = [[2, 2, 2]]
127
```

```
128     WYGRANY = sprawdz_pola(uklad_gracz, 1)  # czy wygrał gracz?
129     if not WYGRANY:  # jeżeli gracz nie wygrywa
130         WYGRANY = sprawdz_pola(uklad_komp, 2)  # czy wygrał komputer?
131 
132     # sprawdź remis
133     if 0 not in POLE_GRY and WYGRANY not in [1, 2]:
134         WYGRANY = 3
135 
136     return WYGRANY
137 
138 # funkcja wyświetlająca komunikat końcowy
139 # tworzy nowy obrazek z tekstem, pobiera jego prostokątny obszar
140 # pozycjonuje go i rysuje w oknie gry
141 
142 
143 def drukuj_wynik(WYGRANY):
144     fontObj = pygame.font.Font('freesansbold.ttf', 16)
145     if WYGRANY == 1:
146         tekst = u'Wygrał gracz!'
147     elif WYGRANY == 2:
148         tekst = u'Wygrał komputer!'
149     elif WYGRANY == 3:
150         tekst = 'Remis!'
151     tekst_obr = fontObj.render(tekst, True, (20, 255, 20))
152     tekst_prost = tekst_obr.get_rect()
153     tekst_prost.center = (75, 75)
154     OKNOGRY.blit(tekst_obr, tekst_prost)
155 
156 
157 # pętla główna programu
158 while True:
159     # obsługa zdarzeń generowanych przez gracza
160     for event in pygame.event.get():
161         # przechwyć zamknięcie okna
162         if event.type == QUIT:
163             pygame.quit()
164             sys.exit()
165 
166         if WYGRANA is False:
167             if RUCH == 1:
168                 if event.type == MOUSEBUTTONDOWN:
169                     if event.button == 1:  # jeżeli naciśnięto 1. przycisk
170                         mouseX, mouseY = event.pos  # rozpakowanie tupli
171                         # wylicz indeks klikniętego pola
172                         pole = (int(mouseY / 50) * 3) + int(mouseX / 50)
173                         RUCH = postaw_znak(pole, RUCH)
174             elif RUCH == 2:
175                 RUCH = ai_ruch(RUCH)
176 
177             WYGRANY = kto_wygral()
178             if WYGRANY is not None:
179                 WYGRANA = True
180 
181     OKNOGRY.fill((0, 0, 0))  # definicja koloru powierzchni w RGB
182     rysuj_plansze()
183     rysuj_pole_gry()
184     if WYGRANA:
185         drukuj_wynik(WYGRANY)
```

```
186     pygame.display.update()
```

W obrębie głównej pętli programu pętla `for` odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Jak widać, w pierwszej kolejności obsługujemy wydarzenie typu (właściwość `.type`) QUIT, czyli zakończenie aplikacji. Później, o ile nikt nie wygrał (zmienna `WYGRANA` ma wartość `False`), a kolej na ruch gracza (zmienna `RUCH` ma wartość 1), przechwytujemy wydarzenie `MOUSEBUTTONDOWN`, tj. kliknięcie myszą. Sprawdzamy, czy naciśnięto pierwszy przycisk, pobieramy współrzędne kursora (`.pos`) i wyliczamy indeks klikniętego pola. Na koniec wywołujemy omówioną wcześniej `funkcję postaw_znak()`. Jeżeli kolej na komputer, uruchamiamy sztuczną inteligencję (`ai_ruch()`).

Po wykonaniu ruchu przez komputer lub gracza trzeba sprawdzić, czy któryś z przeciwników nie wygrał. Korzystamy z funkcji `kto_wygral()`, która definiuje dwa układy wygrywające (`uklad_gracz` i `uklad_komputer`) i za pomocą omówionej wcześniej funkcji `sprawdz_pola()` sprawdza, czy można je odnaleźć w `POLU_GRY`. Na końcu sprawdza możliwość remisu i zwraca wartość symbolizującą wygranego (1, 2, 3) lub `None`, o ile możliwe są kolejne ruchy. Wartość ta wpływa w pętli głównej na zmienną `WYGRANA` kontrolującą obsługę ruchów gracza i komputera.

Funkcja `drukuj_wynik()` ma za zadanie przygotowanie końcowego napisu. W tym celu tworzy obiekt czcionki z podanego pliku (`pygame.font.Font()`), następnie renderuje nowy obrazek z odpowiednim tekstem (`.render()`), pobiera jego powierzchnię prostokątną (`.get_rect()`), pozycjonują ją (`.center()`) i rysują na głównej powierzchni gry (`.blit()`). Ostatnie linie kodu wypełniają okno gry kolorem (`.fill()`), wywołują funkcję rysujące planszę (`rysuj_plansze()`), stan gry (`rysuj_pole_gry()`, czyli znaki gracza i komputera), a także ewentualny komunikat końcowy (`drukuj_wynik()`). Funkcja `pygame.display.update()`, która musi być wykonywana na końcu rysowania, aktualizuje obraz gry na ekranie.

Informacja: Plik wykorzystywany do wyświetlania tekstu (`freesansbold.ttf`) musi znaleźć się w katalogu ze skryptem.

Grę możemy uruchomić poleceniem wpisanym w terminalu:

```
~$ python tictactoe.py
```

Zadania dodatkowe

Zmień grę tak, aby zaczynał ją komputer. Dodaj do gry możliwość rozgrywki wielokrotnej bez konieczności ponownego uruchamiania skryptu. Zmodyfikuj funkcję rysującą pole gry tak, aby komputer rysował krzyżyki, a nie kółka.

Materiały

Źródła:

- `tictactoe_str.zip`

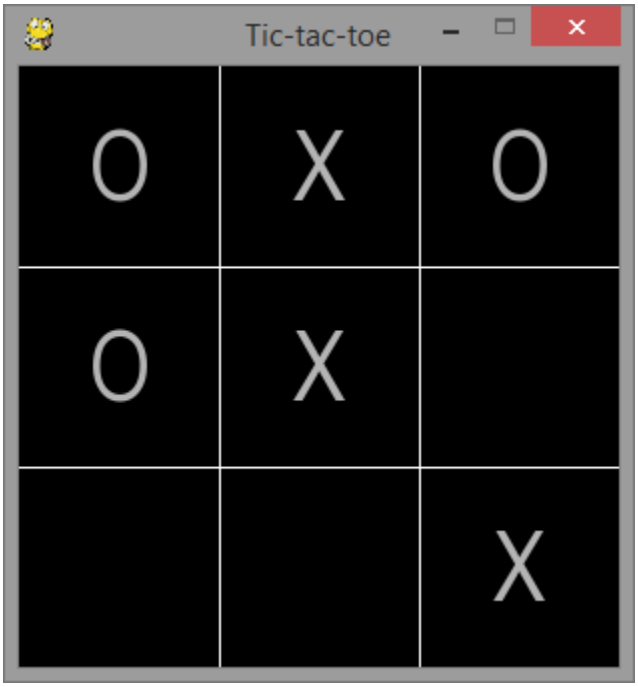
  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik “Autorzy”

Kółko i krzyżyk (obj)

Klasyczna gra w kółko i krzyżyk zrealizowana przy pomocy PyGame.

- *Okienko gry*



Okienko gry

Na wstępie w pliku `~/python101/games/tic_tac_toe.py` otrzymujemy kod który przygotuje okienko naszej gry:

Informacja: Ten przykład zakłada wcześniejsze zrealizowanie przykładu: *Życie Conwaya (obj)*, opisy niektórych cech wspólnych zostały tutaj wyraźnie pominięte. W tym przykładzie wykorzystujemy np. podobne mechanizmy do tworzenia okna i zarządzania główną pętlą naszej gry.

Ostrzeżenie: TODO: Wymaga ewentualnego rozbicia i uzupełnienia opisów.

```
# coding=utf-8
# Copyright 2014 Janusz Skonieczny

"""
Gra w kółko i krzyżyk
"""

import pygame
import pygame.locals
import logging

# Konfiguracja modułu logowania, element dla zaawansowanych
logging_format = '%(asctime)s %(levelname)-7s | %(module)s.%(funcName)s - %(message)s'
```

```
14 logging.basicConfig(level=logging.DEBUG, format=logging_format, datefmt='%H:%M:%S')
15 logging.getLogger().setLevel(logging.INFO)
16
17
18 class Board(object):
19     """
20     Plansza do gry. Odpowiada za rysowanie okna gry.
21     """
22
23     def __init__(self, width):
24         """
25         Konstruktor planszy do gry. Przygotowuje okienko gry.
26
27         :param width: szerokość w pikselach
28         """
29         self.surface = pygame.display.set_mode((width, width), 0, 32)
30         pygame.display.set_caption('Tic-tac-toe')
31
32         # Przed pisaniem tekstów, musimy zainicjować mechanizmy wyboru fontów PyGame
33         pygame.font.init()
34         font_path = pygame.font.match_font('arial')
35         self.font = pygame.font.Font(font_path, 48)
36
37         # tablica znaczników 3x3 w formie listy
38         self.markers = [None] * 9
39
40     def draw(self, *args):
41         """
42         Rysuje okno gry
43
44         :param args: lista obiektów do narysowania
45         """
46         background = (0, 0, 0)
47         self.surface.fill(background)
48         self.draw_net()
49         self.draw_markers()
50         self.draw_score()
51         for drawable in args:
52             drawable.draw_on(self.surface)
53
54         # dopiero w tym miejscu następuje fatyczne rysowanie
55         # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
56         pygame.display.update()
57
58     def draw_net(self):
59         """
60         Rysuje siatkę linii na planszy
61         """
62         color = (255, 255, 255)
63         width = self.surface.get_width()
64         for i in range(1, 3):
65             pos = width / 3 * i
66             # linia pozioma
67             pygame.draw.line(self.surface, color, (0, pos), (width, pos), 1)
68             # linia pionowa
69             pygame.draw.line(self.surface, color, (pos, 0), (pos, width), 1)
70
71     def player_move(self, x, y):
```

```
        """
        Ustawia na planszy znacznik gracza X na podstawie współrzędnych w pikselach
        """
        cell_size = self.surface.get_width() / 3
        x /= cell_size
        y /= cell_size
        self.markers[int(x) + int(y) * 3] = player_marker(True)

    def draw_markers(self):
        """
        Rysuje znaczniki graczy
        """
        box_side = self.surface.get_width() / 3
        for x in range(3):
            for y in range(3):
                marker = self.markers[x + y * 3]
                if not marker:
                    continue
                # zmieniamy współrzędne znacznika
                # na współrzędne w pikselach dla centrum pola
                center_x = x * box_side + box_side / 2
                center_y = y * box_side + box_side / 2

                self.draw_text(self.surface, marker, (center_x, center_y))

    def draw_text(self, surface,  text, center, color=(180, 180, 180)):
        """
        Rysuje wskazany tekst we wskazanym miejscu
        """
        text = self.font.render(text, True, color)
        rect = text.get_rect()
        rect.center = center
        surface.blit(text, rect)

    def draw_score(self):
        """
        Sprawdza czy gra została skończona i rysuje właściwy komunikat
        """
        if check_win(self.markers, True):
            score = u"Wygrałeś(aś)"
        elif check_win(self.markers, True):
            score = u"Przegrałeś(aś)"
        elif None not in self.markers:
            score = u"Remis!"
        else:
            return

        i = self.surface.get_width() / 2
        self.draw_text(self.surface, score, center=(i, i), color=(255, 26, 26))


class TicTacToeGame(object):
    """
    Łączy wszystkie elementy gry w całość.
    """

    def __init__(self, width, ai_turn=False):
        """
```

```
130         Przygotowanie ustawień gry
131         :param width: szerokość planszy mierzona w pikselach
132         """
133         pygame.init()
134         # zegar którego użyjemy do kontrolowania szybkości rysowania
135         # kolejnych klatek gry
136         self.fps_clock = pygame.time.Clock()
137 
138         self.board = Board(width)
139         self.ai = Ai(self.board)
140         self.ai_turn = ai_turn
141 
142     def run(self):
143         """
144         Główna pętla gry
145         """
146         while not self.handle_events():
147             # działaj w pętli do momentu otrzymania sygnału do wyjścia
148             self.board.draw()
149             if self.ai_turn:
150                 self.ai.make_turn()
151                 self.ai_turn = False
152             self.fps_clock.tick(15)
153 
154     def handle_events(self):
155         """
156         Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
157 
158         :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
159         """
160         for event in pygame.event.get():
161             if event.type == pygame.locals.QUIT:
162                 pygame.quit()
163                 return True
164 
165             if event.type == pygame.locals.MOUSEBUTTONDOWN:
166                 if self.ai_turn:
167                     # jeśli jeszcze trwa ruch komputera to ignorujemy zdarzenia
168                     continue
169                 # pobierz aktualną pozycję kursora na planszy mierzoną w pikselach
170                 x, y = pygame.mouse.get_pos()
171                 self.board.player_move(x, y)
172                 self.ai_turn = True
173 
174 
175 class Ai(object):
176     """
177     Kieruje ruchami komputera na podstawie analizy położenia znaczników
178     """
179     def __init__(self, board):
180         self.board = board
181 
182     def make_turn(self):
183         """
184         Wykonuje ruch komputera
185         """
186         if not None in self.board.markers:
187             # brak dostępnych ruchów
```

```
            return
        logging.debug("Plansza: %s" % self.board.markers)
        move = self.next_move(self.board.markers)
        self.board.markers[move] = player_marker(False)

    @classmethod
    def next_move(cls, markers):
        """
        Wybierz następny ruch komputera na podstawie wskazanej planszy
        :param markers: plansza gry
        :return: index tablicy jednowymiarowe w której należy ustawić znacznik kółka
        """
        # pobierz dostępne ruchy wraz z oceną
        moves = cls.score_moves(markers, False)
        # wybierz najlepiej oceniony ruch
        score, move = max(moves, key=lambda m: m[0])
        logging.info("Dostępne ruchy: %s", moves)
        logging.info("Wybrany ruch: %s %s", move, score)
        return move

    @classmethod
    def score_moves(cls, markers, x_player):
        """
        Ocenia rekurencyjne możliwe ruchy

        Jeśli ruch jest zwycięstwem otrzymuje +1, jeśli przegraną -1
        lub 0 jeśli nie nie ma zwycięscy. Dla ruchów bez zwycięscy rekreacyjnie
        analizowane są kolejne ruchy a suma ich punktów jest wynikiem aktualnego
        ruchu.

        :param markers: plansza na podstawie której analizowane są następne ruchy
        :param x_player: True jeśli ruch dotyczy gracza X, False dla gracza O
        """
        # wybieramy wszystkie możliwe ruchy na podstawie wolnych pól
        available_moves = (i for i, m in enumerate(markers) if m is None)
        for move in available_moves:
            from copy import copy
            # tworzymy kopię planszy która na której testowo zostanie
            # wykonany ruch w celu jego późniejszej oceny
            proposal = copy(markers)
            proposal[move] = player_marker(x_player)

            # sprawdzamy czy ktoś wygrywa gracz którego ruch testujemy
            if check_win(proposal, x_player):
                # dodajemy punkty jeśli to my wygrywamy
                # czyli nie x_player
                score = -1 if x_player else 1
                yield score, move
                continue

            # ruch jest neutralny,
            # sprawdzamy rekurencyjne kolejne ruchy zmieniając gracza
            next_moves = list(cls.score_moves(proposal, not x_player))
            if not next_moves:
                yield 0, move
                continue

            # rozdzielamy wyniki od ruchów
```

```
            scores, moves = zip(*next_moves)
            # sumujemy wyniki możliwych ruchów, to będzie nasz wynik
            yield sum(scores), move


def player_marker(x_player):
    """
    Funkcja pomocnicza zwracająca znaczniki graczy
    :param x_player: True dla gracza X False dla gracza O
    :return: odpowiedni znak gracza
    """
    return "X" if x_player else "O"


def check_win(markers, x_player):
    """
    Sprawdza czy przekazany zestaw znaczników gry oznacza zwycięstwo wskazanego gracza

    :param markers: jednowymiarowa sekwencja znaczników w
    :param x_player: True dla gracza X False dla gracza O
    """
    win = [player_marker(x_player)] * 3
    seq = range(3)

    # definiujemy funkcję pomocniczą pobierającą znacznik
    # na podstawie współrzędnych x i y
    def marker(xx, yy):
        return markers[xx + yy * 3]

    # sprawdzamy każdy rząd
    for x in seq:
        row = [marker(x, y) for y in seq]
        if row == win:
            return True

    # sprawdzamy każdą kolumnę
    for y in seq:
        col = [marker(x, y) for x in seq]
        if col == win:
            return True

    # sprawdzamy przekątne
    diagonal1 = [marker(i, i) for i in seq]
    diagonal2 = [marker(i, abs(i-2)) for i in seq]
    if diagonal1 == win or diagonal2 == win:
        return True


# Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
# chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
if __name__ == "__main__":
    game = TicTacToeGame(300)
    game.run()

```

W powyższym kodzie mamy podstawy potrzebne do uruchomienia gry:

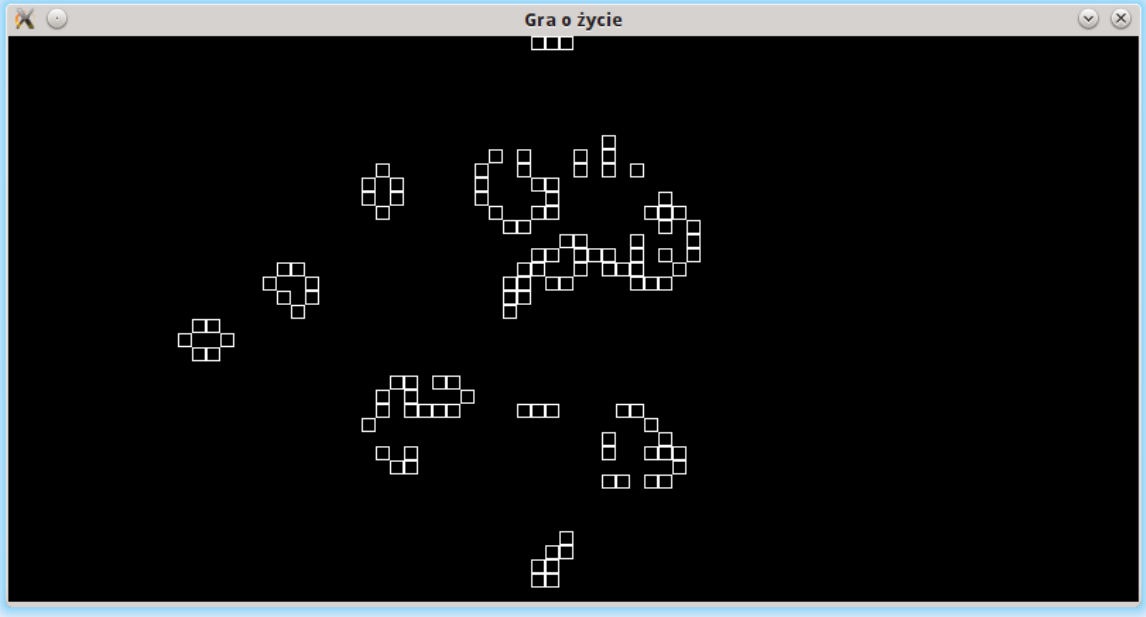
```
~/python101$ python games/tic_tac_toe.py
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Życie Conwaya (str)

Gra w życie zrealizowana z użyciem biblioteki PyGame. Wersja strukturalna. Biblioteka PyGame ułatwia tworzenie aplikacji multimedialnych, w tym gier.



Zmienne i plansza gry

Tworzymy plik `life.py` w terminalu lub w wybranym edytorze i zaczynamy od zdefiniowania zmiennych określających właściwości obiektów w naszej grze.

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

```

```
import pygame
import sys
import random
from pygame.locals import *  # udostępnienie nazw metod z locals

# inicjacja modułu pygame
pygame.init()

# szerokość i wysokość okna gry
OKNOGRY_SZER = 800
OKNOGRY_WYS = 400

# przygotowanie powierzchni do rysowania, czyli inicjacja okna gry
OKNOGRY = pygame.display.set_mode((OKNOGRY_SZER, OKNOGRY_WYS), 0, 32)
# tytuł okna gry
pygame.display.set_caption('Gra o życie')

# rozmiar komórki
ROZ_KOM = 10
# ilość komórek w poziomie i pionie
KOM_POZIOM = int(OKNOGRY_SZER / ROZ_KOM)
KOM_PION = int(OKNOGRY_WYS / ROZ_KOM)

# wartości oznaczające komórki "martwe" i "żywe"
KOM_MARTWA = 0
KOM_ZYWA = 1

# lista opisująca stan pola gry, 0 - komórki martwe, 1 - komórki żywe
# na początku tworzymy listę zawierającą KOM_POZIOM zer
POLE_GRY = [KOM_MARTWA] * KOM_POZIOM
# rozszerzamy listę o listy zagnieżdżone, otrzymujemy więc listę dwuwymiarową
for i in range(KOM_POZIOM):
    POLE_GRY[i] = [KOM_MARTWA] * KOM_PION
```

W instrukcji `pygame.display.set_mode()` inicjalizujemy okno gry o rozmiarach 800x400 pikseli i 32-bitowej głębi kolorów. Tworzymy w ten sposób powierzchnię główną do rysowania zapisaną w zmiennej `OKNOGRY`. Ilość możliwych do narysowania komórek, reprezentowanych przez kwadraty o boku 10 pikseli, wyliczamy w zmiennych `KOM_POZIOM` i `KOM_PION`. Najważniejszą strukturą w naszej grze jest `POLE_GRY`, dwuwymiarowa lista elementów reprezentujących "żywe" i "martwe" komórki, czyli populację. Tworzymy ją w dwóch krokach, na początku inicjujemy zerami jednowymiarową listę o rozmiarze odpowiadającym ilości komórek w poziomie (`POLE_GRY = [KOM_MARTWA] * KOM_POZIOM`). Następnie do każdego elementu listy przypisujemy listę zawierającą tyle zer, ile jest komórek w pionie.

Populacja komórek

Kolejnym krokiem będzie zdefiniowanie funkcji przygotowującej i rysującej populację komórek.

```
# przygotowanie następnej generacji komórek, czyli zaktualizowanego POLA_GRY
def przygotuj_populacje(polegry):
    # na początku tworzymy 2-wymiarową listę wypełnioną zerami
    nast_gen = [KOM_MARTWA] * KOM_POZIOM
    for i in range(KOM_POZIOM):
        nast_gen[i] = [KOM_MARTWA] * KOM_PION

    # iterujemy po wszystkich komórkach
    for y in range(KOM_PION):
```

```
        for x in range(KOM_POZIOM):

            # zlicz populację (żywych komórek) wokół komórki
            populacja = 0
            # wiersz 1
            try:
                if polegry[x - 1][y - 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass
            try:
                if polegry[x][y - 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass
            try:
                if polegry[x + 1][y - 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass

            # wiersz 2
            try:
                if polegry[x - 1][y] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass
            try:
                if polegry[x + 1][y] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass

            # wiersz 3
            try:
                if polegry[x - 1][y + 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass
            try:
                if polegry[x][y + 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass
            try:
                if polegry[x + 1][y + 1] == KOM_ZYWA:
                    populacja += 1
            except IndexError:
                pass

            # "niedoludnienie" lub przeludnienie = śmierć komórki
            if polegry[x][y] == KOM_ZYWA and (populacja < 2 or populacja > 3):
                nast_gen[x][y] = KOM_MARTWA
            # życie trwa
            elif polegry[x][y] == KOM_ZYWA \
                    and (populacja == 3 or populacja == 2):
                nast_gen[x][y] = KOM_ZYWA
            # nowe życie
```

```
106             elif polegry[x][y] == KOM_MARTWA and populacja == 3:
107                 nast_gen[x][y] = KOM_ZYWA
108
109     # zwróć nowe polegry z następną generacją komórek
110     return nast_gen
111
112
113 def rysuj_populacje():
114     """Rysowanie komórek (kwadratów) żywych"""
115     for y in range(KOM_PION):
116         for x in range(KOM_POZIOM):
117             if POLE_GRY[x][y] == KOM_ZYWA:
118                 pygame.draw.rect(OKNOGRY, (255, 255, 255), Rect(
119                     (x * ROZ_KOM, y * ROZ_KOM), (ROZ_KOM, ROZ_KOM)), 1)
```

Najważniejszym fragmentem kodu, implementującym logikę naszej gry, jest funkcja przygotuj_populacje(), która jako parametr przyjmuje omówioną wcześniej strukturę `POLE_GRY` (pod nazwą `polegry`). Funkcja sprawdza, jak rozwija się populacja komórek, według następujących zasad:

1. Jeżeli żywa komórka ma mniej niż 2 żywych sąsiadów, umiera z powodu samotności.
2. Jeżeli żywa komórka ma więcej niż 3 żywych sąsiadów, umiera z powodu przeludnienia.
3. Żywa komórka z 2 lub 3 sąsiadami żyje dalej.
4. Martwa komórka z 3 żywymi sąsiadami ożywa.

Funkcja iteruje po każdym elemencie `POLA_GRY` i sprawdza stan sąsiadów każdej komórki, w wierszu 1 powyżej komórki, w wierszu 2 na tym samym poziomie i w wierszu 3 poniżej. Konstrukcja `try...except` pozwala obsłużyć sytuacje wyjątkowe (błędy), a więc komórki skrajne, które nie mają sąsiadów u góry czy u dołu, z lewej bądź z prawej strony: w takim przypadku wywoływana jest instrukcja `pass`, czyli nie rób nic :-). Końcowa złożona instrukcja warunkowa `if` ożywia lub uśmierca sprawdzaną komórkę w zależności od stanu sąsiednich komórek (czyli zmiennej `populacja`).

Zadaniem funkcji `rysuj_populacje()` jest narysowanie kwadratów (obiekty *Rect*) o białych bokach w rozmiarze 10 pikseli dla pól (elementów), które w liście `POLE_GRY` są żywe (mają wartość 1).

Główna pętla programu

Programy interaktywne, w tym gry, reagujące na działania użytkownika, takie jak ruchy czy kliknięcia myszą, działają w pętli, której zadaniem jest:

1. przechwycenie i obsługa działań użytkownika, czyli tzw. zdarzeń (ruchy, kliknięcia myszą, naciśnięcie klawiszy),
2. aktualizacja stanu gry (przesunięcia elementów, aktualizacja planszy),
3. aktualizacja wyświetlanego okna (narysowanie nowego stanu gry).

Dopisujemy więc do kodu główną pętlę wraz z obsługą zdarzeń:

```
122 # zmienne sterujące wykorzystywane w pętli głównej
123 zycie_trwa = False
124 przycisk_wdol = False
125
126 # pętla główna programu
127 while True:
128     # obsługa zdarzeń generowanych przez gracza
129     for event in pygame.event.get():
```

```
        # przechwyć zamknięcie okna
        if event.type == QUIT:
            pygame.quit()
            sys.exit()

        if event.type == KEYDOWN and event.key == K_RETURN:
            zycie_trwa = True

        if zycie_trwa is False:
            if event.type == MOUSEBUTTONDOWN:
                przycisk_wdol = True
                przycisk_typ = event.button

            if event.type == MOUSEBUTTONUP:
                przycisk_wdol = False

            if przycisk_wdol:
                mouse_x, mouse_y = pygame.mouse.get_pos()
                mouse_x = int(mouse_x / ROZ_KOM)
                mouse_y = int(mouse_y / ROZ_KOM)
                # lewy przycisk myszy ożywia
                if przycisk_typ == 1:
                    POLE_GRY[mouse_x][mouse_y] = KOM_ZYWA
                # prawy przycisk myszy uśmierca
                if przycisk_typ == 3:
                    POLE_GRY[mouse_x][mouse_y] = KOM_MARTWA

    if zycie_trwa is True:
        POLE_GRY = przygotuj_populacje(POLE_GRY)

    OKNOGRY.fill((0, 0, 0))  # ustaw kolor okna gry
    rysuj_populacje()
    pygame.display.update()
    pygame.time.delay(100)
```

W obrębie głównej pętli programu pętla `for` odczytuje kolejne zdarzenia zwracane przez metodę `pygame.event.get()`. Jak widać, w pierwszej kolejności obsługujemy wydarzenie typu (właściwość `.type`) QUIT, czyli zakończenie aplikacji.

Jednak na początku gry gracz klika lewym lub prawym klawiszem myszy i ożywia lub uśmierca kliknięte komórki w obrębie okna gry. Dzieje się tak dopóty, dopóki zmienna `zycie_trwa` ma wartość `False`, a więc dopóki gracz nie naciśnie klawisza ENTER (`if event.type == KEYDOWN and event.key == K_RETURN:`). Każde kliknięcie myszą zostaje przechwycone (`if event.type == MOUSEBUTTONDOWN:`) i zapamiętane w zmiennej `przycisk_wdol`. Jeżeli zmienna ta ma wartość `True`, pobieramy współrzędne kursora myszy (`mouse_x, mouse_y = pygame.mouse.get_pos()`) i obliczamy indeksy elementu listy `POLE_GRY` odpowiadającego klikniętej komórce. Następnie sprawdzamy, który przycisk myszy został naciśnięty; informację tę zapisaliśmy wcześniej za pomocą funkcji `event.button` w zmiennej `przycisk_typ`, która przyjmuje wartość 1 (lewy) lub 3 (prawy przycisk myszy), w zależności od klikniętego przycisku ożywiamy lub uśmiercamy komórkę, zapisując odpowiedni stan w liście `POLE_GRY`.

Naciśnięcie klawisza ENTER uruchamia symulację rozwoju populacji. Zmienna `zycie_trwa` ustawiona zostaje na wartość `True` , co przerywa obsługę kliknięć myszą, i wywoływana jest funkcja `przygotuj_populacje()`, która przygotowuje kolejny stan populacji. Końcowe polecenia wypełniają okno gry kolorem (`.fill()`), wywołują funkcję rysującą planszę (`rysuj_populacje()`). Funkcja `pygame.display.update()`, która musi być wykonywana na końcu rysowania, aktualizuje obraz gry na ekranie. Ostatnie polecenie `pygame.time.delay(100)` dodaje 100-milisekundowe opóźnienie kolejnej aktualizacji stanu populacji. Dzięki temu możemy obserwować jej rozwój na planszy.

Grę możemy uruchomić poleceniem wpisanym w terminalu:

```
~$ python life_str.py
```

Zadania dodatkowe

Spróbuj inaczej zaimplementować funkcję `przygotuj_populacje`. Spróbuj zmodyfikować kod tak, aby plansza gry była biała, a komórki rysowane były jako kolorowe kwadraty o różniącym się od wypełnienia obramowaniu.

Materiały

Źródła:

- `life_str.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Życie Conwaya (obj)

Gra w życie zrealizowana z użyciem biblioteki PyGame.

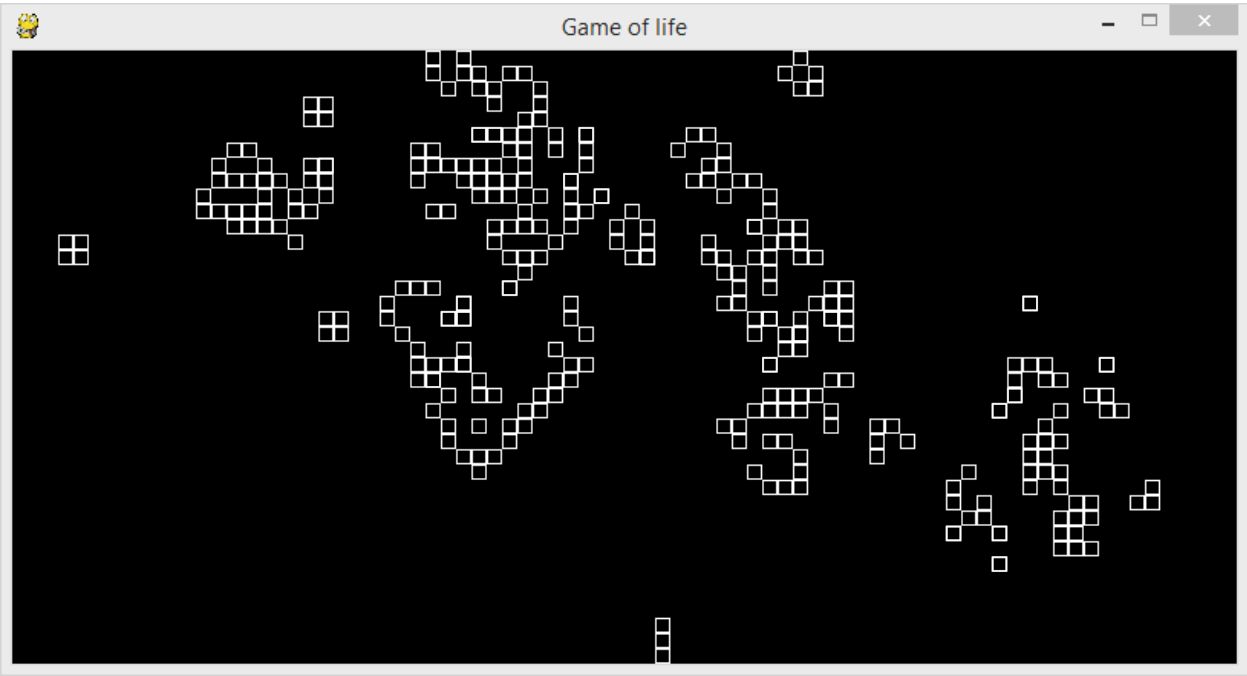
Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szczątkowy kod źródłowy:

```
~/python101$ git checkout -f life/z1
```

Okienko gry

Na wstępie w pliku `~/python101/games/life.py` otrzymujemy kod który przygotuje okienko naszej gry:



Informacja: Ten przykład zakłada wcześniejsze zrealizowanie przykładu: *Pong (obj)*, opisy niektórych cech wspólnych zostały tutaj wyraźnie pominięte. W tym przykładzie wykorzystujemy np. podobne mechanizmy do tworzenia okna i zarządzania główną pętlą naszej gry.

```
# coding=utf-8

import pygame
import pygame.locals


class Board(object):
    """
    Plansza do gry. Odpowiada za rysowanie okna gry.
    """

    def __init__(self, width, height):
        """
        Konstruktor planszy do gry. Przygotowuje okienko gry.

        :param width: szerokość w pikselach
        :param height: wysokość w pikselach
        """
        self.surface = pygame.display.set_mode((width, height), 0, 32)
        pygame.display.set_caption('Game of life')

    def draw(self, *args):
        """
        Rysuje okno gry

        :param args: lista obiektów do narysowania
        """
        background = (0, 0, 0)
```

```
        self.surface.fill(background)
        for drawable in args:
            drawable.draw_on(self.surface)

        # dopiero w tym miejscu następuje fatyczne rysowanie
        # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
        pygame.display.update()


class GameOfLife(object):
    """
    Łączy wszystkie elementy gry w całość.
    """

    def __init__(self, width, height, cell_size=10):
        """
        Przygotowanie ustawień gry
        :param width: szerokość planszy mierzona liczbą komórek
        :param height: wysokość planszy mierzona liczbą komórek
        :param cell_size: bok komórki w pikselach
        """
        pygame.init()
        self.board = Board(width * cell_size, height * cell_size)
        # zegar którego użyjemy do kontrolowania szybkości rysowania
        # kolejnych klatek gry
        self.fps_clock = pygame.time.Clock()

    def run(self):
        """
        Główna pętla gry
        """
        while not self.handle_events():
            # działaj w pętli do momentu otrzymania sygnału do wyjścia
            self.board.draw()
            self.fps_clock.tick(15)

    def handle_events(self):
        """
        Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką

        :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
        """
        for event in pygame.event.get():
            if event.type == pygame.locals.QUIT:
                pygame.quit()
                return True


# Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
# chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
if __name__ == "__main__":
    game = GameOfLife(80, 40)
    game.run()
```

W powyższym kodzie mamy podstawy potrzebne do uruchomienia gry:

```
~/python101$ python games/life.py
```

Tworzymy matrycę życia

Nasza gra polega na ułożenia komórek na planszy i obserwacji jak w kolejnych generacjach życie się zmienia, które komórki giną, gdzie się rozmnażają i wywołują efektowną wędrówkę oraz tworzenie się ciekawych struktur.

Zacznijmy od zadeklarowania zmiennych które zastąpią nam tzw. *magiczne liczby*. W kodzie zamiast wartości `1` dla określenia żywej komórki i wartości `0` dla martwej komórki wykorzystamy zmiennie `ALIVE` oraz `DEAD`. W innych językach takie zmienne czasem są określane jako *stała*.

```
77 # magiczne liczby używane do określenia czy komórka jest żywa
78 DEAD = 0
79 ALIVE = 1
```

Podstawą naszego życia będzie klasa `Population` która będzie przechowywać stan gry, a także realizować funkcje potrzebne do zmian stanu gry w czasie. W przeciwieństwie do *gry w Pong* nie będziemy dzielić odpowiedzialności pomiędzy większą liczbę klas.

```
82 class Population(object):
83     """
84     Populacja komórek
85     """
86
87     def __init__(self, width, height, cell_size=10):
88         """
89         Przygotowuje ustawienia populacji
90
91         :param width: szerokość planszy mierzona liczbą komórek
92         :param height: wysokość planszy mierzona liczbą komórek
93         :param cell_size: bok komórki w pikselach
94         """
95         self.box_size = cell_size
96         self.height = height
97         self.width = width
98         self.generation = self.reset_generation()
99
100    def reset_generation(self):
101        """
102        Tworzy i zwraca macierz pustej populacji
103        """
104        # w pętli wypełnij listę kolumnami
105        # które także w pętli zostają wypełnione wartością 0 (DEAD)
106        return [[DEAD for y in range(self.height)] for x in range(self.width)]
```

Poza ostatnią linią nie ma tutaj wielu niespodzianek, ot konstruktor `__init__` zapamiętujący wartości konfiguracyjne w instancji naszej klasy, tj. w `self`.

W ostatniej linii budujemy macierz dla komórek. Tablicę dwuwymiarową, którą będziemy adresować przy pomocy współrzędnych `x` i `y`. Jeśli plansza miałaby szerokość 4, a wysokość 3 komórek to zadeklarowana ręcznie nasza tablica wyglądałaby tak:

```
1 generation = [
2     [DEAD, DEAD, DEAD, DEAD],
3     [DEAD, DEAD, DEAD, DEAD],
```

```
    [DEAD, DEAD, DEAD, DEAD],
]
```

Jednak ręczne zadeklarowanie byłoby uciążliwe i mało elastyczne, wyobraźmy sobie macierz 40 na 80 — strasznie dużo pisania! Dlatego posłużymy się pętlami i wyliczymy sobie dowolną macierz na podstawie zadanych parametrów.

```
def reset_generation(self)
    generation = []
    for x in range(self.width):
        column = []
        for y in range(self.height)
            column.append(DEAD)
        generation.append(column)
    return generation
```

Powyżej wykorzystaliśmy 2 pętle (jedna zagnieżdżona w drugiej) oraz funkcję `range` która wygeneruje listę wartości od 0 do zadanej wartości - 1. Dzięki temu nasze pętle uzyskają `self.width` i `self.height` przebiegów. Jest lepiej.

Przykład kodu powyżej to konstrukcja którą w taki lub podobny sposób wykorzystuje się co chwila w każdym programie — to chleb powszedni programisty. Każdy program musi w jakiś sposób iterować po elementach list przekształcając je w inne listy.

W linii 113 mamy przykład zastosowania tzw. wyrażeń listowych (ang. *list comprehensions*). Pomiędzy znakami nawiasów kwadratowych `[ ]` mamy pętlę, która w każdym przebiegu zwraca jakiś element. Te zwrócone elementy napełniają nową listę która zostanie zwrócona w wyniku wyrażenia.

Sprawę komplikuje dodaje fakt, że chcemy uzyskać tablicę dwuwymiarową dlatego mamy zagnieżdżone wyrażenie listowe (jak 2 pętle powyżej). Zajrzyjmy najpierw do wewnętrznego wyrażenia:

```
[DEAD for y in range(self.height)]
```

W kodzie powyżej każdym przebiegu pętli uzyskamy `DEAD`. Dzięki temu zyskamy *kolumnę* macierzy od wysokości `self.height`, w każdej z nich będziemy mogli się dostać do pojedynczej komorki adresując ją listę wartością `y` o tak `kolumna[y]`.

Teraz zajmijmy się zewnętrznym wyrażeniem listowym, ale dla uproszczenia w każdym jego przebiegu zwracajmy `nowa_kolumna`

```
[nowa_kolumna for x in range(self.width)]
```

W kodzie powyżej w każdym przebiegu pętli uzyskamy `nowa_kolumna`. Dzięki temu zyskamy listę *kolumn*. Do każdej z nich będziemy mogli się dostać adresując listę wartością `x` o tak `generation[x]`, w wyniku otrzymamy kolumnę którą możemy adresować wartością `y`, co w sumie da nam macierz w której do komórek dostaniemy się o tak: `generation[x][y]`.

Zamieniamy `nowa_kolumna` wyrażeniem listowym dla `y` i otrzymamy 1 linijkę zamiast 7 z przykładu z podwójną pętlą:

```
[[DEAD for y in range(self.height)] for x in range(self.width)]
```

Układamy żywe komórki na planszy

Teraz przygotujemy kod który dzięki wykorzystaniu myszki umożliwi nam ułożenie planszy, będziemy wybierać gdzie na planszy będą żywe komórki. Dodajmy do klasy `Population` metodę `handle_mouse` którą będziemy później

wywoływać w metody `GameOfLife.handle_events` za każdym razem gdy nasz program otrzyma zdarzenie dotyczące myszki.

Chcemy by myszka z naciśniętym lewym klawiszem ustawiała pod kursorem żywą komórkę. Jeśli jest naciśnięty inny klawisz to usuniemy żywą komórkę. Jeśli żaden z klawiszy nie jest naciśnięty to zignorujemy zdarzenie myszki.

Zdarzenia są generowane w przypadku naciśnięcia klawiszy lub ruchu myszką, nie będziemy nic robić jeśli gracz poruszy myszką bez naciskania klawiszy.

```
def handle_mouse(self):
    # pobierz stan guzików myszki z wykorzystaniem funcji pygame
    buttons = pygame.mouse.get_pressed()
    if not any(buttons):
        # ignoruj zdarzenie jeśli żaden z guzików nie jest wciśnięty
        return

    # dodaj żywą komórką jeśli wciśnięty jest pierwszy guzik myszki
    # będziemy mogli nie tylko dodawać żywe komórki ale także je usuwać
    alive = True if buttons[0] else False

    # pobierz pozycję kursora na planszy mierzoną w pikselach
    x, y = pygame.mouse.get_pos()

    # przeliczamy współrzędne komórki z pikseli na współrzędne komórki w macierz
    # gracz może kliknąć w kwadracie o szerokości box_size by wybrać komórkę
    x /= self.box_size
    y /= self.box_size

    # ustaw stan komórki na macierzy
    self.generation[int(x)][int(y)] = ALIVE if alive else DEAD
```

Następnie dodajmy metodę `draw_on` która będzie rysować żywe komórki na planszy. Tą metodę wywołamy w metodzie `GameOfLife.draw`.

```
def draw_on(self, surface):
    """
    Rysuje komórki na planszy
    """
    for x, y in self.alive_cells():
        size = (self.box_size, self.box_size)
        position = (x * self.box_size, y * self.box_size)
        color = (255, 255, 255)
        thickness = 1
        pygame.draw.rect(surface, color, pygame.locals.Rect(position, size),␣
↪thickness)
```

Powyżej wykorzystaliśmy nie istniejącą metodę `alive_cells` która jak wynika z jej użycia powinna zwrócić kolekcję współrzędnych dla żywych komórek. Po jednej parze `x, y` dla każdej żywej komórki. Każdą żywą komórkę narysujemy jako kwadrat w białym kolorze.

Utwórzmy metodę `alive_cells` która w pętli przejdzie po całej macierzy populacji i zwróci tylko współrzędne żywych komórek.

```
def alive_cells(self):
    """
    Generator zwracający współrzędne żywych komórek.
    """
    for x in range(len(self.generation)):
        column = self.generation[x]
```

```
147         for y in range(len(column)):
148             if column[y] == ALIVE:
149                 # jeśli komórka jest żywa zwrócimy jej współrzędne
150                 yield x, y
```

W kodzie powyżej mamy przykład dwóch pętli przy pomocy których sprawdzamy zawartość stan życia komórek dla wszystkich możliwych współrzędnych `x` i `y` w macierzy. Na uwagę zasługują dwie rzeczy. Nigdzie tutaj nie zadeklarowaliśmy listy żywych komórek — którą chcemy zwrócić — oraz instrukcję yield.

Instrukcja `yield` powoduje, że nasza funkcja zamiast zwykłych wartości zwróci *generator*. W skrócie w każdym przebiegu wewnętrznej pętli zostaną *wygenerowane* i zwrócone na zewnątrz wartości `x, y`. Za każdym razem gdy `for x, y in self.alive_cells()` poprosi o współrzędne następnej żywej komórki, `alive_cells` wykona się do instrukcji `yield`.

Wskazówka: Działanie generatora najlepiej zaobserwować w debugerze, będziemy mogli to zrobić za chwilę.

Dodajemy populację do kontrolera gry

Czas by rozwinąć nasz kontroler gry, klasę `GameOfLife` o instancję klasy `Population`

```
38  class GameOfLife(object):
39      """
40      Łączy wszystkie elementy gry w całość.
41      """
42
43      def __init__(self, width, height, cell_size=10):
44          """
45          Przygotowanie ustawień gry
46          :param width: szerokość planszy mierzona liczbą komórek
47          :param height: wysokość planszy mierzona liczbą komórek
48          :param cell_size: bok komórki w pikselach
49          """
50          pygame.init()
51          self.board = Board(width * cell_size, height * cell_size)
52          # zegar którego użyjemy do kontrolowania szybkości rysowania
53          # kolejnych klatek gry
54          self.fps_clock = pygame.time.Clock()
55          self.population = Population(width, height, cell_size)
56
57      def run(self):
58          """
59          Główna pętla gry
60          """
61          while not self.handle_events():
62              # działaj w pętli do momentu otrzymania sygnału do wyjścia
63              self.board.draw(
64                  self.population,
65              )
66              self.fps_clock.tick(15)
67
68      def handle_events(self):
69          """
70          Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
71
72          :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
```

```
        """
        for event in pygame.event.get():
            if event.type == pygame.locals.QUIT:
                pygame.quit()
                return True

            from pygame.locals import MOUSEMOTION, MOUSEBUTTONDOWN
            if event.type == MOUSEMOTION or event.type == MOUSEBUTTONDOWN:
                self.population.handle_mouse()
```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z2
```

Szukamy żyjących sąsiadów

Podstawą do określenia tego czy w danym miejscu na planszy (w współrzędnych x i y macierzy) powstanie nowe życie, przetrwa lub zginie istniejące życie; jest określenie liczby żywych komórek w bezpośrednim sąsiedztwie. Przygotujmy do tego metodę:

```
def neighbours(self, x, y):
    """
    Generator zwracający wszystkich okolicznych sąsiadów
    """
    for nx in range(x-1, x+2):
        for ny in range(y-1, y+2):
            if nx == x and ny == y:
                # pomiń współrzędne centrum
                continue
            if nx >= self.width:
                # sąsiad poza końcem planszy, bierzemy pierwszego w danym rzędzie
                nx = 0
            elif nx < 0:
                # sąsiad przed początkiem planszy, bierzemy ostatniego w danym rzędzie
                nx = self.width - 1
            if ny >= self.height:
                # sąsiad poza końcem planszy, bierzemy pierwszego w danej kolumnie
                ny = 0
            elif ny < 0:
                # sąsiad przed początkiem planszy, bierzemy ostatniego w danej␣
↪kolumnie
                ny = self.height - 1

            # dla każdego nie pominiętego powyżej
            # przejścia pętli zwróć komórkę w tych współrzędnych
            yield self.generation[nx][ny]
```

Następnie przygotujmy funkcję która będzie tworzyć nową populację

```
def cycle_generation(self):
    """
    Generuje następną generację populacji komórek
    """
    next_gen = self.reset_generation()
    for x in range(len(self.generation)):
        column = self.generation[x]
```

```
        for y in range(len(column)):
            # pobieramy wartości sąsiadów
            # dla żywej komórki dostaniemy wartość 1 (ALIVE)
            # dla martwej otrzymamy wartość 0 (DEAD)
            # zwykła suma pozwala nam określić liczbę żywych sąsiadów
            count = sum(self.neighbours(x, y))
            if count == 3:
                # rozmnażamy się
                next_gen[x][y] = ALIVE
            elif count == 2:
                # przechodzi do kolejnej generacji bez zmian
                next_gen[x][y] = column[y]
            else:
                # za dużo lub za mało sąsiadów by przeżyć
                next_gen[x][y] = DEAD

    # nowa generacja staje się aktualną generacją
    self.generation = next_gen
```

Jeszcze ostatnie modyfikacje kontrolera gry tak by komórki zaczęły żyć po wciśnięciu klawisza enter.

```
class GameOfLife(object):
    """
    Łączy wszystkie elementy gry w całość.
    """

    def __init__(self, width, height, cell_size=10):
        """
        Przygotowanie ustawień gry
        :param width: szerokość planszy mierzona liczbą komórek
        :param height: wysokość planszy mierzona liczbą komórek
        :param cell_size: bok komórki w pikselach
        """
        pygame.init()
        self.board = Board(width * cell_size, height * cell_size)
        # zegar którego użyjemy do kontrolowania szybkości rysowania
        # kolejnych klatek gry
        self.fps_clock = pygame.time.Clock()
        self.population = Population(width, height, cell_size)

    def run(self):
        """
        Główna pętla gry
        """
        while not self.handle_events():
            # działaj w pętli do momentu otrzymania sygnału do wyjścia
            self.board.draw(
                self.population,
            )
            if getattr(self, "started", None):
                self.population.cycle_generation()
            self.fps_clock.tick(15)

    def handle_events(self):
        """
        Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką

        :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
```

```
        """
        for event in pygame.event.get():
            if event.type == pygame.locals.QUIT:
                pygame.quit()
                return True

            from pygame.locals import MOUSEMOTION, MOUSEBUTTONDOWN
            if event.type == MOUSEMOTION or event.type == MOUSEBUTTONDOWN:
                self.population.handle_mouse()

            from pygame.locals import KEYDOWN, K_RETURN
            if event.type == KEYDOWN and event.key == K_RETURN:
                self.started = True
```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z3
```

Zadania dodatkowe

1. TODO

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

   Autorzy Patrz plik "Autorzy"

Słownik PyGame

Klatki na sekundę (FPS) liczba klatek wyświetlanych w ciągu sekundy, czyli częstotliwość, z jaką statyczne obrazy pojawiają się na ekranie. Jest ona miarą płynności wyświetlania ruchomych obrazów.

Kanał alfa (ang. alpha channel) w grafice komputerowej jest kanałem, który definiuje przezroczyste obszary grafiki. Jest on zapisywany dodatkowo wewnątrz grafiki razem z trzema wartościami barw składowych RGB.

Inicjalizacja proces wstępnego przypisania wartości zmiennym i obiektom. Każdy obiekt jest inicjalizowany różnymi sposobami zależnie od swojego typu.

Iteracja czynność powtarzania (najczęściej wielokrotnego) tej samej instrukcji (albo wielu instrukcji) w pętli. Mianem iteracji określa się także operacje wykonywane wewnątrz takiej pętli.

Zdarzenie (ang. event) zapis zajścia w systemie komputerowym określonej sytuacji, np. poruszenie myszką, kliknięcie, naciśnięcie klawisza.

pygame.locals moduła zawierający różne stałe używane przez Pygame, np. typy zdarzeń, identyfikatory naciśniętych klawiszy itp.

pygame.time.Clock() tworzy obiekt do śledzenia czasu; `.tick()` – kontroluje ile milisekund upłynęło od poprzedniego wywołania.

pygame.display.set_mode() inicjuje okno lub ekran do wyświetlania, parametry: rozdzielczość w pikselach = (x,y), flagi, głębia koloru.

pygame.display.set_caption() ustawia tytuł okna, parametr: tekst tytułu.

pygame.Surface() obiekt reprezentujący dowolny obrazek (grafikę), który ma określoną rozdzielczość (szerokość i wysokość) oraz format pikseli (głębokość, przezroczystość); SRCALPHA – oznacza, że format pikseli będzie zawierać ustawienie alfa (przezroczystości); `.fill()` – wypełnia obrazek kolorem; `.get_rect()` – zwraca

prostokąt zawierający obrazek, czyli obiekt **Rect**; `.convert_alpha()` – zmienia format pikseli, w tym przezroczystość; `.blit()` – rysuje jeden obrazek na drugim, parametry: źródło, cel.

pygame.draw.ellipse() rysuje okrągły kształt wewnątrz prostokąta, parametry: przestrzeń, kolor, prostokąt.

pygame.draw.rect() rysuje prostokąt na wskazanej powierzchni, parametry: powierzchnia, kolor, obiekt Rect, grubość obramowania.

pygame.font.Font() tworzy obiekt czcionki z podanego pliku; `.render()` – tworzy nową powierzchnię z podanym tekstem, parametry: tekst, antyalias, kolor, tło.

pygame.event.get() pobiera zdarzenia z kolejki zdarzeń; `event.type()` – zwraca identyfikator SDL typu zdarzenia, np. KEYDOWN, KEYUP, MOUSEMOTION, MOUSEBUTTONDOWN, QUIT.

SDL (Simple DirectMedia Layer) międzyplatformowa biblioteka ułatwiająca tworzenie gier i programów multimedialnych.

Rect obiekt pygame.Rect przechowujący współrzędne prostokąta; `.centerx, .x, .y, .top, .bottom, .left, .right` – wirtualne własności obiektu prostokąta określające jego położenie; `.colliderect()` – metoda sprawdza czy dwa prostokąty nachodzą na siebie.

magiczne liczby to takie same wartości liczbowe wielokrotnie używane w kodzie, za każdym razem oznaczające to samo. Stosowanie magicznych liczby jest uważane za złą praktykę ponieważ ich utrudniają czytanie i zrozumienie działania kodu.

stała to zmienna której wartości po początkowym ustaleniu nie będziemy zmieniać. Python nie ma mechanizmów które wymuszają takie zachowanie, jednak przyjmuje się, że zmienne zadeklarowane WIELKIMI_LITERAMI zwykle służą do przechowywania wartości stałych.

generator zwraca jakąś wartość za każdym wywołaniem. Dla świata zewnętrznego generatory zachowują się jak listy (możemy po nich iterować) jedna różnica polega na użyciu pamięci. Listy w całości znajdują się pamięci podczas gdy generatory "tworzą" wartość na zawołanie. Czasem tak samo nazywane są funkcje zwracające generator (ang. generator function).

dziedziczenie w programowaniu obiektowym nazywamy mechanizm współdzielenia funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co oznacza, że oprócz swoich własnych atrybutów oraz zachowań, uzyskuje także te pochodzące z klasy, z której dziedziczy.

przesłanianie w programowaniu obiektowym możemy w klasie dziedziczącej przesłonić metody z klasy nadrzędnej rozszerzając lub całkowicie zmieniając jej działanie

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Materiały

1. `Dokumentacja Pygame (PDF)`

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

   Autorzy Patrz plik "Autorzy"

2.4.5 Bazy danych w Pythonie

Tworzenie i zarządzanie bazami danymi za pomocą Pythona z wykorzystaniem wbudowanego modułu sqlite3 DB-API, a także zewnętrznych bibliotek ORM: Peewee oraz SQLAlchemy.

Poniższe przykłady wykorzystywać będą prostą, wydajną, stosowaną zarówno w prostych, jak i zaawansowanych projektach, bazę danych SQLite3. Gdy zajdzie potrzeba, można je jednak wyorzystać w pracy z innymi bazami, takimi jak np. MySQL, MariaDB czy PostgresSQL.

Do testowania baz danych SQLite można wykorzystać przygotowane przez jej twórców konsolowe narzędzie sqlite3. Zobacz, jak je zainstalować w systemie *Linux* lub *Windows*.

SQL

Jak wiadomo, do obsługi bazy danych wykorzystywany jest strukturalny język zapytań SQL. Jest on m.in. przedmiotem nauki na lekcjach informatyki na poziomie rozszerzonym w szkołach ponadgimnazjalnych. Używając Pythona można łatwo i efektywnie pokazać używanie SQL-a, zarówno z poziomu wiersza poleceń, jak również z poziomu aplikacji internetowych WWW. Na początku zajmiemy się skryptem konsolowym, co pozwala przećwiczyć "surowe" polecenia SQL-a.

Połączenie z bazą

W ulubionym edytorze tworzymy plik `sqlraw.py` i umieszczamy w nim poniższy kod:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import sqlite3

# utworzenie połączenia z bazą przechowywaną na dysku
# lub w pamięci (':memory:')
con = sqlite3.connect('test.db')

# dostęp do kolumn przez indeksy i przez nazwy
con.row_factory = sqlite3.Row

# utworzenie obiektu kursora
cur = con.cursor()
```

Przede wszystkim importujemy moduł `sqlite3` do obsługi baz SQLite3. Następnie w zmiennej `con` tworzymy połączenie z bazą danych przechowywaną w pliku na dysku (`test.db`, nazwa pliku jest dowolona) lub w pamięci, jeśli podamy `':memory:'`. Kolejna instrukcja ustawia właściwość `row_factory` na wartość `sqlite3.Row`, aby możliwy był dostęp do kolumn (pól tabel) nie tylko przez indeksy, ale również przez nazwy. Jest to bardzo przydatne podczas odczytu danych.

Aby móc wykonywać operacje na bazie, potrzebujemy obiektu tzw. kursora, tworzymy go poleceniem `cur = con.cursor()`. I tyle potrzeba, żeby rozpocząć pracę z bazą. Skrypt możemy uruchomić poleceniem podanym niżej, ale

na razie nic się jeszcze nie stanie...

```
~$ python sqlraw.py
```

Model bazy

Zanim będziemy mogli wykonywać podstawowe operacje na bazie danych określane skrótem *CRUD* – *Create* (tworzenie), *Read* (odczyt), *Update* (aktualizacja), *Delete* (usuwanie) - musimy utworzyć tabele i relacje między nimi według zaprojektowanego schematu. Do naszego pliku dopisujemy więc następujący kod:

```
# tworzenie tabel
cur.execute("DROP TABLE IF EXISTS klasa;")

cur.execute("""
    CREATE TABLE IF NOT EXISTS klasa (
        id INTEGER PRIMARY KEY ASC,
        nazwa varchar(250) NOT NULL,
        profil varchar(250) DEFAULT ''
    )""")

cur.executescript("""
    DROP TABLE IF EXISTS uczen;
    CREATE TABLE IF NOT EXISTS uczen (
        id INTEGER PRIMARY KEY ASC,
        imie varchar(250) NOT NULL,
        nazwisko varchar(250) NOT NULL,
        klasa_id INTEGER NOT NULL,
        FOREIGN KEY(klasa_id) REFERENCES klasa(id)
    )""")
```

Jak widać pojedyncze polecenia SQL-a wykonujemy za pomocą metody `.execute()` obiektu kursora. Warto zwrócić uwagę, że w zależności od długości i stopnia skomplikowania instrukcji SQL, możemy je zapisywać w różny sposób. Proste polecenia podajemy w cudzysłowach, bardziej rozbudowane lub kilka instrukcji razem otaczamy potrójnymi cudzysłowami. Ale uwaga: wiele instrukcji wykonujemy za pomocą metody `.executescript()`.

Powyższe polecenia SQL-a tworzą dwie tabele. Tabela "klasa" przechowuje nazwę i profil klasy, natomiast tabela "uczen" zawiera pola przechowujące imię i nazwisko ucznia oraz identyfikator klasy (pole "klasa_id", tzw. klucz obcy), do której należy uczeń. Między tabelami zachodzi relacja jeden-do-wielu, tzn. do jednej klasy może chodzić wielu uczniów.

Po wykonaniu wprowadzonego kodu w katalogu ze skryptem powinien pojawić się plik `test.db`, czyli nasza baza danych. Możemy sprawdzić jej zawartość przy użyciu interpretera *interpretera sqlite3*.

Wstawianie danych

Do skryptu dopisujemy poniższy kod:

```
# wstawiamy jeden rekord danych
cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1A', 'matematyczny'))
cur.execute('INSERT INTO klasa VALUES(NULL, ?, ?);', ('1B', 'humanistyczny'))

# wykonujemy zapytanie SQL, które pobierze id klasy "1A" z tabeli "klasa".
cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1A',))
klasa_id = cur.fetchone()[0]

```

```
# tupla "uczniowie" zawiera tuple z danymi poszczególnych uczniów
uczniowie = (
    (None, 'Tomasz', 'Nowak', klasa_id),
    (None, 'Jan', 'Kos', klasa_id),
    (None, 'Piotr', 'Kowalski', klasa_id)
)

# wstawiamy wiele rekordów
cur.executemany('INSERT INTO uczen VALUES(?,?,?,?)', uczniowie)

# zatwierdzamy zmiany w bazie
con.commit()
```

Do wstawiania pojedynczych rekordów używamy odpowiednich poleceń SQL-a jako argumentów wspominanej metody `.execute()`, możemy też dodawać wiele rekordów na raz posługując się funkcją `.executemany()`. Zarówno w jednym, jak i drugim przypadku wartości pól nie należy umieszczać bezpośrednio w zapytaniu SQL ze względu na możliwe błędy lub ataki typu SQL injection ("wstrzyknięcia" kodu SQL). Zamiast tego używamy zastępników (ang. *placeholder*) w postaci znaków zapytania. Wartości przekazujemy w tupli lub tuplach jako drugi argument.

Warto zwrócić uwagę, na trudności wynikające z relacyjnej struktury bazy danych. Aby dopisać informacje o uczniach do tabeli "Uczeń", musimy znać identyfikator (klucz podstawowy) klasy. Bezpośrednio po zapisaniu danych klasy, możemy go uzyskać dzięki funkcji `.lastrowid()`, która zwraca ostatni *rowid* (unikalny identyfikator rekordu), ale tylko po wykonaniu pojedynczego polecenia *INSERT*. W innych przypadkach trzeba wykonać kwerendę SQL z odpowiednim warunkiem *WHERE*, w którym również stosujemy zastępniki.

Metoda `.fechone()` kursora zwraca listę zawierającą pola wybranego rekordu. Jeżeli interesuje nas pierwszy, i w tym wypadku jedyny, element tej listy dopisujemy `[0]`.

Informacja:

- Wartość `NULL` w poleceniach SQL-a i `None` w tupli z danymi uczniów odpowiadające kluczom głównym umieszczamy po to, aby baza danych utworzyła je automatycznie. Można by je pominąć, ale wtedy w poleceniu wstawiania danych musimy wymienić nazwy pól, np. `INSERT INTO klasa (nazwa, profil) VALUES (?, ?), ('1C', 'biologiczny')`.
- Jeżeli podajemy jedną wartość w tupli jako argument metody .execute(), musimy pamiętać o umieszczeniu dodatkowgo przecinka, np. `('1A',)`, ponieważ w ten sposób tworzymy w Pythonie 1-elementowe tuple. W przypadku wielu wartości przecinek nie jest wymagany.

Metoda `.commit()` zatwierdza, tzn. zapisuje w bazie danych, operacje danej transakcji, czyli grupy operacji, które albo powinny zostać wykonane razem, albo powinny zostać odrzucone ze względu na naruszenie zasad ACID (Atomicity, Consistency, Isolation, Durability – Atomowość, Spójność, Izolacja, Trwałość).

Pobieranie danych

Pobieranie danych (czyli *kwerenda*) wymaga polecenia *SELECT* języka SQL. Dopisujemy więc do naszego skryptu funkcję, która wyświetli listę uczniów oraz klas, do których należą:

```
# pobieranie danych z bazy
def czytajdane():
    """Funkcja pobiera i wyświetla dane z bazy."""
    cur.execute(
        """
        SELECT uczen.id,imie,nazwisko,nazwa FROM uczen,klasa
```

```
64          WHERE uczen.klasa_id=klasa.id
65          """)
66      uczniowie = cur.fetchall()
67      for uczen in uczniowie:
68          print(uczen['id'], uczen['imie'], uczen['nazwisko'], uczen['nazwa'])
69      print()
70
71
72  czytajdane()
```

Funkcja `czytajdane()` wykonuje zapytanie SQL pobierające wszystkie dane z dwóch powiązanych tabel: "uczen" i "klasa". Wydobywamy *id ucznia*, *imię* i *nazwisko*, a także *nazwę* klasy na podstawie warunku w klauzuli *WHERE*. Wynik, czyli wszystkie pasujące rekordy zwrócone przez metodę `.fetchall()`, zapisujemy w zmiennej `uczniowie` w postaci tupli. Jej elementy odczytujemy w pętli `for` jako listę `uczen`. Dzięki ustawieniu właściwości `.row_factory` połączenia z bazą na `sqlite3.Row` odczytujemy poszczególne pola podając nazwy zamiast indeksów, np. `uczen['imie']`.

Informacja: Warto zwrócić uwagę na wykorzystanie w powyższym kodzie potrójnych cudzysłowów (`"..."`). Na początku funkcji umieszczono w nich opis jej działania, dalej wykorzystano do zapisania długiego zapytania SQL-a.

Modyfikacja i usuwanie danych

Do skryptu dodajemy jeszcze kilka linii:

```
74  # zmiana klasy ucznia o identyfikatorze 2
75  cur.execute('SELECT id FROM klasa WHERE nazwa = ?', ('1B',))
76  klasa_id = cur.fetchone()[0]
77  cur.execute('UPDATE uczen SET klasa_id=? WHERE id=?', (klasa_id, 2))
78
79  # usunięcie ucznia o identyfikatorze 3
80  cur.execute('DELETE FROM uczen WHERE id=?', (3,))
81
82  czytajdane()
83
```

Aby zmienić przypisanie ucznia do klasy, pobieramy identyfikor klasy za pomocą metody `.execute()` i polecenia *SELECT* SQL-a z odpowiednim warunkiem. Później konstruujemy zapytanie *UPDATE* wykorzystując zastępniki i wartości przekazywane w tupli (zwróć uwagę na dodatkowy przecinek(!)) – w efekcie zmieniamy przypisanie ucznia do klasy.

Następnie usuwamy dane ucznia o identyfikatorze 3, używając polecenia SQL *DELETE*. Wywołanie funkcji `czytajdane()` wyświetla zawartość bazy po zmianach.

Na koniec zamykamy połącznie z bazą, wywołując metodę `.close()`, dzięki czemu zapisujemy dokonane zmiany i zwalniamy zarezerwowane przez skrypt zasoby.

Zadania

- Przeczytaj *opis przykładowej funkcji pobierającej dane* z pliku tekstowego w formacie *csv*. W skrypcie `sqlraw.py` zaimportuj tę funkcję i wykorzystaj do pobrania i wstawienia danych do bazy.
- Postaraj się przedstawioną aplikację wyposażyć w konsolowy interfejs, który umożliwi operacje odczytu, zapisu, modyfikowania i usuwania rekordów. Dane powinny być pobierane z klawiatury od użytkownika.

- Zobacz, jak zintegrować obsługę bazy danych przy użyciu modułu *sqlite3* Pythona z aplikacją internetową na przykładzie scenariusza "ToDo".

Źródła

- `sqlraw.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Systemy ORM

Znajomość języka SQL jest oczywiście niezbędna, aby korzystać z wszystkich możliwości baz danych, niemniej w wielu niespecjalistycznych projektach można je obsługiwać inaczej, tj. za pomocą systemów ORM (ang. Object-Relational Mapping – mapowanie obiektowo-relacyjne). Pozwalają one traktować tabele w sposób obiektowy, co bywa wygodniejsze w budowaniu logiki aplikacji.

Używanie systemów ORM, takich jak *Peewee* czy *SQLAlchemy*, w prostych projektach sprowadza się do schematu, który poglądowo można opisać w trzech krokach:

1. Nawiązanie połączenia z bazą
2. Deklaracja modelu opisującego bazę i utworzenie struktury bazy
3. Wykonywanie operacji *CRUD*

Poniżej spróbujemy pokazać, jak operacje wykonywane przy użyciu wbudowanego w Pythona modułu *sqlite3* zrealizować przy użyciu technik ORM.

Informacja: Wyjaśnienia podanego niżej kodu są w wielu miejscach uproszczone. Ze względu na przejrzystość i poglądowość instrukcji nie wgłębiamy się w techniczne różnice w implementacji technik ORM w obydwu rozwiązaniach. Poznanie ich interfejsu jest wystarczające, aby efektywnie obsługiwać bazy danych. Co ciekawe, dopóki używamy bazy SQLite3, systemy ORM można traktować jako swego rodzaju nakładkę na owmówiony wyżej moduł *sqlite3* wbudowany w Pythona.

Połączenie z bazą

W ulubionym edytorze utwórz dwa puste pliki o nazwach `ormpw.py` i `ormsa.py`. Pierwszy z nich zawierał będzie kod wykorzystujący ORM Peewee, drugi ORM SQLAlchemy.

```
1 #! /usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
```

```
import os
from peewee import *

if os.path.exists('test.db'):
    os.remove('test.db')
# tworzymy instancję bazy używanej przez modele
baza = SqliteDatabase('test.db')  # ':memory:'


class BazaModel(Model):  # klasa bazowa
    class Meta:
        database = baza
```

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import os
from sqlalchemy import Column, ForeignKey, Integer, String, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship, sessionmaker

if os.path.exists('test.db'):
    os.remove('test.db')
# tworzymy instancję klasy Engine do obsługi bazy
baza = create_engine('sqlite:///test.db')  # ':memory:'

# klasa bazowa
BazaModel = declarative_base()
```

W jednym i drugim przypadku importujemy najpierw potrzebne klasy. Następnie tworzymy instancje `baza` służące do nawiązania połączeń z bazą przechowywaną w pliku `test.db`. Jeżeli zamiast nazwy pliku, podamy `:memory:` bazy umieszczone zostaną w pamięci RAM (przydatne podczas testowania).

Informacja: Moduły `os` i `sys` nie są niezbędne do działania prezentowanego kodu, ale można z nich skorzystać, kiedy chcemy sprawdzić obecność pliku na dysku (`os.path.ispath()`) lub zatrzymać wykonywanie skryptu w dowolnym miejscu (`sys.exit()`). W podanych przykładach usuwamy plik bazy, jeżeli znajduje się na dysku, aby zapewnić bezproblemowe działanie kompletnych skryptów.

Model danych i baza

Przez model rozumiemy tutaj deklaracje klas i ich właściwości (atrybutów) opisujące obiekty, którymi się zajmujemy. Systemy ORM na podstawie klas tworzą odpowiednie tablice, pola, uwzględniając ich typy i powiązania. Wzajemne powiązanie klas i ich właściwości z tabelami i kolumnami w bazie stanowi właśnie istotę mapowania relacyjno-obiektowego.

```
# klasy Klasa i Uczen opisują rekordy tabel "klasa" i "uczen"
# oraz relacje między nimi
class Klasa(BazaModel):
    nazwa = CharField(null=False)
    profil = CharField(default='')


class Uczen(BazaModel):
    imie = CharField(null=False)
```

```
    nazwisko = CharField(null=False)
    klasa = ForeignKeyField(Klasa, related_name='uczniowie')


baza.connect()  # nawiązujemy połączenie z bazą
baza.create_tables([Klasa, Uczen], True)  # tworzymy tabele
```

```
# klasy Klasa i Uczen opisują rekordy tabel "klasa" i "uczen"
# oraz relacje między nimi
class Klasa(BazaModel):
    __tablename__ = 'klasa'
    id = Column(Integer, primary_key=True)
    nazwa = Column(String(100), nullable=False)
    profil = Column(String(100), default='')
    uczniowie = relationship('Uczen', backref='klasa')


class Uczen(BazaModel):
    __tablename__ = 'uczen'
    id = Column(Integer, primary_key=True)
    imie = Column(String(100), nullable=False)
    nazwisko = Column(String(100), nullable=False)
    klasa_id = Column(Integer, ForeignKey('klasa.id'))


# tworzymy tabele
BazaModel.metadata.create_all(baza)
```

W obydwu przypadkach deklarowanie modelu opiera się na pewnej "klasie" podstawowej, którą nazwaliśmy `BazaModel`. Dziedzicząc z niej, deklarujemy następnie własne klasy o nazwach *Klasa* i *Uczen* reprezentujące tabele w bazie. Właściwości tych klas odpowiadają kolumnom; w SQLAlchemy używamy nawet klasy o nazwie `Column()`, która wyraźnie wskazuje na rodzaj tworzonego atrybutu. Obydwa systemy wymagają określenia *typu danych* definiowanych pól. Służą temu odpowiednie klasy, np. `CharField()` lub `String()`. Możemy również definiować dodatkowe cechy pól, takie jak np. nie zezwalanie na wartości puste (`null=False` lub `nullable=False`) lub określenie wartości domyślnych (`default=''`).

Warto zwrócić uwagę, na sposób określania relacji. W *Peewee* używamy konstruktora klasy: `ForeignKeyField(Klasa, related_name = 'uczniowie')`. Przyjmuje on nazwę klasy powiązanej, z którą tworzymy relację, i nazwę atrybutu określającego relację zwrotną w powiązanej klasie. Dzięki temu wywołanie w postaci `Klasa.uczniowie` da nam dostęp do obiektów reprezentujących uczniów przypisanych do danej klasy. Zuważmy, że *Peewee* nie wymaga definiowania kluczy głównych, są tworzone automatycznie pod nazwą `id`.

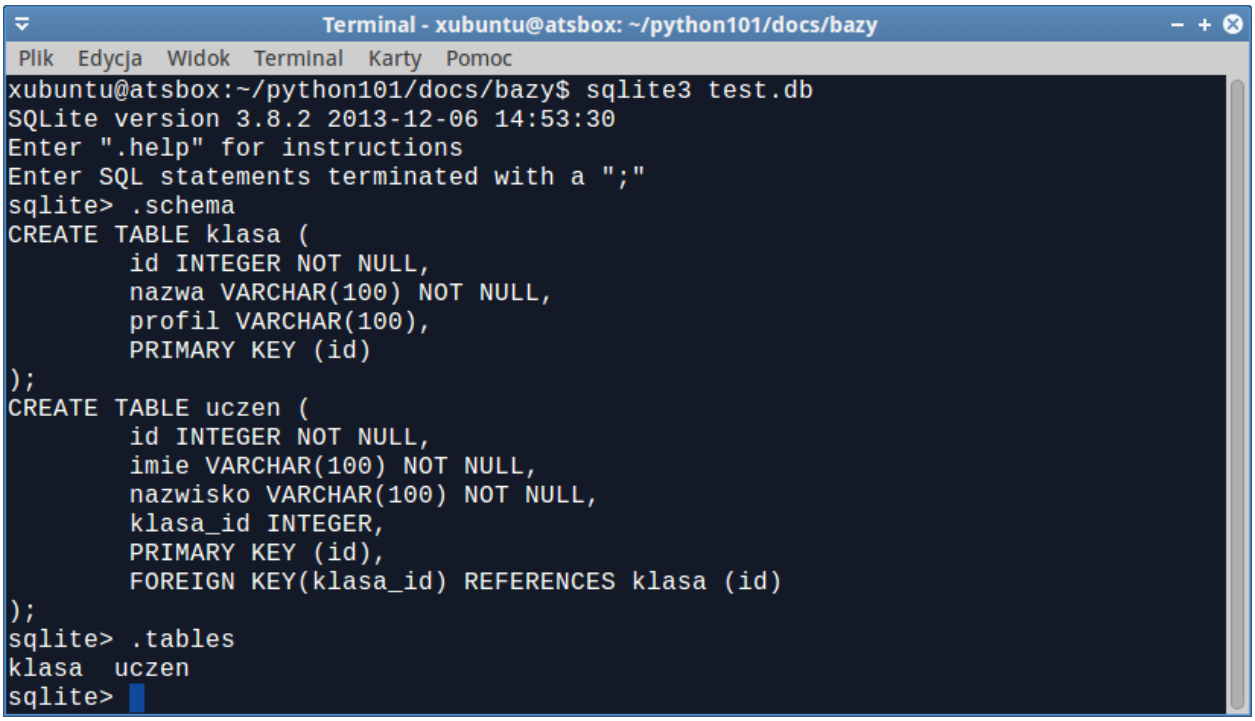
W SQLAlchemy dla odmiany nie tylko jawnie określamy klucze główne (`primary_key=True`), ale i podajemy nazwy tabel (`__tablename__ = 'klasa'`). Klucz obcy oznaczamy odpowiednim parametrem w klasie definiującej pole (`Column(Integer, ForeignKey('klasa.id'))`). Relację zwrotną tworzymy za pomocą konstruktora `relationship('Uczen', backref='klasa')`, w którym podajemy nazwę powiązanej klasy i nazwę atrybutu tworzącego powiązanie. W tym wypadku wywołanie typu `uczen.klasa` udostępni obiekt reprezentujący klasę, do której przypisano ucznia.

Po zdefiniowaniu przemyślanego modelu, co jest relatywnie najtrudniejsze, trzeba przetestować działanie mechanizmów ORM w praktyce, czyli utworzyć tabele i kolumny w bazie. W Peewee łączymy się z bazą i wywołujemy metodę `.create_tables()`, której podajemy nazwy klas reprezentujących tabele. Dodatkowy parametr `True` powoduje sprawdzenie przed utworzeniem, czy tablic w bazie już nie ma. SQLAlchemy wymaga tylko wywołania metody `.create_all()` kontenera *metadata* zawartego w klasie bazowej.

Podane kody można już uruchomić, oba powinny utworzyć bazę `test.db` w katalogu, z którego uruchamiamy

skrypt.

Informacja: Warto wykorzystać *interpreter sqlite3* i sprawdzić, jak wygląda kod tworzący tabele wygenerowany przez ORM-y. Poniżej przykład ilustrujący SQLAlchemy.



Operacje CRUD

Wstawianie i odczytywanie danych

Podstawowe operacje wykonywane na bazie, np, wstawianie i odczytywanie danych, w Peewee wykonywane są za pomocą obiektów reprezentujących rekordy zdefiniowanych tabel oraz ich metod. W SQLAlchemy oprócz obiektów wykorzystujemy metody sesji, w ramach której komunikujemy się z bazą.

```
# dodajemy dwie klasy, jeżeli tabela jest pusta
if Klasa().select().count() == 0:
    inst_klasa = Klasa(nazwa='1A', profil='matematyczny')
    inst_klasa.save()
    inst_klasa = Klasa(nazwa='1B', profil='humanistyczny')
    inst_klasa.save()

# tworzymy instancję klasy Klasa reprezentującą klasę "1A"
inst_klasa = Klasa.select().where(Klasa.nazwa == '1A').get()

# lista uczniów, których dane zapisane są w słownikach
uczniowie = [
    {'imie': 'Tomasz', 'nazwisko': 'Nowak', 'klasa': inst_klasa},
    {'imie': 'Jan', 'nazwisko': 'Kos', 'klasa': inst_klasa},
    {'imie': 'Piotr', 'nazwisko': 'Kowalski', 'klasa': inst_klasa}
]
```

```
50
51  # dodajemy dane wielu uczniów
52  Uczen.insert_many(uczniowie).execute()
53
54  # odczytujemy dane z bazy
55
56
57  def czytajdane():
58      for uczen in Uczen.select().join(Klasa):
59          print(uczen.id, uczen.imie, uczen.nazwisko, uczen.klasa.nazwa)
60      print()
61
62
63  czytajdane()
```

```
39  # tworzymy sesję, która przechowuje obiekty i umożliwia "rozmowę" z bazą
40  BDSesja = sessionmaker(bind=baza)
41  sesja = BDSesja()
42
43  # dodajemy dwie klasy, jeżeli tabela jest pusta
44  if not sesja.query(Klasa).count():
45      sesja.add(Klasa(nazwa='1A', profil='matematyczny'))
46      sesja.add(Klasa(nazwa='1B', profil='humanistyczny'))
47
48  # tworzymy instancję klasy Klasa reprezentującą klasę "1A"
49  inst_klasa = sesja.query(Klasa).filter_by(nazwa='1A').one()
50
51  # dodajemy dane wielu uczniów
52  sesja.add_all([
53      Uczen(imie='Tomasz', nazwisko='Nowak', klasa_id=inst_klasa.id),
54      Uczen(imie='Jan', nazwisko='Kos', klasa_id=inst_klasa.id),
55      Uczen(imie='Piotr', nazwisko='Kowalski', klasa_id=inst_klasa.id),
56  ])
57
58
59  def czytajdane():
60      for uczen in sesja.query(Uczen).join(Klasa).all():
61          print(uczen.id, uczen.imie, uczen.nazwisko, uczen.klasa.nazwa)
62      print()
63
64
65  czytajdane()
```

Dodawanie informacji w systemach ORM polega na utworzeniu instancji odpowiedniego obiektu i podaniu w jego konstruktorze wartości atrybutów reprezentujących pola rekordu: `Klasa(nazwa = '1A', profil = 'matematyczny')`. Utworzony rekord zapisujemy metodą `.save()` obiektu w Peewee lub metodą `.add()` *sesji* w SQLAlchemy. Można również dodawać wiele rekordów na raz. Peewee oferuje metodę `.insert_many()`, która jako parametr przyjmuje listę słowników zawierających dane w formacie "klucz":"wartość", przy czym kluczem jest nazwa pola klasy (tabeli). SQLAlchemy ma metodę `.add_all()` wymagającą listy konstruktorów obiektów, które chcemy dodać.

Zanim dodamy pierwsze informacje sprawdzamy, czy w tabeli *klasa* są jakieś wpisy, a więc wykonujemy prostą kwerendę zliczającą. Peewee używa metod odpowiednich obiektów: `Klasa().select().count()`, natomiast SQLAlchemy korzysta metody `.query()` sesji, która pozwala pobierać dane z określonej jako klasa tabeli. Obydwa rozwiązania umożliwiają łańcuchowe wywoływanie charakterytycznych dla kwerend operacji poprzez "doklejanie" kolejnych metod, np. `sesja.query(Klasa).count()`.

Tak właśnie konstruujemy kwerendy warunkowe. W Peewee definiujemy warunki jako prametry metody `.where(Klasa.nazwa == '1A')`. Podobnie w SQLAlchemy, tyle, że metody sesji inaczej się nazywają i przyjmują postać `.filter_by(nazwa = '1A')` lub `.filter(Klasa.nazwa == '1A')`. Pierwsza wymaga podania warunku w formacie "klucz"="wartość", druga w postaci wyrażenia SQL (należy uważać na użycie poprawnego operatora ==).

Pobieranie danych z wielu tabel połączonych relacjami może być w porównaniu do zapytań SQL-a bardzo proste. W zależności od ORM-a wystarcza polecenie: `Uczen.select()` lub `sesja.query(Uczen).all()`, ale przy próbie odczytu klasy, do której przypisano ucznia (`inst_uczen.klasa.nazwa`), wykonane zostanie dodatkowe zapytanie, co nie jest efektywne. Dlatego lepiej otwarcie wskazywać na powiązania między obiektami, czyli w zależności od ORM-u używać: `Uczen.select().join(Klasa)` lub `sesja.query(Uczen).join(Klasa).all()`. Tak właśnie postępujemy w bliźniaczych funkcjach `czytajdane()`, które pokazują, jak pobierać i wyświetlać wszystkie rekordy z tabel powiązanych relacjami.

Systemy ORM oferują pewne ułatwiania w zależności od tego, ile rekordów lub pól i w jakiej formie chcemy wydobyć. Metody w Peewee:

- `.get()` - zwraca pojedynczy rekord pasujący do zapytania lub wyjątek `DoesNotExist`, jeżeli go brak;
- `.first()` - zwróci z kolei pierwszy rekord ze wszystkich pasujących.

Metody SQLAlchemy:

- `.get(id)` - zwraca pojedynczy rekord na podstawie podanego identyfikatora;
- `.one()` - zwraca pojedynczy rekord pasujący do zapytania lub wyjątek `DoesNotExist`, jeżeli go brak;
- `.scalar()` - zwraca pierwszy element pierwszego zwróconego rekordu lub wyjątek `MultipleResultsFound`;
- `.all()` - zwraca pasujące rekordy w postaci listy.

Informacja: Mechanizm sesji jest unikalny dla SQLAlchemy, pozwala m. in. zarządzać transakcjami i połączeniami z wieloma bazami. Stanowi "przechowalnię" dla tworzonych obiektów, zapamiętuje wykonywane na nich operacje, które mogą zostać zapisane w bazie lub w razie potrzeby odrzucone. W prostych aplikacjach wykorzystuje się jedną instancję sesji, w bardziej złożonych można korzystać z wielu. Instancja sesji (`sesja = BDSesja()`) tworzona jest na podstawie klasy, która z kolei powstaje przez wywołanie konstruktora z opcjonalnym parametrem wskazującym bazę: `BDSesja = sessionmaker(bind=baza)`. Jak pokazano wyżej, obiekt sesji zawiera metody pozwalające komunikować się z bazą. Warto również zauważyć, że po wykonaniu wszystkich zamierzonych operacji w ramach sesji zapisujemy dane do bazy wywołując polecenie `sesja.commit()`.

Modyfikowanie i usuwanie danych

Systemy ORM ułatwiają modyfikowanie i usuwanie danych z bazy, ponieważ operacje te sprowadzają się do zmiany wartości pól klasy reprezentującej tabelę lub do usunięcia instancji danej klasy.

```
65 # zmiana klasy ucznia o identyfikatorze 2
66 inst_uczen = Uczen().select().join(Klasa).where(Uczen.id == 2).get()
67 inst_uczen.klasa = Klasa.select().where(Klasa.nazwa == '1B').get()
68 inst_uczen.save()  # zapisanie zmian w bazie
69
70 # usunięcie ucznia o identyfikatorze 3
71 Uczen.select().where(Uczen.id == 3).get().delete_instance()
72
73 czytajdane()
74
75 baza.close()
```

```
# zmiana klasy ucznia o identyfikatorze 2
inst_uczen = sesja.query(Uczen).filter(Uczen.id == 2).one()
inst_uczen.klasa_id = sesja.query(Klasa.id).filter(
    Klasa.nazwa == '1B').scalar()

# usunięcie ucznia o identyfikatorze 3
sesja.delete(sesja.query(Uczen).get(3))

czytajdane()

# zapisanie zmian w bazie i zamknięcie sesji
sesja.commit()
sesja.close()
```

Załóżmy, że chcemy zmienić przypisanie ucznia do klasy. W obydwu systemach tworzymy więc obiekt reprezentujący ucznia o identyfikatorze "2". Stosujemy omówione wyżej metody zapytań. W następnym kroku modyfikujemy odpowiednie pole tworzące relację z tabelą "klasy", do którego przypisujemy pobrany w zapytaniu obiekt (Peewee) lub identyfikator (SQLAlchemy). Różnice, tzn. przypisywanie obiektu lub identyfikatora, wynikają ze sposobu definiowania modeli w obu rozwiązanich.

Usuwanie jest jeszcze prostsze. W Peewee wystarczy do zapytania zwracającego obiekt reprezentujący ucznia o podanym id "dokleić" odpowiednią metodę: `Uczen.select().where(Uczen.id == 3).get().delete_instance()`. W SQLAlchemy korzystamy jak zwykle z metody sesji, której przekazujemy obiekt reprezentujący ucznia: `sesja.delete(sesja.query(Uczen).get(3))`.

Po zakończeniu operacji wykonywanych na danych powinniśmy pamiętać o zamknięciu połączenia, robimy to używając metody obiektu bazy `baza.close()` (Peewee) lub sesji `sesja.close()` (SQLAlchemy). UWAGA: operacje dokonywane podczas sesji w SQLAlchemy muszą zostać zapisane w bazie, dlatego przed zamknięciem połączenia trzeba umieścić polecenie `sesja.commit()`.

Zadania

- Spróbuj dodać do bazy korzystając z systemu Peewee lub SQLAlchemy wiele rekordów na raz pobranych z pliku. Wykorzystaj i zmodyfikuj funkcję `pobierz_dane()` opisaną w materiale *Dane z pliku*.
- Postaraj się przedstawione aplikacje wyposażyć w konsolowy interfejs, który umożliwi operacje odczytu, zapisu, modyfikowania i usuwania rekordów. Dane powinny być pobierane z klawiatury od użytkownika.
- Przedstawione rozwiązania warto użyć w aplikacjach internetowych jako relatywnie szybki i łatwy sposób obsługi danych. Zobacz, jak to zrobić na przykładzie scenariusza aplikacji *Quiz ORM*.
- Przejrzyj scenariusz aplikacji internetowej *Czat*, zbudowanej z wykorzystaniem frameworku *Django*, korzystającego z własnego modelu ORM.

Źródła

- `orm.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

SQL v. ORM

Bazy danych są niezbędnym składnikiem większości aplikacji. Poniżej zwięźle pokażemy, w jaki sposób z wykorzystaniem Pythona można je obsługiwać przy użyciu języka *SQL*, jak i systemów *ORM* na przykładzie rozwiązania *Peewee*.

Informacja: Niniejszy materiał koncentruje się na poglądowym wyeksponowaniu różnic w kodowaniu, komentarz ograniczono do minimum. Dokładne wyjaśnienia poszczególnych instrukcji znajdziesz w materiale *SQL* oraz *Systemy ORM*. W tym ostatnim omówiono również ORM SQLAlchemy.

Połączenie z bazą

Na początku pliku `sqlraw.py` umieszczamy kod, który importuje moduł do obsługi bazy *SQLite3* i przygotowuje obiekt kursora, który posłuży nam do wydawania poleceń SQL:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import sqlite3

# utworzenie połączenia z bazą przechowywaną w pamięci RAM
con = sqlite3.connect(':memory:')

# dostęp do kolumn przez indeksy i przez nazwy
con.row_factory = sqlite3.Row

# utworzenie obiektu kursora
cur = con.cursor()
```

System ORM Peewee inicjujemy w pliku `ormpw.py` tworząc klasę bazową, która zapewni połączenie z bazą:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import os
from peewee import *

if os.path.exists('test.db'):
    os.remove('test.db')
# tworzymy instancję bazy używanej przez modele
baza = SqliteDatabase('test.db')  # ':memory:'

```

```

# BazaModel to klasa bazowa dla klas Grupa i Uczen, które
# opisują rekordy tabel "grupa" i "uczen" oraz relacje między nimi
class BazaModel(Model):
    class Meta:
        database = baza
```

Informacja: Parametr `:memory:` powoduje utworzenie bazy danych w pamięci operacyjnej, która istnieje tylko w czasie wykonywania programu. Aby utworzyć trwałą bazę, zastąp omawiany parametr nazwę pliku, np. `test.db`.

Model bazy

Dane w bazie zorganizowane są w tabelach, połączonych najczęściej relacjami. Aby utworzyć tabele `grupa` i `uczen` powiązane relacją jeden-do-wielu, musimy wydać następujące polecenia SQL:

```
# tworzenie tabel
cur.executescript("""
    DROP TABLE IF EXISTS grupa;
    CREATE TABLE IF NOT EXISTS grupa (
        id INTEGER PRIMARY KEY ASC,
        nazwa varchar(250) NOT NULL,
        profil varchar(250) DEFAULT ''
    );
    DROP TABLE IF EXISTS uczen;
    CREATE TABLE IF NOT EXISTS uczen (
        id INTEGER PRIMARY KEY ASC,
        imie varchar(250) NOT NULL,
        nazwisko varchar(250) NOT NULL,
        grupa_id INTEGER NOT NULL,
        FOREIGN KEY(grupa_id) REFERENCES grupa(id)
    )""")
```

Wydawanie poleceń SQL-a wymaga koncentracji na poprawności użycia tego języka, systemy ORM izolują nas od takich szczegółów pozwalając skupić się na logice danych. Tworzymy więc klasy opisujące nasze obiekty, tj. grupy i uczniów. Na podstawie właściwości tych obiektów system ORM utworzy odpowiednie pola tabel. Konkretna grupa lub uczeń, czyli instancje klasy, reprezentować będą rekordy w tabelach.

```
class Grupa(BazaModel):
    nazwa = CharField(null=False)
    profil = CharField(default='')


class Uczen(BazaModel):
    imie = CharField(null=False)
    nazwisko = CharField(null=False)
    grupa = ForeignKeyField(Grupa, related_name='uczniowie')


baza.connect()  # nawiązujemy połączenie z bazą
baza.create_tables([Grupa, Uczen], True)  # tworzymy tabele
```

Ćwiczenie 1

Utwórz za pomocą tworzonych skryptów bazy w plikach o nazwach `sqlraw.db` oraz `peewee.db`. Następnie otwórz te bazy w interpreterze Sqlite i wykonaj podane niżej polecenia. Porównaj struktury utworzonych tabel.

```
sqlite> .tables
sqlite> .schema grupa
sqlite> .schema uczen
```

Wstawianie danych

Chcemy wstawić do naszych tabel dane dwóch grup oraz dwóch uczniów. Korzystając z języka SQL użyjemy następujących poleceń:

```
# wstawiamy dane uczniów
cur.execute('INSERT INTO grupa VALUES(NULL, ?, ?);', ('1A', 'matematyczny'))
cur.execute('INSERT INTO grupa VALUES(NULL, ?, ?);', ('1B', 'humanistyczny'))

# wykonujemy zapytanie SQL, które pobierze id grupy "1A" z tabeli "grupa".
cur.execute('SELECT id FROM grupa WHERE nazwa = ?', ('1A',))
grupa_id = cur.fetchone()[0]

# wstawiamy dane uczniów
cur.execute('INSERT INTO uczen VALUES(?,?,?,?)',
            (None, 'Tomasz', 'Nowak', grupa_id))
cur.execute('INSERT INTO uczen VALUES(?,?,?,?)',
            (None, 'Adam', 'Kowalski', grupa_id))

# zatwierdzamy zmiany w bazie
con.commit()
```

W systemie ORM pracujemy z instancjami `inst_grupa` i `inst_uczen`. Nadajemy wartości ich atrybutom i korzystamy z ich metod:

```
# dodajemy dwie klasy, jeżeli tabela jest pusta
if Grupa.select().count() == 0:
    inst_grupa = Grupa(nazwa='1A', profil='matematyczny')
    inst_grupa.save()
    inst_grupa = Grupa(nazwa='1B', profil='humanistyczny')
    inst_grupa.save()

# tworzymy instancję klasy Grupa reprezentującą klasę "1A"
inst_grupa = Grupa.select().where(Grupa.nazwa == '1A').get()
# dodajemy uczniów
inst_uczen = Uczen(imie='Tomasz', nazwisko='Nowak', grupa=inst_grupa)
inst_uczen.save()
inst_uczen = Uczen(imie='Adam', nazwisko='Kowalski', grupa=inst_grupa)
inst_uczen.save()
```

Pobieranie danych

Pobieranie danych (czyli *kwerenda*) wymaga polecenia *SELECT* języka SQL. Aby wyświetlić dane wszystkich uczniów zapisane w bazie użyjemy kodu:

```
def czytajdane():
    """Funkcja pobiera i wyświetla dane z bazy"""
    cur.execute(
        """
        SELECT uczen.id,imie,nazwisko,nazwa FROM uczen,grupa
        WHERE uczen.grupa_id=grupa.id
        """)
    uczniowie = cur.fetchall()
    for uczen in uczniowie:
        print(uczen['id'], uczen['imie'], uczen['nazwisko'], uczen['nazwa'])
    print()


czytajdane()
```

W systemie ORM korzystamy z metody `select()` instancji reprezentującej ucznia. Dostęp do danych przechowywanych w innych tabelach uzyskujemy dzięki wyrażeniom typu `inst_uczen.grupa.nazwa`, które generuje podzapytanie zwracające obiekt grupy przypisanej uczniowi.

```
def czytajdane():
    """Funkcja pobiera i wyświetla dane z bazy"""
    for uczen in Uczen.select():  # lub szybsze: Uczen.select().join(Grupa)
        print(uczen.id, uczen.imie, uczen.nazwisko, uczen.grupa.nazwa)
    print()


czytajdane()
```

Wskazówka: Ze względów wydajnościowych pobieranie danych z innych tabel możemy zasygnalizować już w głównej kwerendzie, używając metody `join()`, np.: `Uczen.select().join(Grupa)`.

Modyfikacja i usuwanie danych

Edycja danych zapisanych już w bazie to kolejna częsta operacja. Jeżeli chcemy przepisać ucznia z grupy do grupy, w przypadku czystego SQL-a musimy pobrać identyfikator ucznia (`uczen_id = cur.fetchone()[0]`), identyfikator grupy (`grupa_id = cur.fetchone()[0]`) i użyć ich w klauzuli `UPDATE`. Usuwany rekord z kolei musimy wskazać w klauzuli `WHERE`.

```
# przepisanie ucznia do innej grupy
cur.execute('SELECT id FROM uczen WHERE nazwisko="Nowak"')
uczen_id = cur.fetchone()[0]
cur.execute('SELECT id FROM grupa WHERE nazwa = ?', ('1B',))
grupa_id = cur.fetchone()[0]
cur.execute('UPDATE uczen SET grupa_id=? WHERE id=?', (grupa_id, uczen_id))
czytajdane()

# usunięcie ucznia o identyfikatorze 1
cur.execute('DELETE FROM uczen WHERE id=?', (1,))
czytajdane()

con.close()
```

W systemie ORM tworzymy instancję reprezentującą ucznia i zmieniamy jej właściwości (`inst_uczen.grupa = Grupa.select().where(Grupa.nazwa == '1B').get()`). Usuwając dane w przypadku systemu ORM,

usuwamy instancję wskazanego obiektu:

```
59 # przepisanie ucznia do innej klasy
60 inst_uczen = Uczen.select().join(Grupa).where(Uczen.nazwisko == 'Nowak').get()
61 inst_uczen.grupa = Grupa.select().where(Grupa.nazwa == '1B').get()
62 inst_uczen.save()  # zapisanie zmian w bazie
63 czytajdane()
64
65 # usunięcie ucznia o identyfikatorze 1
66 inst_uczen = Uczen.select().where(Uczen.id == 1).get()
67 inst_uczen.delete_instance()
68 czytajdane()
69
70 baza.close()
```

Informacja: Po wykonaniu wszystkich założonych operacji na danych połączenie z bazą należy zamknąć, zwalniając w ten sposób zarezerwowane zasoby. W przypadku modułu `sqlite3` wywołujemy polecenie `con.close()`, w Peewee `baza.close()`.

Podsumowanie

Bazę danych można obsługiwać za pomocą języka SQL na niskim poziomie. Zyskujemy wtedy na szybkości działania, ale tracimy przejrzystość kodu, łatwość jego przeglądania i rozwijania. O ile w prostych zastosowaniach można to zaakceptować, o tyle w bardziej rozbudowanych projektach używa się systemów ORM, które pozwalają zarządzać danymi nie w formie tabel, pól i rekordów, ale w formie obiektów reprezentujących logicznie spójne dane. Takie podejście lepiej odpowiada obiektowemu wzorcowi projektowania aplikacji.

Dodatkową zaletą systemów ORM, nie do przecenienia, jest większa odporność na błędy i ewentualne ataki na dane w bazie.

Systemy ORM można łatwo integrować z programami desktopowymi i frameworkami przeznaczonymi do tworzenia aplikacji sieciowych. Wśród tych ostatnich znajdziemy również takie, w których system ORM jest podstawowym składnikiem, np. *Django*.

Zadania

- Wykonaj scenariusz aplikacji *Quiz ORM*, aby zobaczyć przykład wykorzystania systemów ORM w aplikacjach internetowych.
- Wykonaj scenariusz aplikacji internetowej *Czat (cz. 1)*, zbudowanej z wykorzystaniem frameworku *Django*, korzystającego z własnego modelu ORM.

Źródła

- `sqlorm.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik “Autorzy”

Dane z pliku

Dane z tabel w bazach MS Accessa lub LibreOffice Base'a możemy eksportować do formatu *csv*, czyli pliku tekstowego, w którym każda linia reprezentuje pojedynczy rekord, a wartości pól oddzielone są jakimś separatorem, najczęściej przecinkiem.

Załóżmy więc, że mamy plik `uczniowie.csv` zawierający dane uczniów w formacie: `Jan,Nowak,2`. Poniżej podajemy przykład funkcji, która odczyta dane i zwróci je w użytecznej postaci:

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-

import os


def pobierz_dane(plikcsv):
    """
    Funkcja zwraca tuplę tupli zawierających dane pobrane z pliku csv
    do zapisania w tabeli.
    """
    dane = []  # deklarujemy pustą listę
    if os.path.isfile(plikcsv):  # sprawdzamy czy plik istnieje na dysku
        with open(plikcsv, "r") as zawartosc:  # otwieramy plik do odczytu
            for linia in zawartosc:
                linia = linia.replace("\n", "")  # usuwamy znaki końca linii
                linia = linia.replace("\r", "")  # usuwamy znaki końca linii
                linia = linia.decode("utf-8")  # odczytujemy znaki jako utf-8
                # dodajemy elementy do tupli a tuplę do listy
                dane.append(tuple(linia.split(",")))
    else:
        print "Plik z danymi", plikcsv, "nie istnieje!"

    return tuple(dane)  # przekształcamy listę na tuplę i zwracamy ją
```

Na początku funkcji `pobierz_dane()` sprawdzamy, czy istnieje plik podany jako argumet. Wykorzystujemy metodę `isfile()` z modułu `os`, który należy wcześniej zaimportować. Następnie w konstrukcji `with` otwieramy plik i wczytujemy jego treść do zmiennej `zawartosc`. Pętla `for` pobiera kolejne linie, które oczyszczamy ze znaków końca linii (`.replace('\n','')`, `.replace('\r','')`) i dekodujemy jako zapisane w standardzie *utf-8*. Poszczególne wartości oddzielone przecinkiem wyodrębniamy (`.split(',')`) do tupli, którą dodajemy do zdefiniowanej wcześniej listy (`dane.append()`).

Na koniec funkcja zwraca listę przekształconą na tuplę (a więc zagnieżdżone tuple), która po przypisaniu do jakiejś zmiennej może zostać użyta np. jako argument metody `.executemany()` (zob. przykład poniżej).

Powyższy kod można zmodyfikować, aby zwracał dane w strukturę wymaganą przez ORM Peewee, tj. listę słowników zawierających dane w formacie "klucz":"wartość" (zob. *Systemy ORM -> Operacje CRUD*).

Uwaga: Znaki w pliku wejściowym powinny być zakodowane w standardzie `utf-8`.

Przykład użycia

W skrypcie omówionym w materiale *SQL* można wykorzystać poniższy kod:

```
from dane import pobierz_dane
```

```
# ...

uczniowie = pobierz_dane('uczniowie.csv')
cur.executemany(
'INSERT INTO uczen (imie,nazwisko,klasa_id) VALUES(?,?,?)', uczniowie)
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Interpreter Sqlite

Bazy SQLite przechowywane są w pojedynczych plikach, które łatwo archiwizować, przenosić czy badać, podglądając ich zawartość. Podstawowym narzędziem jest interpreter `sqlite3` (`sqlite3.exe` w Windows).

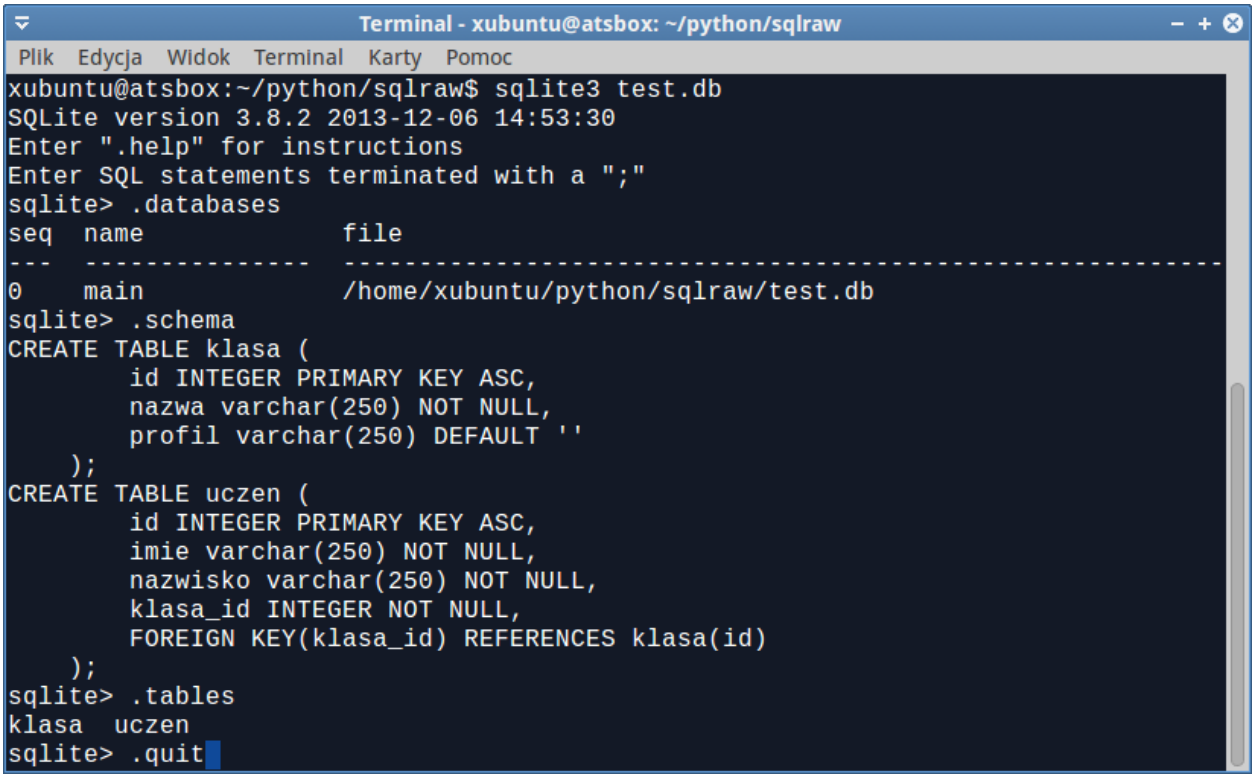
Aby otworzyć bazę zapisaną w przykładowym pliku `test.db` wydajemy w terminalu polecenie:

```
~$ sqlite3 test.db
```

Później do dyspozycji mamy polecenia:

- `.databases` – pokazuje aktualną bazę danych;
- `.schema` – pokazuje schemat bazy danych, czyli polecenia SQL tworzące tabele i relacje;
- `.table` – pokaże tabele w bazie;
- `.quit` – wychodzimy z powłoki interpretera.

Możemy również wydawać komendy SQL-a operujące na bazie, np. kwerendy: `SELECT * FROM klasa;` – polecenia te zawsze kończymy średnikiem.



Informacja: Bardziej zaawansowanym narzędziem umożliwiającym kompleksową obsługę baz SQLite za pomocą interfejsu graficznego jest program `sqlitestudio`.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Słownik baz danych

SQL strukturalny język zapytań używany do tworzenia i zarządzania bazą danych.

SQLite3 silnik bezserwerowej, nie wymagającej dodatkowej konfiguracji, transakcyjnej bazy danych implementującej standard SQL.

CRUD skrót opisujący podstawowe operacje na bazie danych z wykorzystaniem języka SQL, *Create* (tworzenie) odpowiada zapytaniom *INSERT*, *Read* (odczyt) - zapytaniom *SELECT*, *Update* (aktualizacja) - *UPDATE*, *Delete* (usuwanie) - *DELETE*.

Transakcja zbiór powiązanych logicznie operacji na bazie danych, który powinien być albo w całości zapisany, albo odrzucony ze względu na naruszenie zasad spójności (ACID).

ACID Atomicity, Consistency, Isolation, Durability – Atomowość, Spójność, Izolacja, Trwałość; zasady określające kryteria poprawnego zapisu danych w bazie. Więcej o ACID »»»

kwerenda Zapytanie do bazy danych zazwyczaj w oparciu o dodatkowe kryteria, którego celem jest wydobycie z bazy określonych danych lub ich modyfikacja.

obiekt podstawowe pojęcie programowania obiektowego, struktura zawierająca dane i metody (funkcje), za pomocą których wykonuje ię na nich operacje.

klasa definicja obiektu zawierająca opis struktury danych i jej interfejs (metody).

instancja obiekt stworzony na podstawie klasy.

konstruktor metoda wywoływana podczas tworzenia instancji (obiektu) klasy, zazwyczaj przyjmuje jako argumenty inicjalne wartości zdefiniowanych w klasie atrybutów.

ORM (ang. Object-Relational Mapping) – mapowanie obiektowo-relacyjne, oprogramowanie odwzorowujące strukturę relacyjnej bazy danych na obiekty danego języka oprogramowania.

Peewee prosty i mały system ORM, wspiera Pythona w wersji 2 i 3, obsługuje bazy SQLite3, MySQL, Posgresql.

SQLAlchemy rozbudowany zestaw narzędzi i system ORM umożliwiający wykorzystanie wszystkich możliwości SQL-a, obsługuje bazy SQLite3, MySQL, Postgresql, Oracle, MS SQL Server i inne.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Materiały

1. Moduł sqlite3 Pythona
2. Baza SQLite3
3. Język SQL
4. Peewee (ang.)
5. Tutorial Peewee (ang.)

6. SQLAlchemy ORM Tutorial (ang.)
7. Tutorial SQLAlchemy (ang.)

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.6 Aplikacje okienkowe Qt5

PyQt to zbiór bibliotek Pythona tworzonych przez Riverbank Computing umożliwiających szybkie projektowanie interfejsów aplikacji okienkowych opartych o międzyplatformowy framework Qt (zob. również oficjalną stronę Qt Company) dostępny w wersji Open Source na licencji GNU LGPL . Działa na wielu platformach i systemach operacyjnych.

Nasze scenariusze przygotowane zostały z wykorzystaniem Pythona 3 i bilioteki PyQt5.

Instalacja

W systemach Linux opartych na Debianie ((X)Ubuntu, Linux Mint itp.) lub na Arch Linuksie (Manjaro itp.):

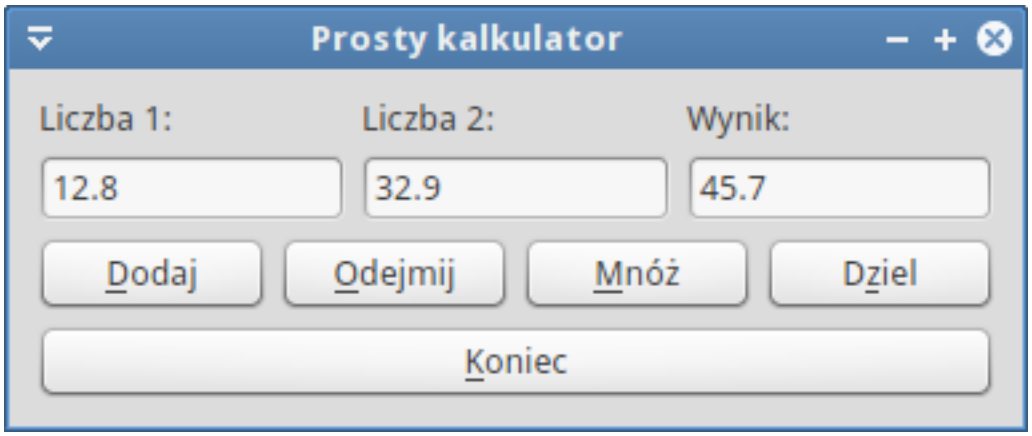
```
~$ sudo apt-get install python3-pyqt5 python3-sip
~# pacman -S python-pyqt5 python-sip
```

W środowisku Windows 64-bitowym(!) (w systemach Linux również) najnowszą wersję zainstalujemy zgodnie z instrukcjami Riverbank za pomocą menedżera pakietów:

```
~$ pip3 install PyQt5 SIP
```

Kalkulator

Prosta 1-okienkowa aplikacja ilustrująca podstawy tworzenia interfejsu graficznego i obsługi działań użytkownika za pomocą Pythona 3, PyQt5 i biblioteki Qt5. Przykład wprowadza również podstawy programowania obiektowego (ang. Object Oriented Programing).



- *Pokaż okno*
- *Widżety*

- *Interfejs*
- *Zamykanie programu*
- *Działania*
- *Materiały*

Pokaż okno

Zaczynamy od utworzenia pliku o nazwie `kalkulator.py` w dowolnym katalogu za pomocą dowolnego edytora. Wstawiamy do niego poniższy kod:

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

from PyQt5.QtWidgets import QApplication, QWidget


class Kalkulator(QWidget):
    def __init__(self, parent=None):
        super().__init__(parent)

        self.interfejs()

    def interfejs(self):

        self.resize(300, 100)
        self.setWindowTitle("Prosty kalkulator")
        self.show()


if __name__ == '__main__':
    import sys

    app = QApplication(sys.argv)
    okno = Kalkulator()
    sys.exit(app.exec_())
```

Import `from __future__ import unicode_literals` ułatwi nam obsługę napisów zawierających znaki narodowe, np. polskie "ogonki".

Podstawą naszego programu będzie moduł `PyQt5.QtWidgets`, z którego importujemy klasy `QApplication` i `QWidget` – podstawową klasę wszystkich elementów interfejsu graficznego.

Wygląd okna naszej aplikacji definiować będziemy za pomocą klasy *Kalkulator* dziedziczącej (zob. *dziedziczenie*) właściwości i metody z klasy *QWidget* (`class Kalkulator(QWidget)`). Instrukcja `super(Kalkulator, self).__init__(parent)` zwraca nam klasę rodzica i wywołuje jego *konstruktor*. Z kolei w konstruktorze naszej klasy wywołujemy metodę `interfejs()`, w której tworzyć będziemy *GUI* naszej aplikacji. Ustawiamy więc właściwości okna aplikacji i jego zachowanie:

- `self.resize(300, 100)` – szerokość i wysokość okna;
- `setWindowTitle("Prosty kalkulator"))` – tytuł okna;
- `self.show()` – wyświetlenie okna na ekranie.

Informacja: Słowa `self` używamy wtedy, kiedy odnosimy się do właściwości lub metod, również odziedziczonych,

jej instancji, czyli obiektów. Słowo to zawsze występuje jako pierwszy parametr metod obiektu definiowanych jako funkcje w definicji klasy. Zob. What is self?

Aby uruchomić program, tworzymy obiekt reprezentujący aplikację: `app = QApplication(sys.argv)`. Aplikacja może otrzymywać parametry z linii poleceń (`sys.argv`). Tworzymy również obiekt reprezentujący okno aplikacji, czyli instancję klasy *Kalkulator*: `okno = Kalkulator()`.

Na koniec uruchamiamy **główną pętlę programu** (`app.exec_()`), która rozpoczyna obsługę zdarzeń (zob. *główna pętla programu*). Zdarzenia (np. kliknięcia) generowane są przez system lub użytkownika i przekazywane do widżetów aplikacji, które mogą je obsługiwać.

Informacja: Jeżeli jakaś metoda, np. `exec_()`, ma na końcu podkreślenie, to dlatego, że jej nazwa pokrywa się z zarezerwowanym słowem kluczowym Pythona. Podkreślenie służy ich rozróżnieniu.

Poprawne zakończenie aplikacji zapewniające zwrócenie informacji o jej stanie do systemu zapewnia metoda `sys.exit()`.

Przetestujmy kod. Program uruchamiamy poleceniem wydanym w terminalu w katalogu ze skryptem:

```
~$ python3 kalkulator.py
```

Widżety

Puste okno być może nie robi wrażenia, zobaczymy więc, jak tworzyć widżety (zob. *widżet*). Najprostszym przykładem będą etykiety.

Dodajemy wymagane importy i rozbudowujemy metodę `interfejs()`:

```
5  from PyQt5.QtGui import QIcon
6  from PyQt5.QtWidgets import QLabel, QGridLayout
```

```
16     def interfejs(self):
17
18         # etykiety
19         etykieta1 = QLabel("Liczba 1:", self)
20         etykieta2 = QLabel("Liczba 2:", self)
21         etykieta3 = QLabel("Wynik:", self)
22
23         # przypisanie widgetów do układu tabelarycznego
24         ukladT = QGridLayout()
25         ukladT.addWidget(etykieta1, 0, 0)
26         ukladT.addWidget(etykieta2, 0, 1)
```

```
27          ukladT.addWidget(etykieta3, 0, 2)
28
29          # przypisanie utworzonego układu do okna
30          self.setLayout(ukladT)
31
32          self.setGeometry(20, 20, 300, 100)
33          self.setWindowIcon(QIcon('kalkulator.png'))
34          self.setWindowTitle("Prosty kalkulator")
35          self.show()
```

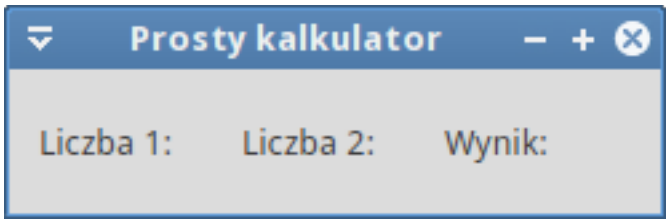
Dodawanie etykiet zaczynamy od utworzenia obiektów na podstawie odpowiedniej klasy, w tym wypadku QtLabel. Do jej konstruktora przekazujemy tekst, który ma się wyświetlać na etykiecie, np.: `etykieta1 = QLabel(Liczba 1:", self)`. Opcjonalny drugi argument wskazuje obiekt rodzica danej kontrolki.

Później tworzymy pomocniczy obiekt służący do rozmieszczenia etykiet w układzie tabelarycznym: `ukladT = QGridLayout()`. Kolejne etykiety dodajemy do niego za pomocą metody `addWidget()`. Przyjmuje ona nazwę obiektu oraz numer wiersza i kolumny definiujących komórkę, w której znaleźć się ma obiekt. Zdefiniowany układ (ang. *layout*) musimy powiązać z oknem naszej aplikacji: `self.setLayout(ukladT)`.

Na koniec używamy metody `setGeometry()` do określenia położenia okna aplikacji (początek układu jest w lewym górnym rogu ekranu) i jego rozmiaru (szerokość, wysokość). Dodajemy również ikonę pokazywaną w pasku tytułowym lub w miniaturze na pasku zadań: `self.setWindowIcon(QIcon('kalkulator.png'))`.

Informacja: Plik graficzny z ikoną musimy `pobrać` i umieścić w katalogu z aplikacją, czyli ze skryptem `kalkulator.py`.

Przetestuj wprowadzone zmiany.



Interfejs

Dodamy teraz pozostałe widżety tworzące graficzny interfejs naszej aplikacji. Jak zwykle, zaczynamy od zaimportowania potrzebnych klas:

```
7   from PyQt5.QtWidgets import QLineEdit, QPushButton, QHBoxLayout
```

Następnie przed instrukcją `self.setLayout(ukladT)` wstawiamy następujący kod:

```
30          # 1-liniowe pola edycyjne
31          self.liczba1Edt = QLineEdit()
32          self.liczba2Edt = QLineEdit()
33          self.wynikEdt = QLineEdit()
34
35          self.wynikEdt.readonly = True
36          self.wynikEdt.setToolTip('Wpisz <b>liczby</b> i wybierz działanie...')
37
38          ukladT.addWidget(self.liczba1Edt, 1, 0)
```

```
39          ukladT.addWidget(self.liczba2Edt, 1, 1)
40          ukladT.addWidget(self.wynikEdt, 1, 2)
41
42          # przyciski
43          dodajBtn = QPushButton("&Dodaj", self)
44          odejmijBtn = QPushButton("&Odejmij", self)
45          dzielBtn = QPushButton("&Mnóż", self)
46          mnozBtn = QPushButton("D&ziel", self)
47          koniecBtn = QPushButton("&Koniec", self)
48          koniecBtn.resize(koniecBtn.sizeHint())
49
50          ukladH = QHBoxLayout()
51          ukladH.addWidget(dodajBtn)
52          ukladH.addWidget(odejmijBtn)
53          ukladH.addWidget(dzielBtn)
54          ukladH.addWidget(mnozBtn)
55
56          ukladT.addLayout(ukladH, 2, 0, 1, 3)
57          ukladT.addWidget(koniecBtn, 3, 0, 1, 3)
```

Jak widać, dodawanie widżetów polega zazwyczaj na:

- **utworzeniu obiektu** na podstawie klasy opisującej potrzebny element interfejsu, np. QLineEdit – 1-liniowe pole edycyjne, lub QPushButton – przycisk;
- **ustawieniu właściwości** obiektu, np. `self.wynikEdt.readonly = True` umożliwia tylko odczyt tekstu pola, `self.wynikEdt.setToolTip('Wpisz <b>liczby</b> i wybierz działanie...')` – ustawia podpowiedź, a `koniecBtn.resize(koniecBtn.sizeHint())` – sugerowany rozmiar obiektu;
- **przypisaniu obiektu do układu** – w powyższym przypadku wszystkie przyciski działań dodano do układu horyzontalnego QHBoxLayout, ponieważ przycisków jest 4, a dopiero jego instancję do układu tabelarycznego: `ukladT.addLayout(ukladH, 2, 0, 1, 3)`. Liczby w tym przykładzie oznaczają odpowiednio wiersz i kolumnę, tj. komórkę, do której wstawiamy obiekt, a następnie ilość wierszy i kolumn, które chcemy wykorzystać.

Informacja: Jeżeli chcemy mieć dostęp do właściwości obiektów interfejsu w zasięgu całej klasy, czyli w innych funkcjach, obiekty musimy definiować jako składowe klasy, a więc poprzedzone słowem `self`, np.: `self.liczba1Edt = QLineEdit()`.

W powyższym kodzie, np. `dodajBtn = QPushButton("&Dodaj", self)`, widać również, że tworząc obiekty można określać ich rodzica (ang. *parent*), tzn. widżet nadrzędny, w tym wypadku `self`, czyli okno główne (ang. *toplevel window*). Bywa to przydatne zwłaszcza przy bardziej złożonych interfejsach.

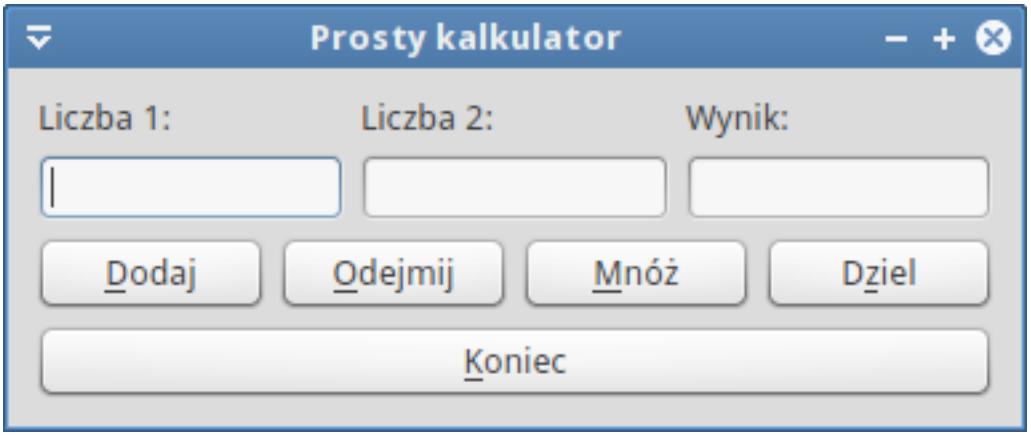
Znak `&` przed jakąś literą w opisie przycisków tworzy z kolei skrót klawiaturowy dostępny po naciśnięciu `ALT + litera`.

Po uruchomieniu programu powinniśmy zobaczyć okno podobne do poniższego:

Zamykanie programu

Mamy okienko z polami edycyjnymi i przyciskami, ale kontrolki te na nic nie reagują. Nauczymy się więc obsługiwać poszczególne zdarzenia. Zacznijmy od zamykania aplikacji.

Na początku zaimportujmy klasę QMessageBox pozwalającą tworzyć komunikaty oraz przestrzeń nazw Qt zawierającą różne stałe:




```
8  from PyQt5.QtWidgets import QMessageBox
9  from PyQt5.QtCore import Qt
```

Dalej po instrukcji `self.setLayout(ukladT)` w metodzie `interfejs()` dopisujemy:

```
64         koniecBtn.clicked.connect(self.koniec)
```

– instrukcja ta wiąże kliknięcie przycisku "Koniec" z wywołaniem metody `koniec()`, którą musimy dopisać na końcu klasy `Kalkulator()`:

```
71     def koniec(self):
72         self.close()
```

Funkcja `koniec()`, obsługująca wydarzenie (ang. *event*) kliknięcia przycisku, wywołuje po prostu metodę `close()` okna głównego.

Informacja: Omówiony fragment kodu ilustruje mechanizm zwany *sygnały i sloty* (ang. *signals & slots*). Zapewnia on komunikację między obiektami. Sygnał powstaje w momencie wystąpienia jakiegoś wydarzenia, np. kliknięcia. Slot może z kolei być wbudowaną w Qt funkcją lub Pythonowym wywołaniem (ang. *callable*), np. klasą lub metodą.

Zamknięcie okna również jest rodzajem wydarzenia (QCloseEvent), które można przechwycić. Np. po to, aby zapobiec utracie niezapisanych danych. Do klasy `Kalkulator()` dopiszmy następujący kod:

```
74     def closeEvent(self, event):
75
76         odp = QMessageBox.question(
77             self, 'Komunikat',
78             "Czy na pewno koniec?",
79             QMessageBox.Yes | QMessageBox.No, QMessageBox.No)
80
81         if odp == QMessageBox.Yes:
82             event.accept()
83         else:
84             event.ignore()
```

W nadpisanej metodzie closeEvent() wyświetlamy użytkownikowi prośbę o potwierdzenie zamknięcia za pomocą metody `question()` (ang. pytanie) klasy QMessageBox. Do konstruktora metody przekazujemy:

- obiekt rodzica – `self` oznacza okno główne;

- tytuł kona dialogowego;
- komunikat dla użytkownika, np. pytanie;
- kombinację standardowych przycisków, np. `QMessageBox.Yes | QMessageBox.No`;
- przycisk domyślny – `QMessageBox.No`.

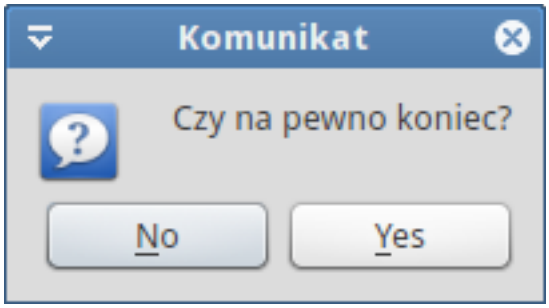
Udzielona odpowiedź `odp`, np. kliknięcie przycisku "Tak", decyduje o zezwoleniu na obsłużenie wydarzenia `event.accept()` lub odrzuceniu go `event.ignore()`.

Może wygodnie byłoby zamykać aplikację naciśnięciem klawisza `ESC`? Dopiszmy jeszcze jedną funkcję:

```
86      def keyPressEvent(self, e):
87          if e.key() == Qt.Key_Escape:
88              self.close()
```

Podobnie jak w przypadku `closeEvent()` tworzymy własną wersję funkcji keyPressEvent obsługującej naciśnięcia klawiszy QKeyEvent. Sprawdzamy naciśnięty klawisz `if e.key() == Qt.Key_Escape:` i zamykamy okno.

Przetestuj działanie aplikacji.



Działania

Kalkulator powinien liczyć. Zaczniemy od dodawania, ale na początku wszystkie sygnały wygenerowane przez przyciski działań połączymy z jednym slotem. Pod instrukcją `koniecBtn.clicked.connect(self.koniec)` dodajemy:

```
65          dodajBtn.clicked.connect(self.dzialanie)
66          odejmijBtn.clicked.connect(self.dzialanie)
67          mnozBtn.clicked.connect(self.dzialanie)
68          dzielBtn.clicked.connect(self.dzialanie)
```

Następnie zaczynamy implementację funkcji `dzialanie()`. Na końcu klasy `Kalkulator()` dodajemy:

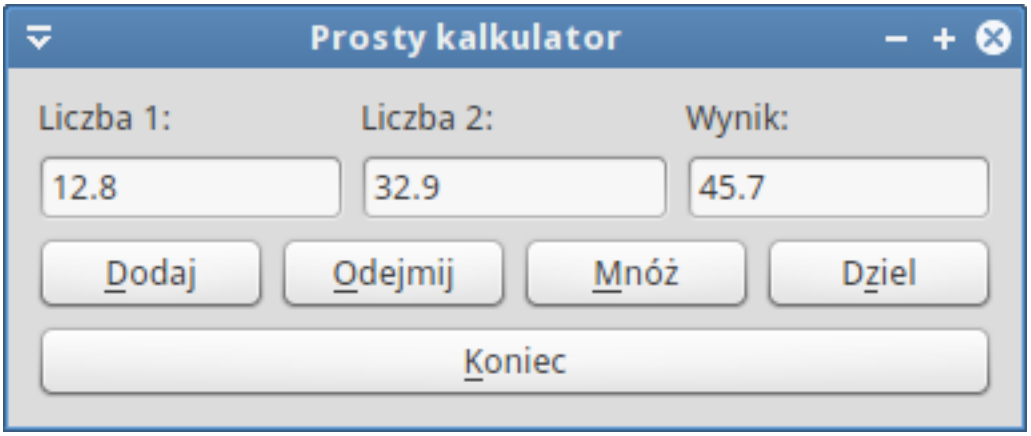
```
94      def dzialanie(self):
95
96          nadawca = self.sender()
97
98          try:
99              liczba1 = float(self.liczba1Edt.text())
100             liczba2 = float(self.liczba2Edt.text())
101             wynik = ""
102
103             if nadawca.text() == "&Dodaj":
104                 wynik = liczba1 + liczba2
105             else:
```

```
                pass

            self.wynikEdt.setText(str(wynik))

        except ValueError:
            QMessageBox.warning(self, "Błąd", "Błędne dane", QMessageBox.Ok)
```

Ponieważ jedna funkcja ma obsłużyć cztery sygnały, musimy znać źródło sygnału (ang. *source*), czyli nadawcę (ang. *sender*): `nadawca = self.sender()`. Dalej rozpoczynamy blok `try: except:` – użytkownik może wprowadzić błędne dane, tj. pusty ciąg znaków lub ciąg, którego nie da się przekształcić na liczbę zmiennoprzecinkową (`float()`). W przypadku wyjątku, wyświetlamy ostrzeżenie o błędnych danych: `QMessageBox.warning()`

Jeżeli dane są liczbami, sprawdzamy nadawcę (`if nadawca.text() == "&Dodaj":`) i jeżeli jest to przycisk dodawania, obliczamy sumę `wynik = liczba1 + liczba2`. Na koniec wyświetlamy ją po zamianie na tekst (`str()`) w polu tekstowym za pomocą metody `setText()`: `self.wynikEdt.setText(str(wynik))`.

Sprawdź działanie programu.

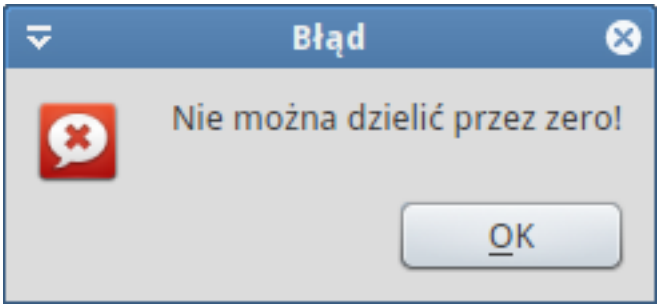


Dopiszemy obsługę pozostałych działań. Instrukcję warunkową w funkcji `dzialanie()` rozbudowujemy następująco:

```
            if nadawca.text() == "&Dodaj":
                wynik = liczba1 + liczba2
            elif nadawca.text() == "&Odejmij":
                wynik = liczba1 - liczba2
            elif nadawca.text() == "&Mnóż":
                wynik = liczba1 * liczba2
            else:  # dzielenie
                try:
                    wynik = round(liczba1 / liczba2, 9)
                except ZeroDivisionError:
                    QMessageBox.critical(
                        self, "Błąd", "Nie można dzielić przez zero!")
                    return
```

Na uwagę zasługuje tylko dzielenie. Po pierwsze określamy dokładność dzielenia do 9 miejsc po przecinku `round(liczba1 / liczba2, 9)`. Po drugie zabezpieczamy się przed dzieleniem przez zero. Znów wykorzystujemy konstrukcję `try: except:`, w której przechwytujemy wyjątek `ZeroDivisionError` i wyświetlamy odpowiednie ostrzeżenie.

Pozostaje przetestować aplikację.



Wskazówka: Jeżeli po zaimplementowaniu działań, aplikacja po uruchomieniu nie aktywuje kursora w pierwszym polu edycyjnym, należy tuż przed ustawianiem właściwości okna głównego (`self.setGeometry()`) umieścić wywołanie `self.liczba1Edt.setFocus()`, które ustawia focus na wybranym elemencie.

Materiały

1. Strona główna dokumentacji Qt5
2. Lista klas Qt5
3. PyQt5 Reference Guide
4. Przykłady PyQt5
5. Signals and slots
6. Kody klawiszy

Źródła:

- `Kalkulator Qt5 Python 3`
- `Kalkulator Qt5 Python 2`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Widżety

1-okienkowa aplikacja prezentująca zastosowanie większości podstawowych widżetów dostępnych w bibliotece Qt5 obsługiwanej za pomocą wiązań PyQt5. Przykład ilustruje również techniki programowania obiektowego (ang. *Object Oriented Programing*).

Uwaga: Wymagana wiedza:

- Znajomość Pythona w stopniu średnim.
- Znajomość podstaw projektowania interfejsu z wykorzystaniem bibliotek Qt (zob. scenariusz *Kalkulator*).

- *QPainter – podstawy rysowania*

- *Klasa* Ksztalt
- *Przyciski CheckBox*
- *Slider i przyciski RadioButton*
- *ComboBox i SpinBox*
- *Przyciski PushButton*
- *QLabel i QLineEdit*
- *Dodatki*
- *Materiały*

QPainter – podstawy rysowania

Zaczynamy od utworzenia głównego pliku o nazwie `widzety.py` w dowolnym katalogu za pomocą dowolnego edytora. Wstawiamy do niego poniższy kod:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals
from PyQt5.QtWidgets import QApplication, QWidget
from gui import Ui_Widget


class Widgety(QWidget, Ui_Widget):
    """ Główna klasa aplikacji """

    def __init__(self, parent=None):
        super(Widgety, self).__init__(parent)
        self.setupUi(self)  # tworzenie interfejsu

if __name__ == '__main__':
    import sys

    app = QApplication(sys.argv)
    okno = Widgety()
    okno.show()

    sys.exit(app.exec_())
```

Podstawową klasą opisującą naszą aplikację będzie klasa `Widgety`. Umieścimy w niej głównie logikę aplikacji, czyli powiązania sygnałów i slotów (zob.: *sygnały i sloty*) oraz implementację tych ostatnich. Klasa ta dziedziczy z zaimportowanej z pliku `gui.py` klasy `Ui_Widget` i w swoim konstruktorze (`def __init__(self, parent=None)`) wywołuję odziedziczoną metodę `self.setupUi(self)`, aby zbudować interfejs. Pozostała część pliku tworzy instancję aplikacji, instancję okna głównego, czyli klasy `Widgety`, wyświetla je i uruchamia pętlę zdarzeń.

Klasę `Ui_Widget` dla przejrzystości umieszczamy we wspomnianym pliku o nazwie `gui.py`. Tworzymy go i wstawiamy poniższy kod:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

```

```
4  from __future__ import unicode_literals
5  from PyQt5.QtGui import QPainter, QColor
6  from PyQt5.QtCore import QRect
7
8
9  class Ui_Widget(object):
10     """ Klasa definiująca GUI """
11
12     def setupUi(self, Widget):
13
14         self.ksztalt = Ksztalty.Ellipse  # kształt do narysowania
15         self.prost = QRect(1, 1, 101, 101)  # współrzędne prostokąta
16         # kolor obramowania i wypełnienia w formacie RGB
17         self.kolorO = QColor(0, 0, 0)
18         self.kolorW = QColor(200, 30, 40)
19
20         self.resize(102, 102)
21         self.setWindowTitle('Widżety')
22
23     def paintEvent(self, e):
24         qp = QPainter()
25         qp.begin(self)
26         self.rysujFigury(e, qp)
27         qp.end()
28
29     def rysujFigury(self, e, qp):
30         qp.setPen(self.kolorO)  # kolor obramowania
31         qp.setBrush(self.kolorW)  # kolor wypełnienia
32         qp.setRenderHint(QPainter.Antialiasing)  # wygładzanie kształtu
33
34         if self.ksztalt == Ksztalty.Rect:
35             qp.drawRect(self.prost)
36         elif self.ksztalt == Ksztalty.Ellipse:
37             qp.drawEllipse(self.prost)
38
39
40 class Ksztalty:
41     """ Klasa pomocnicza, symuluje typ wyliczeniowy """
42     Rect, Ellipse, Polygon, Line = range(4)
```

Klasa pomocnicza `Ksztalty` symulować będzie typ wyliczeniowy. Angielskie nazwy kształtów tworzą dane statyczne (zob. *dana statyczna*) klasy. Przypisujemy im kolejne wartości całkowite zaczynając od 0. Kształty, które będziemy rysowali, to:

- *Rect* – prostokąt, wartość 0;
- *Ellipse* – elipsa, w tym koło, wartość 1;
- *Polygon* – linia łamana zamknięta, np. trójkąt, wartość 2;
- *Line* – linia łącząca dwa punkty, wartość 3.

Określając rodzaj rysowanego kształtu, będziemy używali konstrukcji typu `Ksztalty.Ellipse`, tak jak w głównej metodzie klasy `Ui_Widget` o nazwie `setupUi()`. Definiujemy w niej zmienną wskazującą rysowany kształt (`self.ksztalt = Ksztalty.Ellipse`) oraz jego właściwości, czyli rozmiar, kolor obramowania i wypełnienia. Kolory opisujemy za pomocą klasy QColor, używając formatu RGB, np .: `self.kolorW = QColor(200, 30, 40)`.

Za rysowanie każdego widżetu, w tym wypadku głównego okna, odpowiada funkcja paintEvent(). Nadpisujemy ją, tworzymy instancję klasy QPainter umożliwiającej rysowanie różnych kształtów (`qp = QPainter()`). Między

metodami `begin()` i `end()` wywołujemy funkcję `rysujFigury()`, w której implementujemy właściwy kod rysujący.

Metody `setPen()` i `setBrush()` pozwalają ustawić kolor odpowiednio obramowania i wypełnienia. Po sprawdzeniu w instrukcji warunkowej rodzaju rysowanego kształtu wywołujemy odpowiednią metodę obiektu `QPainter`:

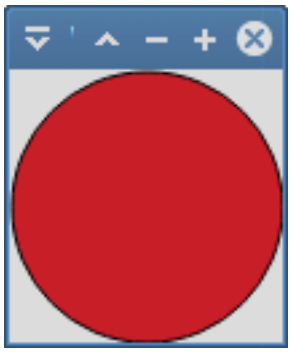
- `drawRect()` – rysuje prostokąty,
- `drawEllipse()` – rysuje elipsy.

Obydwie metody jako parametr przyjmują instancję klasy QRect: `self.prost = QRect(1, 1, 101, 101)`. Pozwala ona opisywać prostokąt do narysowania albo służący do wpisania w niego elipsy. Jako argumenty konstruktora podajemy dwie pary współrzędnych. Pierwsza określa położenie lewego górnego, druga prawego dolnego rogu prostokąta.

Uwaga: Początek układu współrzędnych, w odniesieniu do którego definiujemy w Qt pozycję okien, widżetów czy punkty opisujące kształty, znajduje się w lewym górnym rogu ekranu czy też okna.

Ćwiczenie

- Przetestuj działanie aplikacji wydając w katalogu z plikami źródłowymi polecenie w terminalu: `python widzety.py`.
- Spróbuj zmienić rodzaj rysowanej figury oraz kolory jej obramowania i wypełnienia.



Klasa *Ksztalt*

Przedstawiony wyżej sposób rysowania ma istotne ograniczenia. Przede wszystkim rysowanie odbywa się bezpośrednio w oknie głównym, co utrudnia umieszczanie innych widżetów. Po drugie nie ma wygodnego sposobu dodawania niezależnych od siebie kształtów. Aby poprawić te niedogodności, stworzymy swój widżet do rysowania, czyli klasę `Ksztalt`. Kod umieszczamy w osobnym pliku o nazwie `ksztalt.py` w katalogu z poprzednimi plikami. Jego zawartość jest następująca:

```
1  # -*- coding: utf-8 -*-
2
3  from PyQt5.QtWidgets import QWidget
4  from PyQt5.QtGui import QPainter, QColor, QPolygon
5  from PyQt5.QtCore import QPoint, QRect, QSize
6
7
8  class Ksztalty:
9      """ Klasa pomocnicza, symuluje typ wyliczeniowy """
10     Rect, Ellipse, Polygon, Line = range(4)
```

```
11
12
13  class Ksztalt(QWidget):
14      """ Klasa definiująca widget do rysowania kształtów """
15      # współrzędne prostokąta i trójkąta
16      prost = QRect(1, 1, 101, 101)
17      punkty = QPolygon([
18          QPoint(1, 101),  # punkt początkowy (x, y)
19          QPoint(51, 1),
20          QPoint(101, 101)])
21
22      def __init__(self, parent, ksztalt=Ksztalty.Rect):
23          super(Ksztalt, self).__init__(parent)
24
25          # kształt do narysowania
26          self.ksztalt = ksztalt
27          # kolor obramowania i wypełnienia w formacie RGB
28          self.kolorO = QColor(0, 0, 0)
29          self.kolorW = QColor(255, 255, 255)
30
31      def paintEvent(self, e):
32          qp = QPainter()
33          qp.begin(self)
34          self.rysujFigury(e, qp)
35          qp.end()
36
37      def rysujFigury(self, e, qp):
38          qp.setPen(self.kolorO)  # kolor obramowania
39          qp.setBrush(self.kolorW)  # kolor wypełnienia
40          qp.setRenderHint(QPainter.Antialiasing)  # wygładzanie kształtu
41
42          if self.ksztalt == Ksztalty.Rect:
43              qp.drawRect(self.prost)
44          elif self.ksztalt == Ksztalty.Ellipse:
45              qp.drawEllipse(self.prost)
46          elif self.ksztalt == Ksztalty.Polygon:
47              qp.drawPolygon(self.punkty)
48          elif self.ksztalt == Ksztalty.Line:
49              qp.drawLine(self.prost.topLeft(), self.prost.bottomRight())
50          else:  # kształt domyślny Rect
51              qp.drawRect(self.prost)
52
53      def sizeHint(self):
54          return QSize(102, 102)
55
56      def minimumSizeHint(self):
57          return QSize(102, 102)
58
59      def ustawKsztalt(self, ksztalt):
60          self.ksztalt = ksztalt
61          self.update()
62
63      def ustawKolorW(self, r=0, g=0, b=0):
64          self.kolorW = QColor(r, g, b)
65          self.update()
```

Najważniejsza metoda, tj. `paintEvent()`, w ogóle się nie zmienia. Natomiast funkcję `rysujFigury()` rozbudowujemy o możliwość rysowania kolejnych kształtów:

- `drawPolygon()` – pozwala rysować wielokąty, jako argument podajemy listę typu QPolygon punktów typu QPoint opisujących współrzędne kolejnych wierzchołków; domyślne współrzędne zdefiniowane zostały jako atrybut `punkty` naszej klasy;
- `qp.drawLine()` – pozwala narysować linię wyznaczoną przez współrzędne punktu początkowego i końcowego typu `QPoint`; nasza klasa wykorzystuje tu współrzędne lewego górnego (`self.prost.topLeft()`) i prawego dolnego (`self.prost.bottomRight()`) rogu domyślnego prostokąta: `prost = QRect(1, 1, 101, 101)`.

Konstruktor naszej klasy: `__init__(self, parent, ksztalt=Ksztalty.Rect)` – umożliwia opcjonalne przekazanie w drugim argumencie typu rysowanego kształtu. Domyślnie będzie to prostokąt. Zostanie on przypisany do atrybutu `self.ksztalt`. W konstruktorze definiujemy również domyślne kolory obramowania `self.kolorO` i wypełnienia `self.kolorW`.

Informacja: Warto zrozumieć różnicę pomiędzy **zmiennymi klasy** a **zmiennymi instancji**. Zmienne (właściwości) klasy, określane również jako dane statyczne, są wspólne dla wszystkich jej instancji. W naszej aplikacji zdefiniowaliśmy w ten sposób dostępne kształty, a także zmienne `prost` i `punkty` klasy *Ksztalt*.

Zmienne instancji natomiast są inne dla każdego obiektu. Definiujemy je w konstruktorze, używając słowa `self`. Np. każda instancja klasy *Ksztalt* może rysować inną figurę zapamiętaną w zmiennej `self.ksztalt`. Zob.: Class and Instance Variables

---

Funkcje `ustawKsztalt()` i `ustawKolorW()` – jak wskazują nazwy – pozwalają modyfikować kształt i kolor wypełnienia obiektu kształtu już po jego utworzeniu jako instancji klasy. Metoda `self.update()` wymusza ponowne narysowanie kształtu.

W metodach `sizeHint()` i `minimumSizeHint()` określamy sugerowany i minimalny rozmiar naszego kształtu. Są one niezbędne, aby układy (ang. *layouts*), w których umieścimy kształty, zarezerwowały odpowiednio dużo miejsca na ich wyświetlenie.

Ponieważ wydzieliliśmy klasę opisującą kształty, plik `gui.py` możemy uprościć:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals
from ksztalty import Ksztalty, Ksztalt
from PyQt5.QtWidgets import QHBoxLayout


class Ui_Widget(object):
    """ Klasa definiująca GUI """

    def setupUi(self, Widget):

        # widget rysujący kształty, instancja klasy Ksztalt
        self.ksztalt = Ksztalt(self, Ksztalty.Polygon)

        # układ poziomy, zawiera: self.ksztalt
        ukladH1 = QHBoxLayout()
        ukladH1.addWidget(self.ksztalt)

        self.setLayout(ukladH1)  # przypisanie układu do okna głównego
        self.setWindowTitle('Widżety')
```

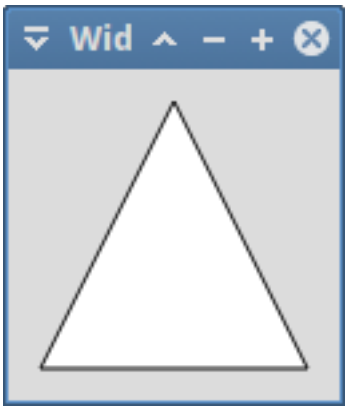
Tworzymy obiekt `self.ksztalt` jako instancję klasy `Ksztalty()` i ustawiamy kolor wypełnienia. Utworzony widżet dodajemy do poziomego układu `ukladH1.addWidget(self.ksztalt)`, a układ przypisujemy do okna

głównego `self.setLayout(ukladH1)`.

Plik `widzety.py` pozostaje bez zmian, jego zadaniem jest uruchomienie aplikacji.

Ćwiczenie

- Ponownie przetestuj działanie aplikacji, spróbuj zmienić rodzaj rysowanej figury oraz kolor jej wypełnienia.



Informacja: W kolejnych krokach będziemy umieszczać w oknie głównym widżety różnego typu. Kod tworzący te obiekty i ustawiający początkowe ich właściwości umieszczać będziemy w pliku `gui.py` w funkcji `setupUi()`. Dodając nowe widżety, musimy pamiętać o zaimportowaniu odpowiedniej klasy Qt na początku pliku. Informacje o importach będą umieszczone na początku każdej sekcji.

Kod wiążący sygnały ze slotami implementować będziemy w pliku `widzety.py`, w konstruktorze klasy `Widgety`. Sloty implementować będziemy jako funkcje tej klasy.

Przyciski CheckBox

Wykorzystując klasę *Ksztalt* utworzymy kolejny obiekt do rysowania figur. Dodamy także przyciski typu QCheckBox umożliwiające zmianę rodzaju wyświetlanej figury.

Importy w pliku `gui.py`:

```
from PyQt5.QtWidgets import QCheckBox, QButtonGroup, QVBoxLayout
```

Funkcja `setupUi()` przyjmuje następującą postać:

```
13      def setupUi(self, Widget):
14
15          # widgety rysujące kształty, instancje klasy Ksztalt
16          self.ksztalt1 = Ksztalt(self, Ksztalty.Polygon)
17          self.ksztalt2 = Ksztalt(self, Ksztalty.Ellipse)
18          self.ksztaltAktywny = self.ksztalt1
19
20          # przyciski CheckBox ###
21          uklad = QVBoxLayout()  # układ pionowy
22          self.grupaChk = QButtonGroup()
23          for i, v in enumerate(('Kwadrat', 'Koło', 'Trójkąt', 'Linia')):
24              self.chk = QCheckBox(v)
25              self.grupaChk.addButton(self.chk, i)
26              uklad.addWidget(self.chk)
```

```
        self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)
        # CheckBox do wyboru aktywnego kształtu
        self.ksztaltChk = QCheckBox('<=')
        self.ksztaltChk.setChecked(True)
        uklad.addWidget(self.ksztaltChk)

        # układ poziomy dla kształtów oraz przycisków CheckBox
        ukladH1 = QHBoxLayout()
        ukladH1.addWidget(self.ksztalt1)
        ukladH1.addLayout(uklad)
        ukladH1.addWidget(self.ksztalt2)
        # koniec CheckBox ###

        self.setLayout(ukladH1)  # przypisanie układu do okna głównego
        self.setWindowTitle('Widżety')
```

Do tworzenia przycisków wykorzystujemy pętlę `for`, która odczytuje z tupli kolejne indeksy i etykiety przycisków. Jeśli masz wątpliwości, jak to działa, przetestuj następujący kod w terminalu:

```
~$ python
>>> for i, v in enumerate(('Kwadrat', 'Koło', 'Trójkąt', 'Linia')):
...   print(i, v)
```

Odczytane etykiety przekazujemy do konstruktora: `self.chk = QCheckBox(v)`.

Przyciski wyboru kształtu działać mają na zasadzie wyłączności, w danym momencie powinien zaznaczony być tylko jeden z nich. Tworzymy więc grupę logiczną dzięki klasie QButtonGroup. Do jej instancji dodajemy przyciski, oznaczając je kolejnymi indeksami: `self.grupaChk.addButton(self.chk, i)`.

Kod `self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)` zaznacza przycisk, który odpowiada aktualnemu kształtowi. Metoda `buttons()` zwraca nam listę przycisków. Ponieważ do oznaczania kształtów używamy kolejnych liczb całkowitych, możemy użyć ich jako indeksu.

Poza pętlą tworzymy jeszcze jeden przycisk (`self.ksztaltChk = QCheckBox("<=")`), niezależny od powyższej grupy. Jego stan wskazuje aktywny kształt. Domyślnie go zaznaczamy: `self.ksztaltChk.setChecked(True)`, co oznacza, że aktywną figurą będzie pierwszy kształt. Inicjujemy również odpowiednią zmienną: `self.ksztaltAktywny = self.ksztalt1`.

Wszystkie elementy interfejsu umieszczamy w układzie poziomym o nazwie `ukladH1`. Po lewej stronie znajdzie się `ksztalt1`, w środku układ przycisków wyboru, a po prawej `ksztalt2`.

Teraz zajmiemy się obsługą sygnałów. W pliku `widzety.py` rozbudowujemy klasę `Widgety`:

```
class Widgety(QWidget, Ui_Widget):
    """ Główna klasa aplikacji """

    def __init__(self, parent=None):
        super(Widgety, self).__init__(parent)
        self.setupUi(self)  # tworzenie interfejsu

        # Sygnały i sloty ###
        # przyciski CheckBox ###
        self.grupaChk.buttonClicked[int].connect(self.ustawKsztalt)
        self.ksztaltChk.clicked.connect(self.aktywujKsztalt)

    def ustawKsztalt(self, wartosc):
        self.ksztaltAktywny.ustawKsztalt(wartosc)

    def aktywujKsztalt(self, wartosc):
```

```
25          nadawca = self.sender()
26          if wartosc:
27              self.ksztaltAktywny = self.ksztalt1
28              nadawca.setText('<=')
29          else:
30              self.ksztaltAktywny = self.ksztalt2
31              nadawca.setText('=>')
32          self.grupaChk.buttons()[self.ksztaltAktywny.ksztalt].setChecked(True)
```

Na początku kliknięcie któregokolwiek z przycisków wyboru wiążemy z funkcją `ustawKsztalt`: `self.grupaChk.buttonClicked[int].connect(self.ustawKsztalt)`. Zapis `buttonClicked[int]` oznacza, że dany sygnał może przekazać do slotu różne dane. W tym wypadku będzie to indeks kliknietego przycisku, czyli liczba całkowita. Gdybyśmy chcieli otrzymać tekst przycisku, użylibyśmy konstrukcji `buttonClicked[str]`. W slocie `ustawKsztalt()` otrzymaną wartość używamy do ustawienia rodzaju rysowanej figury za pomocą odpowiedniej metody klasy `Ksztalt`: `self.ksztaltAktywny.ustawKsztalt(wartosc)`.

Kliknięcie przycisku wskazującego aktywną figurę obsługujemy w kodzie: `self.ksztaltChk.clicked.connect(self.aktywujKsztalt)`. Tym razem funkcja `aktywujKsztalt()` dostaje wartość logiczną `True` lub `False`, która określa, czy przycisk został zaznaczony, czy nie. W zależności od tego ustawiamy jako aktywny odpowiedni obszar rysowania oraz tekst przycisku.

Informacja: Warto zapamiętać, jak uzyskać dostęp do obiektu, który wygenerował dany sygnał. W odpowiednim slocie używamy kodu `self.sender()`.

Ćwiczenie

Jak zwykle uruchom kilkakrotnie aplikację. Spróbuj zmieniać inicjalne rodzaje domyślnych kształtów i kolory wypełnienia figur.

Slider i przyciski RadioButton

Możemy już manipulować rodzajami rysowanych kształtów na obydwu obszarach rysowania. Spróbujemy teraz dodać widżety pozwalające je kolorować.

Importy w pliku `gui.py`:

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QSlider, QLCDNumber, QSplitter
from PyQt5.QtWidgets import QRadioButton, QGroupBox
```

Teraz rozbudowujemy konstruktor klasy `Ui_Widget`. Po komentarzu `# koniec CheckBox ###` wstawiamy:

```
        # Slider i LCDNumber ###
        self.suwak = QSlider(Qt.Horizontal)
        self.suwak.setMinimum(0)
        self.suwak.setMaximum(255)
        self.lcd = QLCDNumber()
        self.lcd.setSegmentStyle(QLCDNumber.Flat)
        # układ poziomy (splitter) dla slajdera i lcd
        ukladH2 = QSplitter(Qt.Horizontal, self)
        ukladH2.addWidget(self.suwak)
        ukladH2.addWidget(self.lcd)
        ukladH2.setSizes((125, 75))

        # przyciski RadioButton ###
        self.ukladR = QHBoxLayout()
        for v in ('R', 'G', 'B'):
            self.radio = QRadioButton(v)
            self.ukladR.addWidget(self.radio)
        self.ukladR.itemAt(0).widget().setChecked(True)
        # grupujemy przyciski
        self.grupaRBtn = QGroupBox('Opcje RGB')
        self.grupaRBtn.setLayout(self.ukladR)
        self.grupaRBtn.setObjectName('Radio')
        self.grupaRBtn.setCheckable(True)
        # układ poziomy dla grupy Radio
        ukladH3 = QHBoxLayout()
        ukladH3.addWidget(self.grupaRBtn)
        # koniec RadioButton ###
```

Do zmiany wartości składowych kolorów RGB wykorzystamy instancję klasy QSlider, czyli popularny suwak, w tym wypadku poziomy. Po utworzeniu obiektu, ustawiamy za pomocą metod `setMinimum()` i `setMaximum()` zakres zmienianych wartości <0-255>. Następnie tworzymy instancję klasy QLCDNumber, którą wykorzystamy do wyświetlania wartości wybranej za pomocą suwaka. Obydwa obiekty dodajemy do poziomego układu, rozdzielając je instancją typu QSplitter. Obiekt tez pozwala płynnie zmieniać rozmiar otaczających go widżetów.

Przyciski typu RadioButton posłużą nam do wskazywania kanału koloru RGB, którego wartość chcemy zmienić. Tworzymy je w pętli, wykorzystując odczytane z tupli nazwy kanałów: `self.radio = QRadioButton(v)`. Przyciski rozmieszczamy w poziomie (`self.ukladR.addWidget(self.radio)`).

Pierwszy z nich zaznaczamy: `self.ukladR.itemAt(0).widget().setChecked(True)`. Metoda `itemAt(0)` zwraca nam pierwszy element danego układu jako typ *QLayoutItem*. Kolejna metoda `widget()` przekształca go w obiekt typu *QWidget*, dzięki czemu możemy wywoływać jego metody.

Układ przycisków dodajemy do grupy typu QGroupBox: `self.grupaRBtn.setLayout(self.ukladR)`. Tego typu grupa zapewnia graficzną ramkę z przyciskiem aktywującym typu CheckBox, który domyślnie zaznaczamy: `self.grupaRBtn.setCheckable(True)`. Za pomocą metody `setObjectName()` grupie nadajemy nazwę *Radio*.

Kończąc zmiany w interfejsie, tworzymy nowy pionowy układ dla elementów głównego okna aplikacji. Przedostatnią linię `self.setLayout(ukladH1)` zastępujemy poniższym kodem:

```
        # główny układ okna, pionowy ###
        ukladOkna = QVBoxLayout()
```

```
        ukladOkna.addLayout(ukladH1)
        ukladOkna.addWidget(ukladH2)
        ukladOkna.addLayout(ukladH3)

        self.setLayout(ukladOkna)  # przypisanie układu do okna głównego
        self.setWindowTitle('Widżety')
```

Ustawienia wstępne i obsługa zdarzeń

Importy w pliku `widzety.py`:

```
from PyQt5.QtGui import QColor
```

Dalej tworzymy dwie zmienne klasy *Widgety*:

```
class Widgety(QWidget, Ui_Widget):
    """ Główna klasa aplikacji """

    kanaly = {'R'}  # zbiór kanałów
    kolorW = QColor(0, 0, 0)  # kolor RGB kształtu 1
```

Następnie uzupełniamy konstruktor (`__init__()`), a za nim dopisujemy dwie funkcje:

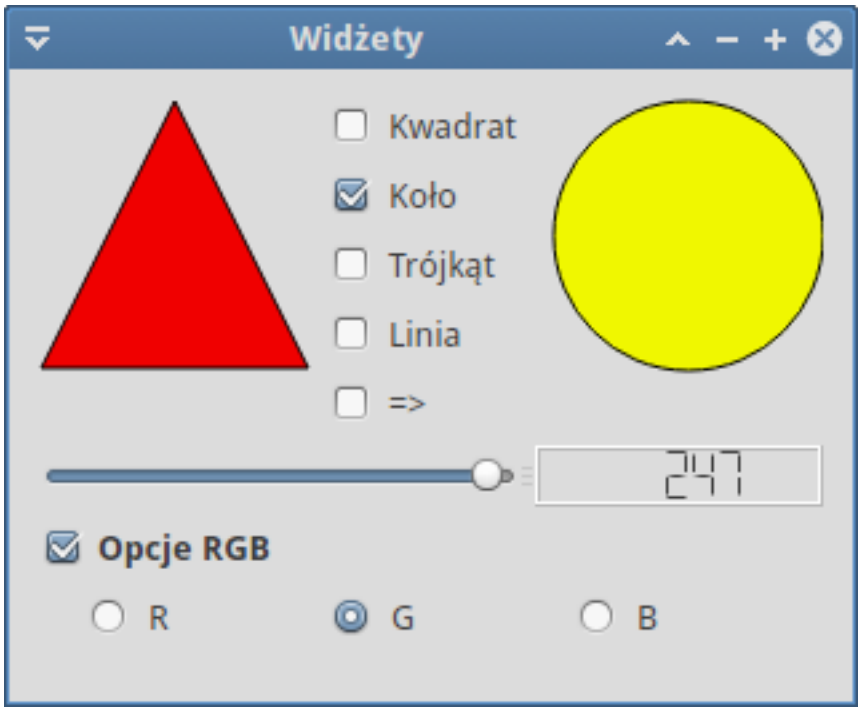
```
        # Slider + przyciski RadioButton ###
        for i in range(self.ukladR.count()):
            self.ukladR.itemAt(i).widget().toggled.connect(self.ustawKanalRBtn)
        self.suwak.valueChanged.connect(self.zmienKolor)

    def ustawKanalRBtn(self, wartosc):
        self.kanaly = set()  # resetujemy zbiór kanałów
        nadawca = self.sender()
        if wartosc:
            self.kanaly.add(nadawca.text())

    def zmienKolor(self, wartosc):
        self.lcd.display(wartosc)
        if 'R' in self.kanaly:
            self.kolorW.setRed(wartosc)
        if 'G' in self.kanaly:
            self.kolorW.setGreen(wartosc)
        if 'B' in self.kanaly:
            self.kolorW.setBlue(wartosc)

        self.ksztaltAktywny.ustawKolorW(
            self.kolorW.red(),
            self.kolorW.green(),
            self.kolorW.blue())
```

Ze zmianą stanu przycisków Radio związany jest sygnał `toggled`. W pętli `for i in range(self.ukladR.count()):` wiążemy go dla każdego przycisku układu z funkcją `ustawKanalRBtn()`. Otrzymuje ona wartość logiczną. Zadaniem funkcji jest zresetowanie zbioru kolorów i dodanie do niego litery opisującej zaznaczony przycisk: `self.kanaly.add(nadawca.text())`.

Manipulowanie suwakiem wyzwala sygnał `valueChanged`, który łączymy ze slotem `zmienKolor()`: `self.suwak.valueChanged.connect(self.zmienKolor)`. Do funkcji przekazywana jest wartość wybrana na suwaku, wyświetlamy ją w widżecie LCD: `self.lcd.display(wartosc)`. Następnie sprawdzamy aktywne kanały w zbiorze kanałów i zmieniamy odpowiadającą im wartość składową w kolorze wypełnienia, np.: `self.kolorW.setRed(wartosc)`. Na koniec przypisujemy otrzymany kolor wypełnienia aktywnemu kształtowi,

osobno podając składowe RGB.

Przetestuj działanie aplikacji.



ComboBox i SpinBox

Modyfikowane kanały koloru można wybierać z rozwijalnej listy typu QComboBox, a ich wartości ustawiać za pomocą widżetu QSpinBox.

Importy w pliku `gui.py`:

```
from PyQt5.QtWidgets import QComboBox, QSpinBox
```

Po komentarzu `# koniec RadioButton ###` uzupełniamy kod funkcji `setupUi()`:

```
72          # Lista ComboBox i SpinBox ###
73          self.listaRGB = QComboBox(self)
74          for v in ('R', 'G', 'B'):
75              self.listaRGB.addItem(v)
76          self.listaRGB.setEnabled(False)
77          # SpinBox
78          self.spinRGB = QSpinBox()
79          self.spinRGB.setMinimum(0)
80          self.spinRGB.setMaximum(255)
81          self.spinRGB.setEnabled(False)
82          # układ pionowy dla ComboBox i SpinBox
83          uklad = QVBoxLayout()
84          uklad.addWidget(self.listaRGB)
85          uklad.addWidget(self.spinRGB)
86          # do układu poziomego grupy Radio dodajemy układ ComboBox i SpinBox
87          ukladH3.insertSpacing(1, 25)
```

```
88          ukladH3.addLayout(uklad)
89          # koniec ComboBox i SpinBox ###
```

Po utworzeniu obiektu listy za pomocą pętli `for` dodajemy kolejne elementy, czyli litery poszczególnych kanałów: `self.listaRGB.addItem(v)`.

Obiekt *SpinBox* podobnie jak *Slider* wymaga ustawienia zakresu wartości <0-255>, wykorzystujemy takie same metody, jak wcześniej, tj. `setMinimum()` i `setMaximum()`.

Obydwa widżety na razie wyłączamy metodą `setEnabled(False)`. Umieszczamy jeden nad drugim, a ich układ dodajemy obok przycisków Radio, rozdzielając je odstępem 25 px: `ukladH3.insertSpacing(1, 25)`.

W pliku `widzety.py` dodajemy do konstruktora kod przechwytujący 3 sygnały i dopisujemy dwie nowe funkcje:

```
28          # Lista ComboBox i SpinBox ###
29          self.grupaRBtn.clicked.connect(self.ustawStan)
30          self.listaRGB.activated[str].connect(self.ustawKanalCBox)
31          self.spinRGB.valueChanged[int].connect(self.zmienKolor)
32
33      def ustawStan(self, wartosc):
34          if wartosc:
35              self.listaRGB.setEnabled(False)
36              self.spinRGB.setEnabled(False)
37          else:
38              self.listaRGB.setEnabled(True)
39              self.spinRGB.setEnabled(True)
40              self.kanaly = set()
41              self.kanaly.add(self.listaRGB.currentText())
42
43      def ustawKanalCBox(self, wartosc):
44          self.kanaly = set()  # resetujemy zbiór kanałów
45          self.kanaly.add(wartosc)
```

Po uruchomieniu aplikacji aktywna jest tylko grupa przycisków Radio. Kliknięcie tej grupy przechwytujemy: `self.grupaRBtn.clicked.connect(self.ustawStan)`. Funkcja `ustawStan()` w zależności od zaznaczenia grupy lub jego braku wyłącza (`setEnabled(False)`) lub włącza (`setEnabled(True)`) widżety *ComboBox* i *SpinBox*. W tym drugim przypadku resetujemy zbiór kanałów i dodajemy do niego tylko kanał wybrany na liście: `self.kanaly.add(self.listaRGB.currentText())`.

Drugie wydarzenie, które obsłużymy, to wybranie nowego kanału z listy. Emitowany jest wtedy sygnał `activated[str]`, który zawiera tekst wybranego elementu. W slocie `ustawKanalCBox()` tekst ten, czyli nazwę składowej koloru, dodajemy do zbioru kanałów.

Zmiana wartości w kontrolce SpinBox, czyli sygnał `valueChanged[int]`, przekierowujemy do funkcji `zmienKolor()`, która obsługuje również zmiany wartości na suwaku.

Uruchom aplikację i sprawdź jej działanie.

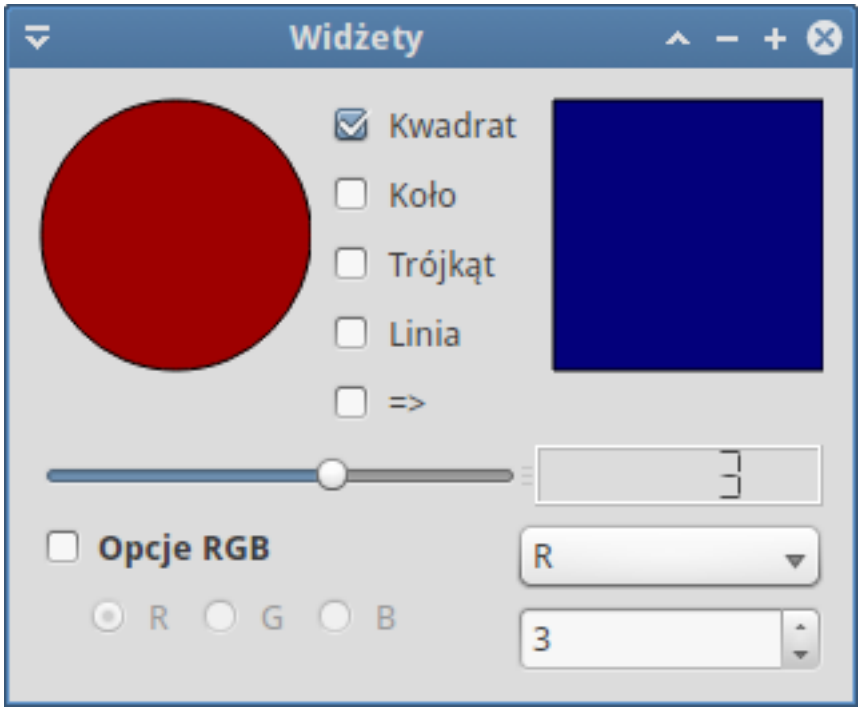
Przyciski PushButton

Do tej pory można było zmieniać kolor każdego kanału składowego osobno. Dodamy teraz grupę przycisków typu QPushButton, które zachowywać się będą jak grupa przycisków wielokrotnego wyboru.

Importy w pliku `gui.py`:

```
from PyQt5.QtWidgets import QPushButton
```

Następnie po komentarzu `# koniec ComboBox i SpinBox ###` dopisujemy kod w funkcji `setupUi()`:




```
92          # przyciski PushButton ###
93          uklad = QHBoxLayout()
94          self.grupaP = QButtonGroup()
95          self.grupaP.setExclusive(False)
96          for v in ('R', 'G', 'B'):
97              self.btn = QPushButton(v)
98              self.btn.setCheckable(True)
99              self.grupaP.addButton(self.btn)
100             uklad.addWidget(self.btn)
101         # grupujemy przyciski
102         self.grupaPBtn = QGroupBox('Przyciski RGB')
103         self.grupaPBtn.setLayout(uklad)
104         self.grupaPBtn.setObjectName('Push')
105         self.grupaPBtn.setCheckable(True)
106         self.grupaPBtn.setChecked(False)
107         # koniec PushButton ###
```

Przyciski, jak poprzednio, tworzymy w pętli, podając w konstruktorze litery składowych koloru RGB: `self.btn = QPushButton(v)`. Każdy przycisk przekształcamy na stanowy (może być trwale wciśnięty) za pomocą metody `setCheckable()`. Kolejne obiekty dodajemy do grupy logicznej typu QButtonGroup: `self.grupaP.addButton(self.btn)`; oraz do układu poziomego. Układ przycisków dodajemy do ramki typu QGropBox z przyciskiem CheckBox: `self.grupaPBtn.setCheckable(True)`. Na początku ramkę wyłączamy: `self.grupaPBtn.setChecked(False)`.

Uwaga: na koniec musimy dodać grupę przycisków do głównego układu okna: `ukladOkna.addWidget(self.grupaPBtn)`. Inaczej nie zobaczymy jej w oknie aplikacji!

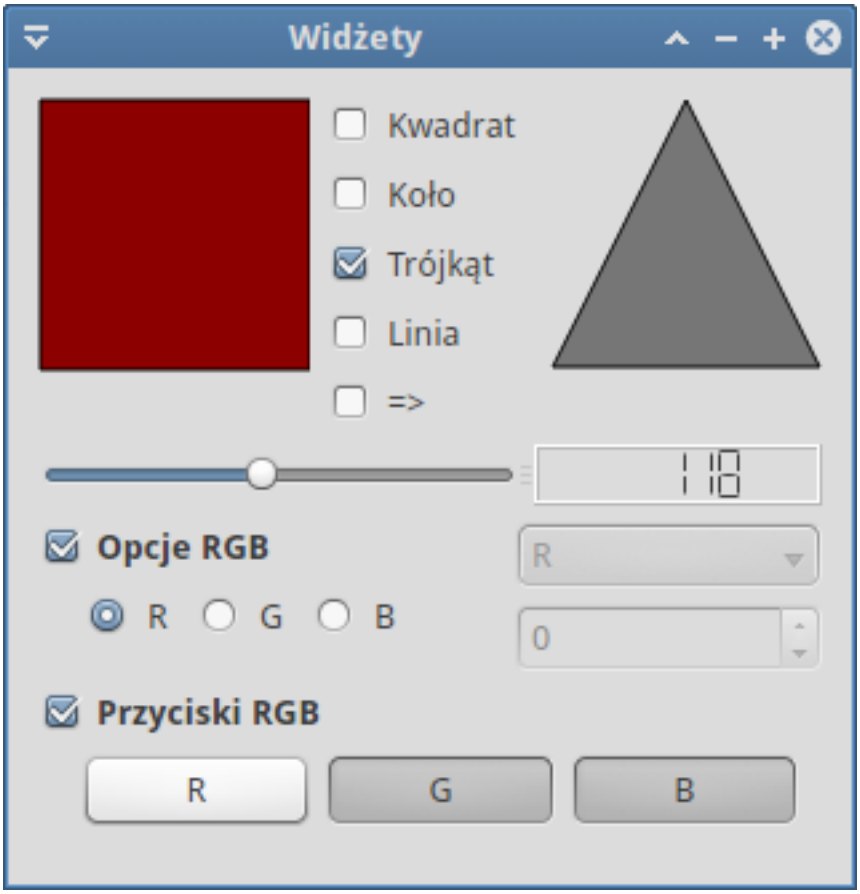
W pliku `widzety.py` jak zwykle dopisujemy obsługę sygnałów w konstruktorze i jedną nową funkcję:

```
32          # przyciski PushButton ###
33          for btn in self.grupaP.buttons():
34              btn.clicked[bool].connect(self.ustawKanalPBtn)
```

```
35          self.grupaPBtn.clicked.connect(self.ustawStan)
36
37      def ustawKanalPBtn(self, wartosc):
38          nadawca = self.sender()
39          if wartosc:
40              self.kanaly.add(nadawca.text())
41          elif wartosc in self.kanaly:
42              self.kanaly.remove(nadawca.text())
```

Pętla `for btn in self.grupaP.buttons():` odczytuje kolejne przyciski z grupy `grupaP`, i kliknięcie każdego wiąże z nową funkcją: `btn.clicked[bool].connect(self.ustawKanalPBtn)`. Zadaniem funkcji jest dodawanie kanału do zbioru, jeżeli przycisk został wciśnięty, i usuwanie ich ze zbioru w przeciwnym razie. Inaczej niż w poprzednich funkcjach, obsługujących przyciski *Radio* i listę *ComboBox*, nie resetujemy tu zbioru kanałów.

Przetestuj zmodyfikowaną aplikację.



QLabel i QLineEdit

Dodamy do aplikacji zestaw widżetów wyświetlających aktywne kanały jako etykiety typu QLabel oraz wartości składowych koloru jako 1-liniowe pola edycyjne typu QLineEdit.

Importy w pliku `gui.py`:

```
from PyQt5.QtWidgets import QLabel, QLineEdit
```

Następnie po komentarzu `# koniec PushButton ###` uzupełnij funkcję `setupUi()`:

```
110         # etykiety QLabel i pola QLineEdit ###
111         ukladH4 = QHBoxLayout()
112         self.labelR = QLabel('R')
113         self.labelG = QLabel('G')
114         self.labelB = QLabel('B')
115         self.kolorR = QLineEdit('0')
116         self.kolorG = QLineEdit('0')
117         self.kolorB = QLineEdit('0')
118         for v in ('R', 'G', 'B'):
119             label = getattr(self, 'label' + v)
120             kolor = getattr(self, 'kolor' + v)
121             ukladH4.addWidget(label)
122             ukladH4.addWidget(kolor)
123             kolor.setMaxLength(3)
124         # koniec QLabel i QLineEdit ###
```

Zaczynamy od utworzenia trzech etykiet i trzech pól edycyjnych dla każdego kanału. W pętli wykorzystujemy funkcję Pythona getattr(obiekt, nazwa), która potrafi zwrócić podany jako `nazwa` atrybut `obiektu`. W tym wypadku kolejne etykiety i pola edycyjne, które umieszczamy obok siebie w poziomie. Przy okazji ograniczamy długość wpisywanego w pola edycyjne tekstu do 3 znaków: `kolor.setMaxLength(3)`.

Uwaga: Pamiętajmy, że aby zobaczyć utworzone obiekty w oknie aplikacji, musimy dołączyć je do głównego układu okna: `ukladOkna.addLayout(ukladH4)`.

W pliku `widzety.py` rozszerzamy konstruktor klasy `Widgety` i dodajemy funkcję informacyjną:

```
36         # etykiety QLabel i pola QEditLine ###
37         for v in ('R', 'G', 'B'):
38             kolor = getattr(self, 'kolor' + v)
39             kolor.textEdited.connect(self.zmienKolor)
40
41     def info(self):
42         fontB = "QWidget { font-weight: bold }"
43         fontN = "QWidget { font-weight: normal }"
44
45         for v in ('R', 'G', 'B'):
46             label = getattr(self, 'label' + v)
47             kolor = getattr(self, 'kolor' + v)
48             if v in self.kanaly:
49                 label.setStyleSheet(fontB)
50                 kolor.setEnabled(True)
51             else:
52                 label.setStyleSheet(fontN)
53                 kolor.setEnabled(False)
54
55         self.kolorR.setText(str(self.kolorW.red()))
56         self.kolorG.setText(str(self.kolorW.green()))
57         self.kolorB.setText(str(self.kolorW.blue()))
```

W pętli, podobnej jak w pliku interfejsu, sygnał zmiany tekstu pola typu *QLineEdit* wiążemy z dodaną wcześniej funkcją `zmienKolor()`. Będziemy mogli wpisywać w tych polach nowe wartości składowych koloru. **Ale uwaga**: do tej pory funkcja `zmienKolor()` otrzymywała wartości typu całkowitego z suwaka *QSlider* lub pola *QSpinBox*. Pole edycyjne zwraca natomiast tekst, który trzeba rzutować na typ całkowity. Dodaj więc na początku funkcji instrukcję: `wartosc = int(wartosc)`.

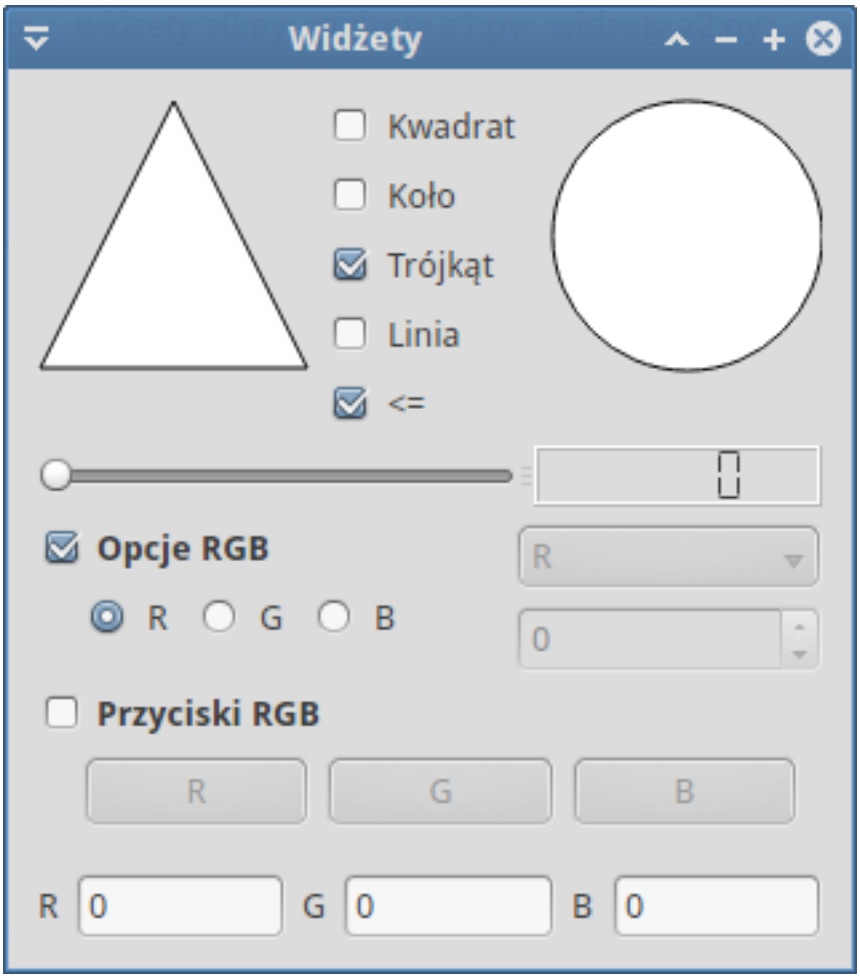
Druga nowa rzecz to funkcja informacyjna `info()`. Jej zadanie polega na wyróżnieniu aktywnych kanałów poprzez pogrubienie czcionki etykiet i uaktywnieniu odpowiednich pól edycyjnych. Jeżeli kanał jest nieaktywny, ustawiamy normalną czcionkę etykiety i wyłączamy pole edycji. Wszystko dzieje się w pętli wykorzystującej omawiane już funkcje `getattr()` oraz `setEnabled()`.

Na uwagę zasługują operacje na czcionce. Zmieniamy ją dzięki stylom CSS zdefiniowanym na początku funkcji pod nazwą `fontB` i `fontN`. Później przypisujemy je etykietom za pomocą metody `setStyleSheet()`.

Na końcu omawianej funkcji do każdego pola edycyjnego wstawiamy aktualną wartość odpowiedniej składowej koloru przekształconą na tekst, np. `self.kolorR.setText(str(self.kolorW.red()))`.

Wywołanie tej funkcji w postaci `self.info()` powinniśmy dopisać przynajmniej do funkcji `zmienKolor()`.

Wprowadź omówione zmiany i przetestuj działanie aplikacji.



Dodatki

Nasza aplikacja działa, ale można dopracować w niej kilka szczegółów. Poniżej zaproponujemy kilka zmian, które potraktować należy jako zachętę do samodzielnych ćwiczeń i przeróbek.

1. Po pierwsze pola edycyjne *QLineEdit* dla składowych zielonej i niebieskiej powinny być na początku nieaktywne. Dodaj odpowiedni kod do pliku `gui.py`, wykorzystaj metodę `setEnabled()`.

2. Zaznaczenie jednej z grup przycisków powinno wyłączać drugą grupę. Jeżeli aktywujemy grupę *Push* dobrze byłoby zaznaczyć przycisk odpowiadający ostatniemu aktywnemu kanałowi. W tym celu trzeba uzupełnić funkcję `ustawStan()`. Spróbuj użyć poniższego kodu:

```
nadawca = self.sender()
if nadawca.objectName() == 'Radio':
    self.grupaPBtn.setChecked(False)
if nadawca.objectName() == 'Push':
    self.grupaRBtn.setChecked(False)
    for btn in self.grupaP.buttons():
        btn.setChecked(False)
        if btn.text() in self.kanaly:
            btn.setChecked(True)
```

Ponieważ w(y)łączanie ramek z przyciskami obsługujemy w jednym slocie, musimy wiedzieć, która ramka wysłała sygnał. Metoda `self.sender()` zwraca nam nadawcę, a za pomocą metody `objectName()` możemy odczytać jego nazwę.

Jeżeli ramką źródłową jest ta z przyciskami PushButton, w pętli `for btn in self.grupaP.buttons():` na początku odznaczamy każdy przycisk po to, żeby zaznaczyć go, o ile wskazywany przez niego kanał jest w zbiorze.

3. Stan pól edycyjnych powinien odpowiadać stanowi przycisków PushButton, wciśnięty przycisk to aktywne pole i odwrotnie. Dopisz odpowiedni kod do slotu `ustawKanalPBtn()`. Wykorzystaj funkcję `getattr`, aby uzyskać dostęp do właściwego pola edycyjnego.

4. Funkcja `zmienKolor()` nie jest zabezpieczona przed błędnymi danymi wprowadzanymi do pól edycyjnych. Prześledź komunikaty w konsoli pojawiające się po wpisaniu wartości ujemnych, albo tekstu. Sytuacje takie można obsłużyć dopisując na początku funkcji np. taki kod:

```
try:
    wartosc = int(wartosc)
except ValueError:
    wartosc = 0
if wartosc > 255:
    wartosc = 255
```

5. Jak zostało pokazane w aplikacji, nic nie stoi na przeszkodzie, żeby podobne sygnały obsługiwane były przez jeden slot. Niekiedy jednak wymaga to pewnych dodatkowych zabiegów. Można by na przykład spróbować połączyć sloty `ustawKanalRBtn()` i `ustawKanalCBox()` w jeden `ustawKanal()`, który mógłby zostać zaimplementowany tak:

```
def ustawKanal(self, wartosc):
    self.kanaly = set()  # resetujemy zbiór kanałów
    try:  # ComboBox
        if len(wartosc) == 1:
            self.kanaly.add(wartosc)
    except TypeError:  # RadioButton
        nadawca = self.sender()
        if wartosc:
            self.kanaly.add(nadawca.text())
```

6. Dodaj dwa osobne przyciski, które umożliwią kopiowanie koloru i kształtu z jednej figury na drugą.

Materiały

1. Qt Widgets
2. Widgets Tutorial

3. Layout Management

Źródła:

- `Widżety Qt5`

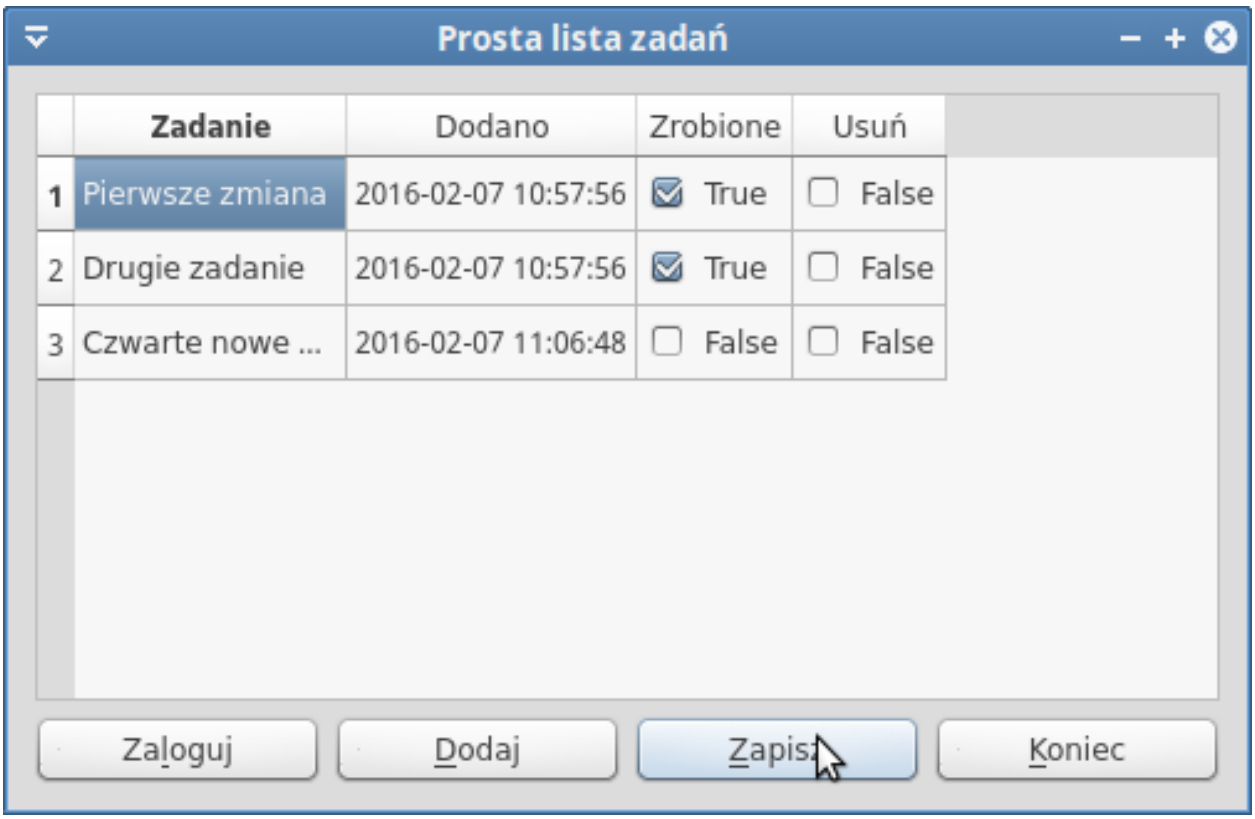
  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

ToDoPw

Realizacja prostej listy zadań do zrobienia jako aplikacji okienkowej, z wykorzystaniem biblioteki Qt5 i wiązań Pythona PyQt5. Aplikacja umożliwia dodawanie, usuwanie, edycję i oznaczanie jako wykonane zadań, zapisywanych w bazie SQLite obsługiwanej za pomocą systemu ORM Peewee. Biblioteka *Peewee* musi być zainstalowana w systemie.

Przykład wykorzystuje programowanie obiektowe (ang. *Object Oriented Programing*) i ilustruje technikę programowania model/widok (ang. *Model/View Programming*).



Uwaga: Wymagana wiedza:

- Znajomość Pythona w stopniu średnim.
- Znajomość podstaw projektowania interfejsu z wykorzystaniem biblioteki Qt (zob. scenariusze *Kalkulator* i *Widżety*).
- Znajomość podstaw systemów ORM (zob. scenariusz *Systemy ORM*).

Interfejs

Budowanie aplikacji zaczniemy od przygotowania podstawowego interfejsu. Na początku utwórzmy katalog aplikacji, w którym zapisywać będziemy wszystkie pliki:

```
~$ mkdir todopw
```

Następnie w dowolnym edytorze tworzymy plik o nazwie `gui.py`, który posłuży do definiowania składników interfejsu. Wklejamy do niego poniższy kod:

```
# -*- coding: utf-8 -*-

from PyQt5.QtWidgets import QTableView, QPushButton
from PyQt5.QtWidgets import QHBoxLayout, QVBoxLayout


class Ui_Widget(object):

    def setupUi(self, Widget):
        Widget.setObjectName("Widget")

        # tabelaryczny widok danych
        self.widok = QTableView()

        # przyciski Push ###
        self.logujBtn = QPushButton("Za&loguj")
        self.koniecBtn = QPushButton("&Koniec")

        # układ przycisków Push ###
        uklad = QHBoxLayout()
        uklad.addWidget(self.logujBtn)
        uklad.addWidget(self.koniecBtn)

        # główny układ okna ###
        ukladV = QVBoxLayout(self)
        ukladV.addWidget(self.widok)
        ukladV.addLayout(uklad)

        # właściwości widżetu ###
        self.setWindowTitle("Prosta lista zadań")
        self.resize(500, 300)
```

Centralnym elementem aplikacji będzie komponent QTableView, który potrafi wyświetlać dane w formie tabeli na podstawie zdefiniowanego modelu. Użyjemy go po to, aby oddzielić dane od sposobu ich prezentacji (zob. Model/View programming). Taka architektura przydaje się zwłaszcza wtedy, kiedy aplikacja okienkowa stanowi przede wszystkim interfejs służący prezentacji i ewentualnie edycji danych, przechowywanych niezależnie, np. w bazie.

Pod kontrolką widoku umieszczamy obok siebie dwa przyciski, za pomocą których będzie się można zalogować do aplikacji i ją zakończyć.

Główne okno i obiekt aplikacji utworzymy w pliku `todopw.py`, który musi zostać zapisany w tym samym katalogu co plik opisujący interfejs. Jego zawartość na początku będzie następująca:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

from __future__ import unicode_literals
from PyQt5.QtWidgets import QApplication, QWidget
from PyQt5.QtWidgets import QMessageBox, QInputDialog
from gui_z0 import Ui_Widget


class Zadania(QWidget, Ui_Widget):

    def __init__(self, parent=None):
        super(Zadania, self).__init__(parent)
        self.setupUi(self)

        self.logujBtn.clicked.connect(self.loguj)
        self.koniecBtn.clicked.connect(self.koniec)

    def loguj(self):
        login, ok = QInputDialog.getText(self, 'Logowanie', 'Podaj login:')
        if ok:
            haslo, ok = QInputDialog.getText(self, 'Logowanie', 'Podaj haslo:')
            if ok:
                if not login or not haslo:
                    QMessageBox.warning(
                        self, 'Błąd', 'Pusty login lub hasło!', QMessageBox.Ok)
                    return
                QMessageBox.information(
                    self, 'Dane logowania',
                    'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)

    def koniec(self):
        self.close()

if __name__ == '__main__':
    import sys

    app = QApplication(sys.argv)
    okno = Zadania()
    okno.show()
    okno.move(350, 200)
    sys.exit(app.exec_())
```

Podobnie jak w poprzednich scenariuszach klasa `Zadania` dziedziczy z klasy `Ui_Widget`, aby utworzyć interfejs aplikacji. W konstruktorze skupiamy się na działaniu aplikacji, czyli wiążemy kliknięcia przycisków z odpowiednimi slotami.

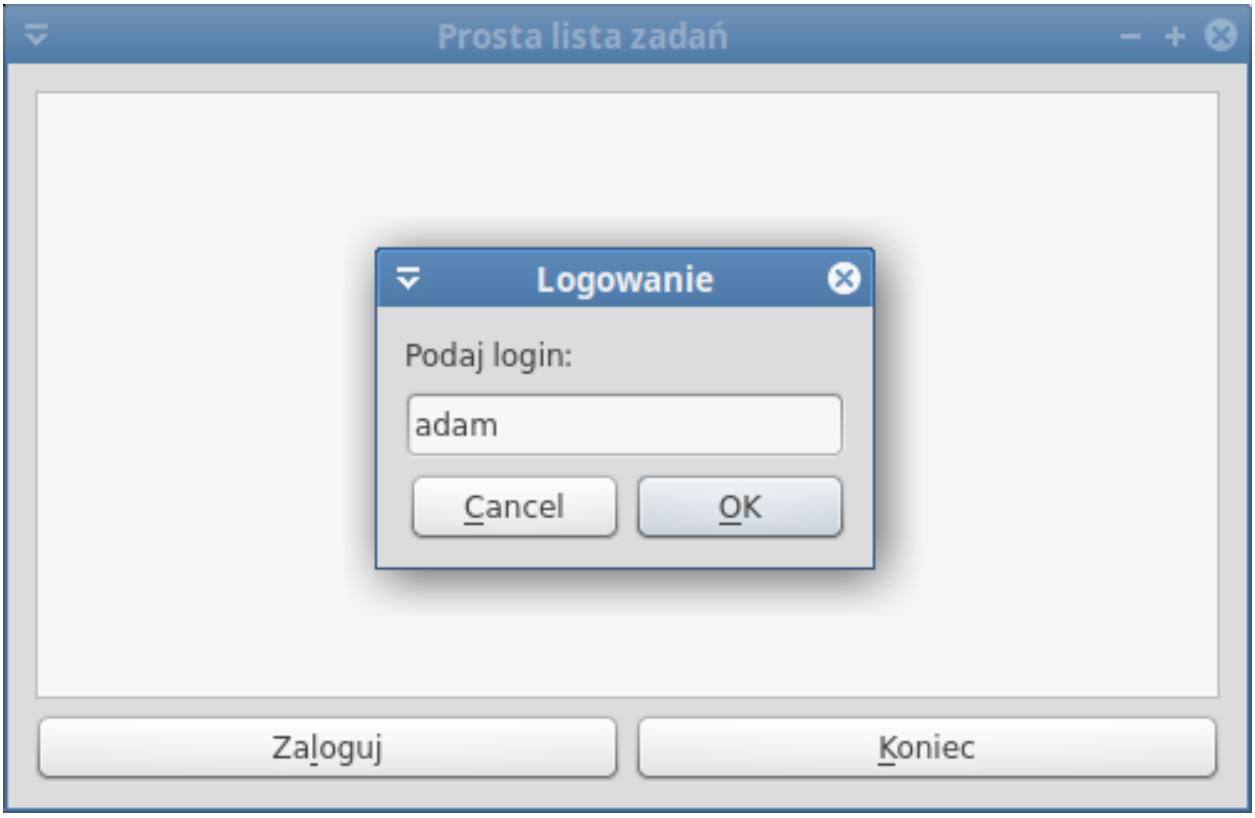
Przeglądanie i dodawanie zadań wymaga zalogowania, które obsługuje funkcja `loguj()`. Login i hasło użytkow-

nika można pobrać za pomocą widżetu QInputDialog, np.: `login, ok = QInputDialog.getText(self, 'Logowanie', 'Podaj login:')`. Zmienna `ok` przyjmie wartość `True`, jeżeli użytkownik zamknie okno naciśnięciem przycisku *OK*.

Jeżeli użytkownik nie podał loginu lub hasła, za pomocą okna dialogowego typu QMessageBox wyświetlamy ostrzeżenie (`warning`). W przeciwnym wypadku wyświetlamy okno informacyjne (`information`) z wprowadzonymi wartościami.

Aplikację testujemy wpisując w terminalu polecenie:

```
~/todopw$ python todopw.py
```



Okno logowania

Pobieranie loginu i hasła w osobnych dialogach nie jest optymalne. Na podstawie klasy QDialog stworzymy specjalne okno dialogowe. Na początku dodajemy importy:

```
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QDialog, QDialogButtonBox
from PyQt5.QtWidgets import QLabel, QLineEdit
from PyQt5.QtWidgets import QGridLayout
```

Na końcu pliku `gui.py` wstawiamy:

```
38 class LoginDialog(QDialog):
39     """ Okno dialogowe logowania """
40
41     def __init__(self, parent=None):
```

```
42          super(LoginDialog, self).__init__(parent)
43
44          # etykiety, pola edycyjne i przyciski ###
45          loginLbl = QLabel('Login')
46          hasloLbl = QLabel('Hasło')
47          self.login = QLineEdit()
48          self.haslo = QLineEdit()
49          self.przyciski = QDialogButtonBox(
50              QDialogButtonBox.Ok | QDialogButtonBox.Cancel,
51              Qt.Horizontal, self)
52
53          # układ główny ###
54          uklad = QGridLayout(self)
55          uklad.addWidget(loginLbl, 0, 0)
56          uklad.addWidget(self.login, 0, 1)
57          uklad.addWidget(hasloLbl, 1, 0)
58          uklad.addWidget(self.haslo, 1, 1)
59          uklad.addWidget(self.przyciski, 2, 0, 2, 0)
60
61          # sygnały i sloty ###
62          self.przyciski.accepted.connect(self.accept)
63          self.przyciski.rejected.connect(self.reject)
64
65          # właściwości widżetu ###
66          self.setModal(True)
67          self.setWindowTitle('Logowanie')
68
69      def loginHaslo(self):
70          return (self.login.text().strip(),
71                  self.haslo.text().strip())
72
73      # metoda statyczna, tworzy dialog i zwraca (login, haslo, ok)
74      @staticmethod
75      def getLoginHaslo(parent=None):
76          dialog = LoginDialog(parent)
77          dialog.login.setFocus()
78          ok = dialog.exec_()
79          login, haslo = dialog.loginHaslo()
80          return (login, haslo, ok == QDialog.Accepted)
```

Okno składa się z dwóch etykiet, odpowiadających im 1-liniowych pól edycyjnych oraz standardowych przycisków. Wywołanie metody `setModal(True)` powoduje, że dopóki użytkownik nie zamknie okna, nie może manipulować oknem rodzica, czyli aplikacją.

Do wywołania okna użyjemy metody statycznej `getLoginHaslo()` (zob. *metoda statyczna*) klasy *LoginDialog*. Można by ją zapisać nawet poza definicją klasy, ale ponieważ ściśle jest z nią związana, używamy dekoratora `@staticmethod`. Metodę wywołamy w pliku `todopw.py` w postaci `LoginDialog.getLoginHaslo(self)`. Tworzy ona okno dialogowe (`dialog = LoginDialog(parent)`) i aktywuje pole loginu. Następnie wyświetla okno i zapisuje odpowiedź użytkownika (wciśnięty przycisk) w zmiennej: `ok = dialog.exec_()`. Po zamknięciu okna pobiera wpisane dane za pomocą funkcji pomocniczej `loginHaslo()` i zwraca je, o ile użytkownik wcisnął przycisk *OK*.

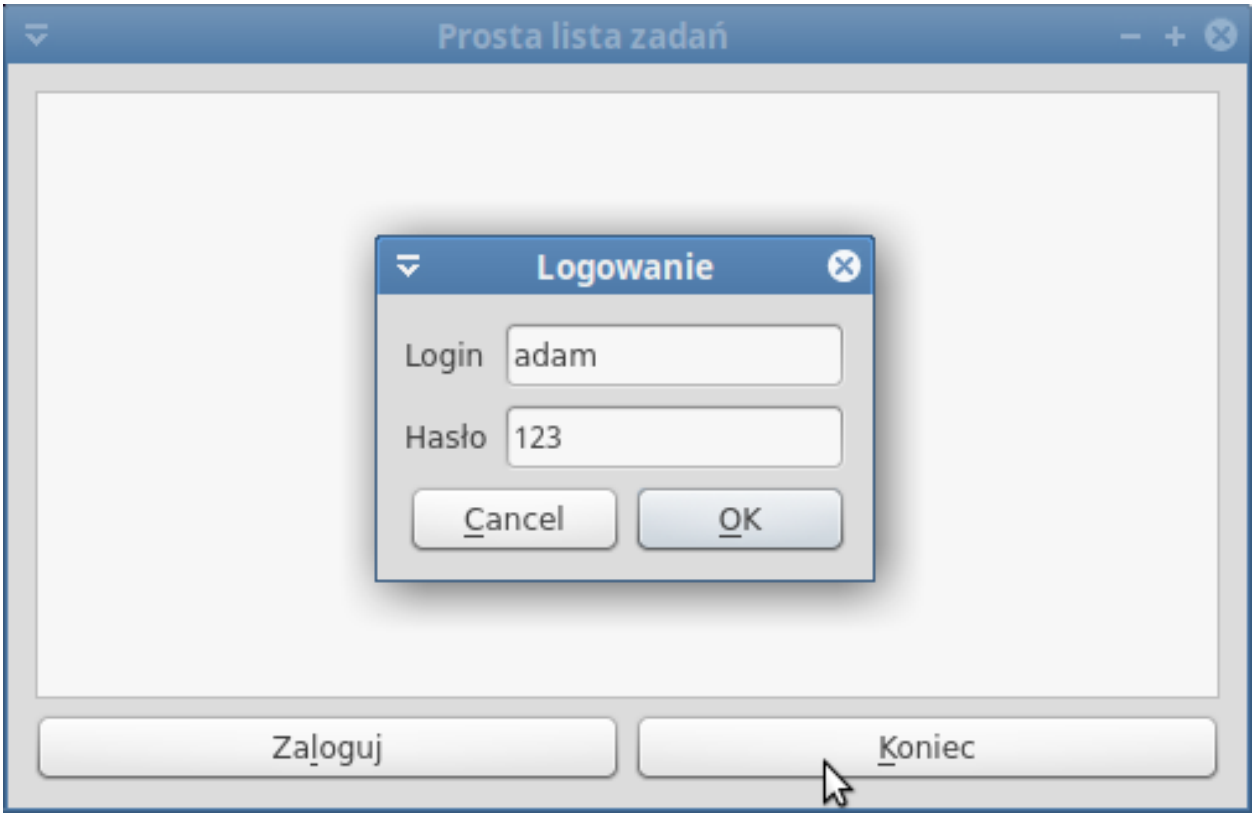
W pliku `todopw.py` uzupełniamy importy:

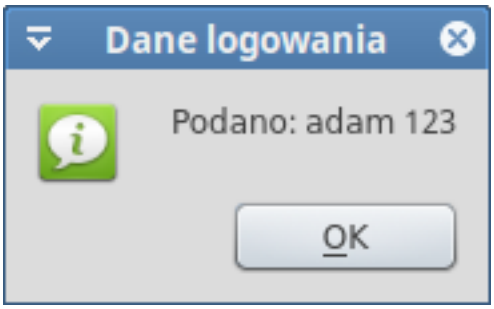
```
from gui import Ui_Widget, LoginDialog
```

– i zmieniamy funkcję `loguj()`:

```
19      def loguj(self):
20          login, haslo, ok = LoginDialog.getLoginHaslo(self)
21          if not ok:
22              return
23
24          if not login or not haslo:
25              QMessageBox.warning(self, 'Błąd',
26                                  'Pusty login lub hasło!', QMessageBox.Ok)
27              return
28
29          QMessageBox.information(self,
30              'Dane logowania', 'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)
```

Przetestuj działanie nowego okna dialogowego.





Podłączamy bazę

Dane użytkowników oraz ich listy zadań zapisywać będziemy w bazie SQLite. Dla uproszczenia jej obsługi wykorzystamy prosty system ORM Peewee. Kod umieścimy w osobnym pliku o nazwie `baza.py`. Po utworzeniu tego pliku wypełniamy go poniższą zawartością:

```
# -*- coding: utf-8 -*-

from peewee import *
from datetime import datetime

baza = SqliteDatabase('adresy.db')


class BazaModel(Model):  # klasa bazowa

    class Meta:
        database = baza


class Osoba(BazaModel):
    login = CharField(null=False, unique=True)
    haslo = CharField()

    class Meta:
        order_by = ('login',)


class Zadanie(BazaModel):
    tresc = TextField(null=False)
    datad = DateTimeField(default=datetime.now)
    wykonane = BooleanField(default=False)
    osoba = ForeignKeyField(Osoba, related_name='zadania')

    class Meta:
        order_by = ('datad',)


def polacz():
    baza.connect()  # nawiązujemy połączenie z bazą
    baza.create_tables([Osoba, Zadanie], True)  # tworzymy tabele
    ladujDane()  # wstawiamy początkowe dane
    return True


def loguj(login, haslo):
    try:
        osoba, created = Osoba.get_or_create(login=login, haslo=haslo)
        return osoba
    except IntegrityError:
        return None


def ladujDane():
    """ Przygotowanie początkowych danych testowych """
    if Osoba.select().count() > 0:
        return
    osoby = ('adam', 'ewa')
```

```
    zadania = ('Pierwsze zadanie', 'Drugie zadanie', 'Trzecie zadanie')
    for login in osoby:
        o = Osoba(login=login, haslo='123')
        o.save()
        for tresc in zadania:
            z = Zadanie(tresc=tresc, osoba=o)
            z.save()
    baza.commit()
    baza.close()
```

Po zaimportowaniu wymaganych modułów mamy definicje klas *Osoba* i *Zadania*, na podstawie których tworzyć będziemy obiekty reprezentujące użytkownika i jego zadania. W pliku definiujemy również instancję bazy w instrukcji: `baza = SqliteDatabase('adresy.db')`. Jako argument podajemy nazwę pliku, w którym zapisywane będą dane.

Dalej mamy trzy funkcje pomocnicze:

- `polacz()` – służy do nawiązania połączenia z bazą, utworzenia tabel, o ile ich w bazie nie ma oraz do wywołania funkcji ładującej początkowe dane testowe;
- `loguj()` – funkcja stara się odczytać z bazy dane użytkownika o podanym loginie i haśle; jeżeli użytkownika nie ma w bazie, zostaje automatycznie utworzony pod warunkiem, że podany login nie został wcześniej wykorzystany; w takim wypadku zamiast obiektu reprezentującego użytkownika zwrócona zostanie wartość `None`;
- `ladujDane()` – jeżeli tabela użytkowników jest pusta, funkcja doda dane dwóch testowych użytkowników.

Resztę zmian nanosimy w pliku `todopw.py`. Przede wszystkim importujemy przygotowany przed chwilą moduł obsługujący bazę:

```
import baza
```

Dalej uzupełniamy funkcję `loguj()`:

```
    def loguj(self):
        """ Logowanie użytkownika """
        login, haslo, ok = LoginDialog.getLoginHaslo(self)
        if not ok:
            return

        if not login or not haslo:
            QMessageBox.warning(self, 'Błąd',
                                'Pusty login lub hasło!', QMessageBox.Ok)
            return

        self.osoba = baza.loguj(login, haslo)
        if self.osoba is None:
            QMessageBox.critical(self, 'Błąd', 'Błędne hasło!', QMessageBox.Ok)
            return

        QMessageBox.information(self,
            'Dane logowania', 'Podano: ' + login + ' ' + haslo, QMessageBox.Ok)
```

Jak widać, dopisujemy kod logujący użytkownika w bazie: `self.osoba = baza.loguj(login, haslo)`.

Na końcu pliku, po utworzeniu obiektu aplikacji (`app = QApplication(sys.argv)`), musimy jeszcze wywołać funkcję ustanawiającą połączenie z bazą, czyli wstawić kod `baza.polacz()`:

```
if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    baza.polacz()
```

Przetestuj działanie aplikacji. Znakiem poprawnego jej działania będzie utworzenie pliku bazy `adresy.db`, komunikat wyświetlający poprawnie podany login i hasło lub komunikat o błędzie, jeżeli login został już w bazie użyty, a hasło do niego nie pasuje.

Model danych

Kluczowym zadaniem podczas programowania z wykorzystaniem techniki model/widok jest zaimplementowanie modelu. Jego zadaniem jest stworzenie interfejsu dostępu do danych dla komponentów pełniących rolę widoków. Zob. Model Classess.

Informacja: Warto zauważyć, ze dane udostępniane przez model mogą być prezentowane za pomocą różnych widoków jednocześnie.

Ponieważ listę zadań przechowujemy w zewnętrznej bazie danych w tabeli, model stworzymy na podstawie klasy QAbstractTableModel. W nowym pliku o nazwie `tabmodel.py` umieszczamy następujący kod:

```
# -*- coding: utf-8 -*-
from __future__ import unicode_literals
from PyQt5.QtCore import QAbstractTableModel, QModelIndex, Qt, QVariant


class TabModel(QAbstractTableModel):
    """ Tabelaryczny model danych """

    def __init__(self, pola=[], dane=[], parent=None):
        super(TabModel, self).__init__()
        self.pola = pola
        self.tabela = dane

    def aktualizuj(self, dane):
        """ Przypisuje źródło danych do modelu """
        print(dane)
        self.tabela = dane

    def rowCount(self, parent=QModelIndex()):
        """ Zwraca ilość wierszy """
        return len(self.tabela)

    def columnCount(self, parent=QModelIndex()):
        """ Zwraca ilość kolumn """
        if self.tabela:
            return len(self.tabela[0])
        else:
            return 0

    def data(self, index, rola=Qt.DisplayRole):
        """ Wyświetlanie danych """
        i = index.row()
        j = index.column()

```

```
        if rola == Qt.DisplayRole:
            return '{0}'.format(self.tabela[i][j])
        else:
            return QVariant()
```

Konstruktor klasy *TabModel* opcjonalnie przyjmuje listę pól oraz listę rekordów – z tych możliwości skorzystamy później. Dane będzie można również przypisać za pomocą metody `aktualizuj()`. Wywołanie `print(dane)` jest w niej umieszczone tylko w celach poglądowych: wydrukuje przekazane dane w konsoli.

Dwie kolejne funkcje `rowCount()` i `columnCount()` są obowiązkowe i zgodnie ze swoimi nazwami zwracają ilość wierszy (`len(self.tabela)`) i kolumn (`len(self.tabela[0])`) w każdym wierszu. Jak widać, dane przekazywać będziemy w postaci listy list, czy też listy dwuwymiarowej.

Funkcja `data()` również jest obowiązkowa i odpowiada za wyświetlanie danych. Wywoływana jest dla każdego wiersza i każdej kolumny osobno. Trzecim parametrem tej funkcji jest tzw. *rola* (zob. ItemDataRole), oznaczająca rodzaj danych wymaganych przez widok do właściwego wyświetlenia danych. Domyślną wartością jest `Qt.DisplayRole`, czyli wyświetlanie danych, dla której zwracamy reprezentację tekstową naszych danych: `return '{0}'.format(self.tabela[i][j])`.

Dane przekazywane do modelu odczytamy za pomocą funkcji, którą dopisujemy do pliku `baza.py`:

```
def czytajDane(osoba):
    """ Pobranie zadań danego użytkownika z bazy """
    zadania = []  # lista zadań
    wpisy = Zadanie.select().where(Zadanie.osoba == osoba)
    for z in wpisy:
        zadania.append([
            z.id,  # identyfikator zadania
            z.tresc,  # treść zadania
            '{0:%Y-%m-%d %H:%M:%S}'.format(z.datad),  # data dodania
            z.wykonane,  # bool: czy wykonane?
            False])  # bool: czy usunąć?
    return zadania
```

Funkcję `czytajDane()` odczytuje wszystkie zadania danego użytkownika z bazy: `wpisy = Zadanie.select().where(Zadanie.osoba == osoba)`. Następnie w pętli do listy `zadania` dodajemy rekordy opisujące kolejne zadania (`zadania.append()`). Każdy rekord to lista, która zawiera: identyfikator, treść, datę dodania, pole oznaczające wykonanie zadania oraz dodatkową wartość logiczną, która pozwoli wskazać zadania do usunięcia.

Pozostaje nam edycja pliku `todopw.py`. Na początku trzeba zaimportować model:

```
from tabmodel import TabModel
```

Następnie tworzymy jego instancję. Uzupełniamy fragment uruchamiający aplikację o kod: `model = TabModel()`:

```
if __name__ == '__main__':
    import sys
    app = QApplication(sys.argv)
    baza.polacz()
    model = TabModel()
    okno = Zadania()
    okno.show()
    okno.move(350, 200)
    sys.exit(app.exec_())
```

Zadania użytkownika odczytujemy w funkcji `loguj()`, w której kod wyświetlający dialog informacyjny (`QMessageBox.information(...)`) zastępujemy oraz dodajemy nową funkcję:

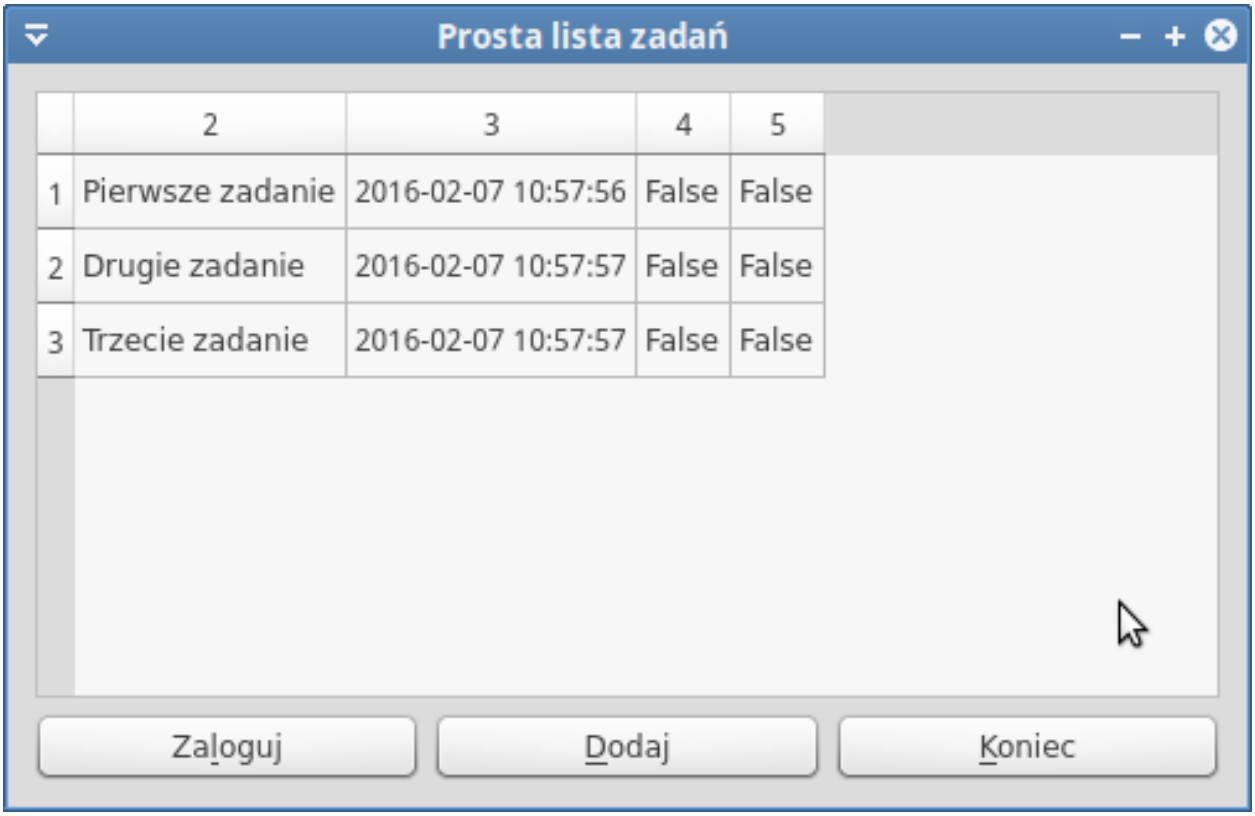
```
37          zadania = baza.czytajDane(self.osoba)
38          model.aktualizuj(zadania)
39          model.layoutChanged.emit()
40          self.odswiezWidok()
41
42      def odswiezWidok(self):
43          self.widok.setModel(model)   # przekazanie modelu do widoku
```

Po odczytaniu zadań `zadania = baza.czytajDane(self.osoba)` przypisujemy dane modelowi `model.aktualizuj(zadania)`.

Instrukcja `model.layoutChanged.emit()` powoduje wysłanie sygnału powiadamiającego widok o zmianie danych. Umieszczamy ją, aby po ewentualnym ponownym zalogowaniu kolejny użytkownik zobaczył swoje zadania.

Dane modelu musimy przekazać widokowi. To zadanie metody `odswiezWidok()`, która wywołuje polecenie: `self.widok.setModel(model)`.

Przetestuj aplikację logując się jako “adam” lub “ewa” z hasłem “123”.



Dodawanie zadań

Możemy już przeglądać zadania, ale jeżeli zalogujemy się jako nowy użytkownik, nic w tabeli nie zobaczymy. Aby umożliwić dodawanie zadań, w pliku `gui.py` tworzymy nowy przycisk “Dodaj”, który po uruchomieniu będzie nieaktywny:

```
        # przyciski Push ###
        self.logujBtn = QPushButton("Za&loguj")
        self.koniecBtn = QPushButton("&Koniec")
        self.dodajBtn = QPushButton("&Dodaj")
        self.dodajBtn.setEnabled(False)

        # układ przycisków Push ###
        uklad = QHBoxLayout()
        uklad.addWidget(self.logujBtn)
        uklad.addWidget(self.dodajBtn)
        uklad.addWidget(self.koniecBtn)
```

W pliku `todopw.py` uzupełniamy konstruktor i dodajemy nową funkcję `dodaj()`:

```
    def __init__(self, parent=None):
        super(Zadania, self).__init__(parent)
        self.setupUi(self)

        self.logujBtn.clicked.connect(self.loguj)
        self.koniecBtn.clicked.connect(self.koniec)
        self.dodajBtn.clicked.connect(self.dodaj)

    def dodaj(self):
        """ Dodawanie nowego zadania """
        zadanie, ok = QInputDialog.getMultiLineText(self,
                                                    'Zadanie',
                                                    'Co jest do zrobienia?')
        if not ok or not zadanie.strip():
            QMessageBox.critical(self,
                                 'Błąd',
                                 'Zadanie nie może być puste.',
                                 QMessageBox.Ok)
            return

        zadanie = baza.dodajZadanie(self.osoba, zadanie)
        model.tabela.append(zadanie)
        model.layoutChanged.emit()  # wyemituj sygnał: zaszła zmiana!
        if len(model.tabela) == 1:  # jeżeli to pierwsze zadanie
            self.odswiezWidok()     # trzeba przekazać model do widoku
```

Kliknięcie przycisku "Dodaj" wiążemy z nową funkcją `dodaj()`. Treść zadania pobieramy za pomocą omawianego okna typu `QInputDialog`. Po sprawdzeniu, czy użytkownik w ogóle coś wpisał, wywołujemy funkcję `dodajZadanie()` z modułu `baza`, która zapisuje nowe dane w bazie. Następnie aktualizujemy dane modelu, czyli do listy zadań dodajemy rekord nowego zadania: `model.tabela.append(zadanie)`. Ponieważ następuje zmiana danych modelu, emitujemy odpowiedni sygnał: `model.layoutChanged.emit()`.

Jeżeli nowe zadanie jest pierwszym w modelu (`if len(model.tabela) == 1`), należy jeszcze odświeżyć widok. Wywołujemy więc funkcję `odswiezWidok()`, którą modyfikujemy do podanej postaci:

```
    def odswiezWidok(self):
        self.widok.setModel(model)  # przekazanie modelu do widoku
        self.widok.hideColumn(0)  # ukrywamy kolumnę id
        # ograniczenie szerokości ostatniej kolumny
        self.widok.horizontalHeader().setStretchLastSection(True)
        # dopasowanie szerokości kolumn do zawartości
        self.widok.resizeColumnsToContents()
```

W uzupełnionej funkcji wywołujemy metody obiektu widoku, które ukrywają pierwszą kolumnę z identyfikatorami

zadań, ograniczają szerokość ostatniej kolumny oraz powodują dopasowanie szerokości kolumn do zawartości.

Musimy jeszcze aktywować przycisk dodawania po zalogowaniu się użytkownika. Na końcu funkcji `loguj()` dopisujemy:

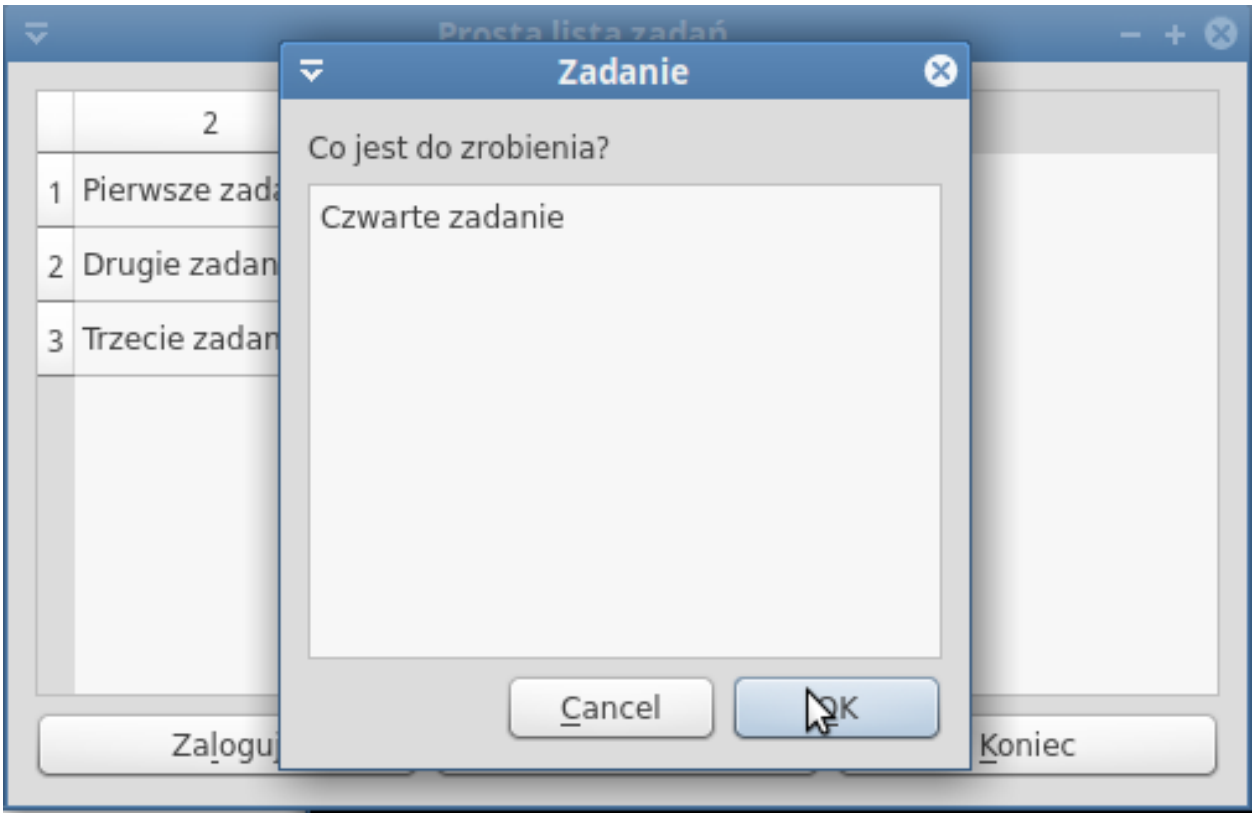
```
self.dodajBtn.setEnabled(True)
```

W pliku `baza.py` dopisujemy jeszcze wspomnianą funkcję `dodajZadanie()`:

```
78 def dodajZadanie(osoba, tresc):
79     """ Dodawanie nowego zadania """
80     zadanie = Zadanie(tresc=tresc, osoba=osoba)
81     zadanie.save()
82     return [
83         zadanie.id,
84         zadanie.tresc,
85         '{0:%Y-%m-%d %H:%M:%S}'.format(zadanie.datad),
86         zadanie.wykonane,
87         False]
```

Zapisanie zadania jest proste dzięki wykorzystaniu systemu ORM. Tworzymy instancję klasy *Zadanie*: `zadanie = Zadanie(tresc=tresc, osoba=osoba)` – podając tylko wymagane dane. Wartości pozostałych pól utworzone zostaną na podstawie wartości domyślnych określonych w definicji klasy. Wywołanie metody `save()` zapisuje zadanie w bazie. Funkcja zwraca listę – rekord o takiej samej strukturze, jak funkcja `czytajDane()`.

Pozostaje uruchomienie aplikacji i dodanie nowego zadania.



Edycja i widok danych

Edycję zadań można zrealizować za pomocą funkcjonalności modelu. Rozszerzamy więc funkcję `data()` i uzupełniamy definicję klasy *TabModel* w pliku `tabmodel.py`:

```
30      def data(self, index, rola=Qt.DisplayRole):
31          """ Wyświetlanie danych """
32          i = index.row()
33          j = index.column()
34
35          if rola == Qt.DisplayRole:
36              return '{0}'.format(self.tabela[i][j])
37          elif rola == Qt.CheckStateRole and (j == 3 or j == 4):
38              if self.tabela[i][j]:
39                  return Qt.Checked
40              else:
41                  return Qt.Unchecked
42          elif rola == Qt.EditRole and j == 1:
43              return self.tabela[i][j]
44          else:
45              return QVariant()
46
47      def flags(self, index):
48          """ Zwraca właściwości kolumn tabeli """
49          flags = super(TabModel, self).flags(index)
50          j = index.column()
51          if j == 1:
52              flags |= Qt.ItemIsEditable
53          elif j == 3 or j == 4:
54              flags |= Qt.ItemIsUserCheckable
55
56          return flags
57
58      def setData(self, index, value, rola=Qt.DisplayRole):
59          """ Zmiana danych """
60          i = index.row()
61          j = index.column()
62          if rola == Qt.EditRole and j == 1:
63              self.tabela[i][j] = value
64          elif rola == Qt.CheckStateRole and (j == 3 or j == 4):
65              if value:
66                  self.tabela[i][j] = True
67              else:
68                  self.tabela[i][j] = False
69
70          return True
71
72      def headerData(self, sekcja, kierunek, rola=Qt.DisplayRole):
73          """ Zwraca nagłówki kolumn """
74          if rola == Qt.DisplayRole and kierunek == Qt.Horizontal:
75              return self.pola[sekcja]
76          elif rola == Qt.DisplayRole and kierunek == Qt.Vertical:
77              return sekcja + 1
78          else:
79              return QVariant()
```

W funkcji `data()` dodajemy obsługę roli `Qt.CheckStateRole`, pozwalającej w polach typu prawda/fałsz wyświetlić kontrolki *checkbox*. Rozpoczęcie edycji danych, np. poprzez dwukrotne kliknięcie, wywołuje rolę `Qt.`

`EditRole`, wtedy zwracamy do dotychczasowe dane.

Właściwości danego pola danych określa funkcja `flags()`, która wywoływana jest dla każdego pola osobno. W naszej implementacji, po sprawdzeniu indeksu pola, pozwalamy na zmianę treści zadania: `flags |= Qt.ItemIsEditable`. Pozwalamy również na oznaczenie zadania jako wykonanego i przeznaczonego do usunięcia: `flags |= Qt.ItemIsUserCheckable`.

Faktyczną edycję danych zatwierdza funkcja `setData()`. Po sprawdzeniu roli i indeksu pola aktualizuje ona treść zadania oraz stan pól typu *checkbox* w modelu.

Ostatnia funkcja, `headerData()`, odpowiada za wyświetlanie nagłówków kolumn. Nagłówki pól (resp. kolumn, `kierunek == Qt.Horizontal`), odczytywane są z listy: `return self.pola[sekcja]`. Kolejne rekordy (resp. wiersze, `kierunek == Qt.Vertical`) są kolejno numerowane: `return sekcja+1`. Zmienna `sekcja` oznacza numer kolumny lub wiersza.

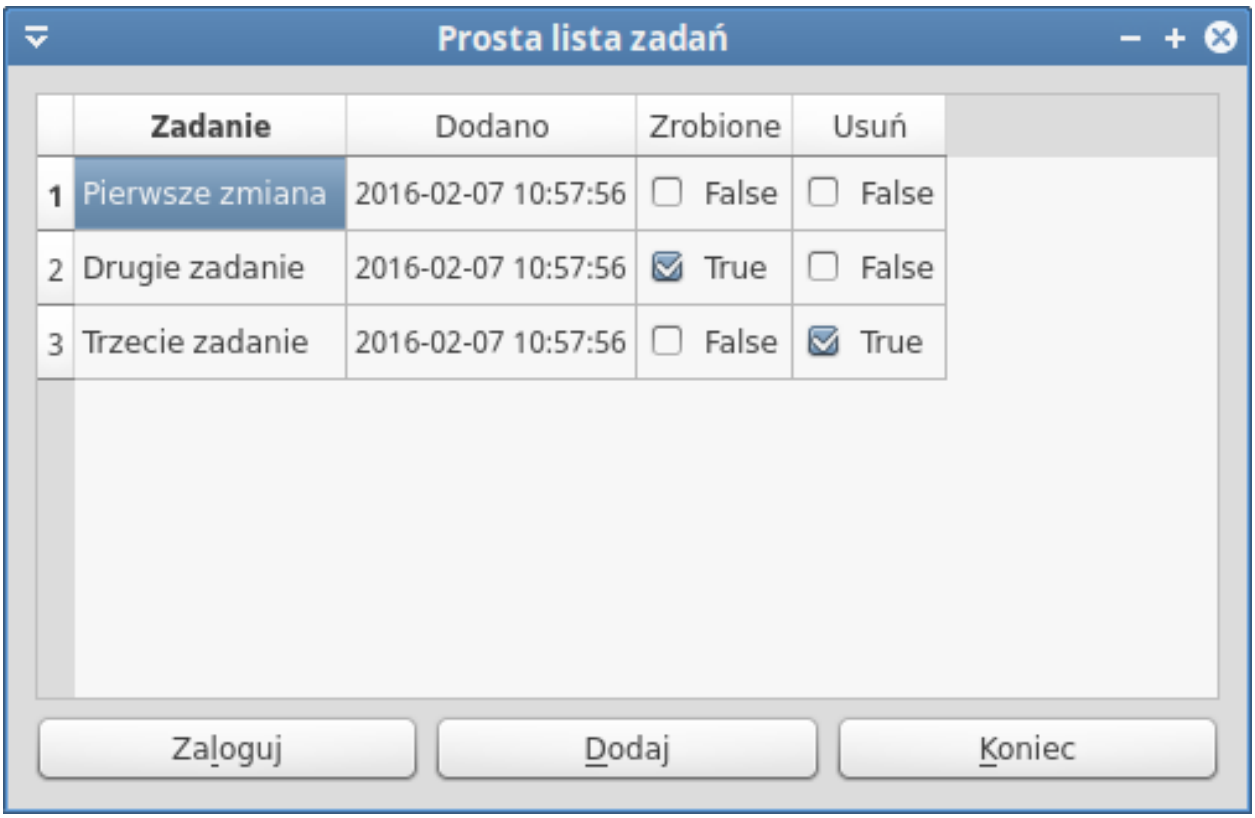
Listę nagłówków kolumn definiujemy w pliku `baza.py` dopisując na końcu:

```
90 pola = ['Id', 'Zadanie', 'Dodano', 'Zrobione', 'Usuń']
```

W pliku `todopw.py` uzupełniamy jeszcze kod tworzący instancję modelu:

```
72 if __name__ == '__main__':
73     import sys
74     app = QApplication(sys.argv)
75     baza.polacz()
76     model = TabModel(baza.pola)
```

Uruchom zmodyfikowaną aplikację. Spróbuj zmienić treść zadania dwukrotnie klikając. Oznacz wybrane zadania jako wykonane lub przeznaczone do usunięcia.



Zapisywanie zmian

Możemy już edytować zadania, oznaczać je jako wykonane i przeznaczone do usunięcia, ale zmiany te nie są zapisywane. Dodamy więc taką możliwość. W pliku `gui.py` tworzymy jeszcze jeden przycisk i dodajemy go do układu:

```
19          # przyciski Push ###
20          self.logujBtn = QPushButton("Za&loguj")
21          self.koniecBtn = QPushButton("&Koniec")
22          self.dodajBtn = QPushButton("&Dodaj")
23          self.dodajBtn.setEnabled(False)
24          self.zapiszBtn = QPushButton("&Zapisz")
25          self.zapiszBtn.setEnabled(False)
26
27          # układ przycisków Push ###
28          uklad = QHBoxLayout()
29          uklad.addWidget(self.logujBtn)
30          uklad.addWidget(self.dodajBtn)
31          uklad.addWidget(self.zapiszBtn)
32          uklad.addWidget(self.koniecBtn)
```

W pliku `todopw.py` kliknięcie przycisku "Zapisz" wiążemy z nową funkcją `zapisz()`:

```
14      def __init__(self, parent=None):
15          super(Zadania, self).__init__(parent)
16          self.setupUi(self)
17
18          self.logujBtn.clicked.connect(self.loguj)
19          self.koniecBtn.clicked.connect(self.koniec)
20          self.dodajBtn.clicked.connect(self.dodaj)
21          self.zapiszBtn.clicked.connect(self.zapisz)
22
23      def zapisz(self):
24          baza.zapiszDane(model.tabela)
25          model.layoutChanged.emit()
```

Slot `zapisz()` wywołuje funkcję zdefiniowaną w module `baza.py`, przekazując jej listę z rekordami: `baza.zapiszDane(model.tabela)`. Na koniec emitujemy sygnał zmiany, aby widok mógł uaktualnić dane, jeżeli jakieś zadania zostały usunięte.

Przycisk "Zapisz" podobnie jak "Dodaj" powinien być uaktywniony po zalogowaniu użytkownika. Na końcu funkcji `loguj()` należy dopisać kod:

```
self.zapiszBtn.setEnabled(True)
```

Pozostaje dopisanie na końcu pliku `baza.py` funkcji zapisującej zmiany:

```
93  def zapiszDane(zadania):
94      """ Zapisywanie zmian """
95      for i, z in enumerate(zadania):
96          # utworzenie instancji zadania
97          zadanie = Zadanie.select().where(Zadanie.id == z[0]).get()
98          if z[4]:  # jeżeli zaznaczono zadanie do usunięcia
99              zadanie.delete_instance()  # usunięcie zadania z bazy
100             del zadania[i]  # usunięcie zadania z danych modelu
101         else:
102             zadanie.tresc = z[1]
```

```
103             zadanie.wykonane = z[3]
104             zadanie.save()
```

W pętli odczytujemy indeksy i rekordy z danymi zadań: `for i, z in enumerate(zadania)`. Tworzymy instancję każdego zadania na podstawie identyfikatora zapisanego jako pierwszy element listy: `zadanie = Zadanie.select().where(Zadanie.id == z[0]).get()`. Później albo usuwamy zadanie, albo aktualizujemy przypisując polom "tresc" i "wykonane" dane z modelu.

To wszystko, przetestuj gotową aplikację.

Materiały

1. Model/View Programming
2. Model/View Tutorial
3. Presenting Data in a Table View
4. Layout Management

Źródła:

- `ToDoPw Qt5`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Słownik (Py)Qt

GUI (ang. Graphical User Interface) – graficzny interfejs użytkownika, czyli sposób prezentacji informacji na komputerze i innych urządzeniach oraz interakcji z użytkownikiem.

widżet (ang. *widget*) – podstawowy element graficzny interfejsu, zwany czasami kontrolką, nie tylko główne okno aplikacji, ale również etykiety, pola edycyjne, przycicki itd.

główna pętla programu (ang. *mainloop*) – mechanizm komunikacji między aplikacją, systemem i użytkownikiem. Zapewnia przekazywanie zdarzeń do aplikacji. Zdarzenia wynikają z zachowania systemu lub użytkownika (kliknięcia, użycie klawiatury, czyli edycja danych itd.) i przekazywane są do widżetów apliakcji, które mogą – choć nie muszą – na nie reagować, np. wywołując jakąś metodę (funkcję).

klasa – schematyczny model obiektu, czyli opis jego właściwości i działań na nich. Właściwości tworzą dane, którymi manipuluje się za pomocą metod klasy implementowanych jako funkcje.

konstruktor – metoda wykonywana domyślnie w momncie tworzenia instancji klasy, czyli obiektu. Służy do inicjowania danych klasy. W Pythonie nazywa się `__init()__`.

obiekt – termin wieloznaczny; w kontekście OOP (ang. Object Oriented Programing), czyli programowania zorientowanego obiektowo, oznacza element rzeczywistości, który próbujemy opisać za pomocą klas. Np. osobę, ale też okno aplikacji.

instancja – obiekt utworzony na podstawie klasy, która go opisuje. Posiada konkretne właściwości, które odróżniają go od innych instancji klasy.

sygnały i sloty – (ang. signals and slots), sygnały powstają kiedy zachodzi jakieś wydarzenie. W odpowiedzi na sygnał wywoływane są sloty, czyli funkcje. Wiele sygnałów można łączyć z jednym slotem i odwrotnie. Można też łączyć ze sobą sygnały. Widżety Qt mają wiele predefiniowanych zarówno sygnałów, jak i slotów. Można jednak tworzyć własne. Dzięki temu obsługuje się tylko te zdarzenia, które nas interesują.

dziedziczenie w programowaniu obiektowym nazywamy mechanizm współdzielenia funkcjonalności między klasami. Klasa może dziedziczyć po innej klasie, co w najprostszym przypadku oznacza, że oprócz swoich własnych atrybutów oraz zachowań, uzyskuje także te pochodzące z klasy, z której dziedziczy. Jest wiele odmian dziedziczenia .

metoda statyczna – (ang. static method), metody powiązane z klasą, a nie z jej instancjami, czyli obiektami. Tworzymy je używając w ciele klasy dekoratora `@staticmethod`. Do metody takiej trzeba odwoływać się podając nazwę klasy, np. Klasa.metoda(). Metoda statyczna nie otrzymuje parametru `self`.

dana statyczna – (ang. static data), dane powiązane z klasą, a nie z jej instancjami, czyli obiektami. Tworzymy je definiując atrybuty klasy. Korzystamy z nich podając nazwę klasy, np.: `Klasa.dana`. Wszystkie instancje klasy dzielą ze sobą jeden egzemplarz danych statycznych.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Informacja: Aplikacje okienkowe w Pythonie można tworzyć z wykorzystaniem innych rozwiązań, takich jak:

- Tkinter – wykorzystuje bibliotekę Tk;
- PyGTK – wykorzytuje bibliotekę GTK+;
- wxPython – wykorzystuje bibliotekę wxWidgets;
- PySide – wykorzystuje bibliotekę Qt4, alternatywa dla PyQt4.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.7 Aplikacje WWW (Flask)

Python znakomicie nadaje się do tworzenia aplikacji internetowych dzięki takim rozszerzeniom jak micro-framework Flask. Upraszcza on projektowanie zapewniając przejrzysty schemat łączenia adresów URL, źródła danych, widoków i szablonów. Domyślnie dostajemy również deweloperski serwer WWW, nie musimy instalować żadnych dodatkowych narzędzi typu LAMP (WAMP).

Zobacz, jak zainstalować wymagane biblioteki w systemie *Linux* lub *Windows*.

Quiz

Realizacja aplikacji internetowej Quiz w oparciu o *framework* Flask 0.12.x. Na stronie wyświetlamy pytania, użytkownik zaznacza poprawne odpowiedzi, przesyła je na serwer i otrzymuje informację o wynikach.

- *Projekt i aplikacja*
- *Strona główna*
- *Pytania i odpowiedzi*
- *Oceniamy odpowiedzi*
- *Materiały*

Projekt i aplikacja

W katalogu użytkownika tworzymy nowy katalog aplikacji o nazwie `quiz`:

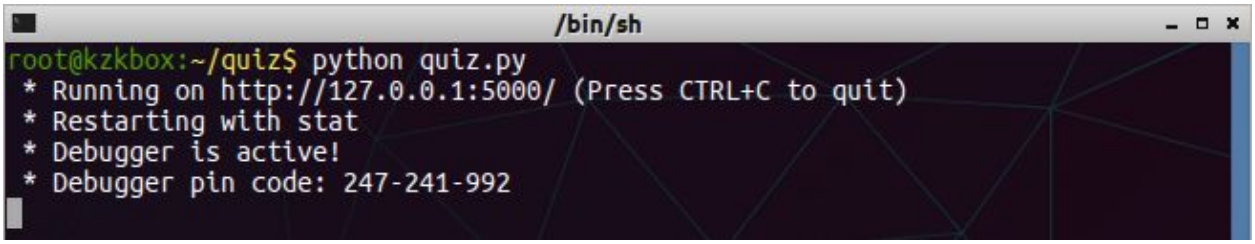
```
~$ mkdir quiz; cd quiz;
```

Utworzymy szkielet aplikacji Flask, co pozwoli na uruchomienie testowego serwera www, umożliwiającego wygodne rozwijanie kodu. W nowym pliku o nazwie `quiz.py` wpisujemy poniższy kod i zapisujemy w katalogu aplikacji.

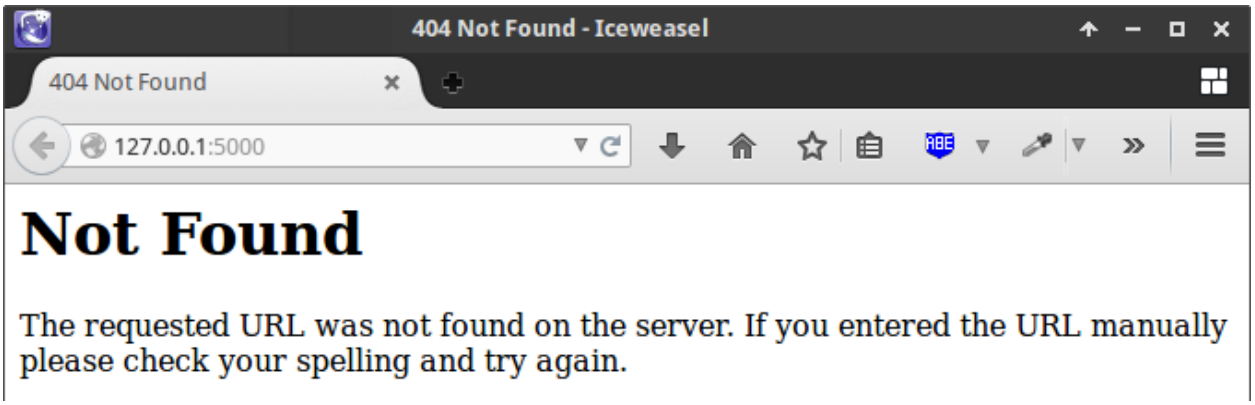
```
1 # -*- coding: utf-8 -*-
2 # quiz/quiz.py
3
4 from flask import Flask
5
6 app = Flask(__name__)
7
8 if __name__ == '__main__':
9     app.run(debug=True)
```

Serwer uruchamiamy komendą:

```
~/quiz$ python3 quiz.py
```



Domyślnie serwer uruchamia się pod adresem *http://127.0.0.1:5000*. Po wpisaniu go do przeglądarki internetowej otrzymamy *kod odpowiedzi HTTP* 404, tj. błąd "nie znaleziono", co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego widoku dla tego adresu.



Wskazówka: Działanie serwera w terminalu zatrzymujemy skrótem `CTRL+C`.

Strona główna

Jeżeli chcemy, aby nasza aplikacja zwracała użytkownikowi jakieś strony www, tworzymy tzw. *widok*. Jest to funkcja Pythona powiązana z określonymi adresami URL za pomocą tzw. dekoratorów. Widoki pozwalają nam obsługiwać podstawowe żądania protokołu *HTTP*, czyli: *GET*, wysyłane przez przeglądarkę, kiedy użytkownik chce zobaczyć stronę, i *POST*, kiedy użytkownik przesyła dane na serwer za pomocą formularza.

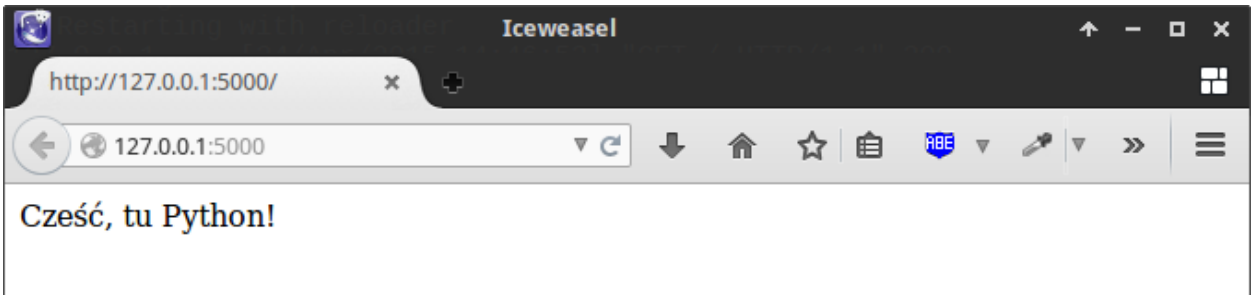
W odpowiedzi aplikacja może odsyłać różne dane. Najczęściej będą to znaczniki *HTML* oraz treści, np. wyniki quizu. Flask ułatwia tworzenie takich dokumentów za pomocą szablonów.

W pliku `quiz.py` umieszczamy kod:

```
# -*- coding: utf-8 -*-
# quiz/quiz.py

from flask import Flask

app = Flask(__name__)


@app.route('/')
def index():
    return 'Cześć, tu Python!'


if __name__ == '__main__':
    app.run(debug=True)
```

Widok (czyli funkcja) `index()` powiązany jest z adresem głównym (/) za pomocą dekoratora `@app.route('/')`. Funkcja zostanie wykonana w odpowiedzi na żądanie GET wysłane przez przeglądarkę po wpisaniu i zatwierdzeniu przez użytkownika adresu serwera.

Najprostszą odpowiedzią jest zwrócenie jakiegoś tekstu: `return 'Cześć, tu Python!'`.



Zazwyczaj będziemy prezentować bardziej skomplikowane dane, w dodatku sformatowane wizualnie. Potrzebujemy szablonów. Będziemy je zapisywać w katalogu `quiz/templates`, który utworzymy np. poleceniem:

```
~/quiz$ mkdir templates
```

Następnie w nowym pliku `templates/index.html` umieszczamy kod:

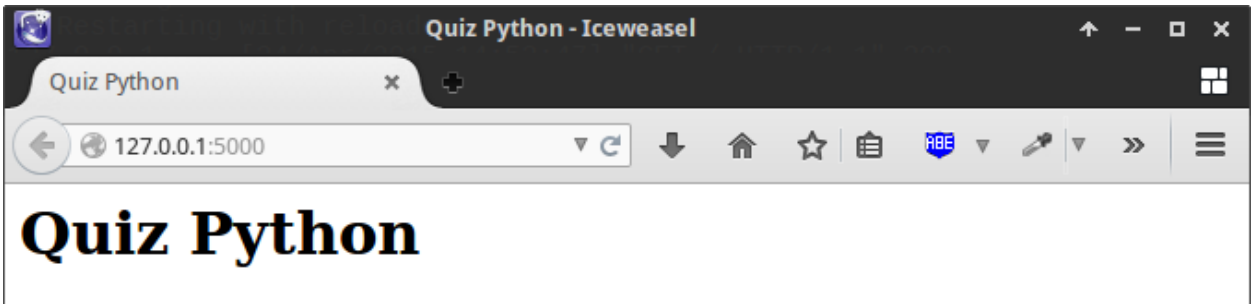
```
<!-- quiz/templates/index.html -->
<html>
  <head>
    <title>Quiz Python</title>
  </head>
<body>
```

```
  <h1>Quiz Python</h1>
</body>
</html>
```

Na koniec modyfikujemy funkcję `index()` w pliku `quiz.py`:

```
# -*- coding: utf-8 -*-
# quiz/quiz.py

from flask import Flask
from flask import render_template

app = Flask(__name__)


@app.route('/')
def index():
    # return 'Cześć, tu Python!'
    return render_template('index.html')


if __name__ == '__main__':
    app.run(debug=True)
```

Do renderowania szablonu (zob: *renderowanie szablonu*) używamy funkcji `render_template('index.html')`, która jako argument przyjmuje nazwę pliku szablonu. Pod adresem *http://127.0.0.1:5000* strony głównej, zobaczymy dokument HTML:



Pytania i odpowiedzi

Dane aplikacji, a więc pytania i odpowiedzi, umieścimy w liście `DANE` w postaci słowników zawierających: treść pytania, listę możliwych odpowiedzi oraz poprawną odpowiedź.

Modyfikujemy plik `quiz.py`. Podany kod wstawiamy po inicjacji zmiennej `app`, ale przed dekoratorem widoku `index()`:

```
# -*- coding: utf-8 -*-
# quiz/quiz.py

from flask import Flask
from flask import render_template

app = Flask(__name__)

```

```
# konfiguracja aplikacji
app.config.update(dict(
    SECRET_KEY='bradzosekretnawartosc',
))

# lista pytań
DANE = [{
    'pytanie': 'Stolica Hiszpani, to:',  # pytanie
    'odpowiedzi': ['Madryt', 'Warszawa', 'Barcelona'],  # możliwe odpowiedzi
    'odpok': 'Madryt'},  # poprawna odpowiedź
    {
    'pytanie': 'Objętość sześcianu o boku 6 cm, wynosi:',
    'odpowiedzi': ['36', '216', '18'],
    'odpok': '216'},
    {
    'pytanie': 'Symbol pierwiastka Helu, to:',
    'odpowiedzi': ['Fe', 'H', 'He'],
    'odpok': 'He'},
]


@app.route('/')
def index():
    # return 'Cześć, tu Python!'
    return render_template('index.html', pytania=DANE)


if __name__ == '__main__':
    app.run(debug=True)
```

W konfiguracji aplikacji dodaliśmy sekretny klucz, wykorzystywany podczas korzystania z sesji (zob *sesja*).

Dane aplikacji

Każda aplikacja korzysta z jakiegoś źródła danych. W najprostszym przypadku dane zawarte są w samej aplikacji. Dodaliśmy więc listę słowników `DANE`, którą przekazujemy dalej jako drugi argument do funkcji `render_template()`. Dzięki temu będziemy mogli odczytać je w szablonie w zmiennej `pytania`.

Do szablonu `index.html` wstawiamy poniższy kod po nagłówku `<h1>`.

```
    <!-- formularz z quizem -->
    <form method="POST">
        <!-- przeglądamy listę pytań -->
        {% for p in pytania %}
          <p>
            <!-- wyświetlamy treść pytania -->
            {{ p.pytanie }}
            <br>
            <!-- zapamiętujemy numer pytania licząc od zera -->
            {% set pnr = loop.index0 %}
            <!-- przeglądamy odpowiedzi dla danego pytania -->
            {% for o in p.odpowiedzi %}
              <label>
                <!-- odpowiedzi wyświetlamy jako pole typu radio -->
                <input type="radio" value="{{ o }}" name="{{ pnr }}">
                {{ o }}
              </label>
              <br>
            {% endfor %}
```

```
20            </p>
21          {% endfor %}
22
23          <!-- przycisk wysyłający wypełniony formularz -->
24          <button type="submit">Sprawdź odpowiedzi</button>
25      </form>
```

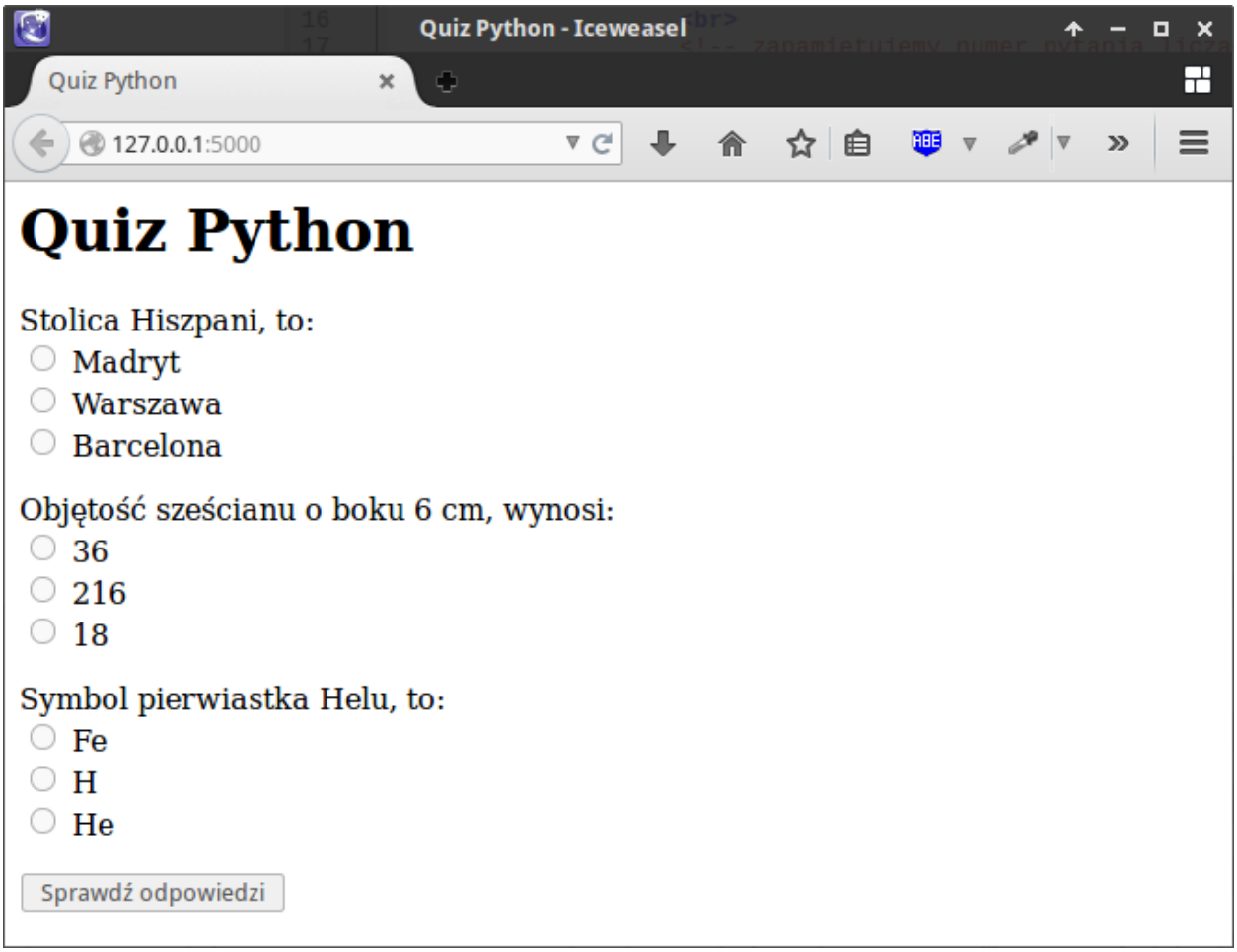
Znaczniki HTML w powyższym kodzie tworzą formularz (`<form>`). Natomiast tagi, czyli polecenia dostępne w szablonach, pozwalają wypełnić go danymi.

- `{% instrukcja %}` – tak wstawiamy instrukcje sterujące;
- `{{ zmienna }}` – tak wstawiamy wartości zmiennych przekazanych do szablonu.

Z przekazanej do szablonu listy pytań, czyli ze zmiennej `pytania` odczytujemy w pętli `{% for p in pytania %}` kolejne słowniki; dalej tworzymy elementy formularza, czyli wyświetlamy treść pytania `{{ p.pytanie }}`, a w kolejnej pętli `{% for o in p.odpowiedz %}` odpowiedzi w postaci grupy opcji typu radio.

Każda grupa odpowiedzi nazywana jest dla odróżnienia numerem pytania liczonym od 0. Odpowiednią zmienną ustawiamy w instrukcji `{% set pnr = loop.index0 %}`, a używamy w postaci `name="{{ pnr }}"`. Dzięki temu przyporządkujemy przesłane odpowiedzi do kolejnych pytań podczas ich sprawdzania.

Po ponownym uruchomieniu serwera powinniśmy otrzymać następującą stronę internetową:



Oceniamy odpowiedzi

Mechanizm sprawdzana liczby poprawnych odpowiedzi umieścimy w funkcji `index()`. Na początku pliku `quiz.py` dodajemy potrzebne importy:

```
6  from flask import request, redirect, url_for, flash
```

– i uzupełniamy kod funkcji `index()`:

```
30 @app.route('/', methods=['GET', 'POST'])
31 def index():
32
33     if request.method == 'POST':
34         punkty = 0
35         odpowiedzi = request.form
36
37         for pnr, odp in odpowiedzi.items():
38             if odp == DANE[int(pnr)]['odpok']:
39                 punkty += 1
40
41         flash('Liczba poprawnych odpowiedzi, to: {0}'.format(punkty))
42         return redirect(url_for('index'))
43
44     # return 'Cześć, tu Python!'
45     return render_template('index.html', pytania=DANE)
```

- `methods=['GET', 'POST']` – lista zawiera obsługiwane typy żądań, chcemy obsługiwać zarówno żądania *GET* (odesłanie żądanej strony), jak i *POST* (ocena przesłanych odpowiedzi i odesłanie wyniku);
- `if request.method == 'POST':` – instrukcja warunkowa, która wykrywa żądania POST i wykonuje blok kodu zliczający poprawne odpowiedzi;
- `odpowiedzi = request.form` – przesyłane dane z formularza pobieramy z obiektu `request` i zapisujemy w zmiennej odpowiedzi;
- `for pnr, odp in odpowiedzi.items()` – w pętli odczytujemy kolejne pary danych, czyli numer pytania i udzieloną odpowiedź;
- `if odp == DANE[int(pnr)]['odpok']:` – sprawdzamy, czy nadesłana odpowiedź jest zgodna z poprawną, którą wydobywamy z listy pytań.

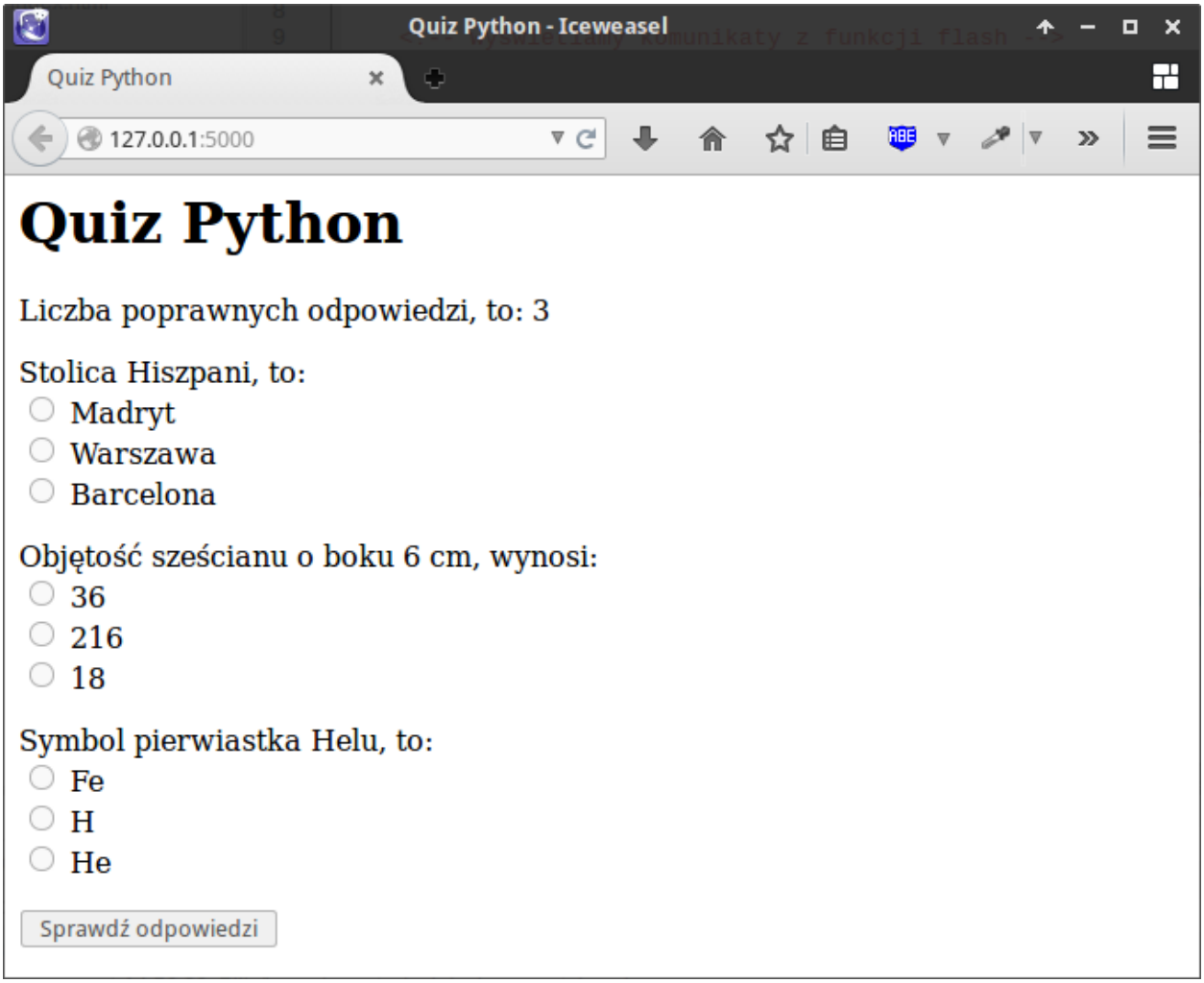
Zwróćmy uwagę, że wartości zmiennej `pnr`, czyli numery pytań liczone od zera, ustaliliśmy wcześniej w szablonie.

Jeżeli nadesłana odpowiedź jest poprawna, doliczamy punkt (`punkty += 1`). Informacje o wyniku przekazujemy użytkownikowi za pomocą funkcji `flash()`, która korzysta z tzw. sesji HTTP (wykorzystującej `SECRET_KEY`), czyli mechanizmu pozwalającego na rozróżnianie żądań przychodzących w tym samym czasie od różnych użytkowników.

W szablonie `index.html` między znacznikami `<h1>` i `<form>` wstawiamy instrukcje wyświetlające wynik:

```
9      <!-- wyświetlamy komunikaty z funkcji flash -->
10     <p>
11       {% for message in get_flashed_messages() %}
12         {{ message }}
13       {% endfor %}
14     </p>
```

Po uruchomieniu aplikacji, zaznaczeniu odpowiedzi i ich przesłaniu otrzymujemy ocenę.


Quiz Python - Iceweasel
Quiz Python
127.0.0.1:5000
Quiz Python
Liczba poprawnych odpowiedzi, to: 3
Stolica Hiszpani, to:
Madryt
Warszawa
Barcelona
Objętość sześcianu o boku 6 cm, wynosi:
36
216
18
Symbol pierwiastka Helu, to:
Fe
H
He
Sprawdź odpowiedzi

Materiały

Źródła:

- `Instrukcja w pdf dla Pythona 2`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

ToDo

Realizacja aplikacji internetowej ToDo (lista zadań do zrobienia) w oparciu o *framework* Flask 0.12.x. Aplikacja umożliwia dodawanie z określoną datą, przeglądanie i oznaczanie jako wykonane różnych zadań, które zapisywane będą w bazie danych SQLite.

Początek pracy jest taki sam, jak w przypadku aplikacji *Quiz*. Wykonujemy dwa pierwsze punkty "Projekt i aplikacja" oraz "Strona główna", tylko katalog aplikacji nazywamy `todo`, a kod zapisujemy w pliku `todo.py`.

Po wykonaniu wszystkich kroków i uruchomieniu serwera testowego powinniśmy w przeglądarce zobaczyć stronę główną:

Model danych i baza

Jako źródło danych aplikacji wykorzystamy tym razem bazę SQLite3 obsługiwaną za pomocą Pythonowego modułu sqlite3.

Model danych: w katalogu aplikacji tworzymy plik `schema.sql`, który zawiera instrukcje języka SQL tworzące tabelę z zadaniami i dodające przykładowe dane.

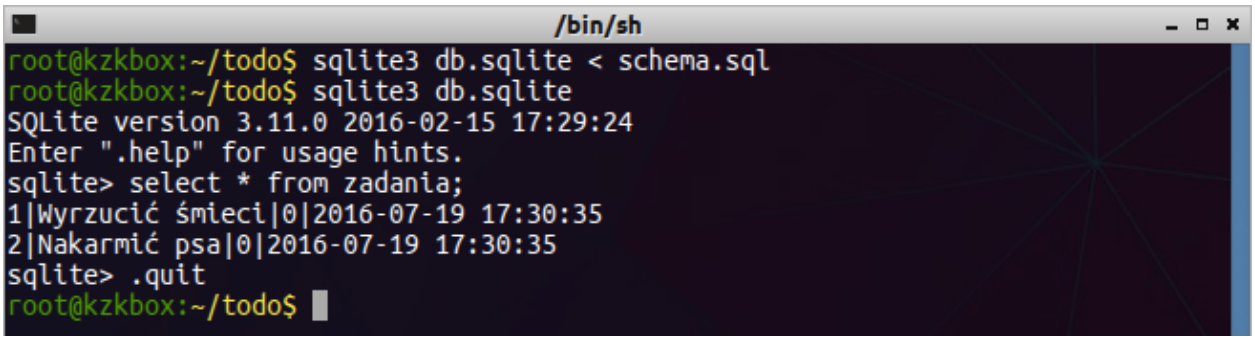
```
-- todo/schema.sql

-- tabela z zadaniami
DROP TABLE IF EXISTS zadania;
CREATE TABLE zadania (
    id integer primary key autoincrement, -- unikalny indentyfikator
    zadanie text not null, -- opis zadania do wykonania
    zrobione boolean not null, -- informacja czy zadania zostalo juz wykonane
    data_pub datetime not null -- data dodania zadania
);

```


```
-- pierwsze dane
INSERT INTO zadania (id, zadanie, zrobione, data_pub)
VALUES (null, 'Wyrzucić śmieci', 0, datetime(current_timestamp));
INSERT into zadania (id, zadanie, zrobione, data_pub)
VALUES (null, 'Nakarmić psa', 0, datetime(current_timestamp));
```

W terminalu wydajemy teraz następujące polecenia:

```
~/todo$ sqlite3 db.sqlite < schema.sql
~/todo$ sqlite3 db.sqlite
sqlite> select * from zadania;
sqlite> .quit
```

Pierwsze polecenie tworzy bazę danych w pliku `db.sqlite`. Drugie otwiera ją w interpreterze. Trzecie to zapytanie SQL, które pobiera wszystkie dane z tabeli *zadania*. Interpreter zamykamy poleceniem `.quit`.



Połączenie z bazą

Bazę danych już mamy, teraz pora napisać funkcje umożliwiające łączenie się z nią z poziomu naszej aplikacji. W pliku `todo.py` dodajemy brakujący kod:

```
# -*- coding: utf-8 -*-
# todo/todo.py

from flask import Flask, g
from flask import render_template
import os
import sqlite3

app = Flask(__name__)

app.config.update(dict(
    SECRET_KEY='bardzosekretnawartosc',
    DATABASE=os.path.join(app.root_path, 'db.sqlite'),
    SITE_NAME='Moje zadania'
))


def get_db():
    """Funkcja tworząca połączenie z bazą danych"""
    if not g.get('db'):  # jeżeli brak połączenia, to je tworzymy
        con = sqlite3.connect(app.config['DATABASE'])
        con.row_factory = sqlite3.Row
        g.db = con  # zapisujemy połączenie w kontekście aplikacji
    return g.db  # zwracamy połączenie z bazą


@app.teardown_appcontext
def close_db(error):
    """Zamykanie połączenia z bazą"""
    if g.get('db'):
        g.db.close()


@app.route('/')
def index():
    # return 'Cześć, tu Python!'
    return render_template('index.html')


if __name__ == '__main__':
    app.run(debug=True)
```

Konfiguracja aplikacji przechowywana jest w obiekcie `config`, który jest podklasą słownika i w naszym przypadku zawiera:

- `SECRET_KEY` – sekretna wartość wykorzystywana do obsługi sesji;
- `DATABSE` – ścieżka do pliku bazy;
- `SITE_NAME` – nazwa aplikacji.

Funkcja `get_db()`:

- `if not g.get('db'):` – sprawdzamy, czy obiekt `g` aplikacji, służący do przechowywania danych kontekstowych, nie zawiera właściwości `db`, czyli połączenia z bazą;

- dalsza część kodu tworzy połączenie w zmiennej `con` i zapisuje w kontekście (obiekcie `g`) aplikacji.

Funkcja `close_db()`:

- `@app.teardown_appcontext` – dekorator, który rejestruje funkcję zamykającą połączenie z bazą do wykonania po zakończeniu obsługi żądania;
- `g.db.close()` – zamknięcie połączenia z bazą.

Lista zadań

Dodajemy widok, czyli funkcję `zadania()` powiązaną z adresem URL `/zadania`:

```
40 @app.route('/zadania')
41 def zadania():
42     db = get_db()
43     kursor = db.execute('SELECT * FROM zadania ORDER BY data_pub DESC;')
44     zadania = kursor.fetchall()
45     return render_template('zadania_lista.html', zadania=zadania)
```

- `db = get_db()` – utworzenie obiektu bazy danych ();
- `db.execute('select...')` – wykonanie podanego zapytania SQL, czyli pobranie wszystkich zadań z bazy;
- `fetchall()` – metoda zwraca pobrane dane w formie listy;

Szablon tworzymy w pliku `todo/templates/zadania_lista.html`:

```
1  <!-- todo/templates/zadania_lista.html -->
2  <html>
3    <head>
4    <!-- nazwa aplikacji pobrana z ustawień -->
5      <title>{{ config.SITE_NAME }}</title>
6    </head>
7    <body>
8      <h1>{{ config.SITE_NAME }}:</h1>
9
10     <ol>
11       <!-- wypisujemy kolejno wszystkie zadania -->
12       {% for zadanie in zadania %}
13         <li>{{ zadanie.zadanie }} - <em>{{ zadanie.data_pub }}</em></li>
14       {% endfor %}
15     </ol>
16
17   </body>
18 </html>
```

- `{% %}` – tagi używane w szablonach do instrukcji sterujących;
- `{{ }}` – tagi używane do wstawiania wartości zmiennych;
- `{{ config.SITE_NAME }}` – w szablonie mamy dostęp do obiektu ustawień `config`;
- `{% for zadanie in zadania %}` – pętla odczytująca zadania z listy przekazanej do szablonu w zmiennej `zadania`;

Odnośniki

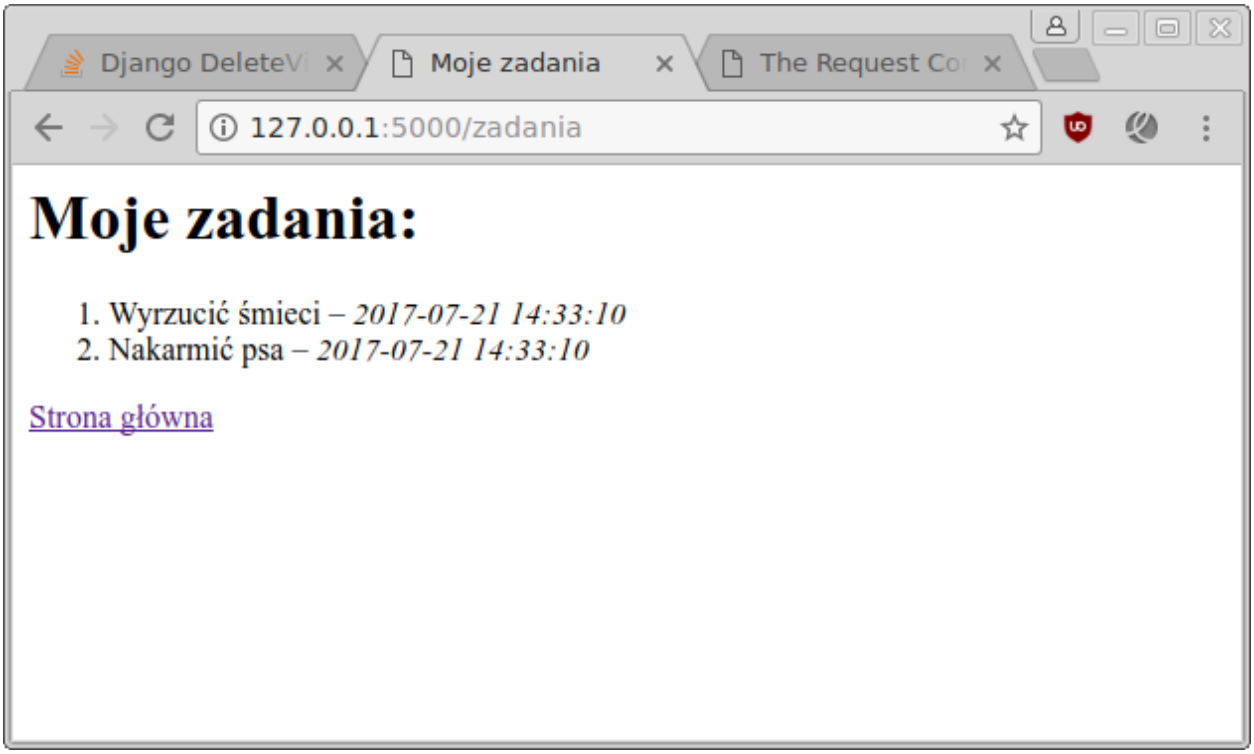
W szablonie `index.html` warto wstawić link do strony z listą zadań, czyli kod:

```
<p><a href="{{ url_for('zadania') }}">Lista zadań</a></p>
```

- `url_for('zadania')` – funkcja dostępna w szablonach, generuje adres powiązany z podaną nazwą funkcji.

Ćwiczenie

Wstaw link do strony głównej w szablonie listy zadań. Po odwiedzeniu strony *127.0.0.1:5000/zadania* powinniśmy zobaczyć listę zadań.



Dodawanie zadań

Po wpisaniu adresu w przeglądarce i naciśnięciu Enter, wysyłamy do serwera żądanie typu *GET*, które obsługujemy zwracając klientowi odpowiednie dane (listę zadań). Dodawanie zadań wymaga przesłania danych z formularza na serwer – są to żądania typu *POST*, które modyfikują dane aplikacji.

Na początku pliku `todo.py` trzeba, jak zwykle, zaimportować wymagane funkcje:

```
8  from datetime import datetime
9  from flask import flash, redirect, url_for, request
```

Następnie rozbudujemy widok listy zadań:

```
43 @app.route('/zadania', methods=['GET', 'POST'])
44 def zadania():
45     error = None
46     if request.method == 'POST':
```

```
        zadanie = request.form['zadanie'].strip()
        if len(zadanie) > 0:
            zrobione = '0'
            data_pub = datetime.now()
            db = get_db()
            db.execute('INSERT INTO zadania VALUES (?, ?, ?, ?);',
                       [None, zadanie, zrobione, data_pub])
            db.commit()
            flash('Dodano nowe zadanie.')
            return redirect(url_for('zadania'))

        error = 'Nie możesz dodać pustego zadania!'  # komunikat o błędzie

    db = get_db()
    kursor = db.execute('SELECT * FROM zadania ORDER BY data_pub DESC;')
    zadania = kursor.fetchall()
    return render_template('zadania_lista.html', zadania=zadania, error=error)
```

- `methods=['GET', 'POST']` – w liście wymieniamy typy obsługiwanych żądań;
- `request.form['zadanie']` – dane przesyłane w żądaniach POST odczytujemy ze słownika `form`;
- `db.execute(...)` – wykonujemy zapytanie, które dodaje nowe zadanie, w miejsce symboli zastępczych `(?, ?, ?, ?)` wstawione zostaną dane z listy podanej jako drugi parametr;
- `flash()` – funkcja pozwala przygotować komunikaty dla użytkownika, które można będzie wstawić w szablonie;
- `redirect(url_for('zadanie'))` – przekierowanie użytkownika na adres związany z podanym widokiem – żądanie typu GET.

Warto zauważyć, że do szablonu przekazujemy dodatkową zmienną `error`.

W szablonie `zadania_lista.html` po znaczniku `<h1>` umieszczamy kod:

```
    <!-- formularz dodawania zadania -->
    <form class="add-form" method="POST" action="{{ url_for('zadania') }}">
      <input name="zadanie" value=""/>
      <button type="submit">Dodaj zadanie</button>
    </form>

    <!-- informacje o sukcesie lub błędzie -->
    <p>
      {% if error %}
        <strong class="error">Błąd: {{ error }}</strong>
      {% endif %}

      {% for message in get_flashed_messages() %}
        <strong class="success">{{ message }}</strong>
      {% endfor %}
    </p>
```

- `{% if error %}` – sprawdzamy, czy zmienna `error` cokolwiek zawiera;
- `{% for message in get_flashed_messages() %}` – pętla odczytująca komunikaty;

Style CSS

O wyglądzie aplikacji decydują arkusze stylów CSS. Umieszczamy je w podkatalogu `static` folderu aplikacji. Tworzymy więc plik `~/todo/static/style.css` z przykładowymi definicjami:

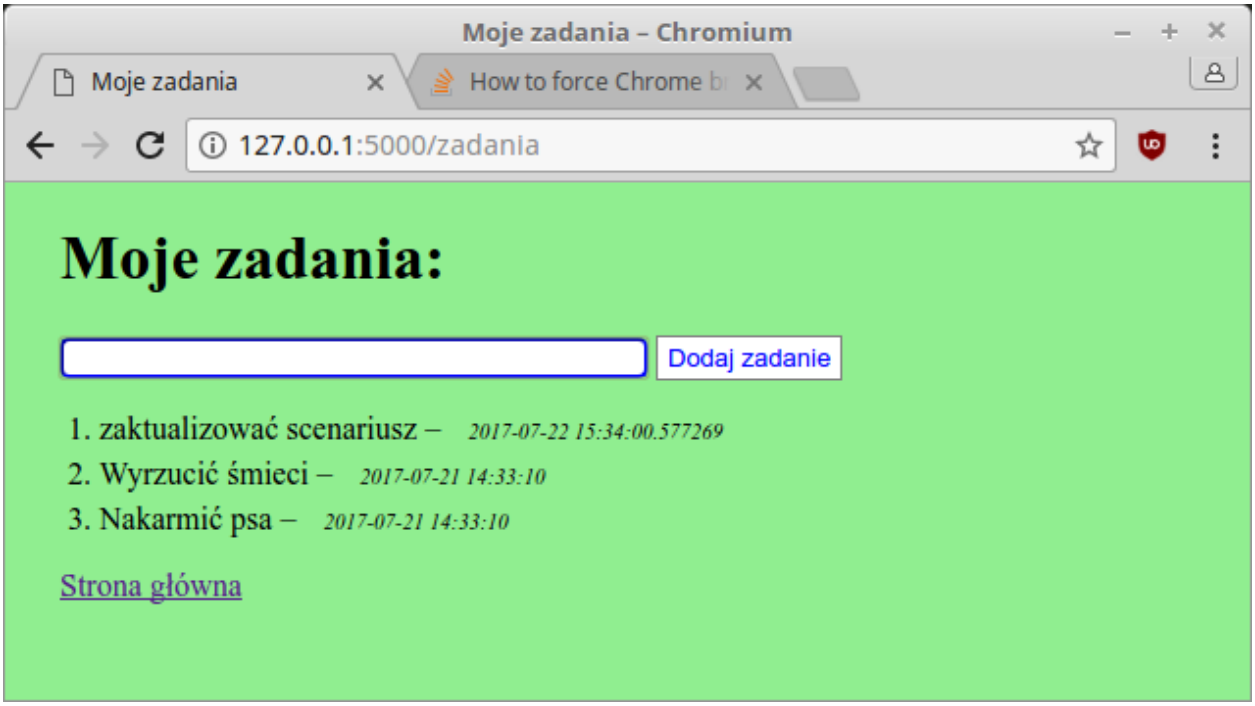
```
/* todo/static/style.css */

body { margin-top: 20px; background-color: lightgreen; }
h1, p { margin-left: 20px; }
.add-form { margin-left: 20px; }
ol { text-align: left; }
em { font-size: 11px; margin-left: 10px; }
form { display: inline-block; margin-bottom: 0;}
input[name="zadanie"] { width: 300px; }
input[name="zadanie"]:focus {
  border-color: blue;
  border-radius: 5px;
}
li { margin-bottom: 5px; }
button {
  padding: 3px 5px;
  cursor: pointer;
  color: blue;
  font-size: 12px/1.5em;
  background: white;
  border: 1px solid grey;
}
.error { color: red; }
.success { color: green; }
.done { text-decoration: line-through; }
```

Arkusz CSS dołączamy do pliku `zadania_lista.html` w sekcji `head`:

```
  <head>
  <!-- nazwa aplikacji pobrana z ustawień -->
    <title>{{ config.SITE_NAME }}</title>
  <!-- dołączamy arkusz CSS -->
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.
↪css') }}">
  </head>
```

Ćwiczenie

Dołącz arkusz stylów CSS również do szablonu `index.html`. Odśwież aplikację w przeglądarce.



Zadania wykonane

Do każdego zadania dodamy formularz, którego wysłanie będzie oznaczało, że wykonaliśmy dane zadanie, czyli zmienimy atrybut `zrobione` wpisu z *0* (niewykonane) na *1* (wykonane). Odpowiednie żądanie typu POST obsłuży nowy widok w pliku `todo.py`, który wstawiamy przed kodem uruchamiającym aplikację (`if __name__ == '__main__':`):

```
@app.route('/zrobione', methods=['POST'])
def zrobione():
    """Zmiana statusu zadania na wykonane."""
    zadanie_id = request.form['id']
    db = get_db()
    db.execute('UPDATE zadania SET zrobione=1 WHERE id=?', [zadanie_id])
    db.commit()
    flash('Zmieniono status zadania.')
    return redirect(url_for('zadania'))
```

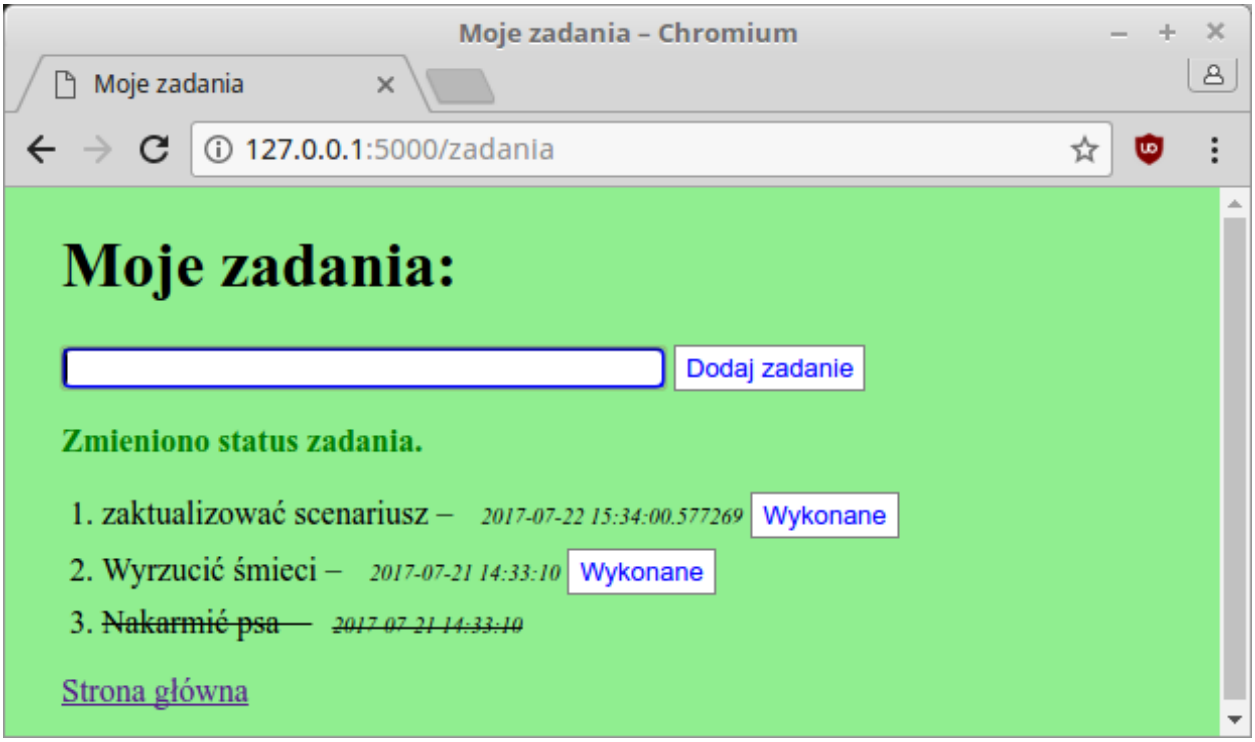
- `zadanie_id = request.form['id']` – odczytujemy przesłany identyfikator zadania;

- `db.execute('UPDATE zadania SET zrobione=1 WHERE id=?', [zadanie_id])` – wykonujemy zapytanie aktualizujące staus zadania.

W szablonie `zadania_lista.html` modyfikujemy fragment wyświetlający listę zadań i dodajemy formularz:

```
    <ol>
      <!-- wypisujemy kolejno wszystkie zdania -->
      {% for zadanie in zadania %}
        <li>
          <!-- wyróżnienie zadań zakończonych -->
          {% if zadanie.zrobione %}
            <span class="done">{{ zadanie.zadanie }} - <em>{{ zadanie.data_pub }}</em>
↪</span>
          {% else %}
            {{ zadanie.zadanie }} - <em>{{ zadanie.data_pub }}</em>
          {% endif %}

          <!-- formularz zmiany statusu zadania -->
          {% if not zadanie.zrobione %}
            <form method="POST" action="{{ url_for('zrobione') }}">
              <!-- wysyłamy jedynie informacje o id zadania -->
              <input type="hidden" name="id" value="{{ zadanie.id }}"/>
              <button type="submit">Wykonane</button>
            </form>
          {% endif %}
        </li>
      {% endfor %}
    </ol>
```

Możemy dodawać zadania oraz zmieniać ich status.



Zadania dodatkowe

- Dodaj możliwość usuwania zadań.
- Dodaj mechanizm logowania użytkownika tak, aby użytkownik mógł dodawać i edytować tylko swoją listę zadań.

Materiały

Źródła:

- `todo.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Quiz ORM

Realizacja aplikacji internetowej Quiz w oparciu o *framework* Flask 0.12.x i bazę danych SQLite zarządzaną systemem ORM Peewee lub SQLAlchemy.

Zalecamy zapoznanie się z materiałami zawartymi w scenariuszach:

- *Podstawy Pythona*,
- *Bazy danych w Pythonie*,
- *Quiz*,
- *ToDo*.

Wykorzystywane biblioteki instalujemy przy użyciu instalatora `pip`:

```
~$ sudo pip install peewee flask-wtf
```

Informacja: W budowanym poniżej kodzie wykorzystamy ORM Peewee, na końcu omówimy różnice w przypadlku użycia SQLAlchemy.

- *Modularyzacja*
- *Szablon podstawowy*
- *Baza danych*
- *Modele*
- *Dane początkowe*
- *Odczyt*
- *Quiz*
- *Dodawanie*
- *Edycja*
- *Usuwanie*

- *SQLAlchemy*
- *Źródła*

Modularyzacja

Scenariusze *Quiz* i *ToDo* pokazują możliwość umieszczenia całego kodu aplikacji obsługiwanej przez Flaska w jednym pliku. Dla celów szkoleniowych to dobre rozwiązanie, ale w bardziej rozbudowanych projektach wygodniej umieścić poszczególne części aplikacji w osobnych plikach.

Kod rozmieścimy więc następująco:

- `app.py` – konfiguracja aplikacji Flaska i połączeń z bazą,
- `models.py` – klasy opisujące tabele, pola i relacje w bazie,
- `views.py` – widoki, czyli funkcje, powiązane z adresami URL, obsługujące żądania użytkownika,
- `forms.py` – definicje formularza wykorzystywanego w aplikacji,
- `main.py` – główny plik naszej aplikacji wiążący wszystkie powyższe, odpowiada za utworzenie początkowej bazy,
- `dane.py` – moduł opcjonalny, odczytanie przykładowych danych z pliku `pytania.csv` i dodanie ich do bazy.

Wszystkie pliki muszą znajdować się w katalogu aplikacji `quiz-orm`, który zawierać będzie również podkatalogi:

- `templates` – tu umieścimy szablony html,
- `static` – to miejsce dla arkuszy stylów, obrazki i/lub skryptów *js*.

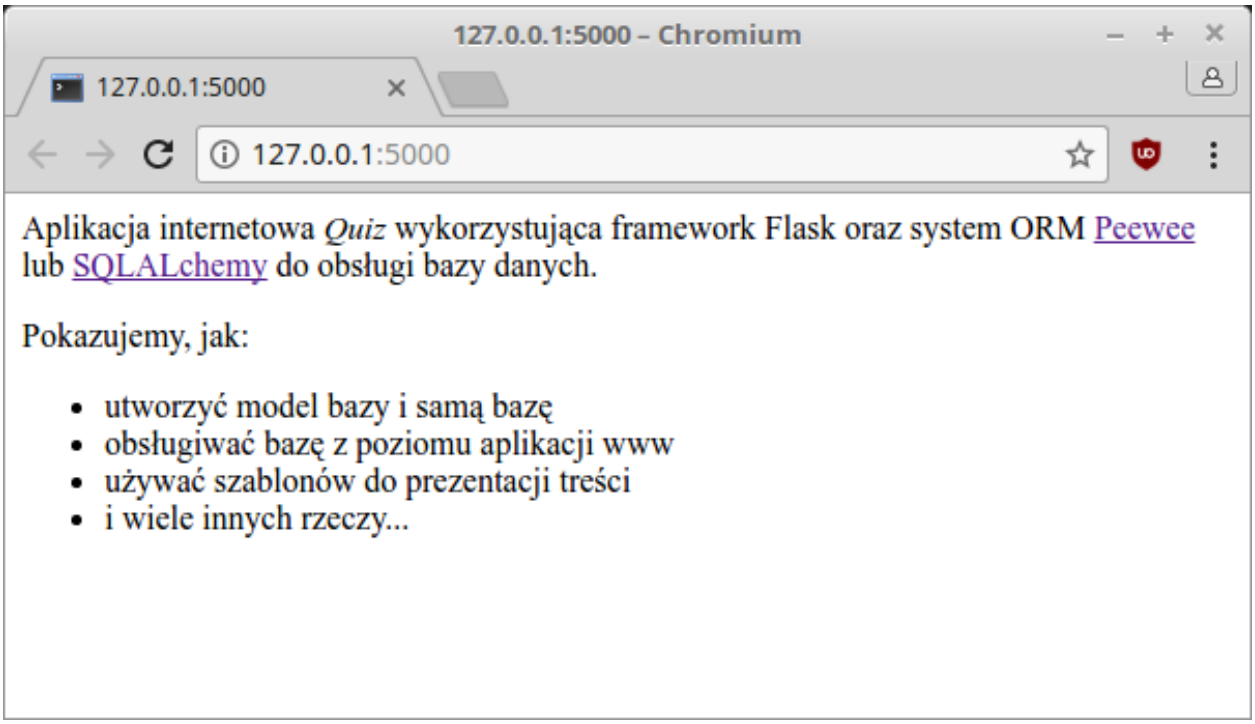
Ściągamy przygotowane przez nas archiwum `quiz-orm_skel.zip` i rozpakowujemy w wybranym katalogu. Początkowy kod pozwoli uruchomić aplikację i wyświetlić zawartość strony głównej. Aplikację uruchamiamy wydając w katalogu `quiz-orm` polecenie:

```
~/quiz-orm$ python3 main.py
```

Szablon podstawowy

W omówionych do tej pory, wspomnianych wyżej, scenariuszach aplikacji internetowych każdy szablon zawierał kompletny kod strony. W praktyce jednak duża część kodu HTML powtarza się na każdej stronie w ramach danego serwisu. Tę wspólną część kodu umieścimy w szablonie podstawowym `templates/szkielet.html`:

```
<!doctype html>
<!-- quiz-orm/templates/szkielet.html -->
<html>
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>{% block tytul %}{% endblock %} &#8211; {{ config.TYTUL }}</title>
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet"
     href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css"
     integrity="sha384-BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/
↪K68vbdEjh4u"
     crossorigin="anonymous">
```




```
14      <!-- Optional theme -->
15      <link rel="stylesheet"
16       href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap-theme.min.css
↪"
17       integrity="sha384-rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQIXwl/
↪Sp"
18       crossorigin="anonymous">
19      <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.
↪css') }}">
20    </head>
21    <body>
22      <div class="container">
23        <!-- Static navbar -->
24        <nav class="navbar navbar-default">
25          <div class="container-fluid">
26            <div class="navbar-header">
27              <button type="button" class="navbar-toggle collapsed" data-toggle=
↪"collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
28                <span class="sr-only">Przełącz nawigację</span>
29                <span class="icon-bar"></span>
30                <span class="icon-bar"></span>
31                <span class="icon-bar"></span>
32              </button>
33              <a class="navbar-brand" href="http://flask.pocoo.org/">
34                <img src="{{ url_for('static', filename='flask.png') }}" style="max-
↪width: 100%; max-height: 100%;">
35              </a>
36            </div>
37
38  {% set navigation_bar = [
39    ('/', 'index', 'Strona główna'),
40    ('/lista', 'lista', 'Lista pytań'),
```

```
  ('/quiz', 'quiz', 'Quiz'),
  ('/dodaj', 'dodaj', 'Dodaj pytania'),
] %}
{% set active_page = active_page|default('index') %}

          <div id="navbar" class="navbar-collapse collapse">
            <ul class="nav navbar-nav">

            {% for href, id, tekst in navigation_bar %}
              <li{% if id == active_page %} class="active"{% endif %}>
                <a href="{{ href|e }}">{{ tekst|e }}</a>
              </li>
            {% endfor %}

            </ul>
          </div><!--/.nav-collapse -->
        </div><!--/.container-fluid -->
      </nav>

      <div class="row">
        <div class="col-md-12">
          <h1>{% block h1 %}{% endblock %}</h1>

          {% with komunikaty = get_flashed_messages(with_categories=true) %}
          {% if komunikaty %}
          <div id="komunikaty" class="well">
            {% for kategoria, komunikat in komunikaty %}
              <span class="{{ kategoria }}">{{ komunikat }}</span><br>
            {% endfor %}
          </div>
          {% endif %}
          {% endwith %}

          <div id="tresc" class="cb">
          {% block tresc %}
          {% endblock %}
          </div>

        </div>
      </div> <!-- /row -->
    </div> <!-- /container -->

    <!-- jQuery CDN -->
    <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
     integrity="sha256-k2WSCIexGzOj3Euiig+TlR8gA0EmPjuc79OEeY5L45g="
     crossorigin="anonymous"></script>
    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"
    integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa
↪"
    crossorigin="anonymous"></script>

  </body>
</html>
```

Szablon oparty jest na frameworku Bootstrap. Odpowiednie linki do stylów CSS, pobieranych z systemu CDN zostały skopiowane ze strony Getting started i wklejone w podświetlonych liniach. Do szablonu dołączono również wymaganą przez Bootstrapa bibliotekę jQuery.

- `{{ url_for('static', filename='style.css') }}` – funkcja `url_for()` pozwala wygenerować ścieżkę do zasobów umieszczonych w podkatalogu `static`;
- `{% tag %}...{% endtag %}` – tagi sterujące, wymagają zamknięcia(!),
- `{% block nazwa_bloku %}` – tag pozwala definiować miejsca, w których szablony dziedziczące mogą wstawiać swój kod,
- `{{ zmienna }}` – tagi pozwalające wstawiać wartości zmiennych dostępnych domyślnie i przekazanych do szablonu,
- `container`, `row`, `navbar` itd. – klasy Bootstrapa tworzące podstawowy układ (ang. *layout*) strony,
- `navigation_bar` – lista na podstawie której generowane są pozycje menu,
- `active_page` – zmienna zawierająca identyfikator aktywnej strony,
- `get_flashed_messages(with_categories=true)` – funkcja zwracająca komunikaty dla użytkownika oznaczone kategoriami, wykorzystywanymi jako klasy CSS.

Dodatkowo szablon wykorzystuje zawarty w początkowym archiwum plik `static/style.css`.

Szablon strony głównej z pliku `index.html` zmieniamy następująco:

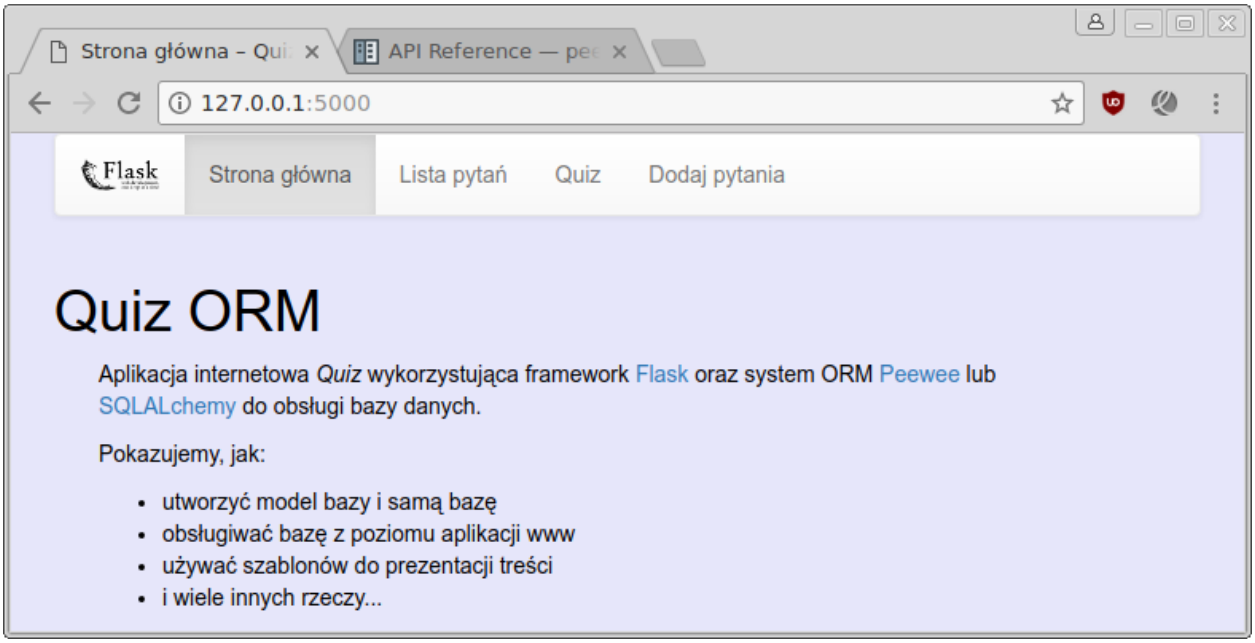
```
<!-- quiz-orm/templates/index.html -->
{% extends "szkielet.html" %}
{% set active_page = "index" %}
{% block tytul %}Strona główna{% endblock%}
{% block h1 %}Quiz ORM{% endblock%}
{% block tresc %}
  <p>
    Aplikacja internetowa <i>Quiz</i> wykorzystująca framework
    <a href="http://flask.pocoo.org/">Flask</a>
    oraz system ORM <a href="http://docs.peewee-orm.com/en/latest/">Peewee</a>
    lub <a href="https://www.sqlalchemy.org/">SQLALchemy</a>
    do obsługi bazy danych.</p>
  <p>
    Pokazujemy, jak:
    <ul>
      <li>utworzyć model bazy i samą bazę</li>
      <li>obsługiwać bazę z poziomu aplikacji www</li>
      <li>używać szablonów do prezentacji treści</li>
      <li>i wiele innych rzeczy...</li>
    </ul>
  </p>
{% endblock %}
```

- `{% extends szkielet.html"%}` – wskazanie dziedziczenia z szablonu podstawowego;
- `{% block tresc %} treść {% endblock %}` – zastąpienie lub uzupełnienie treści bloków zdefiniowanych w szablonie podstawowym.

Po odświeżeniu strony powinniśmy zobaczyć w przeglądarce nowy wygląd strony:

Baza danych

Konfigurację bazy danych obsługiwanej przez wybrany system ORM umieścimy w pliku `app.py`. Zaczynamy od uzupełnienia ustawień w słowniku `config` i utworzenia obiektu bazy danych:




```
# -*- coding: utf-8 -*-
# quiz-orm/app.py

import os
from flask import Flask, g
from peewee import SqliteDatabase

app = Flask(__name__)

# konfiguracja aplikacji
app.config.update(dict(
    SECRET_KEY='bardzosekretnawartosc',
    TYTUL='Quiz ORM Peewee',
    DATABASE=os.path.join(app.root_path, 'quiz.db'),
))

# tworzymy instancję bazy używanej przez modele
baza = SqliteDatabase(app.config['DATABASE'])


@app.before_request
def before_request():
    g.db = baza
    g.db.get_conn()


@app.after_request
def after_request(response):
    g.db.close()
    return response
```

- `before_request()`, `after_request()` – funkcje wykorzystywane do otwierania i zamykania połączenia z bazą SQLite przed żądaniem i po żądaniu (ang. *request*),
- `g` – specjalny obiekt Flaska do przechowywania danych kontekstowych aplikacji.

Modele

Modele pozwalają opisać strukturę naszej bazy danych w postaci definicji klas i ich właściwości. Na podstawie tych definicji system ORM utworzy odpowiednie tabele i kolumny. Wykorzystamy tabelę `Pytanie`, zawierającą treść pytania i poprawną odpowiedź, oraz tabelę `Odpowiedź`, która przechowywać będzie wszystkie możliwe odpowiedzi. Relację *jeden-do-wielu* między tabelami tworzyć będzie pole `pnr`, czyli klucz obcy, przechowujący identyfikator pytania.

```
# -*- coding: utf-8 -*-
# quiz-orm/models.py

from peewee import *
from app import baza


class BaseModel(Model):
    class Meta:
        database = baza


class Pytanie(BaseModel):
    pytanie = CharField(unique=True)
    odpok = CharField()

    def __str__(self):
        return self.pytanie


class Odpowiedz(BaseModel):
    pnr = ForeignKeyField(
        Pytanie, related_name='odpowiedzi', on_delete='CASCADE')
    odpowiedz = CharField()

    def __str__(self):
        return self.odpowiedz
```

- `BaseModel` – klasa określająca obiekt bazy,
- `unique=True` – właściwość wymagająca niepowtarzalnej zawartości pola,
- `ForeignKeyField()` – definicja klucza obcego, tworzenie relacji,
- `on_delete = 'CASCADE'` – usuwanie rekordów z powiązanych tabel.

Identyfikatory pytań i odpowiedzi, czyli pola `id` w każdej tabeli tworzone są automatycznie.

Metody `__str__(self)` służą "autoprezentacji" obiektów utworzonych na podstawie danego modelu, są wykorzystywane np. podczas używania funkcji `print()`.

Dane początkowe

Moduł `dane.py`:

```
# -*- coding: utf-8 -*-
# quiz-orm/dane.py

import os
import csv
```

```
from models import Pytanie, Odpowiedz


def pobierz_dane(plikcsv):
    """Funkcja zwraca tuplę zawierającą tuple z danymi z pliku csv."""
    dane = []
    if os.path.isfile(plikcsv):
        with open(plikcsv, newline='') as plikcsv:
            tresc = csv.reader(plikcsv, delimiter='#')
            for rekord in tresc:
                dane.append(tuple(rekord))
    else:
        print("Plik z danymi", plikcsv, "nie istnieje!")

    return tuple(dane)


def dodaj_pytania(dane):
    """Funkcja dodaje pytania i odpowiedzi przekazane w tupli do bazy."""
    for pytanie, odpowiedzi, odpok in dane:
        p = Pytanie(pytanie=pytanie, odpok=odpok)
        p.save()
        for o in odpowiedzi.split(","):
            odp = Odpowiedz(pnr=p.id, odpowiedz=o.strip())
            odp.save()
    print("Dodano przykładowe pytania")
```

- `pobierz_dane()` – funkcja wykorzystuje moduł `csv`, który ułatwia odczytywanie danych zapisanych w tym formacie, zobacz format CSV, zwraca tuplę 3-elementowych tupli (:-));
- `dodaj_pytania()` – funkcja dodaje przykładowe pytania i odpowiedzi wykorzystując składnię wykorzystywanego systemu ORM;
- `for pytanie,odpowiedzi,odpok in dane:` – pętla rozpakowuje pytanie, listę odpowiedzi i odpowiedź poprawną z przekazanych tupli;
- `p = Pytanie(pytanie=pytanie, odpok=odpok)` – utworzenie obiektu pytania;
- `odp = Odpowiedz(pnr=p.id, odpowiedz=o.strip())` – utworzenie obiektu odpowiedzi;
- `save()` – metoda zapisująca utworzony/zmieniony obiekt w bazie danych.

Zawartość dołączonego do archiwum pliku `pytania.csv`:

```
Stolica Hiszpani, to:#Madryt, Warszawa, Barcelona#Madryt
Objętość sześcianu o boku 6 cm, wynosi:#36, 216, 18#216
Symbol pierwiastka Helu, to:#Fe, H, He#He
```

Kod uruchamiający utworzenie bazy i dodanie do niej przykładowych danych umieścimy w pliku `main.py`:

```
import os
from app import app, baza
from models import *
from views import *
from dane import *

if __name__ == '__main__':
    if not os.path.exists(app.config['DATABASE']):
        baza.create_tables([Pytanie, Odpowiedz], True)  # tworzymy tabele
```

```
        dodaj_pytania(pobierz_dane('pytania.csv'))
    app.run(debug=True)
```

[todo]

Odczyt

Skrót *CRUD* (*Create* (tworzenie), *Read* (odczyt), *Update* (aktualizacja), *Delete* (usuwanie)) oznacza podstawowe operacje wykonywane na bazie danych.

Zaczniemy od widoku `lista()` pobierającego wszystkie pytania i zwracającego szablon z ich listą:

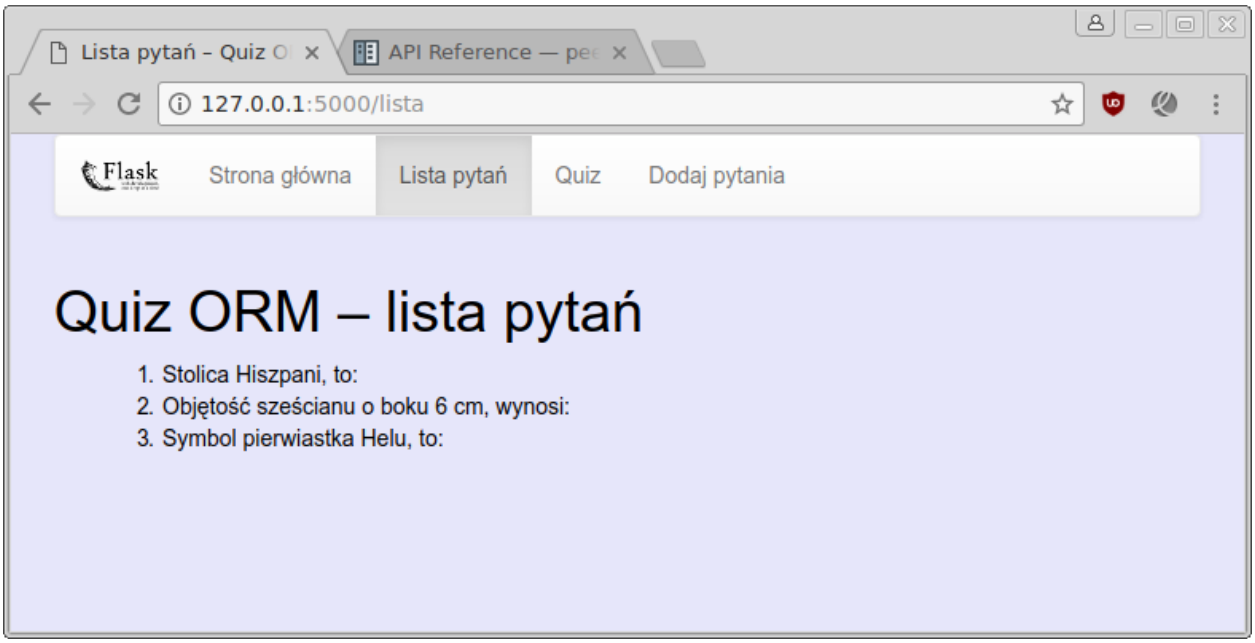
```
from flask import render_template, request, redirect, url_for, abort, flash
from app import app
from models import Pytanie, Odpowiedz
from forms import *


@app.route('/')
def index():
    """Strona główna"""
    return render_template('index.html')


@app.route('/lista')
def lista():
    """Pobranie wszystkich pytań z bazy i zwrócenie szablonu z listą pytań"""
    pytania = Pytanie().select().annotate(Odpowiedz)
    if not pytania.count():
        flash('Brak pytań w bazie.', 'kom')
        return redirect(url_for('index'))

    return render_template('lista.html', pytania=pytania)
```

- `pytania = Pytanie().select()` – pobranie z bazy wszystkich pytań.
- `redirect(url_for('index'))` – przekierowanie użytkownika na adres obsługiwany przez podany jako argument widok.

Kod szablonu `lista.html`:

```
<!-- quiz-orm/templates/lista.html -->
{% extends "szkielet.html" %}
{% set active_page = "lista" %}
{% block tytul %}Lista pytań{% endblock%}
{% block h1 %}Quiz ORM &#8211; lista pytań{% endblock%}
{% block tresc %}
  <ol>
  <!-- pętla odczytująca kolejne pytania z listy -->
  {% for p in pytania %}
    <li>
      <!-- wypisujemy pytanie -->
      {{ p.pytanie }}
    </li>
  {% endfor %}
  </ol>
{% endblock %}
```

Po uzupełnieniu kodu w przeglądarce powinniśmy zobaczyć listę pytań:



Quiz

Widok wyświetlający pytania i odpowiedzi w formie quizu i sprawdzający udzielone przez użytkownika odpowiedzi to również przykład operacji odczytu danych danych z bazy. Dodajemy funkcję `quiz()`:

```
@app.route('/quiz', methods=['GET', 'POST'])
def quiz():
    """Wyświetlenie pytań i odpowiedzi w formie quizu oraz ocena poprawności
    przesłanych odpowiedzi"""
    if request.method == 'POST':
        wynik = 0
        for pid, odp in request.form.items():
            odpok = Pytanie.select(Pytanie.odpok).where(
                Pytanie.id == int(pid)).scalar()
            if odp == odpok:
                wynik += 1

        flash('Liczba poprawnych odpowiedzi, to: {0}'.format(wynik), 'sukces')
        return redirect(url_for('index'))

    # GET, wyświetl pytania
    pytania = Pytanie().select().annotate(Odpowiedz)
    if not pytania.count():
        flash('Brak pytań w bazie.', 'kom')
        return redirect(url_for('index'))

    return render_template('quiz.html', pytania=pytania)
```

- `@app.route('/quiz', methods=['GET', 'POST'])` – określenie obsługiwanego adresu URL oraz akcpetowanych metod żądań,
- `request.method` – wykorzystana metoda: GET lub POST,

- `request.form` – formularz przesłany w żądaniu POST,
- `for pid, odp in request.form.items():` – pętla odczytująca przesłane identyfikatory pytań i udzielone odpowiedzi.

Zapytania ORM:

- `Pytanie().select().annotate(Odpowiedz)` – pobranie wszystkich pytań razem z odpowiedziami,
- `Pytanie.select(Pytanie.odpok).where(Pytanie.id == int(pid)).scalar()` – pobranie poprawnej odpowiedzi dla pytania o podanym identyfikatorze, metoda `scalar()` zwraca pojedynczą wartość.

Szablon `quiz.html` – oparty na omówionym wcześniej wzorcu – wyświetla pytania i możliwe odpowiedzi jako pola opcji typu radio button:

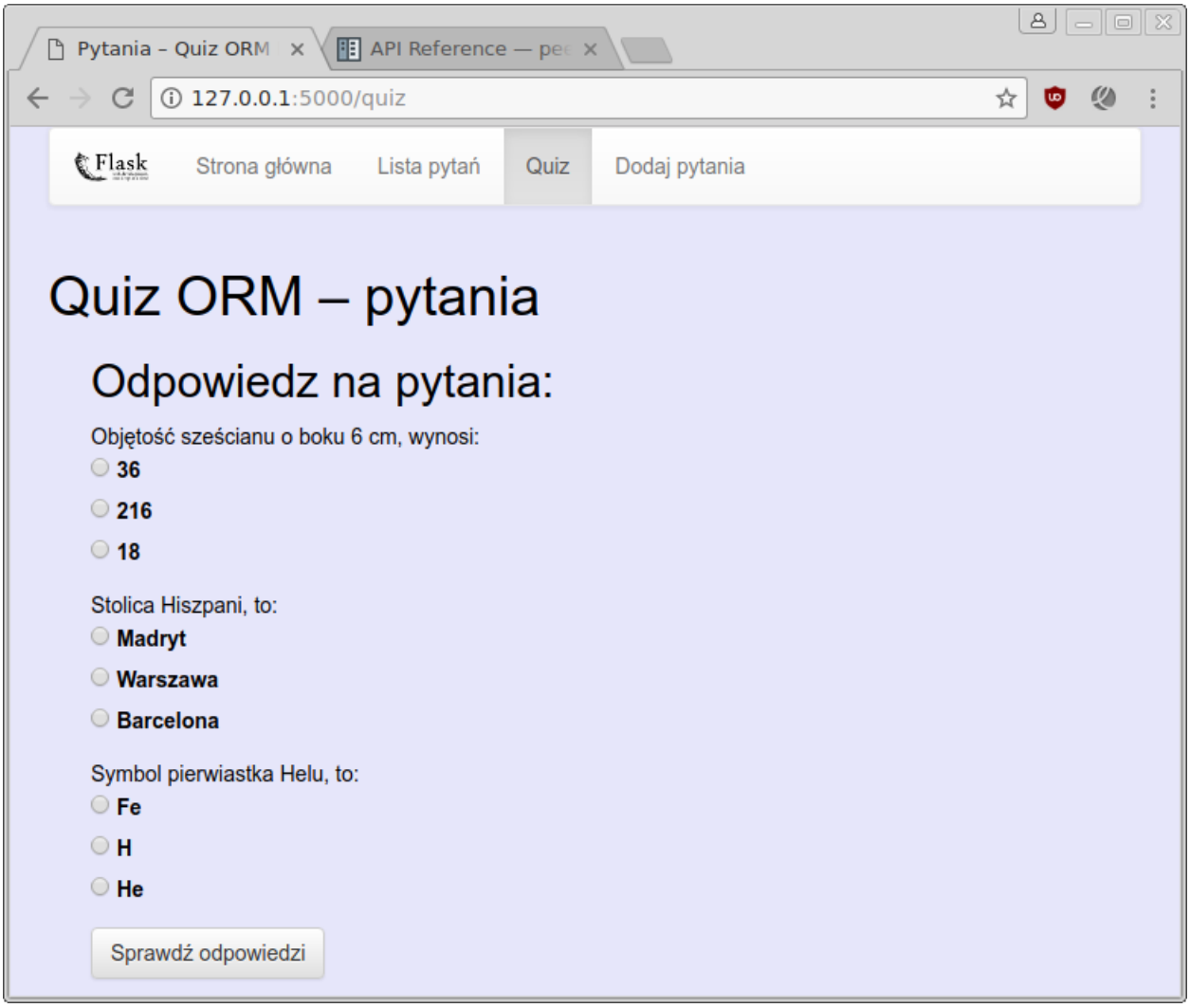
```
<!-- quiz-orm/templates/quiz.html -->
{% extends "szkielet.html" %}
{% set active_page = "quiz" %}
{% block tytul %}Pytania{% endblock%}
{% block h1 %}Quiz ORM &#8211; pytania{% endblock%}
{% block tresc %}
  <h2>Odpowiedz na pytania:</h2>
  <!-- formularz z quizem -->
  <form method="POST">
    <!-- pętla odczytująca kolejne pytania z listy -->
    {% for p in pytania %}
      <p>
        <!-- wypisujemy pytanie -->
        {{ p.pytanie }}
        <br>
        <!-- pętla odczytująca możliwe odpowiedzi dla danego pytania -->
        {% for o in p.odpowiedzi %}
          <label>
            <!-- pole radio button aby można było zaznaczyć odpowiedź -->
            <input type="radio" value="{{ o.odpowiedz }}" name="{{ p.id }}">
            {{ o.odpowiedz }}
          </label>
          <br>
        {% endfor %}
      </p>
    {% endfor %}

    <!-- przycisk wysyłający wypełniony formularz -->
    <button type="submit" class="btn btn-default">Sprawdź odpowiedzi</button>
  </form>
{% endblock %}
```

Dodawanie

Dodawanie nowych pytań i odpowiedzi wymaga formularza. Gdybyśmy stworzyli go "ręcznie" w szablonie html, musielibyśmy napisać sporo kodu sprawdzającego poprawność przesyłanych danych. Dlatego skorzystamy z biblioteki Flask-wtf, pozwalającej wykorzystać formularze WTForms.

Formularz definiujemy w pliku `forms.py`:

```
# -*- coding: utf-8 -*-
# quiz-orm/forms.py
```


Pytania - Quiz ORM
API Reference — pe
127.0.0.1:5000/quiz
Flask
Strona główna
Lista pytań
Quiz
Dodaj pytania
Quiz ORM – pytania
Odpowiedz na pytania:
Objętość sześcianu o boku 6 cm, wynosi:
36
216
18
Stolica Hiszpani, to:
Madryt
Warszawa
Barcelona
Symbol pierwiastka Helu, to:
Fe
H
He
Sprawdź odpowiedzi

```
3
4  from flask_wtf import FlaskForm
5  from wtforms import StringField, RadioField, HiddenField, FieldList
6  from wtforms.validators import Required
7
8  blad1 = 'To pole jest wymagane'
9  blad2 = 'Brak zaznaczonej poprawnej odpowiedzi'
10
11
12 class DodajForm(FlaskForm):
13     pytanie = StringField('Treść pytania:',
14                           validators=[Required(message=blad1)])
15     odpowiedzi = FieldList(StringField(
16                            'Odpowiedź',
17                            validators=[Required(message=blad1)]),
18                            min_entries=3,
19                            max_entries=3)
20     odpok = RadioField(
21         'Poprawna odpowiedź',
22         validators=[Required(message=blad2)],
23         choices=[('0', 'o0'), ('1', 'o1'), ('2', 'o2')]
24     )
25     pid = HiddenField("Pytanie id")
```

- `StringField()` – definicja pola tekstowego,
- `FieldList(StringField())` – definicja trzech pól tekstowych,
- `Required(message=blad1)` – pole wymagane,
- `RadioField()` – pola jednokrotnego wyboru, opcje definiuje się w postaci listy `choices` zawierającej pary wartość - etykieta,
- `HiddenField()` – pole ukryte.

Funkcja pomocnicza i widok obsługujący dodawanie:

```
51 def flash_errors(form):
52     """Odczytanie wszystkich błędów formularza i przygotowanie komunikatów"""
53     for field, errors in form.errors.items():
54         for error in errors:
55             if type(error) is list:
56                 error = error[0]
57             flash("Błąd: {}. Pole: {}".format(
58                 error,
59                 getattr(form, field).label.text))
60
61
62 @app.route('/dodaj', methods=['GET', 'POST'])
63 def dodaj():
64     """Dodawanie pytań i odpowiedzi"""
65     form = DodajForm()
66     if form.validate_on_submit():
67         odp = form.odpowiedzi.data
68         p = Pytanie(pytanie=form.pytanie.data, odpok=odp[int(form.odpok.data)])
69         p.save()
70         for o in odp:
71             inst = Odpowiedz(pnr=p.id, odpowiedz=o)
72             inst.save()
73         flash("Dodano pytanie: {}".format(form.pytanie.data))
```

```
74          return redirect(url_for("lista"))
75      elif request.method == 'POST':
76          flash_errors(form)
77
78      return render_template("dodaj.html", form=form, radio=list(form.odpok))
```

- `flash_errors()` – zadaniem funkcji jest przygotowanie komunikatów dla użytkownika zawierających ewentualne błędy walidacji formularza dostępne w słowniku `form.errors`,
- `form = DodajForm()` – utworzenie pustego formularza,
- `form.validate_on_submit()` – funkcja zwraca prawdę, jeżeli żądanie jest typu POST i formularz zawiera poprawne dane, czyli przechodzi procedurę walidacji, funkcja automatycznie wypełnia obiekt formularza przesłanymi danymi,
- `form.pole.data` – odczyt wartości danego pola formularza,
- `odpok=odp[int(form.odpok.data)]` – jako poprawną odpowiedź zapisujemy tekst odpowiedzi.

Do szablonu przekazujemy formularz i osobno listę opcji odpowiedzi. Kod szablonu `dodaj.html`:

```
1  <!-- quiz-orm/templates/dodaj.html -->
2  {% extends "szkielet.html" %}
3  {% set active_page = "dodaj" %}
4  {% block tytul %}Dodawanie{% endblock%}
5  {% block h1 %}Quiz ORM &#8211; dodawanie pytań{% endblock%}
6  {% block tresc %}
7    {% include "pytanie_form.html" %}
8  {% endblock %}
```

- `{% include "pytanie_form.html"%}` – instrukcja włączania kodu z innego pliku.

Kod renderujący formularz jest taki sam podczas dodawania, jak i edycji danych. Dlatego umieścimy go w osobnym pliku:

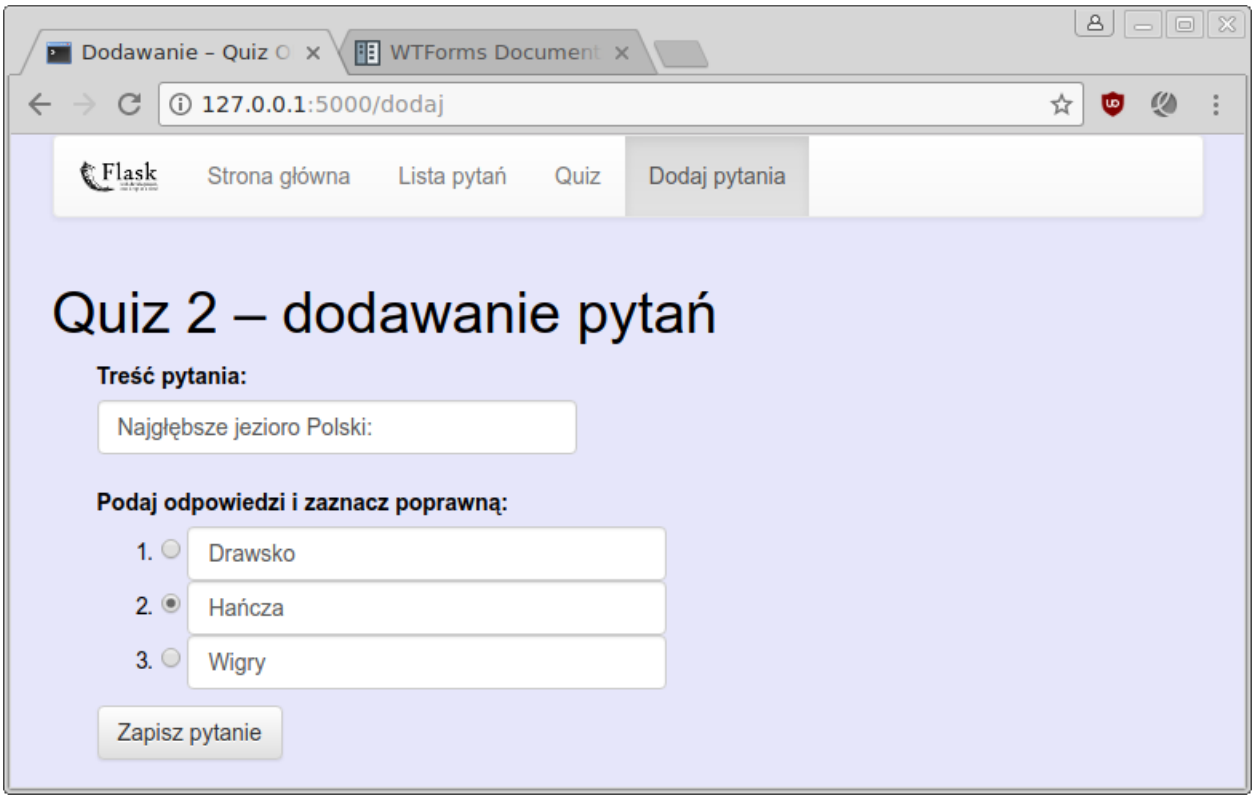
```
1   <form method="POST" class="form-inline" action="">
2     {{ form.csrf_token }}
3
4     {{ form.pytanie.label }}<br>
5     {{ form.pytanie(class="form-control") }}<br>
6     <br>
7     <label>Podaj odpowiedzi i zaznacz poprawną:</label><br>
8     <ol>
9     {% for o in form.odpowiedzi %}
10      <li>{{ radio[loop.index0] }} {{ o(class="form-control") }}</li>
11    {% endfor %}
12    </ol>
13
14    <button type="submit" class="btn btn-default">Zapisz pytanie</button>
15  </form>
```

Formularz renderujemy "ręcznie", aby uzyskać odpowiedni układ pól. Po nazwie pola można opcjonalnie podawać klasy CSS, które mają zostać użyte w kodzie HTML, np. `form.pytanie(class="form-control")`.

Efekt prezentuje się następująco:

Edycja

Zaczniemy od dodania w pliku `views.py` funkcji pomocniczych i widoku `edytuj()`:




```
def get_or_404(pid):
    """Pobranie i zwrócenie obiektu z bazy lub wywołanie szablonu 404.html"""
    try:
        p = Pytanie.select().annotate(Odpowiedz).where(Pytanie.id == pid).get()
        return p
    except Pytanie.DoesNotExist:
        abort(404)


@app.errorhandler(404)
def page_not_found(e):
    """Zwrócenie szablonu 404.html w przypadku nie odnalezienia strony"""
    return render_template('404.html'), 404


@app.route('/edytuj/<int:pid>', methods=['GET', 'POST'])
def edytuj(pid):
    """Edycja pytania o identyfikatorze pid i odpowiedzi"""
    p = get_or_404(pid)
    form = DodajForm()

    if form.validate_on_submit():
        odp = form.odpowiedzi.data
        p.pytanie = form.pytanie.data
        p.odpok = odp[int(form.odpok.data)]
        p.save()
        for i, o in enumerate(p.odpowiedzi):
            o.odpowiedz = odp[i]
            o.save()
```

```
        flash("Zaktualizowano pytanie: {}".format(form.pytanie.data))
        return redirect(url_for("lista"))
    elif request.method == 'POST':
        flash_errors(form)

    for i in range(3):
        if p.odpok == p.odpowiedzi[i].odpowiedz:
            p.odpok = i
            break
    form = DodajForm(obj=p)
    return render_template("edytuj.html", form=form, radio=list(form.odpok))
```

Żądanie wyświetlenia aktualizowanego pytania (GET):

- `'/edytuj/<int:pid>'` – definicja adresu URL mówiąca, że oczekujemy wywołań w postaci */edytuj/1*, przy czym końcowa liczba to identyfikator pytania,
- `p = get_or_404(pid)` – próbujemy pobrać z bazy dane pytania o podanym identyfikatorze, funkcja pomocnicza `get_or_404()` zwróci obiekt, a jeżeli nie będzie to możliwe, wywoła błąd `abort(404)` – co oznacza, że żądanego zasobu nie odnaleziono,
- `page_not_found(e)` – funkcja, którą za pomocą dekoratora rejestrujemy do obsługi błędów HTTP 404, zwraca szablon `404.html`,
- `for i in range(3)` – pętla, w której ustalamy numer poprawnej odpowiedzi (`p.odpok=i`), który przekażemy do formularza, aby zaznaczony został właściwy przycisk radio,
- `form = DodajForm(obj=p)` – przed przekazaniem formularza do szablonu wypełniamy go danymi używając parametru `obj`.

Żądanie zapisania danych z formularza (POST):

- `p.pytanie = form.pytanie.data` – aktualizujemy dane pytania po sprawdzeniu ich poprawności;
- `for i, o in enumerate(p.odpowiedzi)` – pętla, w której aktualizujemy kolejne odpowiedzi: `o.odpowiedz = odp[i]`.

Szablon `404.html` może wyglądać np. tak:

```
<!-- quiz-orm/templates/dodaj.html -->
{% extends "szkielet.html" %}
{% set active_page = "index" %}
{% block tytul %}Błąd: strony nie znaleziono{% endblock%}
{% block h1 %}Quiz ORM &#8211; błąd{% endblock%}
{% block tresc %}
<style type="text/css">
.error-template {padding: 40px 15px;text-align: center;}
.error-actions {margin-top:15px;margin-bottom:15px;}
.error-actions .btn { margin-right:10px; }
</style>
<div class="container">
  <div class="row">
    <div class="error-template">
      <h2>404 Nie znaleziono</h2>
      <div class="error-details">
        Przepraszamy, wystąpił błąd, żądanej strony nie znaleziono!<br>
      </div>
      <div class="error-actions">
        <a href="{{ url_for('index') }}" class="btn btn-primary">
        <i class="icon-home icon-white"></i> Strona główna</a>
      </div>
```

```
    </div>
  </div>
</div>
{% endblock %}
```

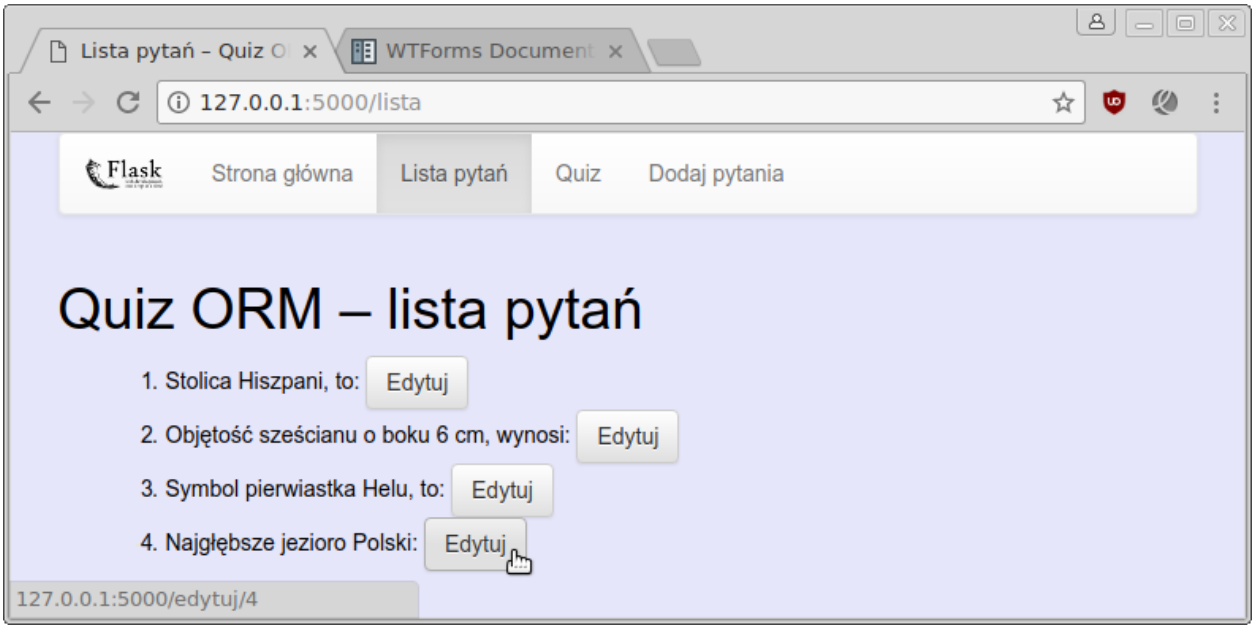
Szablon edycji jest bardzo podobny do szablonu dodawania, ponieważ wykorzystujemy ten sam formularz. Tworzymy więc plik `edytuj.html`:

```
<!-- quiz-orm/templates/edytuj.html -->
{% extends "szkielet.html" %}
{% set active_page = "edytuj" %}
{% block tytul %}Edycja{% endblock%}
{% block h1 %}Quiz ORM &#8211; edycja pytań{% endblock%}
{% block tresc %}
  {% include "pytanie_form.html" %}
{% endblock %}
```

Linki umożliwiające edycję pytań wygenerujemy w na liście pytań. W pliku `lista.html` po kodzie `{{ pytanie }}` wstawiamy:

```
      {{ p.pytanie }}
      <a href="{{ url_for('edytuj', pid=p.id ) }}" class="btn btn-default">Edytuj</a>
```

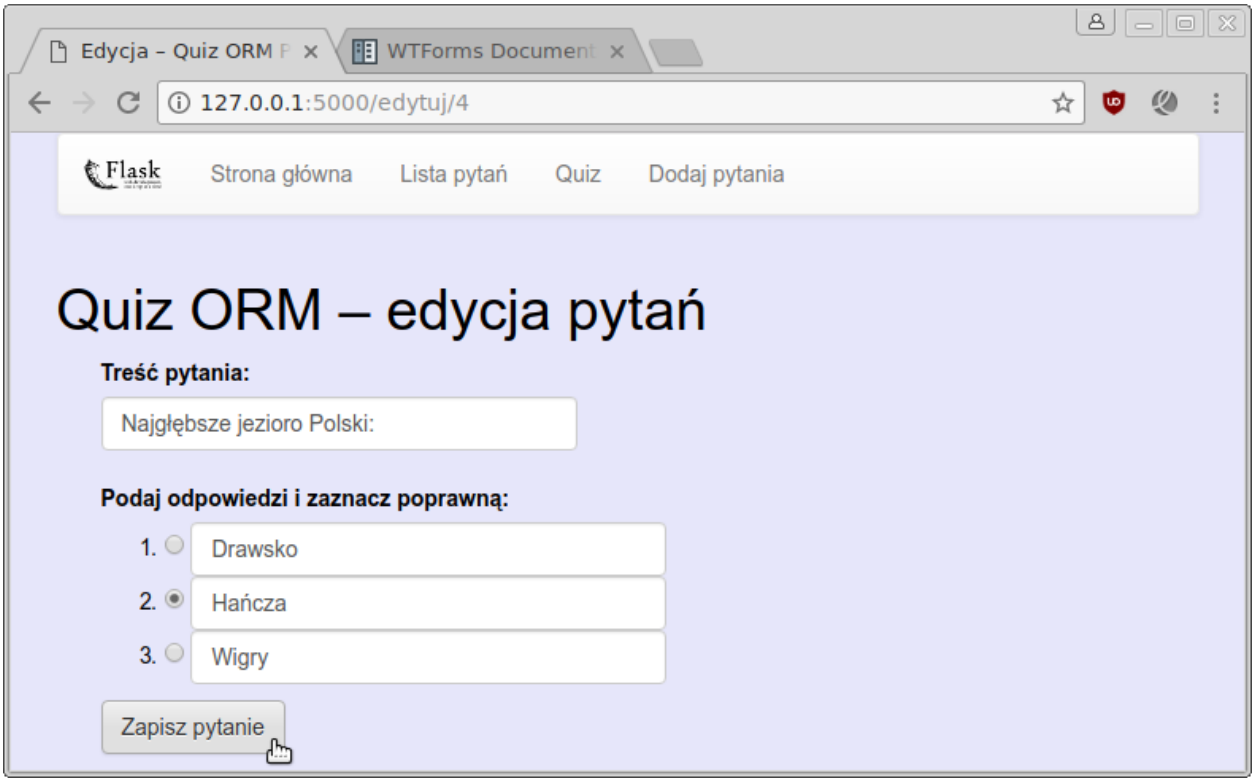
- `{{ url_for('edytuj', pid=p.id ) }}` – funkcja generuje adres dla podanego widoku dodając na końcu identyfikator pytania.



Usuwanie

Pozostaje umożliwienie usuwania pytań i odpowiedzi. W pliku `views.py` dodajemy widok `usun()`:

```
@app.route('/usun/<int:pid>', methods=['GET', 'POST'])
def usun(pid):
    """Usunięcie pytania o identyfikatorze pid"""
```




```
126     p = get_or_404(pid)
127     if request.method == 'POST':
128         flash('Usunięto pytanie {0}'.format(p.pytanie), 'sukces')
129         p.delete_instance(recursive=True)
130         return redirect(url_for('index'))
131     return render_template("pytanie_usun.html", pytanie=p)
```

- `'/usun/<int:pid>'` – podobnie jak w przypadku edycji widok obsłuży adres URL zawierający identyfikator pytania, który wykorzystujemy do pobrania obiektu z bazy danych,
- `p.delete_instance(recursive=True)` – obsługując żądanie typu POST, usuwamy pytania, a także wszystkie skojarzone z nim odpowiedzi (opcja `recursive`).

W przypadku żądania typu GET zwracamy formularz potwierdzenia usunięcia `pytanie_usun.html`:

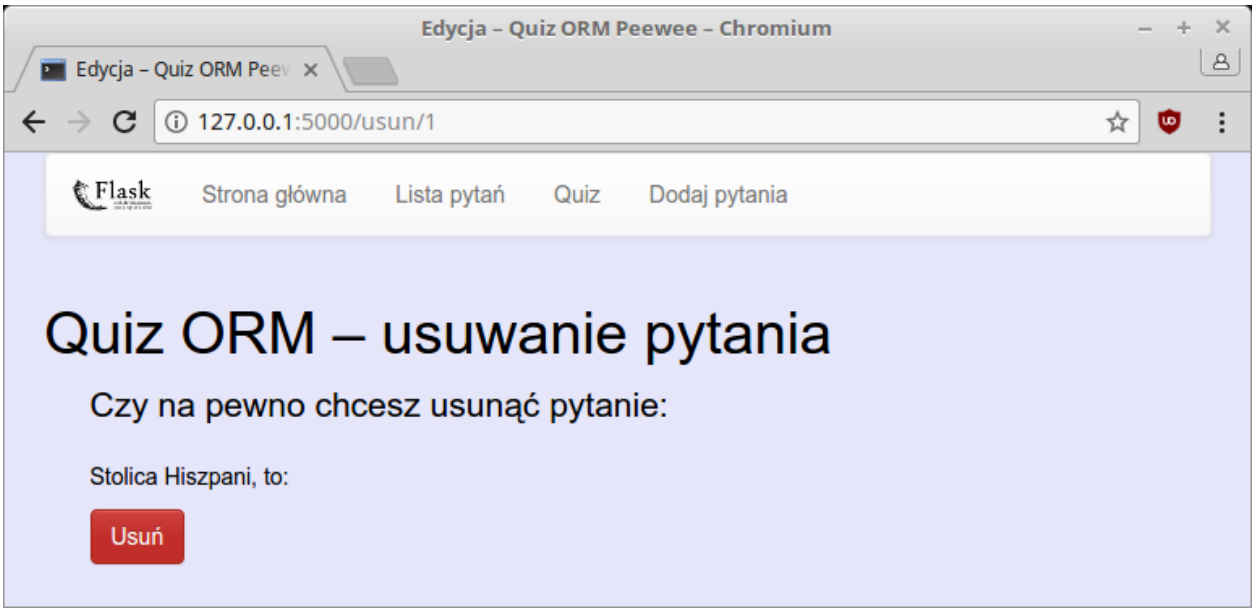
```
1  <!-- quiz-orm/templates/pytanie_usun.html -->
2  {% extends "szkielet.html" %}
3  {% set active_page = "usun" %}
4  {% block tytul %}Edycja{% endblock%}
5  {% block h1 %}Quiz ORM &#8211; usuwanie pytania{% endblock%}
6  {% block tresc %}
7    <form method="POST" action="{{ url_for('usun', pid=pytanie.id) }}">
8      <!-- wstawiamy id pytania -->
9      <p class="lead">Czy na pewno chcesz usunąć pytanie:</p>
10     <p>{{ pytanie.pytanie }}</p>
11     <button id="btn-delete" type="submit" class="btn btn-danger">Usuń</button>
12   </form>
13 {% endblock %}
```

- `action="{{ url_for('usun', pid=pytanie.id) }}` – generujemy adres, pod który wysłane zo-

stanie potwierdzenie.

Na koniec należy wstawić link umożliwiający usunięcie pytania do szablonu `lista.html`:

```
13          <a href="{{ url_for('edytuj', pid=p.id ) }}" class="btn btn-default">Edytuj</a>
14          <a href="{{ url_for('usun', pid=p.id ) }}" class="btn btn-danger">Usuń</a>
```



SQLAlchemy

Instalacja wymaganych modułów:

```
~$ sudo pip install  sqlalchemy flask-sqlalchemy flask-wtf
```

Obsługa bazy nie wymaga w przypadku SQLAlchemy funkcji nawiązujących i kończących połączenia z bazą. Wszystko odbywa się w sesji tworzonej automatycznie.

```
# -*- coding: utf-8 -*-
# quiz-orm/app.py

from flask import Flask
from flask_sqlalchemy import SQLAlchemy
import os

app = Flask(__name__)

# konfiguracja aplikacji
app.config.update(dict(
    SECRET_KEY='bardzosekretnawartosc',
    DATABASE=os.path.join(app.root_path, 'quiz.db'),
    SQLALCHEMY_DATABASE_URI='sqlite:///' +
                            os.path.join(app.root_path, 'quiz.db'),
    SQLALCHEMY_TRACK_MODIFICATIONS=False,
    TYTUL='Quiz ORM SQLAlchemy'
))

# tworzymy instancję bazy używanej przez modele
baza = SQLAlchemy(app)
```

- `SQLALCHEMY_TRACK_MODIFICATIONS=False` – wyłączenie nieużywanego przez nas śledzenia modyfikacji obiektów i emitowania sygnałów.

W pliku `dane.py` należy zaimportować obiekt umożliwiający zarządzanie bazą danych, następnie modyfikujemy funkcję `dodaj_pytanie()`:

```
from app import baza
```

```
def dodaj_pytania(dane):
    """Funkcja dodaje pytania i odpowiedzi przekazane w tupli do bazy."""
    for pytanie, odpowiedzi, odpok in dane:
        p = Pytanie(pytanie=pytanie, odpok=odpok)
        baza.session.add(p)
        baza.session.commit()
        for o in odpowiedzi.split(","):
            odp = Odpowiedz(pnr=p.id, odpowiedz=o.strip())
            baza.session.add(odp)
        baza.session.commit()
    print("Dodano przykładowe pytania")
```

W operacjach dodawania, również w funkcji `dodaj()` (zob. niżej) korzystamy z metod obiektu sesji:

- `session.add()` – dodaje obiekt,
- `session.commit()` – zatwierdza zmiany w bazie.

W pliku `main.py` zmieniamy tylko jedną linią:

```
    if not os.path.exists(app.config['DATABASE']):
        baza.create_all()  # tworzymy tabele
```

- `create_all()` – funkcja tworzy wszystkie tabele na podstawie zadeklarowanych modeli:

```
# -*- coding: utf-8 -*-
# quiz-orm/models.py

from app import baza


class Pytanie(baza.Model):
    id = baza.Column(baza.Integer, primary_key=True)
    pytanie = baza.Column(baza.Unicode(255), unique=True)
    odpok = baza.Column(baza.Unicode(100))
    odpowiedzi = baza.relationship(
        'Odpowiedz', backref=baza.backref('pytanie'),
        cascade="all, delete, delete-orphan")

    def __str__(self):
        return self.pytanie


class Odpowiedz(baza.Model):
    id = baza.Column(baza.Integer, primary_key=True)
    pnr = baza.Column(baza.Integer, baza.ForeignKey('pytanie.id'))
    odpowiedz = baza.Column(baza.Unicode(100))

    def __str__(self):
        return self.odpowiedz
```

- `from app import baza` – jedyny import, którego potrzebujemy, to obiekt `baza` udostępniający wszystkie klasy i metody SQLAlchemy,
- `primary_key=True` – definicja klucza podstawowego, czyli identyfikatora pytania i odpowiedzi,
- `ForeignKey()` – określenie klucza obcego, czyli relacji,
- `relationship()` – relacja zwrotna, właściwość `Pytanie.odpowiedzi`,
- `backref=baza.backref('pytanie')` – relacja zwrotna, właściwość `Odpowiedz.pytanie`,
- `cascade=ąll, delete, delete-orphan"` – usuwanie rekordów z powiązanych tabel.

Zmiany w pliku `views.py` dotyczą głównie innej składni zapytań do bazy. Na początku drobne zmiany w importach: usuwamy obiekt `abort` i dodajemy import obiektu `baza`:

```
from flask import render_template, request, redirect, url_for, flash
from app import app, baza
```

Funkcja `lista()` – zmieniamy instrukcje odczytujące pytania z bazy:

```
    """Pobranie wszystkich pytań z bazy i zwrócenie szablonu z listą pytań"""
    pytania = Pytanie.query.all()
    if not pytania:
```

- `pytania = Pytanie.query.all()` – pobranie z bazy wszystkich pytań w formie listy.

Funkcja `quiz()` – zmieniamy zapytanie odczytujące poprawną odpowiedź:

```
            odpok = baza.session.query(Pytanie.odpok).filter(
                Pytanie.id == int(pid)).scalar()
```

– a także zapytanie odczytujące pytania oraz odpowiedzi z bazy:

```
42      # GET, wyświetl pytania
43      pytania = Pytanie.query.join(Odpowiedz).all()
```

- `.join()` – metoda pozwala odczytać odpowiedzi powiązane relacją z pytaniem.

Funkcja `dodaj()` – zmieniamy polecenia dodające obiekty do bazy:

```
68          p = Pytanie(pytanie=form.pytanie.data, odpok=odp[int(form.odpok.data)])
69          baza.session.add(p)
70          baza.session.commit()
71          for o in odp:
72              inst = Odpowiedz(pnr=p.id, odpowiedz=o)
73              baza.session.add(inst)
74          baza.session.commit()
75          flash("Dodano pytanie: {}".format(form.pytanie.data))
76          return redirect(url_for("lista"))
77      elif request.method == 'POST':
78          flash_errors(form)
79
80      return render_template("dodaj.html", form=form, radio=list(form.odpok))
81
82
83  @app.errorhandler(404)
84  def page_not_found(e):
```

Funkcje `edytuj()` i `usun()` – zmieniamy kod pobierający obiekt o podanym identyfikatorze z bazy:

```
92      p = Pytanie.query.get_or_404(pid)
```

Funkcja `get_or_404()` – jest niepotrzebna i należy ją usunąć. Zamiast niej używamy metody dostępnej w SQLAlchemy.

Funkcja `edytuj()` – upraszczamy kod aktualizujący obiekty w bazie, :

```
98          p.odpok = odp[int(form.odpok.data)]
99          for i, o in enumerate(p.odpowiedzi):
100             o.odpowiedz = odp[i]
101         baza.session.commit()
```

Funkcja `usun()` – kod usuwający obiekty z bazy przyjmuje postać:

```
120         flash('Usunięto pytanie {0}'.format(p.pytanie), 'sukces')
121         baza.session.delete(p)
122         baza.session.commit()
```

Źródła

- `quiz-orm-peewee.zip`
- `quiz-orm-sqlalchemy.zip`

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Słownik aplikacji internetowych

aplikacja program komputerowy.

framework zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji, przykładem jest biblioteka Pythona Flask.

HTML język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.

CSS język służący do opisu formy prezentacji stron WWW.

HTTP protokół przesyłania dokumentów WWW. Więcej o HTTP »»»

GET typ żądania HTTP, służący do pobierania zasobów z serwera WWW. Więcej o GET »»»

POST typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW. Więcej o POST »»»

Kod odpowiedzi HTTP numeryczne oznaczenie stanu realizacji zapytania klienta, np. *200 (OK)* lub *404 (Not Found)*. Więcej o kodach HTTP »»»

logowanie proces autoryzacji i uwierzytelniania użytkownika w systemie.

ORM (ang. Object-Relational Mapping) – mapowanie obiektowo-relacyjne, oprogramowanie odwzorowujące strukturę relacyjnej bazy danych na obiekty danego języka oprogramowania.

Peewee prosty i mały system ORM, wspiera Pythona w wersji 2 i 3, obsługuje bazy SQLite3, MySQL, Posgresql.

SQLAlchemy rozbudowany zestaw narzędzi i system ORM umożliwiający wykorzystanie wszystkich możliwości SQL-a, obsługuje bazy SQLite3, MySQL, Postgresql, Oracle, MS SQL Server i inne.

serwer deweloperski testowy serwer www używany w czasie prac nad oprogramowaniem.

serwer WWW serwer obsługujący protokół HTTP.

baza danych program przeznaczony do przechowywania i przetwarzania danych.

szablon wzorzec (nazywany czasem templatką) strony WWW wykorzystywany do renderowania widoków.

renderowanie szablonu przetwarzanie szkieletowego kodu HTML oraz specjalnych tagów w celu uzyskania kompletnego kodu HTML strony zawierającego przekazane do szablonu dane.

URL ustandaryzowany format adresowania zasobów w internecie (przykład).

MVC (ang. Model-View-Controller) – Model-Widok-Kontroler, wzorzec projektowania aplikacji rozdzielający dane (model) od sposobu ich prezentacji (widok) i zarządzania ich przepływem (kontroler).

model schemat opisujący strukturę danych w bazie, np. klasa definiująca tabele i relacje między nimi. Więcej o modelu bazy danych »»»

widok we Flasku lub Django jest to funkcja lub klasa, która obsługuje żądania wysyłane przez użytkownika, przeprowadza operacje na danych i najczęściej zwraca je np. w formie strony WWW do przeglądarki.

kontroler logika aplikacji, we Flasku lub Django mechanizm obsługujący zadania HTTP powiązane z określonymi adresami URL za pomocą widoków (funkcji lub klas).

sesja w kontekście aplikacji wykorzystujących protokół HTTP sposób zapamiętywania po stronie serwera danych związanych z konkretnym użytkownikiem.

ciasteczka (ang. *cookies*) zaszyfrowane dane tekstowe wysyłane przez serwer i zapamiętywane po stronie klienta, zawierają np. identyfikator sesji użytkownika.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Materiały

1. Python
2. Flask
3. SQLite
4. Peewee
5. SQLAlchemy

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.8 Aplikacje WWW (Django)

Python znakomicie nadaje się do tworzenia aplikacji internetowych dzięki frameworkowi Django. Upraszcza on projektowanie serwisów oferując gotowe rozwiązania wielu powtarzalnych i pracochłonnych mechanizmów wymaganych w serwisach internetowych. Oferuje również gotowe środowisko testowe, czyli deweloperski serwer WWW, nie musimy instalować żadnych dodatkowych narzędzi typu LAMP (WAMP).

Zobacz, jak zainstalować wymagane biblioteki w systemie *Linux* lub *Windows*.

Czat (cz. 1)

Zastosowanie Pythona i frameworka Django do stworzenia aplikacji internetowej Czat; prostego czata, w którym zarejestrowani użytkownicy będą mogli wymieniać się krótkimi wiadomościami.

Uwaga: Wymagane oprogramowanie:

- Python v. 3.x
- Django v. 1.11.2
- Interpreter bazy SQLite3

Środowisko

W katalogu domowym tworzymy wirtualne środowisko Pythona:

```
~$ virtualenv -p python3 pve3
~$ source pve3/bin/activate
(pve3) ~$ pip install Django==1.11.2
```

Ostrzeżenie: Polecenie `source pve3/bin/activate` aktywuje wirtualne środowisko Pythona. Zawsze wydajemy je przed rozpoczęciem pracy nad projektem. Innymi słowy w terminalu ścieżka katalogu musi być poprzedzona prefiksem wirtualnego środowiska: `(pve3)`.

Projekt i aplikacja

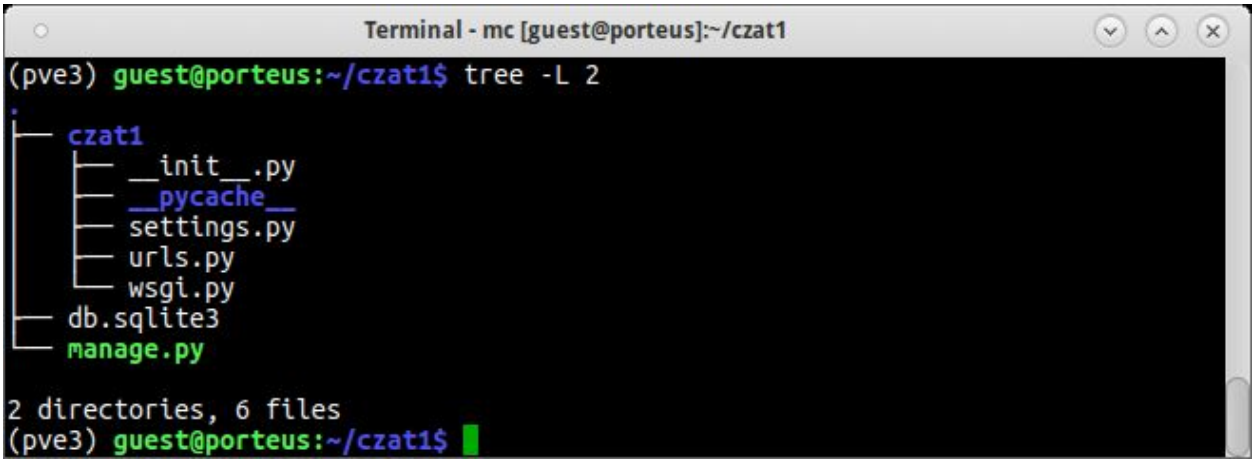
Utworzymy nowy projekt Django. Wydajemy polecenia:

```
(pve3) ~/$ django-admin.py startproject czat1
(pve3) ~$ cd czat1
(pve3) ~$ python manage.py migrate
```

- `startproject` – tworzy katalog `czat1` z **podkatalogiem ustawień projektu** o takiej samej nazwie (`czat1`),
- `migrate` – tworzy inicjalną bazę danych z tabelami wykorzystywanymi przez Django.

Struktura plików projektu – w terminalu wydajemy jedno z poleceń:

```
(.pve) ~/czat1$ tree -L 2
[lub]
(.pve) ~/czat1$ ls -R
```



Zewnętrzny katalog `czat1` to tylko pojemnik na projekt, jego nazwę można zmieniać. Zawiera on:

- `manage.py` – skrypt Pythona do zarządzania projektem;
- `db.sqlite3` – bazę danych w domyślnym formacie SQLite3.

Katlog projektu `czat1/czat1` zawiera:

- `settings.py` – konfiguracja projektu;
- `urls.py` – lista obsługiwanych adresów URL;
- `wsgi.py` – plik konfiguracyjny wykorzystywany przez serwery WWW.

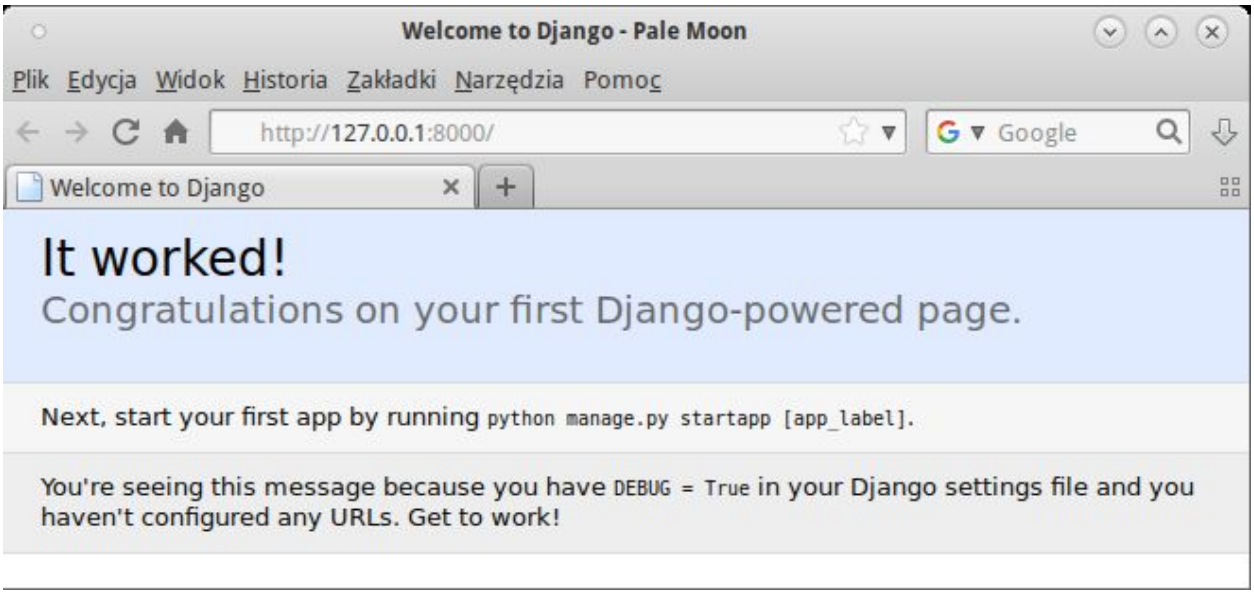
Plik `__init__.py` obecny w danym katalogu wskazuje, że dany katalog jest modułem Pythona.

Serwer deweloperski

Serwer uruchamiamy poleceniem w terminalu:

```
(pve3) ~/czat1$ python manage.py runserver
```

Łączymy się z serwerem wpisując w przeglądarce adres: `127.0.0.1:8000`. W terminalu możemy obserwować żądania obsługiwane przez serwer. Większość zmian w kodzie nie wymaga restartowania serwera. Serwer zatrzymujemy naciskając w terminalu skrót `CTRL+C`.

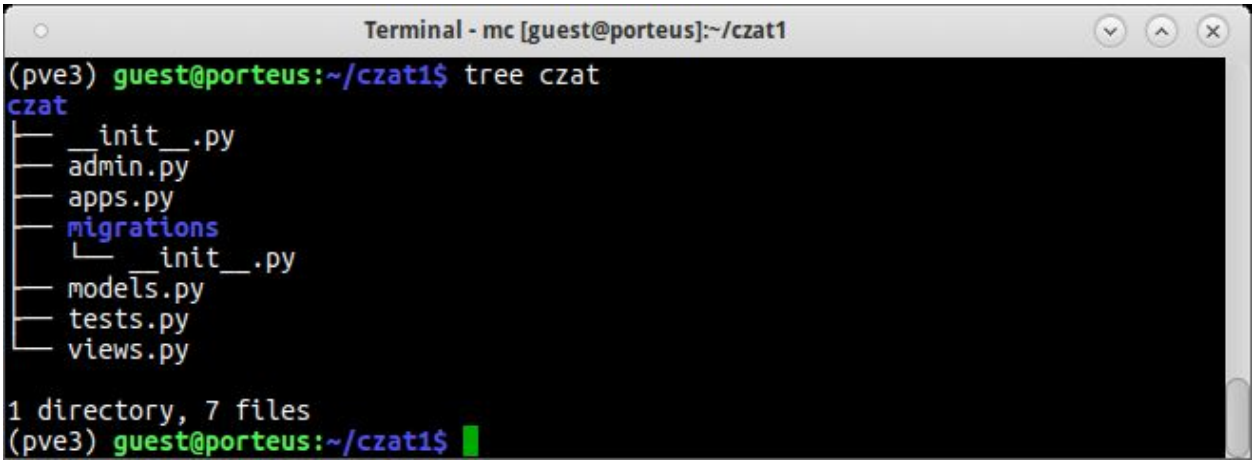


Aplikacja

W ramach jednego projektu (serwisu internetowego) może działać wiele aplikacji. Utworzymy teraz aplikację *czat* i zbadamy jej strukturę plików:

```
(.pve) ~/czat1$ python manage.py startapp czat
(.pve) ~/czat1$ tree czat
lub:
(.pve) ~/czat1$ ls -R czat
```

Katalog aplikacji `czat1/czat` zawiera:



- `apps.py` – ustawienia aplikacji;
- `admin.py` – konfigurację panelu administracyjnego;
- `models.py` – plik definiujący modele danych przechowywanych w bazie;
- `views.py` – plik zawierający funkcje lub klasy definiujące tzw. *widoki* (ang. *views*), obsługujące żądania klienta przychodzące do serwera.

Ustawienia projektu

Dostosujemy ustawienia projektu: zarejestrujemy aplikację w projekcie, ustawimy polską wersję językową oraz zlokalizujemy datę i czas. Edytujemy plik `czat1/settings.py`:

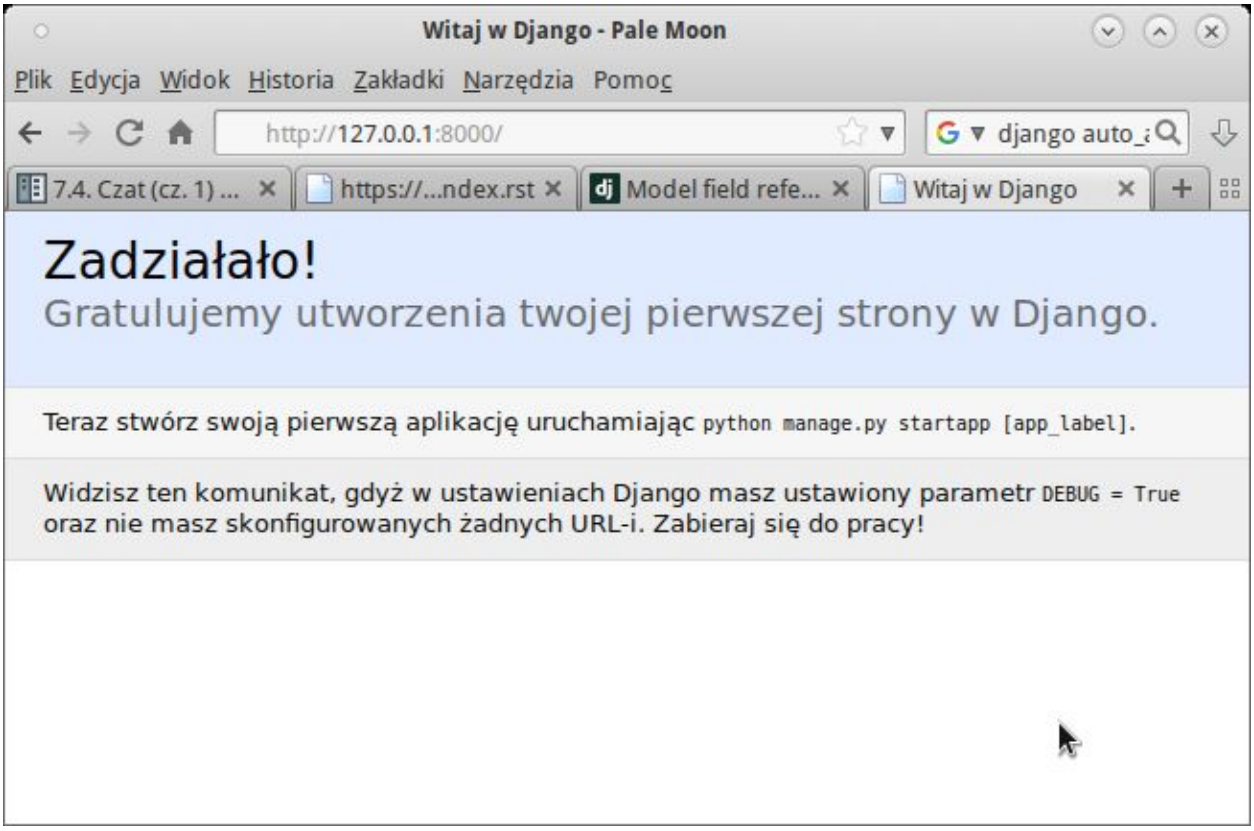
```
# czat1/settings.py

INSTALLED_APPS = [
    'czat.apps.CzatConfig', # rejestrujemy aplikacje czat
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

LANGUAGE_CODE = 'pl'  # ustawienie jezyka

TIME_ZONE = 'Europe/Warsaw'  # ustawienie strefy czasowej
```

Uruchom ponownie serwer deweloperski i sprawdź w przeglądarce, jak wygląda strona powitalna.

Model danych

Budowanie aplikacji w Django nawiązuje do wzorca projektowego *MVC*, czyli Model-Widok-Kontroler. Więcej informacji na ten temat umieściliśmy w osobnym materiale *MVC*.

Zaczynamy więc od zdefiniowania modelu (zob. *model*), czyli klasy opisującej tabelę zawierającą wiadomości. Atrybuty klasy odpowiadają polom tabeli. Instancje tej klasy będą reprezentować wiadomości utworzone przez użytkowników, czyli rekordy tabeli. Każda wiadomość będzie zwierała treść, datę dodania oraz wskazanie autora (użytkownika).



W pliku `czat/models.py` wpisujemy:

```
# -*- coding: utf-8 -*-
# czat/models.py

from django.db import models
from django.contrib.auth.models import User


class Wiadomosc(models.Model):

    """Klasa reprezentująca wiadomość w systemie"""
    tekst = models.CharField('treść wiadomości', max_length=250)
    data_pub = models.DateTimeField('data publikacji', auto_now_add=True)
    autor = models.ForeignKey(User)
```

Opisując klasę `Wiadomosc` podajemy nazwy poszczególnych właściwości (pól) oraz typy przechowywanych w nich danych.

Informacja: Typy pól:

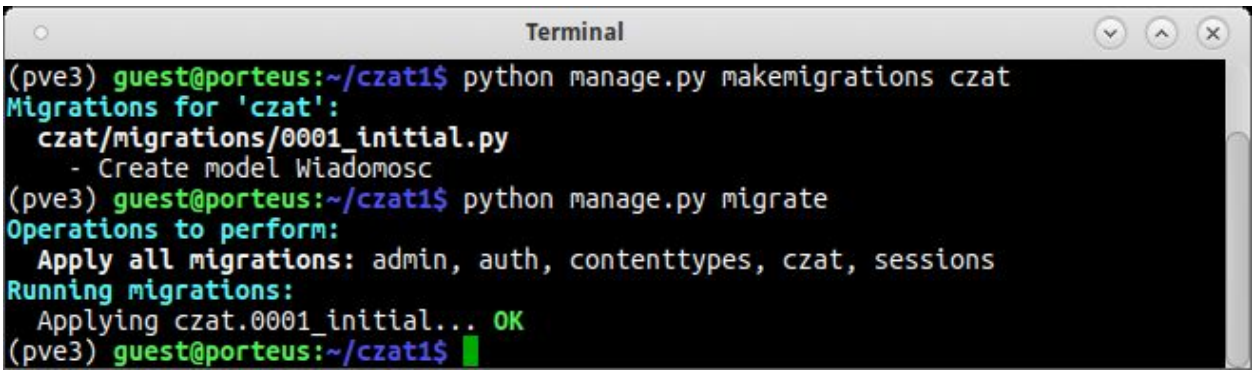
- `CharField` – pole znakowe, przechowuje niezbyt długie napisy, np. nazwy;
- `Date(Time)Field` – pole daty (i czasu);
- `ForeignKey` – pole klucza obcego, czyli relacji; wymaga nazwy powiązanego modelu jako pierwszego argumentu.

Właściwości pól:

- `verbose_name` lub napis podany jako pierwszy argument – przyjazna nazwa pola;
- `max_length` – maksymalna długość pola znakowego;
- `help_text` – tekst podpowiedzi;
- `auto_now_add=True` – data (i czas) wstawione zostaną automatycznie.

Utworzenie migracji – po dodaniu lub zmianie modelu należy zaktualizować bazę danych, tworząc tzw. migrację, czyli zapis zmian:

```
(.pve) ~/czat1$ python manage.py makemigrations czat
(.pve) ~/czat1$ python manage.py migrate
```



Informacja: Domyślnie Django korzysta z bazy SQLite zapisanej w pliku `db.sqlite3`. Warto zobaczyć, jak wygląda. W terminalu wydajemy polecenie `python manage.py dbshell`, które otworzy bazę w interpreterze `sqlite3`. Następnie: * `.tables` - pokaże listę tabel; * `.schema czat_wiadomosc` - pokaże instrukcje SQL-a użyte do utworzenia podanej tabeli * `.quit` - wyjście z interpretera.

Panel administracyjny

Panel administratora pozwala dodawać użytkowników i wprowadzać dane. W pliku `czat/admin.py` umieszczamy kod:

```
1 # -*- coding: utf-8 -*-
2 # czatpro/czat/admin.py
3
4 from django.contrib import admin
5 from czat import models  # importujemy nasz model
6
7 # rejestrujemy model Wiadomosc w panelu administracyjnym
8 admin.site.register(models.Wiadomosc)
```

Po zaimportowaniu modelu rejestrujemy go w panelu: `admin.site.register(models.Wiadomosc)`.

Informacja: Warto zapamiętać, że każdy model, funkcję, formularz czy widok, których chcemy użyć, musimy najpierw zaimportować za pomocą klauzuli typu `from <skąd> import <co>`.

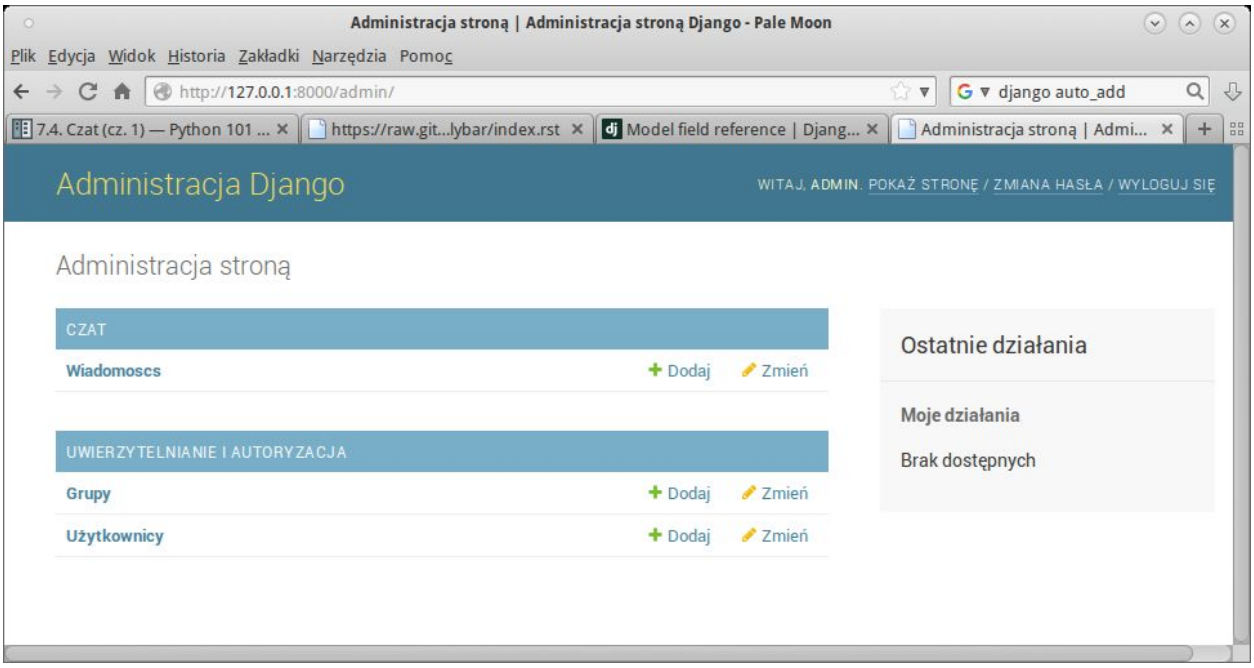
Konto administratora tworzymy wydając w terminalu polecenie:

```
(.pve) ~/czat1$ python manage.py createsuperuser
```

– na pytanie o nazwę, email i hasło administratora, podajemy: “admin”, “”, “zaq1@WSX”.

Ćwiczenie

1. Uruchom/zrestartuj serwer, w przeglądarce wpisz adres *127.0.0.1:8000/admin/* i zaloguj się na konto administratora.



2. Dodaj użytkowników “adam” i “ewa” z hasłami “zaq1@WSX”.

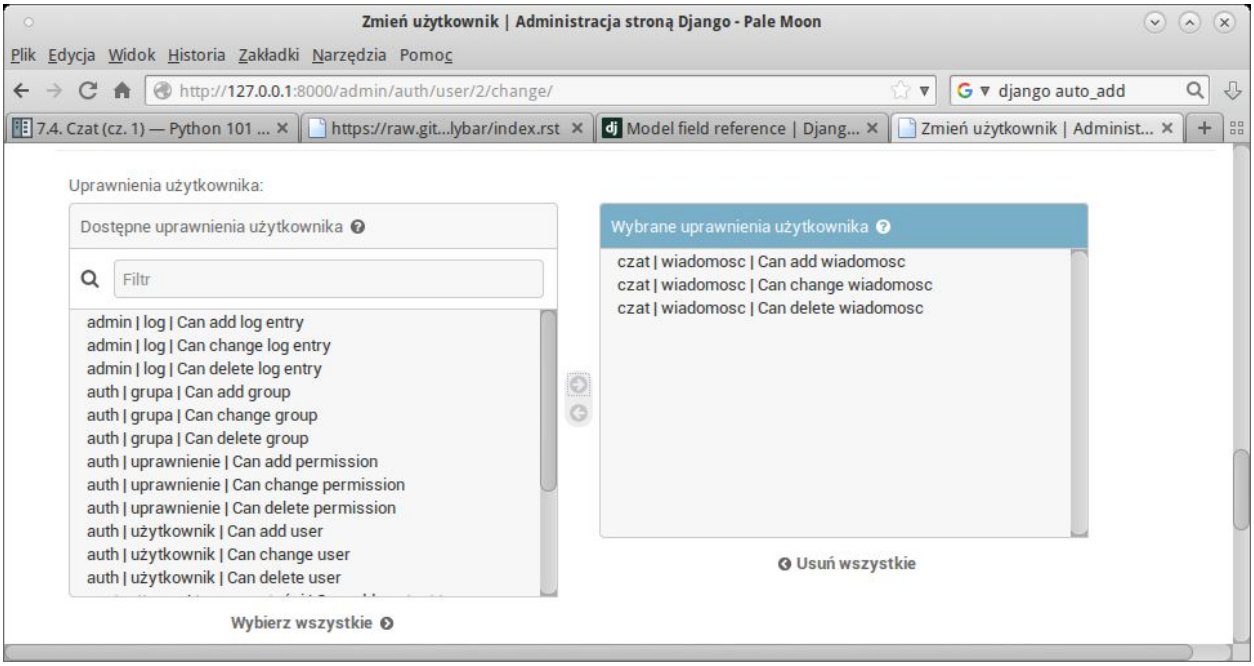
   Na stronie, która wyświetla się po utworzeniu konta, zaznacz opcję “W zespole”. W sekcji “Dostępne uprawnienia” zaznacz prawa dodawania (*add*), zmieniania (*change*) oraz usuwania (*del*) wiadomości (wpisy typu: “czat | wiadomosc | Can add wiadomosc”) i przypisz je użytkownikowi naciskając strzałkę w prawo.

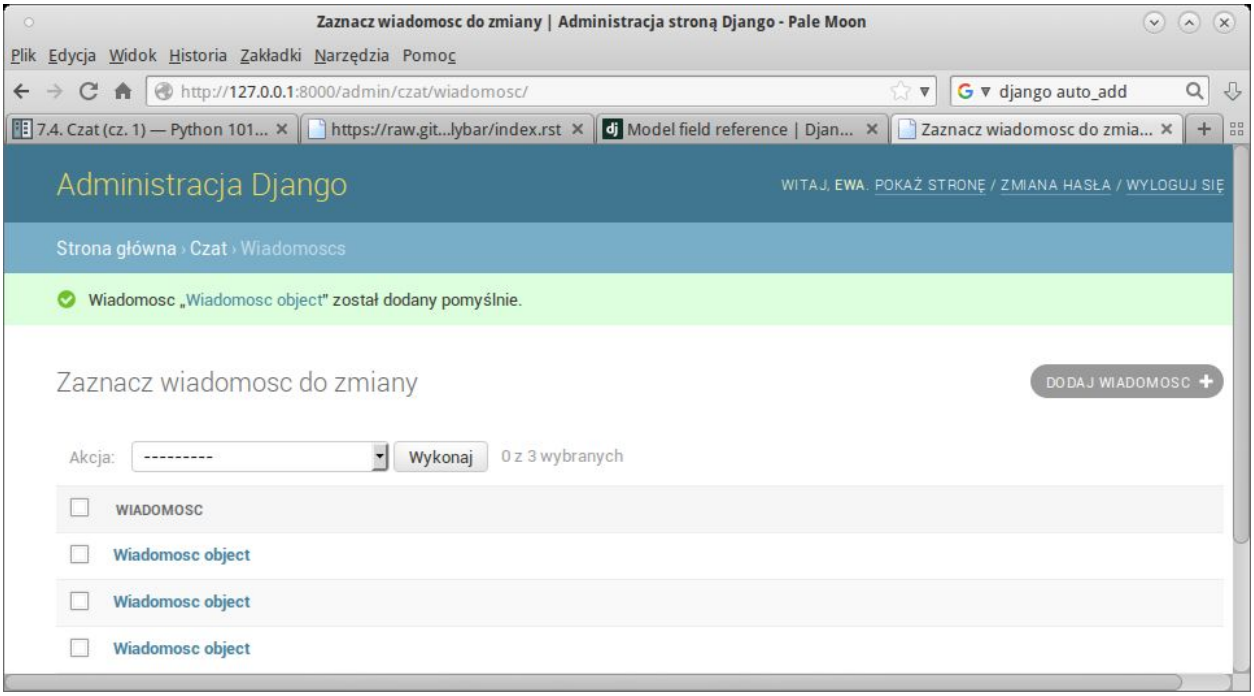
3. Z konta “adam” dodaj dwie przykładowe wiadomości, a z konta “ewa” – jedną.

Uzupełnienie modelu

W formularzu dodawania wiadomości widać, że etykiety opisujące nasz model nie są spolszczone. Uzupełniamy więc plik `czat/models.py`:

```
8  class Wiadomosc(models.Model):
9
10     """Klasa reprezentująca wiadomość w systemie"""
11     tekst = models.CharField('treść wiadomości', max_length=250)
12     data_pub = models.DateTimeField('data publikacji', auto_now_add=True)
13     autor = models.ForeignKey(User)
14
15     class Meta:  # ustawienia dodatkowe
16         verbose_name = u'wiadomość'  # nazwa obiektu w języku polskim
17         verbose_name_plural = u'wiadomości'  # nazwa obiektów w l.m.
```

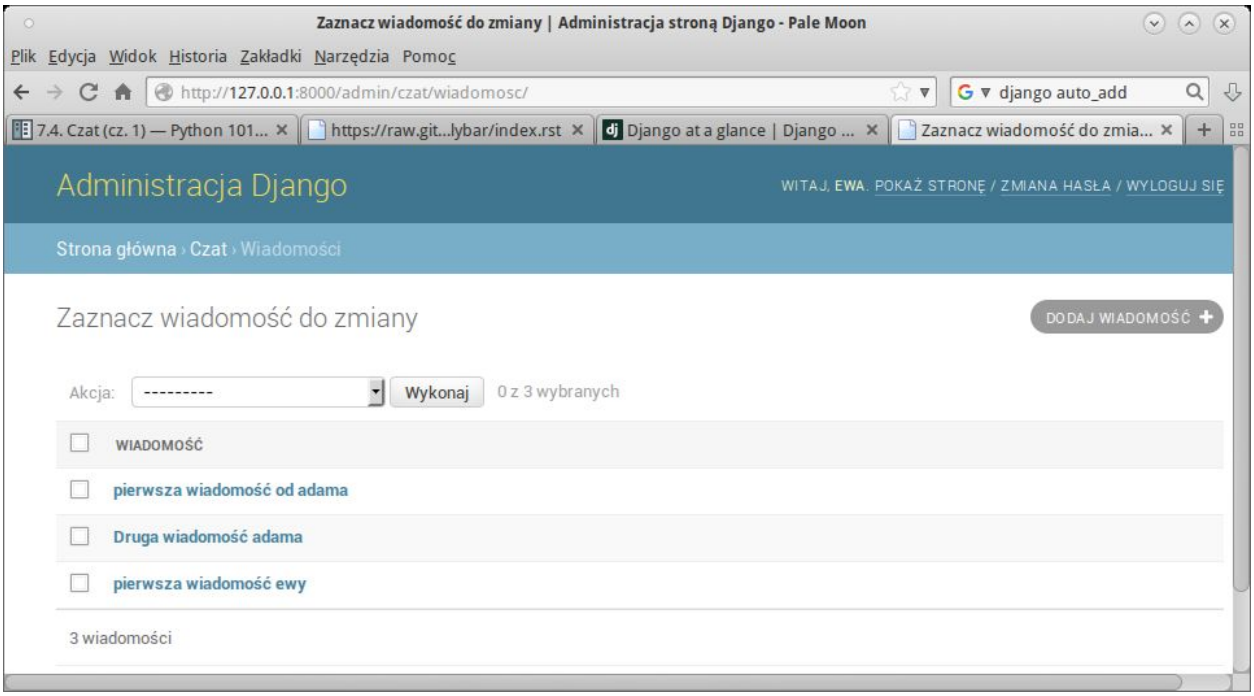

Zmień użytkownik | Administracja stroną Django - Pale Moon
Plik Edycja Widok Historia Zakładki Narzędzia Pomoc
http://127.0.0.1:8000/admin/auth/user/2/change/
django auto_add
7.4. Czat (cz. 1) — Python 101 ...
https://raw.git...lybar/index.rst
Model field reference | Djang...
Zmień użytkownik | Administ...
Uprawnienia użytkownika:
Dostępne uprawnienia użytkownika
Filtr
admin | log | Can add log entry
admin | log | Can change log entry
admin | log | Can delete log entry
auth | grupa | Can add group
auth | grupa | Can change group
auth | grupa | Can delete group
auth | uprawnienie | Can add permission
auth | uprawnienie | Can change permission
auth | uprawnienie | Can delete permission
auth | użytkownik | Can add user
auth | użytkownik | Can change user
auth | użytkownik | Can delete user
Wybierz wszystkie
Wybrane uprawnienia użytkownika
czat | wiadomosc | Can add wiadomosc
czat | wiadomosc | Can change wiadomosc
czat | wiadomosc | Can delete wiadomosc
Usuń wszystkie


Zaznacz wiadomosc do zmiany | Administracja stroną Django - Pale Moon
Plik Edycja Widok Historia Zakładki Narzędzia Pomoc
http://127.0.0.1:8000/admin/czat/wiadomosc/
django auto_add
7.4. Czat (cz. 1) — Python 101...
https://raw.git...lybar/index.rst
Model field reference | Djan...
Zaznacz wiadomosc do zmia...
Administracja Django
WITAJ, EWA. POKAŻ STRONĘ / ZMIANA HASŁA / WYLOGUJ SIĘ
Strona główna › Czat › Wiadomoscs
Wiadomosc „Wiadomosc object" został dodany pomyślnie.
Zaznacz wiadomosc do zmiany
DODAJ WIADOMOSC
Akcja: ---------
Wykonaj
0 z 3 wybranych
WIADOMOSC
Wiadomosc object
Wiadomosc object
Wiadomosc object

```
        ordering = ['data_pub']  # domyślne porządkowanie danych

    def __str__(self):
        return self.tekst  # "autoprezentacja"
```

Podklasa `Meta` pozwala zdefiniować formy liczby pojedynczej i mnogiej oraz domyślny sposób sortowania wiadomości (`ordering = ['data_pub']`). Zadaniem funkcji `__str__()` jest "autoprezentacja" klasy, czyli w naszym wypadku wyświetlenie treści wiadomości.

Odśwież panel administracyjny (np. klawiszem `F5`).



Strona główna

Aby utworzyć stronę główną, zakodujemy pierwszy *widok* (zob. *więcej »»»*), czyli funkcję o zwyczajowej nazwie `index()`. W pliku `views.py` umieszczamy:

```
# -*- coding: utf-8 -*-
# czat/views.py

from django.shortcuts import render
from django.http import HttpResponse


def index(request):
    """Strona główna aplikacji."""
    return HttpResponse("Witaj w aplikacji Czat!")
```

Najprostszy widok zwraca do klienta (przeglądarki) jakiś tekst: `return HttpResponse("Witaj w aplikacji Czat!")`.

Adresy URL, które ma obsługiwać nasza aplikacja, definiujemy w pliku `czat/urls.py`. Tworzymy nowy plik i uzupełniamy go kodem:

```
# -*- coding: utf-8 -*-
# czat/urls.py

from django.conf.urls import url
from . import views  # import widoków aplikacji

app_name = 'czat'  # przestrzeń nazw aplikacji
urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

- `app_name = 'czat'` – określamy przestrzeń nazw, w której dostępne będą mapowania między adresami url a widokami naszej aplikacji,
- `url()` – funkcja, która wiąże zdefiniowany adres URL z widokiem,
- `r'^$'` – wyrażenie regularne opisujące adres URL, symbol `^` to początek, `$` – koniec łańcucha. Zapis `r'^$'` to adres główny serwera;
- `views.index` – przykładowy widok, czyli funkcja zdefiniowana w pliku `czat/views.py`;
- `name='index'` – nazwa, która pozwoli na generowanie adresów url dla linków w kodzie HTML.

Konfigurację adresów URL naszej aplikacji musimy włączyć do konfiguracji adresów URL projektu. W pliku `czat1/urls.py` dopisujemy:

```
from django.conf.urls import url, include
from django.contrib import admin


urlpatterns = [
    url(r'^', include('czat.urls')),
    url(r'^admin/', admin.site.urls),
]
```

- `include()` – funkcja pozwala na import adresów URL wskazanej aplikacji,
- `'czat.urls'` – plik konfiguracyjny aplikacji.

Przetestuj stronę główną wywołując adres `127.0.0.1:8000`.

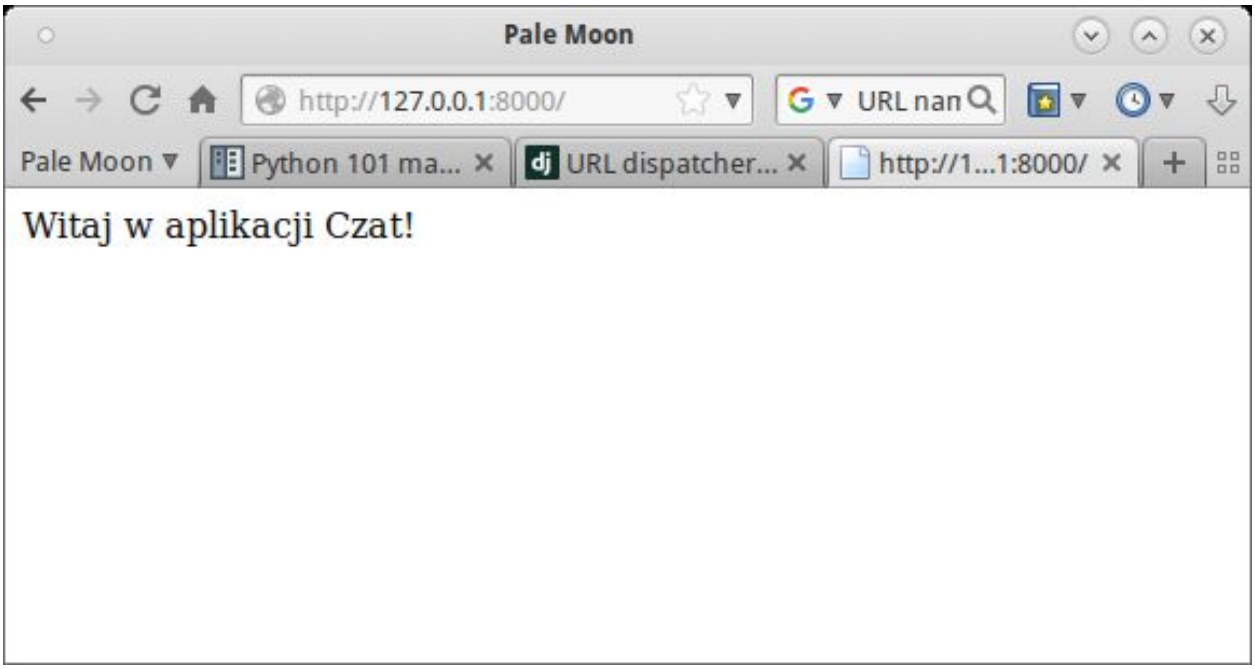
Widoki i szablony

Typową odpowiedzią na wywołanie jakiegoś adresu URL są strony zapisane w języku HTML. **Szablony** takich stron umieszczamy w podkatalogu `aplikacja/templates/aplikacja`. Tworzymy więc katalog:

```
(pve3) ~/czat1$ mkdir -p czat/templates/czat
```

Następnie tworzymy szablon `templates/czat/index.html`, który zawiera:

```
<!-- templates/czat/index.html -->
<html>
  <head></head>
  <body>
    <h1>Witaj w aplikacji Czat!</h1>
  </body>
</html>
```



W pliku `views.py` zmieniamy instrukcję odpowiedzi:

```
4  from django.shortcuts import render
5  # from django.http import HttpResponse
6
7
8  def index(request):
9      """Strona główna aplikacji."""
10     # return HttpResponse("Witaj w aplikacji Czat!")
11     return render(request, 'czat/index.html')
```

Funkcja `render()` jako pierwszy parametr pobiera obiekt typu `HttpRequest` zawierający informacje o żądaniu, jako drugi nazwę szablonu z katalogiem nadrzędnym.

Po uruchomieniu serwera i wpisaniu adresu *127.0.0.1:8000* zobaczymy tekst, który umieściliśmy w szablonie:

(Wy)logowanie

Udostępnimy użytkownikom możliwość logowania i wylogowywania się, aby mogli dodawać i przeglądać wiadomości.

Na początku w pliku `views.py`, dopisujemy importy wymaganych obiektów, później dodajemy widoki `loguj()` i `wyloguj()`:

```
6  from django.contrib.auth import login, logout
7  from django.shortcuts import redirect
8  from django.core.urlresolvers import reverse
9  from django.contrib import messages
```

```
19 def loguj(request):
20     """Logowanie użytkownika"""
21     from django.contrib.auth.forms import AuthenticationForm
22     if request.method == 'POST':
```


```
23          form = AuthenticationForm(request, request.POST)
24          if form.is_valid():
25              login(request, form.get_user())
26              messages.success(request, "Zostałeś zalogowany!")
27              return redirect(reverse('czat:index'))
28
29      kontekst = {'form': AuthenticationForm()}
30      return render(request, 'czat/loguj.html', kontekst)
31
32
33  def wyloguj(request):
34      """Wylogowanie użytkownika"""
35      logout(request)
36      messages.info(request, "Zostałeś wylogowany!")
37      return redirect(reverse('czat:index'))
```

Logowanie rozpoczyna się od wyświetlenia odpowiedniej strony – to żądanie typu *GET*. Widok logowania zwraca wtedy szablon: `return render(request, 'czat/loguj.html', kontekst)`. Parametr `kontekst` to słownik, który pod kluczem `form` zawiera pusty formularz logowania utworzony w instrukcji `AuthenticationForm()`.

Wypełnienie formularza danymi i przesłanie ich na serwer to żądanie typu *POST*. Wykrywamy je w instrukcji `if request.method == 'POST':`. Następnie tworzymy instancję formularza wypełnioną przesłanymi danymi: `form = AuthenticationForm(request, request.POST)`. Jeżeli dane są poprawne `if form.is_valid():`, możemy zalogować użytkownika za pomocą funkcji `login(request, form.get_user())`.

Tworzymy również informację zwrotną dla użytkownika, wykorzystując system komunikatów: `messages.error(request, "...")`. Tak utworzone komunikaty możemy odczytać w każdym szablonie ze zmiennej `messages`.

Wylogowanie polega na użyciu funkcji `logout(request)` – wyloguje ona użytkownika, którego dane zapisane są w przesłanym żądaniu. Po utworzeniu informacji zwrotnej podobnie jak po udanym logowaniu przekierowujemy użytkownika na stronę główną (`return redirect(reverse('index'))`) z żądaniem jej wyświetlenia (typu GET).

Szablon logowania `templates/czat/loguj.html` zawiera kod:

```
<!-- templates/czat/loguj.html -->
<html>
  <head></head>
  <body>
    <h1>Witaj w aplikacji Czat!</h1>

    <h2>Logowanie użytkownika</h2>
    {% if not user.is_authenticated %}
      <form action="." method="POST">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Zaloguj</button>
      </form>
    {% else %}
      <p>Jesteś już zalogowany jako {{ user.username }}</p>
      <ul>
        <li><a href="{% url 'czat:index'%}">Strona główna</a></li>
      </ul>
    {% endif %}

  </body>
</html>
```

W szablonach wykorzystujemy tagi dwóch rodzajów:

- `{% instrukcja %}` – pozwalają używać instrukcji sterujących, np. warunkowych lub pętli,
- `{{ zmienna }}` – służą wyświetlaniu wartości zmiennych lub wywoływaniu metod obiektów przekazanych do szablonu.
- `{% if not user.is_authenticated %}` – instrukcja sprawdza, czy aktualny użytkownik jest zalogowany,
- `{% csrf_token %}` – zabezpieczenie formularza przed atakiem typu csrf,
- `{{ form.as_p }}` – automatyczne wyświetlenie pól formularza w akapitach,
- `{% url 'czat:index' %}` – wstawienie adresu do odnośnika: w cudzysłowach podajemy przestrzeń nazw naszej aplikacji (`app_name`), a później nazwę widoku (`name`) zdefiniowane w pliku `czat/urls.py`,
- `{{ user.username }}` – tak wyświetlamy nazwę zalogowanego użytkownika.

Komunikaty zwrotne przygotowane dla użytkownika w widokach wyświetlimy po uzupełnieniu szablonu `index.html`. Po znaczniku `<h1>` wstawiamy poniższy kod:

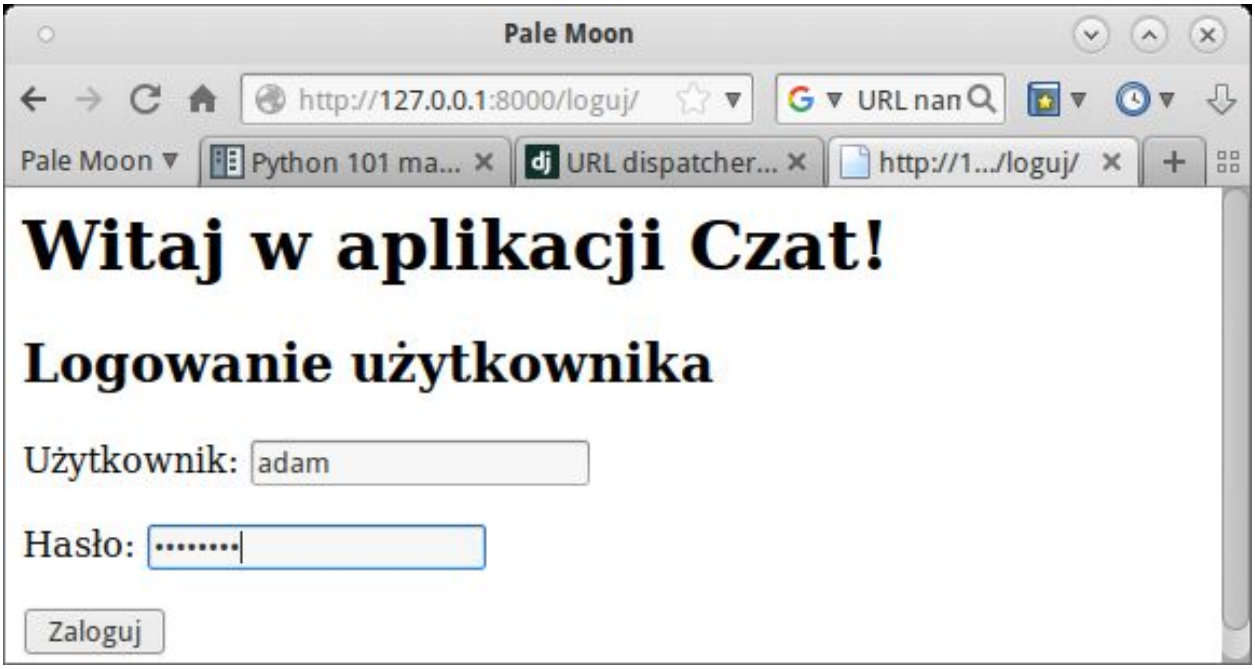
```
      {% if messages %}
        <ul>
        {% for komunikat in messages %}
           <li>{{ komunikat|capfirst }}</li>
        {% endfor %}
        </ul>
      {% endif %}
```

- `{% if messages %}` – sprawdzamy, czy mamy jakieś komunikaty,
- `{% for komunikat in messages %}` – w pętli pobieramy kolejne komunikaty...
- `{{ komunikat|capfirst }}` – i wyświetlamy z dużej litery za pomocą filtra.

Mapowanie adresów URL na widoki – w pliku `czat/urls.py` dopisujemy reguły:

```
10      url(r'^loguj/$', views.loguj, name='loguj'),
11      url(r'^wyloguj/$', views.wyloguj, name='wyloguj'),
```

Działanie dodanych funkcji testujemy pod adresami: `127.0.0.1:8000/loguj` i `127.0.0.1:8000/wyloguj`. Używamy nazw i haseł utworzonych wcześniej użytkowników. Przykładowy formularz wygląda tak:



Ćwiczenie

Adresów logowania i wylogowywania nikt nie wpisuje ręcznie. Wstaw kod generujący odpowiednie linki do szablonu strony głównej po bloku wyświetlającym komunikaty. Użytkownik niezalogowany powinien zobaczyć odnośnik *Zaloguj*, użytkownik zalogowany – *Wyloguj*. Przykładowe działanie stron może wyglądać tak:

Dodawanie wiadomości

Chcemy, by zalogowani użytkownicy mogli dodawać wiadomości, a także przeglądać wiadomości innych.

Zaczynamy od dodania **widoku** o nazwie np. `wiadomosci()`. Do pliku `views.py` dodajemy import i kod funkcji:

```
10  from czat.models import Wiadomosc
```

```
40  def wiadomosci(request):
41      """Dodawanie i wyświetlanie wiadomości"""
42      if request.method == 'POST':
43          tekst = request.POST.get('tekst', '')
44          if not 0 < len(tekst) <= 250:
45              messages.error(
46                  request,
47                  "Wiadomość nie może być pusta, może mieć maks. 250 znaków!")
48          else:
```


Pale Moon
http://127.0.0.1:8000/
URL nan
Pale Moon
Python 101 ma...
URL dispatcher...
http://1...1:8000/
Witaj w aplikacji Czat!
Jesteś zalogowany jako adam
Wyloguj


Pale Moon
http://127.0.0.1:8000/
URL nan
Pale Moon
Python 101 ma...
URL dispatcher...
http://1...1:8000/
Witaj w aplikacji Czat!
Zostałeś wylogowany!
Zaloguj się

```
            wiadomosc = Wiadomosc(
                tekst=tekst,
                autor=request.user)
            wiadomosc.save()
            return redirect(reverse('czat:wiadomosci'))

    wiadomosci = Wiadomosc.objects.all()
    kontekst = {'wiadomosci': wiadomosci}
    return render(request, 'czat/wiadomosci.html', kontekst)
```

Obsługa żądania typu GET (wyświetlenie wiadomości i formularza):

- `wiadomosci = Wiadomosc.objects.all()` – pobieramy wszystkie wiadomości z bazy, używając wbudowanego w Django systemu ORM.
- `return render(request, 'czat/wiadomosci.html', kontekst)` – zwracamy szablon, któremu przekazujemy słownik `kontekst` zawierający wiadomości.

Obsługa żądania typu POST (przesłanie danych z formularza):

- `tekst = request.POST.get('tekst', '')` – wiadomość pobieramy ze słownika `request.POST` za pomocą metody `get('tekst', '')`, pierwszy argument to nazwa pola formularza użytego w szablonie, drugi argument to wartość domyślna, jeśli pole będzie niedostępne.
- `if not 0 < len(tekst) <= 250:` – sprawdzenie minimalnej i maksymalnej długości wiadomości,
- `Wiadomosc(tekst=tekst, autor=request.user)` – utworzenie instancji wiadomości za pomocą konstruktora modelu, któremu przekazujemy wartości wymaganych pól,
- `wiadomosc.save()` – zapisanie nowej wiadomości w bazie.

Szablon zapisany w pliku `templates/czat/wiadomosci.html` będzie wyświetlał komunikaty zwrotne, np. błędy, a także formularz dodawania i listę wiadomości:

```
<!-- templates/czat/wiadomosci.html -->
<html>
  <head></head>
  <body>
    <h1>Witaj w aplikacji Czat!</h1>

    {% if messages %}
      <ul>
        {% for komunikat in messages %}
           <li>{{ komunikat|capfirst }}</li>
        {% endfor %}
        </ul>
    {% endif %}

    <h2>Dodaj wiadomość</h2>
    <form action="." method="POST">
      {% csrf_token %}
      <input type="text" name="tekst" />
      <input type="submit" value="Zapisz" />
    </form>

    <h2>Lista wiadomości:</h2>
    <ol>
      {% for wiadomosc in wiadomosci %}
        <li>
          <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
```

```
27            <br /> {{ wiadomosc.tekst }}
28          </li>
29        {% endfor %}
30      </ol>
31
32    </body>
33  </html>
```

- `<input type="textńame="tekst"/>` – "ręczne" przygotowanie formularza, czyli wstawienie kodu HTML pola do wprowadzania tekstu wiadomości,
- `{{ wiadomosc.tekst }}` – wyświetlenie właściwości obiektu przekazanego w kontekście.

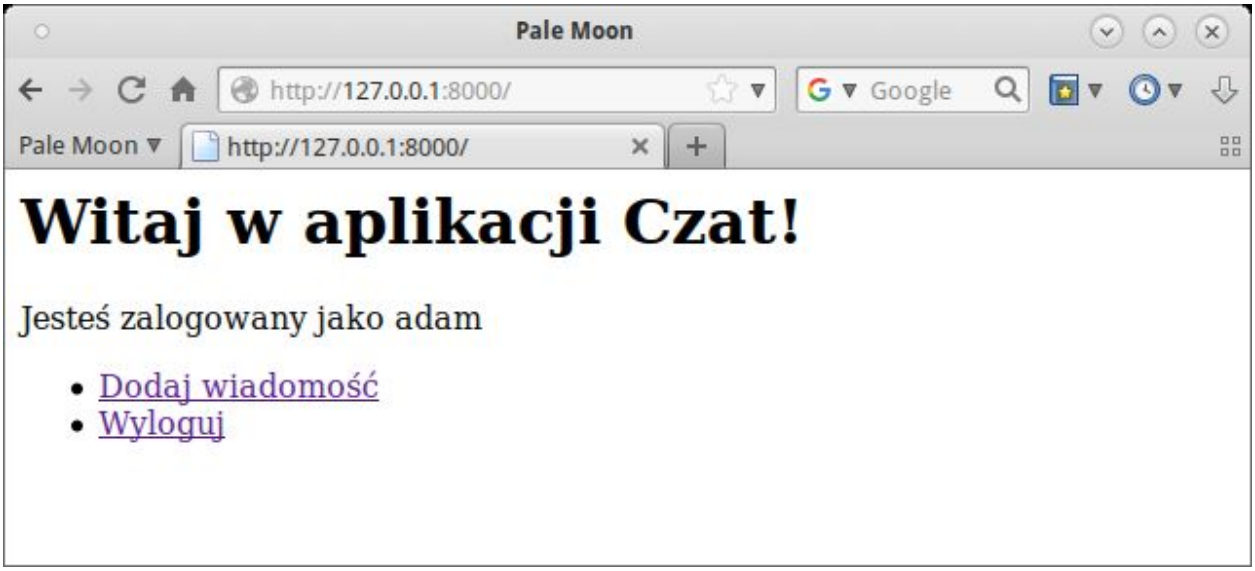
Adres URL, obsługiwany przez widok `wiadomosci()`, definiujemy w pliku `czat/urls.py`, nadając mu nazwę *wiadomosci*:

```
12      url(r'^wiadomosci/$', views.wiadomosci, name='wiadomosci'),
```

Ćwiczenie

- W szablonie widoku strony głównej dodaj link "Dodaj wiadomość" dla zalogowanych użytkowników.
- W szablonie wiadomości dodaj link "Strona główna".
- Zaloguj się i przetestuj wyświetlanie[1] i dodawanie wiadomości pod adresem *127.0.0.1:8000/wiadomosci/*. Sprawdź, co się stanie po wysłaniu pustej wiadomości.

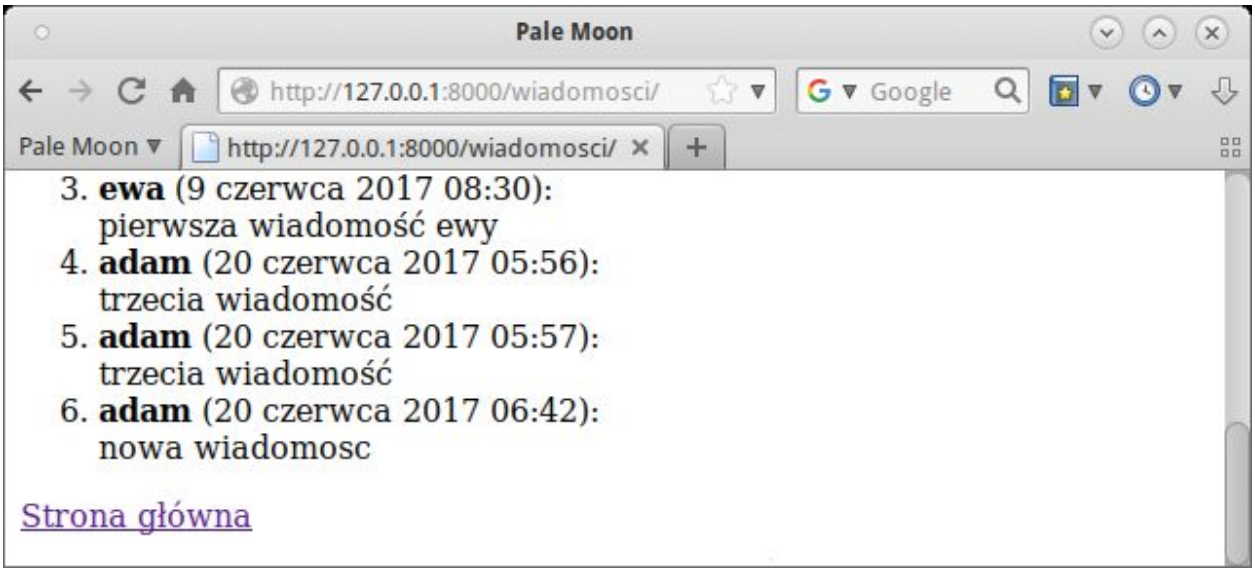
Poniższe zrzuty prezentują efekty naszej pracy:



Materiały

1. O Django http://pl.wikipedia.org/wiki/Django_(informatyka)
2. Strona projektu Django https://www.djangoproject.com/

[1] Jeżeli nie dodałeś do tej pory żadnej wiadomości, lista na początku będzie pusta.



3. Co to jest framework? http://pl.wikipedia.org/wiki/Framework
4. Co nieco o HTTP i żądaniach GET i POST http://pl.wikipedia.org/wiki/Http

Źródła:

- `czat1.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Czat (cz. 2)

Dodawanie, edycja, usuwanie czy przeglądanie danych zgromadzonych w bazie są typowymi czynnościami w aplikacjach internetowych. Utworzony w scenariuszu *Czat (cz. 1)* kod ilustruje "ręczną" obsługę żądań GET i POST, w tym tworzenie formularzy, walidację danych itp. Django zawiera jednak gotowe mechanizmy, których użycie skraca i ulepsza pracę programisty eliminując potencjalne błędy.

Będziemy rozwijać kod uzyskany po zrealizowaniu punktów **8.1.1 – 8.1.10** scenariusza *Czat (cz. 1)*. Pobierz więc `archiwum` i rozpakuj w katalogu domowym użytkownika. Następnie wydaj polecenia:

```
~$ source pve3/bin/activate
(pve3) ~$ cd czat2
(pve3) ~/czat2$ python manage.py check
```

Ostrzeżenie: Przypominamy, że pracujemy w wirtualnym środowisku Pythona z zainstalowanym frameworkiem Django, które powinno znajdować się w katalogu `pve3`. Zobacz w scenariuszu Czat (cz. 1), jak utworzyć takie *środowisko*.

Rejestrowanie

Na początku zajmiemy się obsługą użytkowników. Umożliwimy im samodzielne zakładanie kont w serwisie, logowanie i wylogowywanie się. Inaczej niż w cz. 1 zadania te zrealizujemy za pomocą tzw. widoków wbudowanych opartych na klasach (ang. class-based generic views).

Na początku pliku `czat2/czat/urls.py` importujemy formularz tworzenia użytkownika (`UserCreationForm`) oraz wbudowany widok przeznaczony do dodawania danych (`CreateView`):

```
from django.contrib.auth.forms import UserCreationForm
from django.views.generic.edit import CreateView
```

Następnie do listy `urlpatterns` dopisujemy:

```
    url(r'^rejestruj/', CreateView.as_view(
        template_name='czat/rejestruj.html',
        form_class=UserCreationForm,
```

Powyższy kod łączy adres URL */rejestruj* z wywołaniem widoku wbudowanego jako funkcji `CreateView.as_view()`. Przekazujemy jej trzy parametry:

- `template_name` – szablon, który zostanie użyty do zwrócenia odpowiedzi;
- `form_class` – formularz, który zostanie przekazany do szablonu;
- `success_url` – adres, na który nastąpi przekierowanie w przypadku braku błędów (np. po udanej rejestracji).

Teraz tworzymy szablon formularza rejestracji, który zapisać należy w pliku `templates/czat/rejestruj.html`:

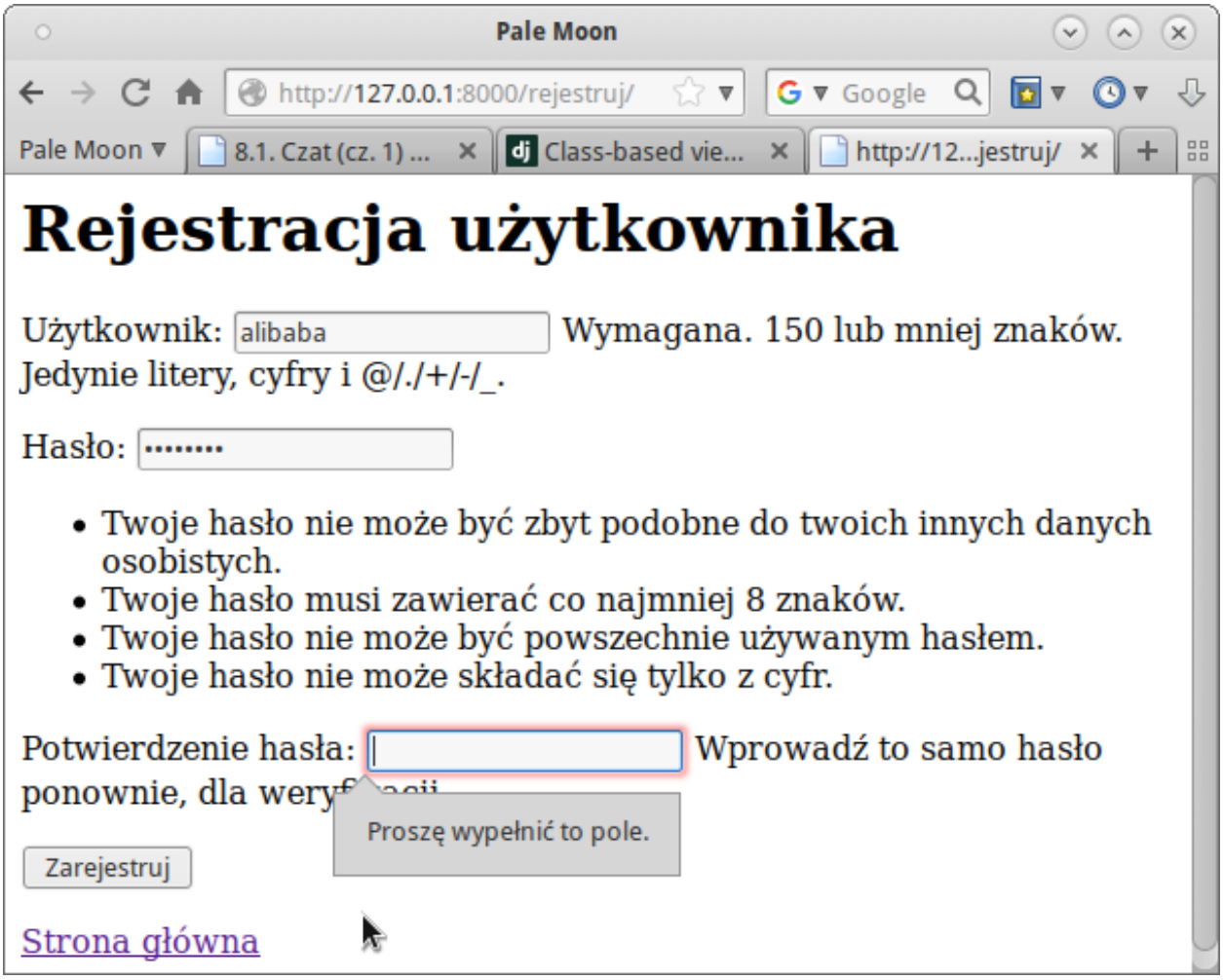
```
<!-- templates/czat/rejestruj.html -->
<html>
  <body>
    <h1>Rejestracja użytkownika</h1>
    {% if user.is_authenticated %}
      <p>Jesteś już zarejestrowany jako {{ user.username }}.
      <br /><a href="/">Strona główna</a></p>
    {% else %}
    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zarejestruj</button>
    </form>
    {% endif %}
  </body>
</html>
```

Na koniec wstawimy link na stronie głównej, a więc uzupełniamy plik `index.html`:

```
<!-- templates/czat/index.html -->
<html>
  <head></head>
  <body>
    <h1>Witaj w aplikacji Czat!</h1>

    {% if user.is_authenticated %}
      <p>Jesteś zalogowany jako {{ user.username }}.</p>
    {% else %}
      <p><a href="{% url 'czat:rejestruj' %}">Zarejestruj się</a></p>
    {% endif %}

  </body>
</html>
```

Ćwiczenie: dodaj link do strony głównej w szablonie `rejestruj.html`.

Uruchom aplikację (`python manage.py runserver`) i przetestuj dodawanie użytkowników: spróbuj wysłać niepełne dane, np. bez hasła; spróbuj dodać dwa razy tego samego użytkownika.



Wy(logowanie)

Na początku pliku `urls.py` aplikacji dopisujemy wymagany import:

```
8  from django.core.urlresolvers import reverse_lazy
9  from django.contrib.auth import views as auth_views
```

– a następnie:

```
22     url(r'^loguj/', auth_views.login,
23         {'template_name': 'czat/loguj.html'},
24         name='loguj'),
25     url(r'^wyloguj/', auth_views.logout,
26         {'next_page': reverse_lazy('czat:index')},
27         name='wyloguj'),
```

Widać, że z adresami */loguj* i */wyloguj* wiążemy wbudowane w Django widoki `login` i `logout` importowane z modułu `django.contrib.auth.views`. Jedynym nowym parametrem jest `next_page`, za pomocą którego

wskazujemy stronę wyświetlaną po wylogowaniu (`reverse_lazy('czat:index')`).

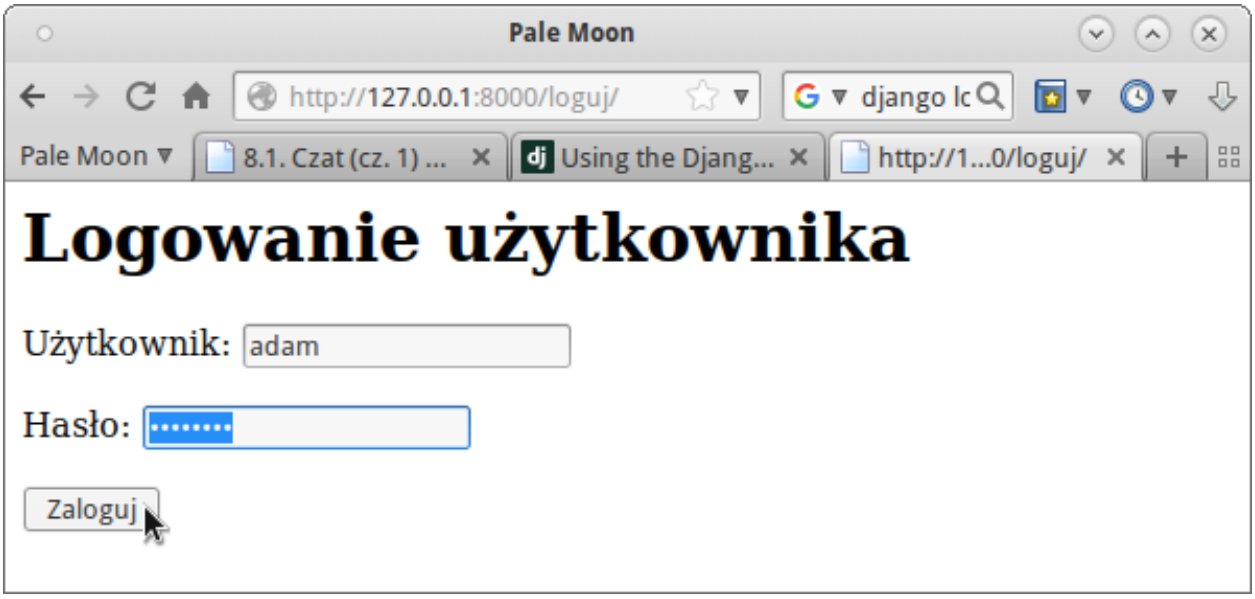
Logowanie wymaga szablonu `loguj.html`, który tworzymy i zapisujemy w podkatalogu `templates/czat`:

```
<!-- templates/czat/loguj.html -->
<html>
  <body>
    <h1>Logowanie użytkownika</h1>
    {% if user.is_authenticated %}
      <p>Jesteś już zalogowany jako {{ user.username }}.
      <br /><a href="/">Strona główna</a></p>
    {% else %}
    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zaloguj</button>
    </form>
    {% endif %}
  </body>
</html>
```

Musimy jeszcze określić stronę, na którą powinien zostać przekierowany użytkownik po udanym zalogowaniu. W tym wypadku na końcu pliku `czat2/settings.py` definiujemy wartość zmiennej `LOGIN_REDIRECT_URL`:

```
# czat2/settings.py

from django.core.urlresolvers import reverse_lazy
LOGIN_REDIRECT_URL = reverse_lazy('czat:index')
```

Ćwiczenie: Uzupełnij plik `index.html` o linki służące do logowania i wylogowania.



Lista wiadomości

Chcemy, by zalogowani użytkownicy mogli przeglądać wiadomości wszystkich użytkowników, zmieniać, usuwać i dodawać własne. Najprostszy sposób to skorzystanie z widoków wbudowanych.

Informacja: Django oferuje wbudowane widoki przeznaczone do typowych operacji:

- DetailView i ListView – (ang. *generic display view*) widoki przeznaczone do prezentowania szczegółów i listy danych;
- FormView, CreateView, UpdateView i DeleteView – (ang. *generic editing views*) widoki przeznaczone do wyświetlania formularzy ogólnych, w szczególności służących dodawaniu, uaktualnianiu, usuwaniu obiektów (danych).

Do wyświetlania listy wiadomości użyjemy klasy `ListView`. Do pliku `urls.py` dopisujemy importy:

```
from django.contrib.auth.decorators import login_required
from django.views.generic import ListView
from czat.models import Wiadomosc
```

– i wiążemy adres */wiadomosci* z wywołaniem widoku:

```
    url(r'^wiadomosci/', login_required(
        ListView.as_view(
            model=Wiadomosc,
            context_object_name='wiadomosci',
            paginate_by=2)),
        name='wiadomosci'),
```

Zakładamy, że wiadomości mogą oglądać tylko użytkownicy zalogowani. Dlatego całe wywołanie widoku umieszczamy w funkcji `login_required()`.

W funkcji `ListView.as_view()` podajemy kolejne parametry modyfikujące działanie widoków:

- `model` – podajemy model, którego dane zostaną pobrane z bazy;
- `context_object_name` – pozwala zmienić domyślną nazwę (object_list) listy obiektów przekazanych do szablonu;
- `paginate_by`– pozwala określić ilość obiektów wyświetlanych na stronie.

Na końcu pliku `czat2/settings.py` określamy adres logowania, na który przekierowani zostaną niezalogowani użytkownicy, którzy próbowaliby zobaczyć listę wiadomości:

```
# czat2/settings.py

LOGIN_URL = reverse_lazy('czat:loguj')
```

Potrzebujemy szablonu, którego Django szuka pod domyślną nazwą *<nazwa modelu>_list.html*, czyli w naszym przypadku tworzymy plik `czat/wiadomosc_list.html`:

```
<!-- czat/wiadomosc_list.html -->
<html>
  <body>
    <h1>Wiadomości</h1>

    <h2>Lista wiadomości:</h2>
    <ol>
      {% for wiadomosc in wiadomosci %}
      <li>
        <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
        <br /> {{ wiadomosc.tekst }}
      </li>
      {% endfor %}
```

```
    </ol>

    {% if is_paginated %}
      <p>
      {% if page_obj.has_previous %}
        <a href="?page={{ page_obj.previous_page_number }}">Poprzednie</a>
      {% endif %}
        Strona {{ page_obj.number }} z {{ page_obj.paginator.num_pages }}.
      </span>
      {% if page_obj.has_next %}
        <a href="?page={{ page_obj.next_page_number }}">Następne</a>
      {% endif %}
      </p>
    {% endif %}

    <p><a href="{% url 'czat:index' %}">Strona główna</a></p>

  </body>
</html>
```

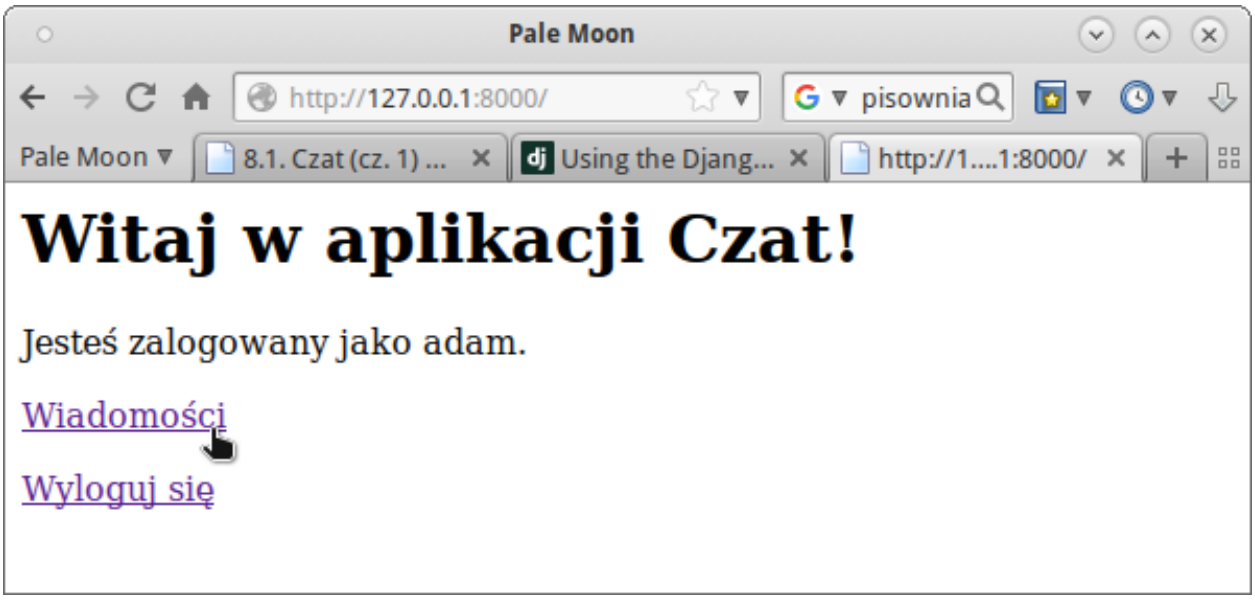
Kolejne wiadomości odczytujemy i wyświetlamy w pętli przy użyciu tagu `{% for %}`. Dostęp do właściwości obiektów umożliwia operator kropki, np.: `{{ wiadomosc.autor.username }}`.

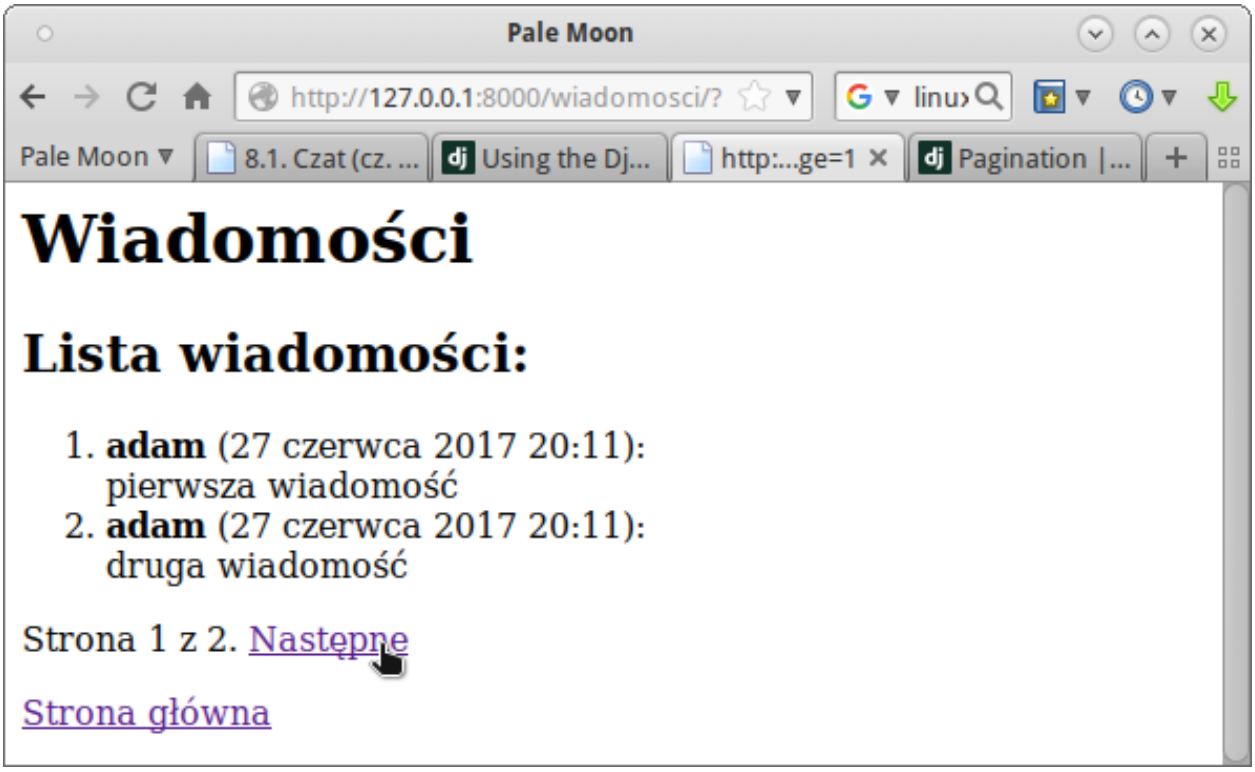
Linki nawigacyjne tworzymy w instrukcji warunkowej `{% if is_paginated %}`. Obiekt `page_obj` zawiera następujące właściwości:

- `has_previous` – zwraca `True`, jeżeli jest poprzednia strona;
- `previous_page_number` – numer poprzedniej strony;
- `next_page_number` – numer następnej strony;
- `number` – numer aktualnej strony;
- `paginator.num_pages` – ilość wszystkich stron.

Numer strony do wyświetlenia przekazujemy w zmiennej `page` adresu URL.

Ćwiczenie: Dodaj link do strony wyświetlającej wiadomości na stronie głównej dla zalogowanych użytkowników.





Dodawanie wiadomości

Zadanie to zrealizujemy wykorzystując widok `CreateView`. Aby ułatwić dodawanie wiadomości **dostosujemy klasę widoku** tak, aby użytkownik nie musiał wprowadzać pola autor.

Na początek dopiszemy w pliku `urls.py` skojarzenie adresu URL *wiadomosc/* z wywołaniem klasy `CreateView` jako funkcji:

```
34      url(r'^dodaj/$', login_required(
35          views.DodajWiadomosc.as_view(),
36          login_url='/loguj'),
37          name='dodaj'),
```

Dalej kodujemy w pliku `views.py`. Na początku dodajemy importy:

```
6   from django.views.generic.edit import CreateView
7   from czat.models import Wiadomosc
8   from django.utils import timezone
9   from django.contrib import messages
```

```
19  class DodajWiadomosc(CreateView):
20      model = Wiadomosc
21      fields = ['tekst', 'data_pub']
22      context_object_name = 'wiadomosci'
23      success_url = '/dodaj'
24
25      def get_initial(self):
26          initial = super(DodajWiadomosc, self).get_initial()
27          initial['data_pub'] = timezone.now()
28          return initial
```

```

    def get_context_data(self, **kwargs):
        context = super(DodajWiadomosc, self).get_context_data(**kwargs)
        context['wiadomosci'] = Wiadomosc.objects.all()
        return context

    def form_valid(self, form):
        wiadomosc = form.save(commit=False)
        wiadomosc.autor = self.request.user
        wiadomosc.save()
        messages.success(self.request, "Dodano wiadomość!")
        return super(DodajWiadomosc, self).form_valid(form)
```

Tworzymy klasę opartą na widoku ogólnym (`class DodajWiadomosc(CreateView)`), określamy jej podstawowe właściwości i nadpisujemy wybrane metody:

- `fields` – pozwala wskazać pola, które mają znaleźć się na formularzu;
- `get_initial()` – metoda pozwala ustawić domyślne wartości dla wybranych pól. Wykorzystujemy ją do zainicjowania pola `data_pub` aktualna datą: `initial['data_pub'] = timezone.now()`.
- `get_context_data()` – metoda pozwala przekazać do szablonu dodatkowe dane, w tym wypadku jest to lista wszystkich wiadomości: `context['wiadomosci'] = Wiadomosc.objects.all()`.
- `form_valid()` – metoda, która sprawdza poprawność przesłanych danych i zapisuje je w bazie:
  - `wiadomosc = form.save(commit=False)` – tworzymy obiekt wiadomości, ale go nie zapisujemy;
  - `wiadomosc.autor = self.request.user` – uzupełniamy dane autora;
  - `wiadomosc.save()` – zapisujemy obiekt;
  - `messages.success(self.request, "Dodano wiadomość!")` – przygotowujemy komunikat, który wyświetlony zostanie po dodaniu wiadomości.

Domyślny szablon dodawania danych nazywa się *<nazwa modelu>_form.html*. W nowym pliku wstawiamy poniższą treść i zapisujemy pod nazwą `templates/czat/wiadomosc_form.html`:

```
<!-- czat/wiadomosc_form.html -->
<html>
  <body>
    <h1>Wiadomości</h1>

    <h2>Dodaj wiadomość:</h2>
    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zapisz</button>
    </form>

    <h2>Lista wiadomości:</h2>
    <ol>
      {% for wiadomosc in wiadomosci %}
      <li>
        <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
        <br /> {{ wiadomosc.tekst }}
      </li>
      {% endfor %}
    </ol>

```

```
    <p><a href="{% url 'czat:index' %}">Strona główna</a></p>

  </body>
</html>
```

W szablonie `templates/czat/wiadomosc_list.html` wstawimy jeszcze po nagłówku `<h1>` kod wyświetlający komunikaty:

```
    {% if messages %}
      <ul>
        {% for komunikat in messages %}
          <li>{{ komunikat|capfirst }}</li>
        {% endfor %}
      </ul>
    {% endif %}
```

Ostrzeżenie: W pliku `czat/models.py` trzeba usunąć parametr `auto_now_add=True` z definicji pola `data_pub`, aby użytkownik mógł modyfikować datę dodania wiadomości w formularzu.

Ćwiczenie: Jak zwykle, umieść link do dodawanie wiadomości na stronie głównej.

Edycja wiadomości

Widok pozwalający na edycję wiadomości i jej aktualizację dostępny będzie pod adresem **/edytuj/id_wiadomości**, gdzie **id_wiadomosci** będzie identyfikatorem obiektu do zaktualizowania. Zaczniemy od uzupełnienia pliku `urls.py`:

```
    url(r'^edytuj/(?P<pk>\d+)/', login_required(
        views.EdytujWiadomosc.as_view(),
        login_url='/loguj'),
        name='edytuj'),
```

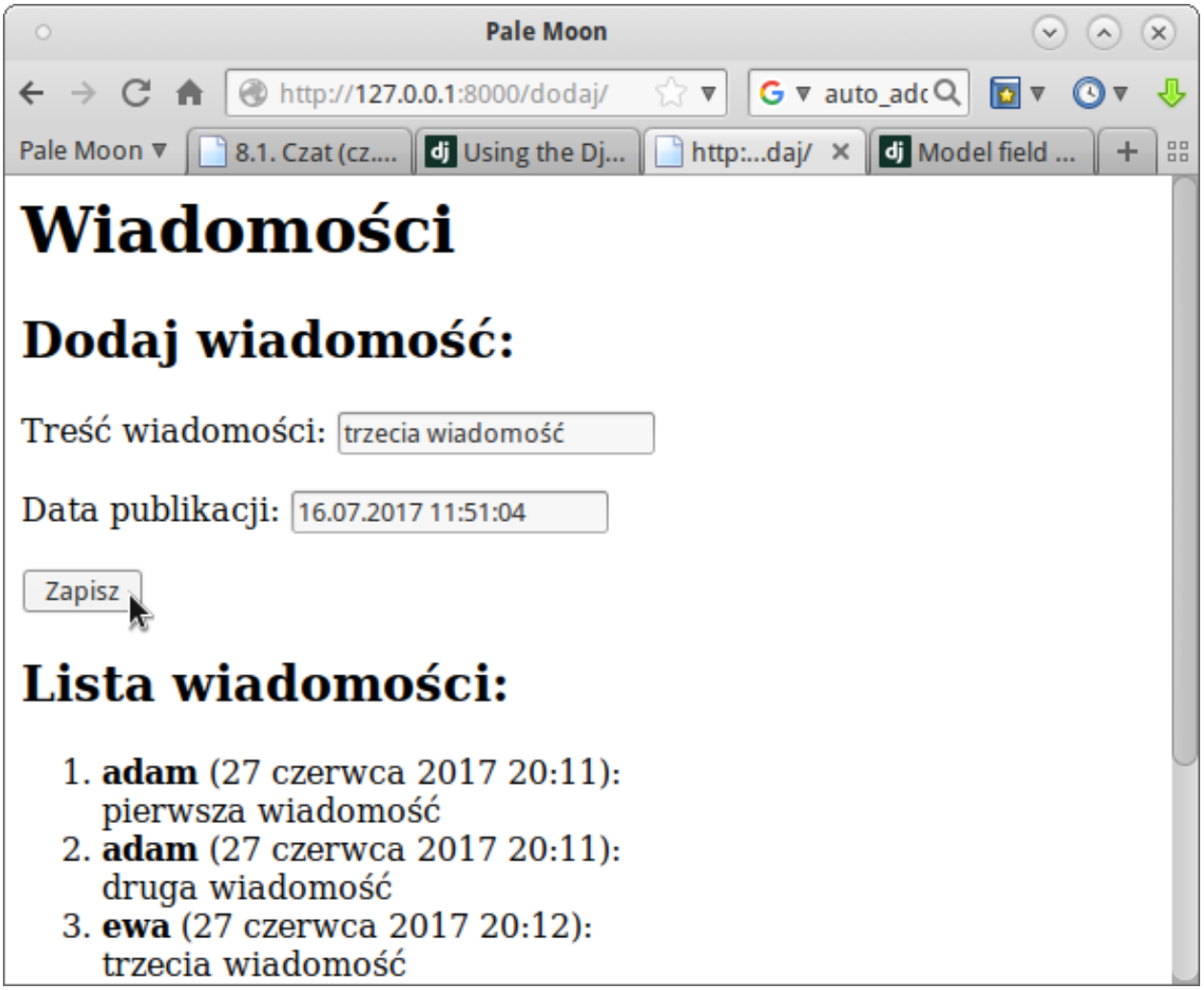
Nowością w powyższym kodzie są wyrażenia regularne definiujące adresy z dodatkowym parametrem, np. `r'^edytuj/(?P<pk>\d+)/'`. Część `/(?P<pk>\d+)` oznacza, że oczekujemy 1 lub więcej cyfr (`\d+`), które zostaną zapisane w zmiennej o nazwie `pk` (`?P<pk>`) – nazwa jest tu skrótem od ang. wyrażenia *primary key*, co znaczy "klucz główny". Zmienna ta zawierać będzie identyfikator wiadomości i dostępna będzie w klasie widoku, który obsłuży edycję wiadomości.

Na początku pliku `views.py` importujemy więc potrzebny widok:

```
from django.views.generic.edit import UpdateView
```

Dalej tworzymy klasę `EdytujWiadomosc`, która dziedziczy, czyli dostosowuje wbudowany widok `UpdateView`:

```
class EdytujWiadomosc(UpdateView):
    model = Wiadomosc
    from czat.forms import EdytujWiadomoscForm
    form_class = EdytujWiadomoscForm
    context_object_name = 'wiadomosci'
    template_name = 'czat/wiadomosc_form.html'
    success_url = '/wiadomosci'

    def get_context_data(self, **kwargs):
        context = super(EdytujWiadomosc, self).get_context_data(**kwargs)
```


Pale Moon
http://127.0.0.1:8000/dodaj/
auto_adc
Pale Moon
8.1. Czat (cz....
Using the Dj...
http:...daj/
Model field ...
Wiadomości
Dodaj wiadomość:
Treść wiadomości:
trzecia wiadomość
Data publikacji:
16.07.2017 11:51:04
Zapisz
Lista wiadomości:
1. adam (27 czerwca 2017 20:11):
pierwsza wiadomość
2. adam (27 czerwca 2017 20:11):
druga wiadomość
3. ewa (27 czerwca 2017 20:12):
trzecia wiadomość

```
        context['wiadomosci'] = Wiadomosc.objects.filter(
            autor=self.request.user)
        return context

    def get_object(self, queryset=None):
        wiadomosc = Wiadomosc.objects.get(id=self.kwargs['pk'])
        return wiadomosc
```

Najważniejsza jest tu metoda `get_object()`, która pobiera i zwraca wskazaną przez identyfikator w zmiennej *pk* wiadomość: `wiadomosc = Wiadomosc.objects.get(id=self.kwargs['pk'])`. Omawianą już metodę `get_context_data()` wykorzystujemy, aby przekazać do szablonu listę wiadomości, ale tylko zalogowanego użytkownika (`context['wiadomosci'] = Wiadomosc.objects.filter(autor=self.request.user)`).

Właściwości `model`, `context_object_name`, `template_name` i `success_url` wyjaśniliśmy wcześniej. Jak widać, do edycji wiadomości można wykorzystać ten sam szablon, którego użyliśmy podczas dodawania.

Formularz jednak dostosujemy. Wykorzystamy właściwość `form_class`, której przypisujemy utworzoną w nowym pliku `forms.py` klasę zmieniającą domyślne ustawienia:

```
# -*- coding: utf-8 -*-
# czat/forms.py

from django.forms import ModelForm, TextInput
from czat.models import Wiadomosc


class EdytujWiadomoscForm(ModelForm):
    class Meta:
        model = Wiadomosc
        fields = ['tekst', 'data_pub']
        exclude = ['autor']
        widgets = {'tekst': TextInput(attrs={'size': 60})}
```

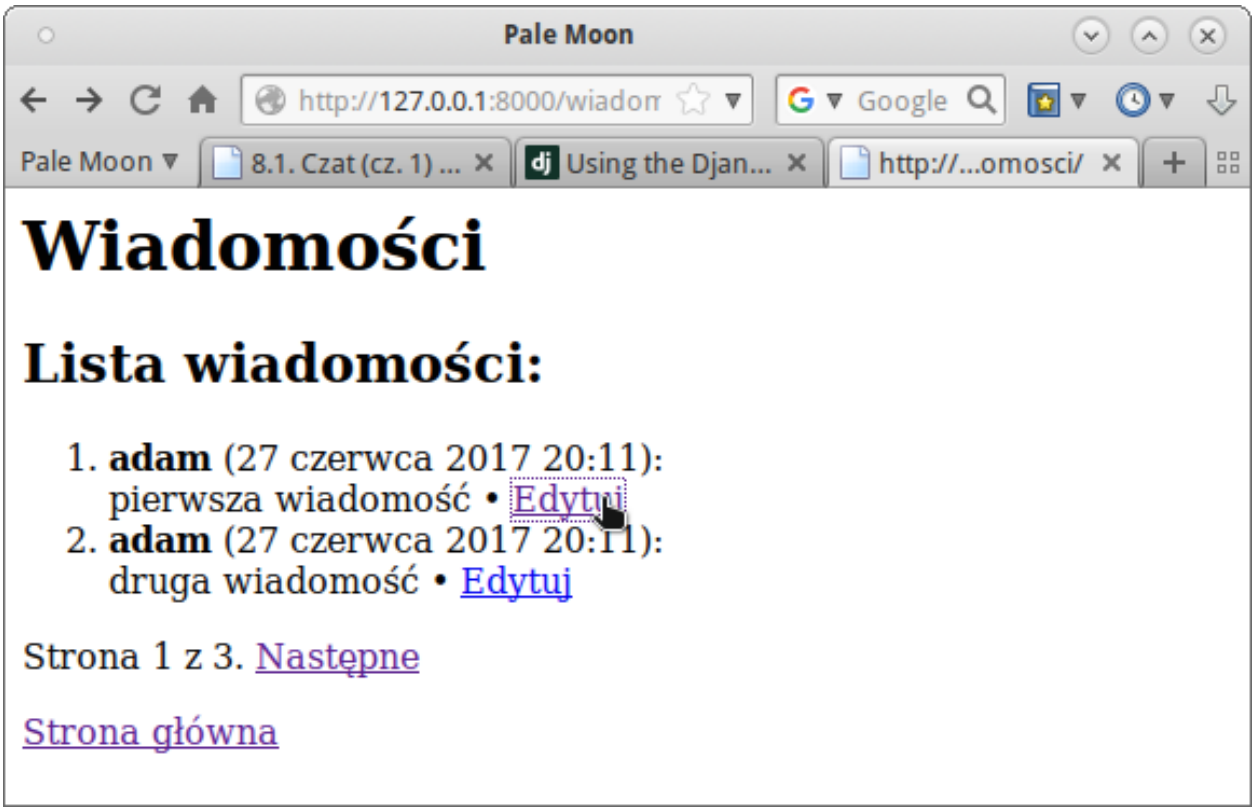
Klasa `EdytujWiadomoscForm` oparta jest na wbudowanej klasie `ModelForm`. Właściwości formularza określamy w podklasie `Meta`:

- `model` – oznacza to samo co w widokach, czyli model, dla którego tworzony jest formularz;
- `fields` – to samo co w widokach, lista pól do wyświetlenia;
- `exclude` – opcjonalnie lista pól do pominięcia;
- `widgets` – słownik, którego klucze oznaczają pola danych, a ich wartości odpowiadające im w formularzach HTML typy pól i ich właściwości, np. rozmiar.

Żeby przetestować aktualizowanie wiadomości, w szablonie `wiadomosc_list.html` trzeba wygenerować linki *Edytuj* dla wiadomości utworzonych przez zalogowanego użytkownika. Wstaw w odpowiednie miejsce szablonu, tzn po tagu wyświetlającym tekst wiadomości (`{{ wiadomosc.tekst }}`) poniższy kod:

```
        {% if wiadomosc.autor.username == user.username %}
          &bull; <a href="{% url 'czat:edytuj' wiadomosc.id %}">Edytuj</a>
        {% endif %}
```

Ćwiczenie: Ten sam link "Edytuj" umieść również w szablonie dodawania.



Usuwanie wiadomości

Usuwanie danych realizujemy za pomocą widoku `DeleteView`, który importujemy na początku pliku `urls.py`:

```
13 from django.views.generic import DeleteView
```

Podobnie, jak w przypadku edycji, usuwanie powiążemy z adresem URL zawierającym identyfikator wiadomości */usun/id_wiadomości*. W pliku `urls.py` dopisujemy:

```
42     url(r'^usun/(?P<pk>\d+)/', login_required(
43         DeleteView.as_view(
44             model=Wiadomosc,
45             template_name='czat/wiadomosc_usun.html',
46             success_url='/wiadomosci'),
47         login_url='/loguj'),
48         name='usun'),
```

Warto zwrócić uwagę, że podobnie jak w przypadku listy wiadomości, o ile wystarcza nam domyślna funkcjonalność widoku wbudowanego, nie musimy niczego implementować w pliku `views.py`.

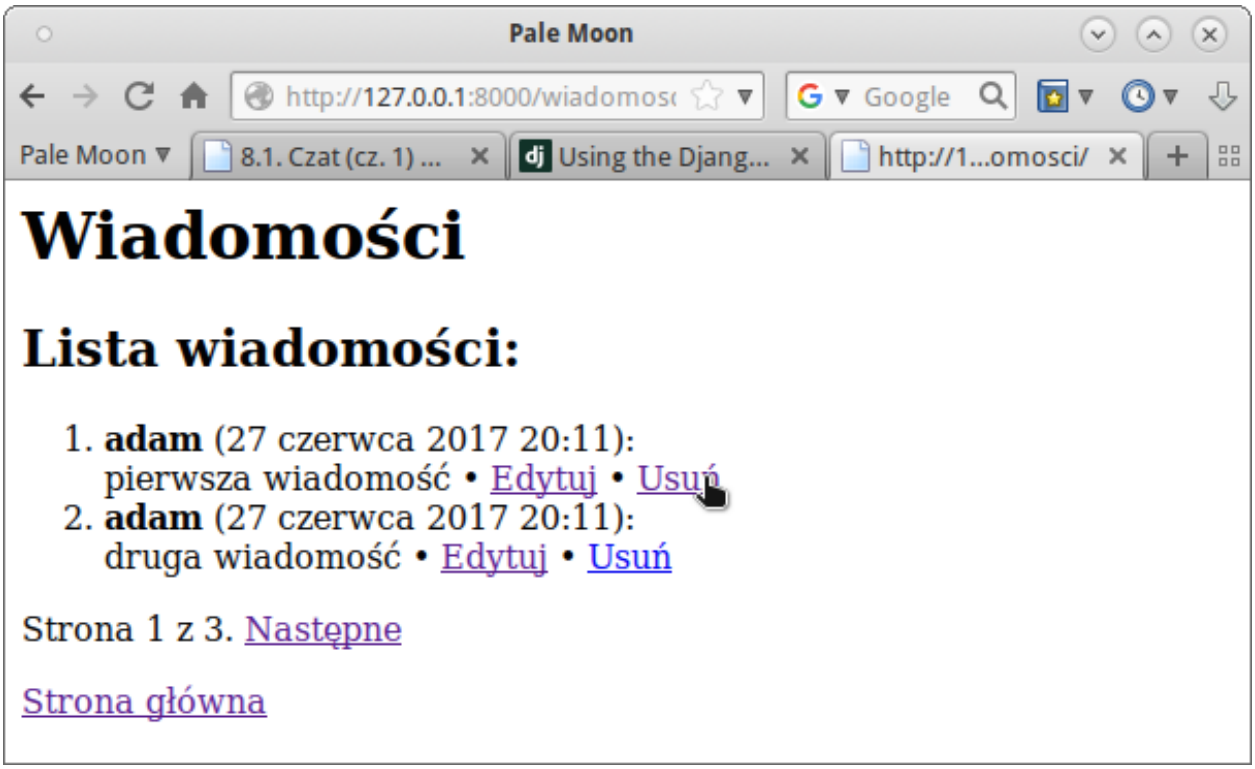
Domyślny szablon dla tego widoku przyjmuje nazwę *<nazwa-modelu>_confirm_delete.html*, dlatego uprościliśmy jego nazwę we właściwości `template_name`. Tworzymy więc plik `wiadomosc_usun.html`:

```
1 <!-- templates/czat/wiadomosc_usun.html -->
2 <html>
3   <body>
4     <h1>Wiadomości</h1>
5
6     <h2>Usuń wiadomość</h2>
```

```
    <form method="POST">
      {% csrf_token %}
      <p>Czy na pewno chcesz usunąć wiadomość:<br /><i>{{ object }}</i>?</p>
      <button type="submit">Usuń</button>
    </form>

    <p><a href="{% url 'czat:index' %}">Strona główna</a></p>
  </body>
</html>
```

Tag `{{ object }}` zostanie zastąpiony treścią wiadomości zwróconą przez funkcję "autoprezentacji" `__str__()` modelu.

Ćwiczenie: Wstaw link "Usuń" (`&bull; <a href="{% url 'czat:usun' wiadomosc.id %}">Usuń</a>`) za linkiem "Edytuj" w szablonach wyświetlających listę wiadomości.



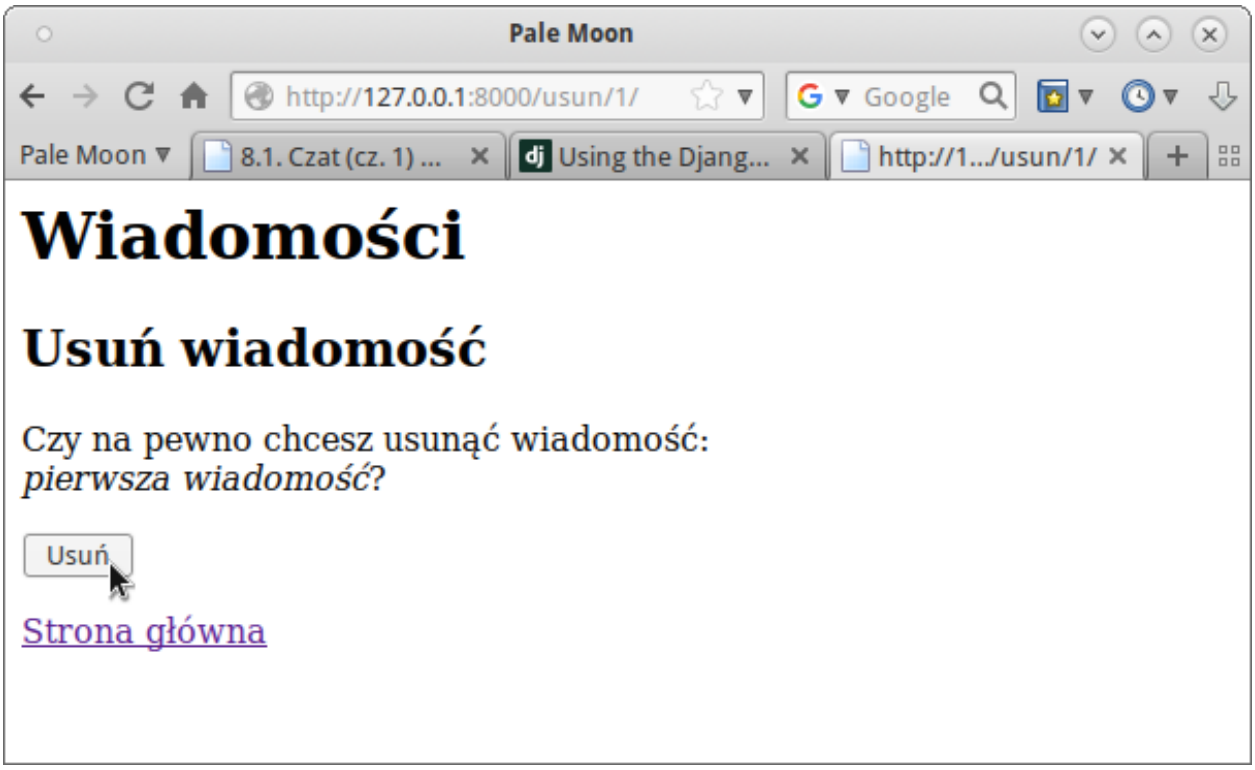
Materiały

1. O Django http://pl.wikipedia.org/wiki/Django_(informatyka)
2. Strona projektu Django https://www.djangoproject.com/
3. Co to jest framework? http://pl.wikipedia.org/wiki/Framework
4. Co nieco o HTTP i żądaniach GET i POST http://pl.wikipedia.org/wiki/Http

Źródła:

- `czat2.zip`

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3



Autorzy Patrz plik "Autorzy"

Czat (cz. 3)

Poniższy materiał koncentruje się na obsłudze szablonów (ang. *templates*) wykorzystywanych w Django. Stanowi kontynuację projektu zrealizowanego w scenariuszu *Czat (cz. 2)*.

Na początku pobierz `archiwum` z potrzebnymi plikami i rozpakuj je w katalogu domowym użytkownika. Następnie wydaj polecenia:

```
~$ source pve3/bin/activate
(pve3) ~$ cd czat3
(pve3) ~/czat3$ python manage.py check
```

Ostrzeżenie: Przypominamy, że pracujemy w wirtualnym środowisku Pythona z zainstalowanym frameworkiem Django, które powinno znajdować się w katalogu `pve3`. Zobacz w scenariuszu Czat (cz. 1), jak utworzyć takie *środowisko*.

Szablony

Zapewne zauważyłeś, że większość kodu w szablonach i stronach HTML, które z nich powstają, powtarza się albo jest bardzo podobna. Biorąc pod uwagę schematyczną budowę stron WWW jest to nieuniknione.

Szablony, jak można było zauważyć, składają się ze **zmiennych** i **tagów**. Zmienne, które ujmowane są w podwójne nawiasy sześciokątne `{{ zmienna }}`, zastępowane są konkretnymi wartościami. Tagi z kolei, oznaczane notacją `{% tag %}`, tworzą mini-język szablonów i pozwalają kontrolować logikę budowania treści. Najważniejsze tagi, `{% if warunek %}`, `{% for wyrażenie %}`, `{% url nazwa %}` – już stosowaliśmy.

Spróbujmy uprościć i ujednolicić nasze szablony. Zacznijmy od szablonu bazowego, który umieścimy w pliku `templates/czat/baza.html`:

```
<!-- templates/czat/baza.html -->
<!DOCTYPE html>
<html lang="pl">
  <meta charset="utf-8" />
  <head>
    <title>{% block tytul %} System wiadomości Czat {% endblock tytul %}</title>
  </head>
  <body>
    <h1>{% block naglowek %} Witaj w aplikacji Czat! {% endblock %}</h1>

    {% block komunikaty %}
      {% if messages %}
        <ul>
          {% for komunikat in messages %}
            <li>{{ komunikat|capfirst }}</li>
          {% endfor %}
        </ul>
      {% endif %}
    {% endblock %}

    {% block tresc %} {% endblock %}

    {% if user.is_authenticated %}
      {% block linki1 %} {% endblock %}
      <p><a href="{% url 'czat:wyloguj' %}">Wyloguj się</a></p>
    {% else %}
      {% block linki2 %} {% endblock %}
    {% endif %}

    {% block linki3 %} {% endblock %}

  </body>
</html>
```

Jest to zwykły tekstowy dokument, zawierający schemat strony utworzony z wymaganych znaczników HTML oraz bloki zdefiniowane za pomocą tagów `{% block %}`. W pliku tym umieszczamy wspólną strukturę stron w serwisie (np. nagłówek, menu, sekcja treści, stopka itp.) oraz wydzielamy bloki, których treść będzie można zmieniać w szablonach konkretnych stron.

Wykorzystując szablon podstawowy, zmieniamy stronę główną, czyli plik `index.html`:

```
<!-- templates/czat/index.html -->
{% extends "czat/baza.html" %}

{% block naglowek %}Witaj w aplikacji Czat!{% endblock %}

{% block linki1 %}
  <p>Jesteś zalogowany jako {{ user.username }}.</p>
  <p><a href="{% url 'czat:dodaj' %}">Dodaj wiadomość</a></p>
  <p><a href="{% url 'czat:wiadomosci' %}">Lista wiadomości</a></p>
{% endblock %}

{% block linki2 %}
  <p><a href="{% url 'czat:loguj' %}">Zaloguj się</a></p>
  <p><a href="{% url 'czat:rejestruj' %}">Zarejestruj się</a></p>
{% endblock %}
```

Jak widać, szablon dziedziczy z szablonu bazowego – tag `{% extends plik_bazowy %}`. Dalej podajemy zawartość bloków, które są potrzebne na danej stronie.

Postępując na tej samej zasadzie modyfikujemy szablon rejestracji:

```
<!-- templates/czat/rejestruj.html -->
{% extends "czat/baza.html" %}

{% block naglowek %}Rejestracja użytkownika{% endblock %}

{% block tresc %}
  {% if not user.is_authenticated %}
    <form method="POST">
      {% csrf_token %}
      {{ form.as_p }}
      <button type="submit">Zarejestruj</button>
    </form>
  {% endif %}
{% endblock %}

{% block linki1 %}
  <p>Jesteś już zarejestrowany jako {{ user.username }}.</p>
{% endblock %}

{% block linki3 %}
  <p><a href="{% url 'czat:index' %}">Strona główna</a></p>
{% endblock %}
```

Ćwiczenie

Wzorując się na podanych przykładach zmień pozostałe szablony tak, aby opierały się na szablonie bazowym. Następnie przetestuj działanie aplikacji. Wygląd stron nie powinien ulec zmianie!

Dla przykładu szablon `wiadomosc_list.html` powinien wyglądać tak:

```
<!-- czat/wiadomosc_list.html -->
{% extends "czat/baza.html" %}

{% block naglowek %}Wiadomości{% endblock %}

{% block tresc %}
  <h2>Lista wiadomości:</h2>
  <ol>
    {% for wiadomosc in wiadomosci %}
    <li>
      <strong>{{ wiadomosc.autor.username }}</strong> ({{ wiadomosc.data_pub }}):
      <br /> {{ wiadomosc.tekst }}
      {% if wiadomosc.autor.username == user.username %}
        &bull; <a href="{% url 'czat:edytuj' wiadomosc.id %}">Edytuj</a>
        &bull; <a href="{% url 'czat:usun' wiadomosc.id %}">Usuń</a>
      {% endif %}
    </li>
    {% endfor %}
  </ol>

  {% if is_paginated %}
```

```
    <p>
    {% if page_obj.has_previous %}
      <a href="?page={{ page_obj.previous_page_number }}">Poprzednie</a>
    {% endif %}
      Strona {{ page_obj.number }} z {{ page_obj.paginator.num_pages }}.
    </span>
    {% if page_obj.has_next %}
      <a href="?page={{ page_obj.next_page_number }}">Następne</a>
    {% endif %}
    </p>
  {% endif %}
{% endblock %}

{% block linki3 %}
  <p><a href="{% url 'czat:index' %}">Strona główna</a></p>
{% endblock %}
```

Style CSS i obrazki

Nasze szablony zyskały na zwięzłości i przejrzystości, ale nadal pozbawione są elementarnych dla dzisiejszych stron WWW zasobów, takich jak style CSS, skrypty JavaScript czy zwykłe obrazki. Jak je dołączyć?

Przede wszystkim potrzebujemy osobnego katalogu `czat/static/czat`. W terminalu w katalogu projektu (!) wydajemy polecenia:

```
(.pve) ~/czat3$  mkdir -p czat/static/czat
(.pve) ~/czat3$  cd czat/static/czat
(.pve) ~/czat3/czat/static/czat$ mkdir css js img
```

Ostatnie polecenie tworzy podkatalogi dla różnych typów zasobów:

- `css` – arkusze stylów CSS,
- `js` – skrypty Java Script,
- `img` – obrazki.

Tworzymy przykładowy arkusz stylów CSS `style.css` i zapisujemy w katalogu `static/czat/css`:

```
body {
        margin: 10px;
        font-family: Helvetica, Arial, sans-serif;
        font-size: 12pt;
        background: lightgreen url('../img/django.png') no-repeat fixed top right;
}
a { text-decoration: none; }
a:hover { text-decoration: underline; }
a:visited { text-decoration: none; }
.clearfix { clear: both; }
h1 { font-size: 1.8em; font-weight: bold; margin-top: 20px; }
h2 { font-size: 1.4em; font-weight: bold; margin-top: 20px; }
p { font-szie: 1em; font-family: Arial, sans-serif; }
.fr { float: right; }
```

Do podkatalogu `static/czat/img` rozpakuj obrazki z pobranego `archiwum`.

Teraz musimy dołączyć style i obrazki do szablonu bazowego `baza.html`:

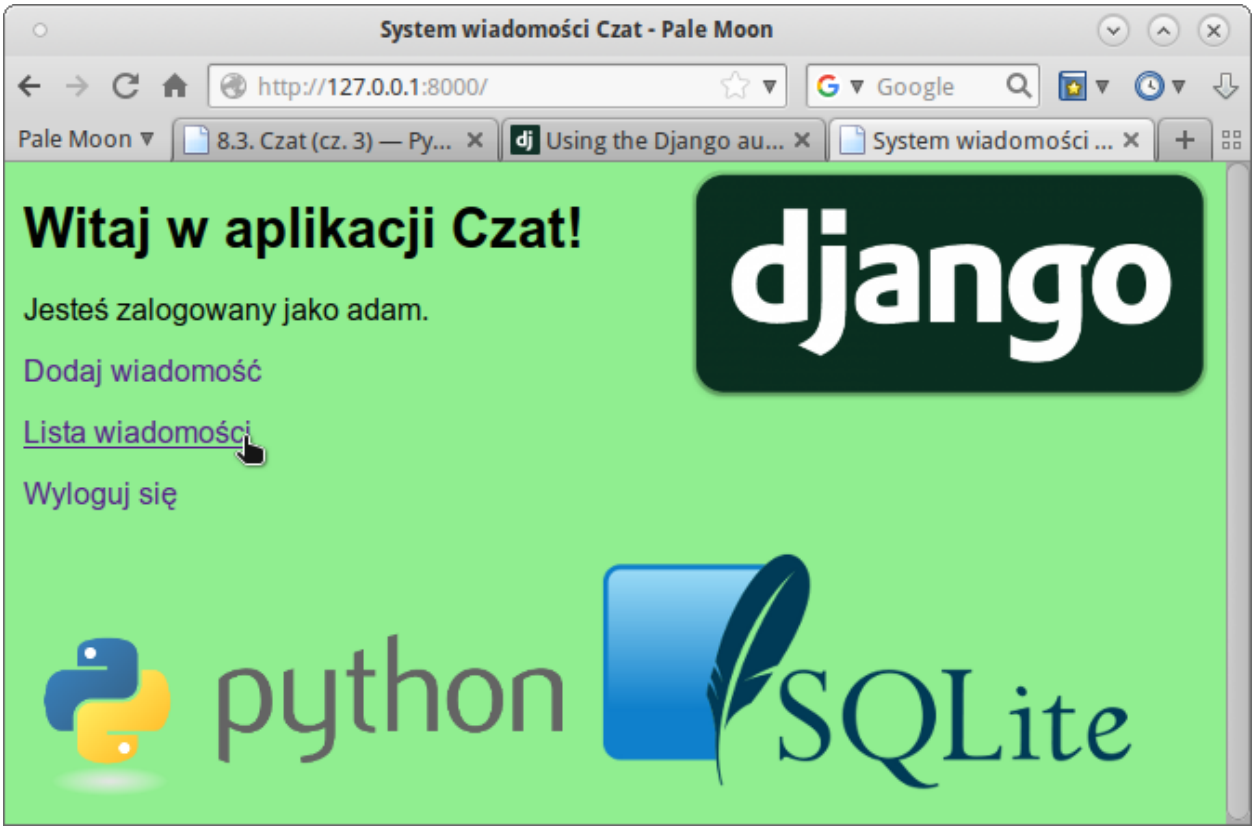
```
<!-- templates/czat/baza.html -->
{% load staticfiles %}
<!DOCTYPE html>
<html lang="pl">
  <meta charset="utf-8" />
  <head>
    <title>{% block tytul %} System wiadomości Czat {% endblock tytul %}</title>
    <!-- dołączamy arkusz stylów: -->
    <link rel="stylesheet" type="text/css" href="{% static 'czat/css/style.css' %}" />
  </head>
  <body>
    <h1>{% block naglowek %} Witaj w aplikacji Czat! {% endblock %}</h1>

    {% block komunikaty %}
      {% if messages %}
        <ul>
        {% for komunikat in messages %}
          <li>{{ komunikat|capfirst }}</li>
        {% endfor %}
        </ul>
      {% endif %}
    {% endblock %}

    {% block tresc %} {% endblock %}

    {% if user.is_authenticated %}
      {% block linki1 %} {% endblock %}
      <p><a href="{% url 'czat:wyloguj' %}">Wyloguj się</a></p>
    {% else %}
      {% block linki2 %} {% endblock %}
    {% endif %}

    {% block linki3 %} {% endblock %}

    <!-- wstawiamy obrazki: -->
    <div id="obrazki">
      <img src="{% static 'czat/img/python.png' %}" width="300" />
      <img src="{% static 'czat/img/sqlite.png' %}" width="300" />
    </div>

  </body>
</html>
```

- `{% load staticfiles %}` – ten kod umieszczamy na początku dokumentu, konfiguruje on ścieżkę do zasobów;
- `{% static plik %}` – ten tag wskazuje lokalizację dołączanego do strony pliku, np. arkusza CSS czy obrazka, tag umieszczamy jako wartość atrybutu `href`.

Ćwiczenie

W szablonie bazowym stwórz blok umożliwiający zastępowanie domyślnych obrazków. Następnie zmień szablon `rejestracja.html` tak, aby wyświetlał inne obrazki, które znajdziesz w podkatalogu `czat/static/img`.

Wskazówka: Tag `{% load staticfiles %}` musisz wstawić zaraz po tagu `{% extends %}` do każdego



szablonu, w którym chcesz odwoływać się do plików z katalogu `static`.

Java Script

Na ostatnim zrzucie widać wykonane ćwiczenie, czyli użycie dodatkowych obrazków. Jednak strona nie wygląda dobrze, ponieważ treść podpowiedzi nachodzi na logo Django (oczywiście przy małym rozmiarze okna przeglądarki). Spróbujemy temu zaradzić.

Wykorzystamy prosty skrypt wykorzystujący bibliotekę jQuery. Ściągamy `archiwum` i rozpakowujemy do katalogu `static/js`. Następnie do szablonu podstawowego `baza.html` dodajemy przed tagiem zamykającym `</body>` znaczniki `<script>`, w których wskazujemy położenie skryptów:

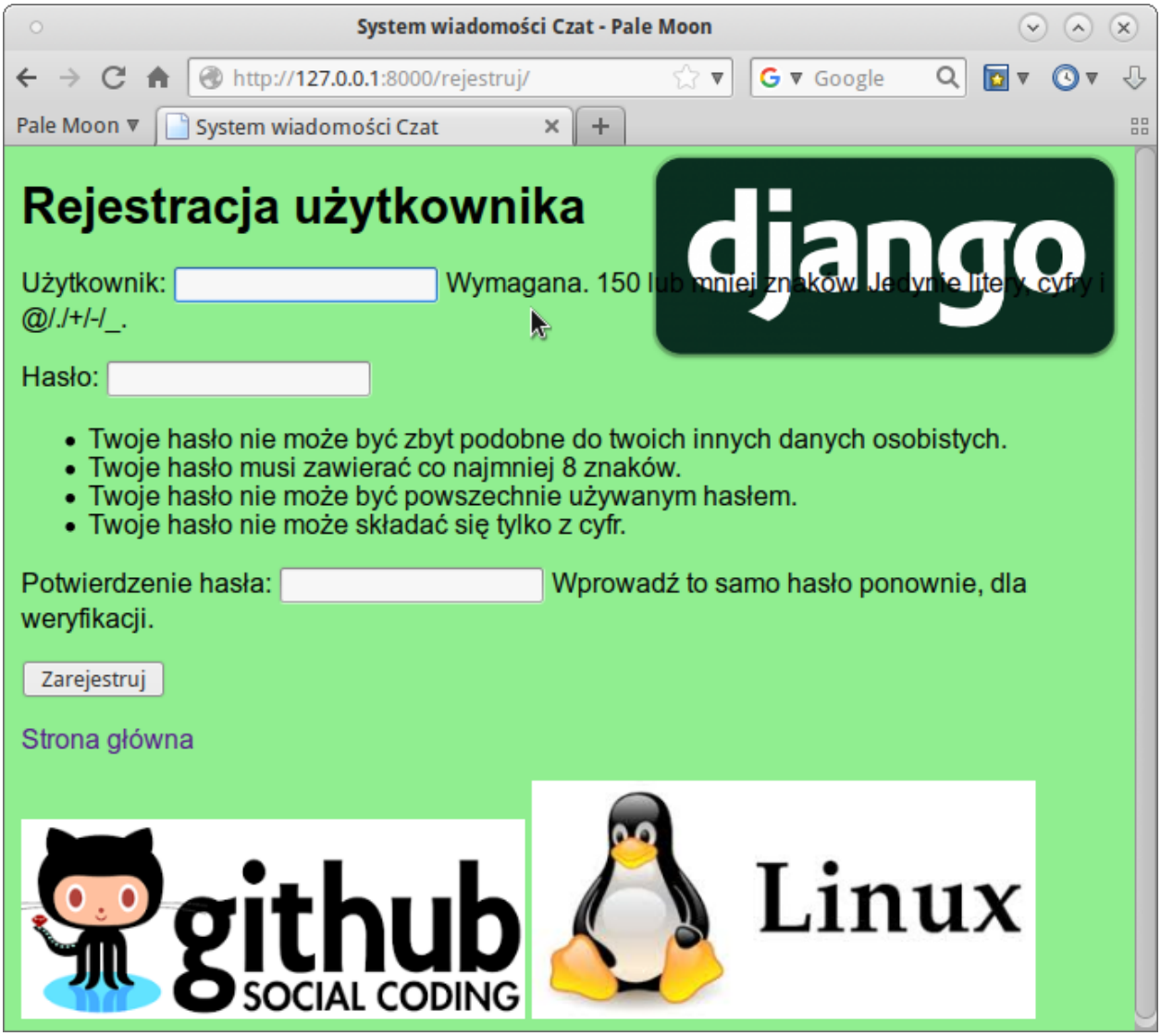
```
    <script type="text/javascript" src="{% static 'czat/js/jquery.js' %}"></script>
    <script type="text/javascript" src="{% static 'czat/js/czat.js' %}"></script>
```

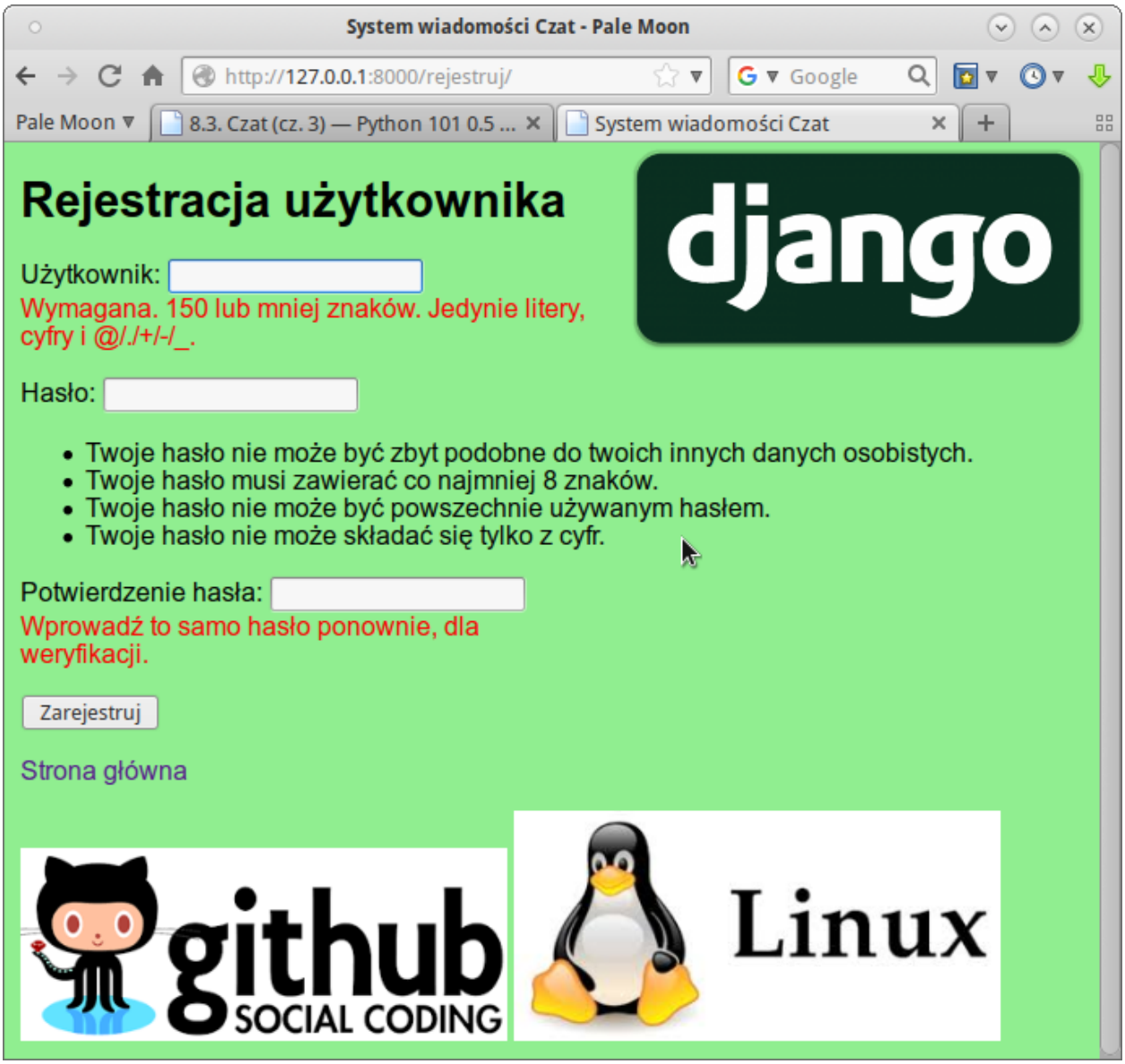
Po odświeżeniu adresu */rejestruj* powinieneś zobaczyć poprawioną stronę:

Bootstrap

Bootstrap to jeden z najpopularniejszych frameworków, który z wykorzystaniem języków HTML, CSS i JS ułatwia tworzenie responsywnych aplikacji sieciowych. Zintegrowanie go z naszą aplikacją przy wykorzystaniu omówionych mechanizmów jest całkiem proste.

Wchodzimy na stronę Getting started, kopiujemy linki dołączające arkusze CSS i wklejamy je za znacznikiem `<title>` w szablonie bazowym:


System wiadomości Czat - Pale Moon
http://127.0.0.1:8000/rejestruj/
Google
Pale Moon
System wiadomości Czat
Rejestracja użytkownika
django
Użytkownik:
Wymagana. 150 lub mniej znaków. Jedynie litery, cyfry i @/./+/-/_.
Hasło:
Twoje hasło nie może być zbyt podobne do twoich innych danych osobistych.
Twoje hasło musi zawierać co najmniej 8 znaków.
Twoje hasło nie może być powszechnie używanym hasłem.
Twoje hasło nie może składać się tylko z cyfr.
Potwierdzenie hasła:
Wprowadź to samo hasło ponownie, dla weryfikacji.
Zarejestruj
Strona główna
github
SOCIAL CODING
Linux


System wiadomości Czat - Pale Moon
http://127.0.0.1:8000/rejestruj/
Google
Pale Moon
8.3. Czat (cz. 3) — Python 101 0.5 ...
System wiadomości Czat
Rejestracja użytkownika
django
Użytkownik:
Wymagana. 150 lub mniej znaków. Jedynie litery, cyfry i @/./+/-/_.
Hasło:
Twoje hasło nie może być zbyt podobne do twoich innych danych osobistych.
Twoje hasło musi zawierać co najmniej 8 znaków.
Twoje hasło nie może być powszechnie używanym hasłem.
Twoje hasło nie może składać się tylko z cyfr.
Potwierdzenie hasła:
Wprowadź to samo hasło ponownie, dla weryfikacji.
Zarejestruj
Strona główna
github
SOCIAL CODING
Linux

```
    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
↪bootstrap.min.css" integrity="sha384-
↪BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin=
↪"anonymous">

    <!-- Optional theme -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/
↪bootstrap-theme.min.css" integrity="sha384-
↪rHyoN1iRsVXV4nD0JutlnGaslCJuC7uwjduW9SVrLvRYooPp2bWYgmgJQIXwl/Sp" crossorigin=
↪"anonymous">
```

Następnie kopiujemy link dołączający Java Script i wklejamy na końcu szablonu bazowego po linku włączającym *jQuery*.

```
    <script type="text/javascript" src="{% static 'czat/js/jquery.js' %}"></script>
    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"␣
↪integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa"␣
↪crossorigin="anonymous"></script>
    <script type="text/javascript" src="{% static 'czat/js/czat.js' %}"></script>
```

Na koniec użyjemy dla przykładu klas `img-thumbnail` i `img-circle` Bootstrapa w znacznikach `<img>` szablonu bazowego:

```
      <img src="{% static 'czat/img/python.png' %}" width="300" class="img-thumbnail␣
↪img-circle" />
      <img src="{% static 'czat/img/sqlite.png' %}" width="300" class="img-thumbnail␣
↪img-circle" />
```

Informacja: Można oczywiście dołączać pliki Bootstrapa po pobraniu i umieszczeniu ich w podkatalogach folderu `static` za pomocą omawianego tagu `{% static %}`.

Materiały

1. O Django http://pl.wikipedia.org/wiki/Django_(informatyka)
2. Strona projektu Django https://www.djangoproject.com/
3. Co to jest framework? http://pl.wikipedia.org/wiki/Framework
4. Co nieco o HTTP i żądaniach GET i POST http://pl.wikipedia.org/wiki/Http

Źródła:

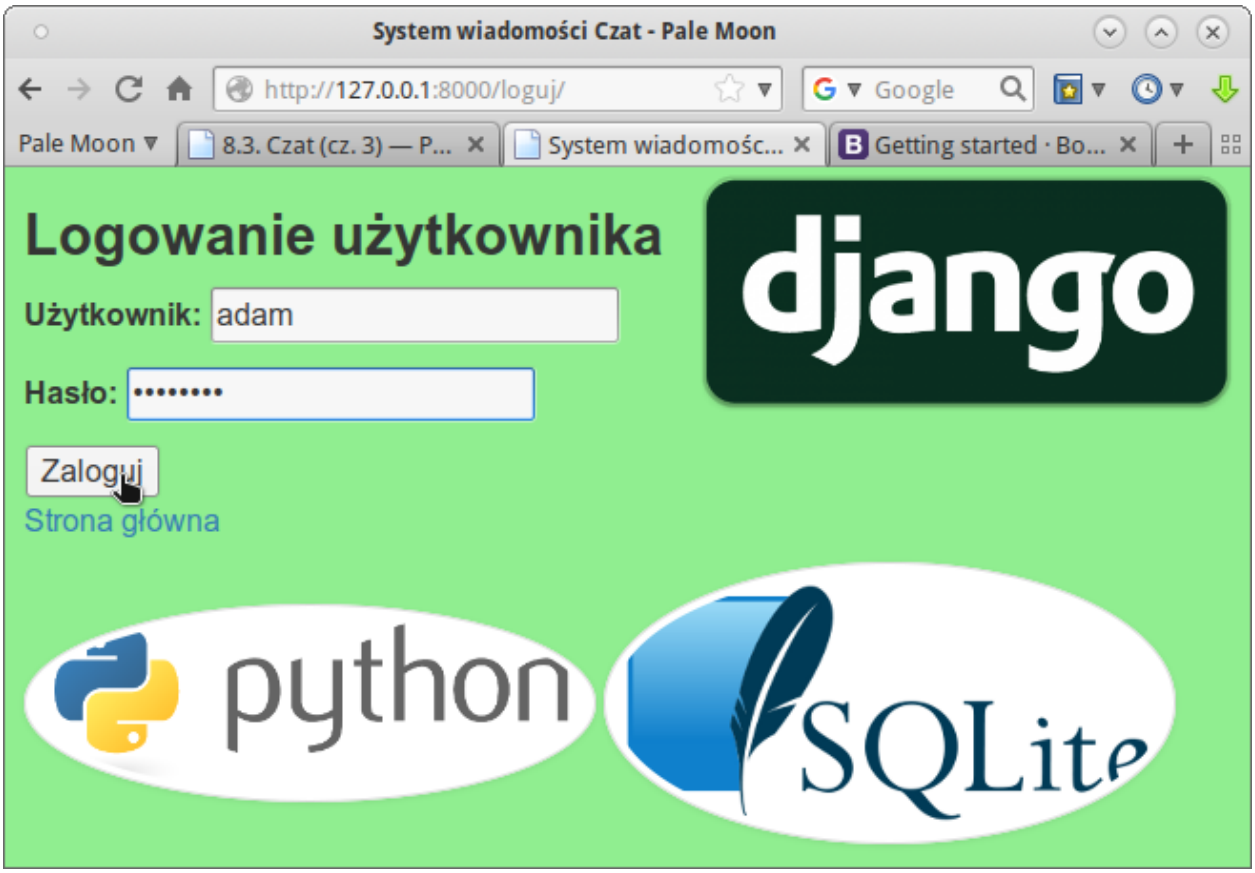
- `czat3.zip`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik “Autorzy”

MVC

W projektowaniu aplikacji internetowych odwołujemy się do wzorca M(odel)V(iew)C(ontroller), czyli Model–Widok–Kontroler, co pozwala na oddzielenie danych od ich prezentacji oraz logiki aplikacji. Frameworki takie jak Flask czy Django korzystają z tego wzorca



Model

Modele – *model* w Django reprezentuje źródło informacji; są to klasy Pythona opisujące pojedyncze tabele w bazie danych (zob. *ORM*); atrybuty klasy odpowiadają polom tabeli, ewentualne funkcje wykonują operacje na danych. Instancja klasy odpowiada rekordowi danych. Modele definiujemy w pliku `models.py`.

Widok

Widoki – *widok* we Flasku lub Django to funkcja lub klasa Pythona, która odpowiada na żądania www, np. zwraca kod HTML generowany w szablonie (ang. *template*), jakiś dokument, obrazek lub przekierowuje na inny adres.

W Django Widoki definiujemy w pliku `views.py`. Django zawiera wiele widoków wbudowanych (ang. *generic views*), w tym opartych na klasach opisujących modele, umożliwiających przeglądanie (np. *ListView, DetailView*) i edycję danych (np. *CreateView, UpdateView*).

Każda funkcja pełniąca rolę widoku jako pierwszy argument otrzymuje obiekt `HttpRequest` zawierający informacje o żądaniu, np. jego typ (GET lub POST), nazwę użytkownika, a zwłaszcza dane przesłane do serwera. Obiekt *request* jest słownikiem. Widok musi zwrócić jakąś odpowiedź. W Django jest to obiekt typu `HttpResponse`.

Widoki wykonują jakieś operacje po stronie serwera w odpowiedzi na żądania klienta. Widoki powiązane są z określonymi adresami url.

Dane z bazy przekazywane są do szablonów za pomocą Pythonowego słownika. Renderowanie polega na odszukaniu pliku szablonu, zastąpieniu przekazanych zmiennych danymi i odesłaniu całości (HTML + dane) do użytkownika.

W Django szablony zapisywane są w podkatalogu `templates/nazwa_aplikacji`.

Kontroler

Kontroler – *kontroler* to mechanizm kierujący kolejne żądania do odpowiednich widoków na podstawie wzorców adresów URL. We Flasku adresy powiązane z widokiem definiujemy w dekoratorach typu `@app.route('/', methods=['GET', 'POST'])`. W Django adresy wiążemy z widokami w pliku `urls.py` np.: `url(r'^loguj/$', views.loguj, name='loguj')`.

Wzorce dopasowania

Fragment `r'^loguj/$'` to wyrażenie regularne, często określane w języku angielskim skrótowo *regex*. Najczęściej będzie zawierać następujące symbole:

1. `r` – początek, `$` – koniec, ograniczniki granic wyrażenia
2. `^` – dopasowuje początek ciągu lub nowej linii
3. `.` – dowolny pojedynczy znak
4. `\d` lub `[0-9]` – pojedyncza cyfra dziesiętna
5. `[a-z]`, `[A-Z]`, `[a-zA-Z]` – małe i/lub duże litery
6. `+`, np. `\d+` – jedno lub więcej wystąpień poprzedniego wyrażenia
7. `?`, np. `\d?` – zero lub 1 wystąpienie poprzedniego wyrażenia
8. `*`, np. `\d*` – zero lub więcej wystąpień poprzedniego wyrażenia
9. `{1,3}`, np. `\d{1,3}` – od 1 do 3 wystąpień poprzedniego wyrażenia

Więcej nt wyrażeń regularnych w Pythonie znajdziesz w dokumentacji: Regular Expression Syntax.

Django

Twórcy Django traktują wzorzec MVC elastycznie, twierdząc że ich framework wykorzystuje taczej wzorzec MTV, czyli model (Model), szablon (Template), widok (View). Oznacza to, że powiązanie widoków z adresami URL oraz same widoki decydują o tym, co zostanie zwrócone i pełnią w ten sposób rolę kontrolera. Szablony natomiast decydują o tym, jak to zostanie zaprezentowane użytkownikowi, a więc pełnią rolę widoków w sensie MVC.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Materiały

1. Python
2. Django
3. SQLite

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.4.9 Minecraft Pi

Minecraft Pi Edition to specjalna wersja gry Minecraft uruchamianej jako serwer na minikomputerze Raspberry Pi z systemem Raspbian. Wyjątkową cechą tej wersji jest możliwość kontrolowanie niektórych elementów gry za pomocą Minecraft API zawartych w bibliotekach **mcpi** napisanych w języku Python i preinstalowanych w Raspbianie (w wersji dla Pythona 2 i 3). Całość bardzo dobrze nadaje się do nauki programowania z wykorzystaniem języka Python.

Wymagania wstępne

1. Serwer Minecrafta Pi, czyli minikomputer Raspberry Pi w wersji B+, 2 lub 3 z najnowszą wersją systemu Raspbian.
2. Klient, czyli dowolny komputer z systemem Linux lub Windows, zawierający interpreter Pythona 2, bibliotekę mcpi oraz symulator mcpi-sim.
3. Adresy IP serwera i klienta muszą należeć do tej samej sieci lokalnej.

Instalacja bibliotek

Wszystkie biblioteki oraz symulator umieściliśmy w archiwum `mcpi-sim.zip`, które należy pobrać i rozpakować w katalogu użytkownika. W kolejnych scenariuszach zakładamy, że tworzone skrypty zapisujemy w katalogu `~/mcpi-sim`.

Informacja:

- Do działania symulatora potrzebna jest biblioteka *PyGame*. Zobacz, jak ją zainstalować w systemie *Linux* lub *Windows*. Symulator działa tylko w Pythonie 2.
- Dystrybucje *Linux Live* przygotowane na potrzeby naszego projektu zawierają już symulator.
- Opisane poniżej scenariusze można realizować bezpośrednio na Raspberry Pi.
- Symulator dostępny jest w repozytorium `https://github.com/pddring/mcpi-sim.git`.
- Biblioteki *mcpi* dostępne są w repozytorium `https://github.com/martinohanlon/mcpi.git`.

Podstawy mcpi

Połączenie z serwerem

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu `mcpi-sim` pod nazwą `mcpi-podst.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
import os
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

# utworzenie połączenia z minecraftem
mc = minecraft.Minecraft.create("192.168.1.10")  # podaj adres IP Rpi


```

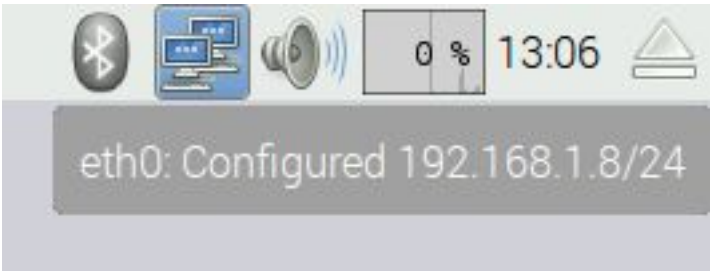
```
16 def main(args):
17     mc.postToChat("Czesc! Tak dziala MC chat!")  # wysłanie komunikatu do mc
18     return 0
19
20
21 if __name__ == '__main__':
22     sys.exit(main(sys.argv))
```

Na początku importujemy moduły do obsługi Minecrafta i za pomocą instrukcji `os.environ[ŹMIENNA"]` ustawiamy wymagane przez `mcpi` zmienne środowiskowe z nazwami użytkownika i komputera:

Informacja: Udany import wymaga, aby w katalogu ze skryptem znajdował się katalog `mcpi`, z którego importujemy wymagane moduły. Jeżeli katalog ten byłby w innym folderze, np. `biblioteki`, przed instrukcjami importu musielibyśmy wskazać ścieżkę do niego, np: `sys.path.append("/home/user/biblioteki")`.

Po wykonaniu czynności wstępnych tworzymy podstawowy obiekt reprezentujący grę Minecraft: `mc = minecraft.Minecraft.create("192.168.1.8")`.

Wskazówka: Adres IP serwera Minecrafta, czyli minikomputera Raspberry Pi, odczytamy po najechaniu myszą na ikonę połączenia sieciowego w prawym górnym rogu pulpitu (zob. zrzut poniżej). Możemy też wydać w terminalu polecenie `ip addr` i odczytać adres poprzedzony przedrostkiem *inet* dla interfejsu *eth0* (łącze kablowe) lub *wlan0* (łącze radiowe).



Na końcu w funkcji `main()`, czyli głównej, wywołujemy metodę `postToChat()`, która pozwala wysłać i wyświetlić podaną wiadomość na czacie Minecrafta.

Skrypt uruchamiamy z poziomu edytora, jeśli to możliwe, lub wykonując w terminalu polecenie:

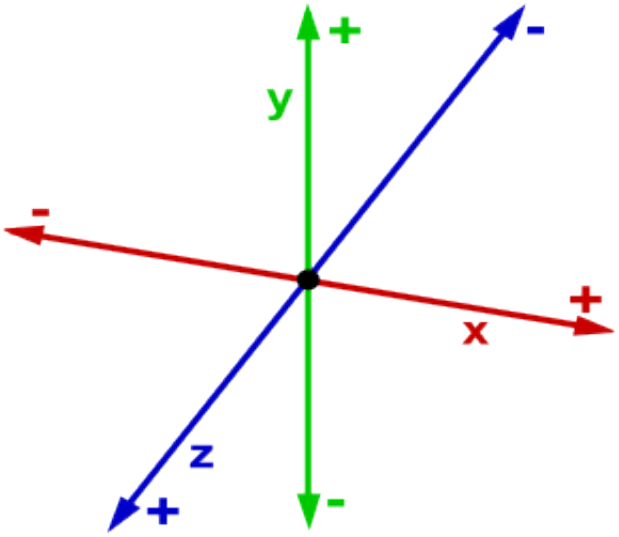
```
~/mcpi-sim$ python mcpi-podst.py
```

Informacja: Omówiony kod (linie 4-14) stanowi **niezbędne minimum**, które **musi znaleźć się** w każdym skrypcie lub w sesji interpretera (konsoli), jeżeli chcemy widzieć efekty naszych działań na serwerze. Dla wygody kopiowania podajemy go w skondensowanej formie:

```
1 import mcpi.minecraft as minecraft  # import modułu minecraft
2 import mcpi.block as block  # import modułu block
3 import os
4 os.environ["USERNAME"] = "Steve"  # wpisz dowolną nazwę użytkownika
5 os.environ["COMPUTERNAME"] = "mykomp"  # wpisz dowolną nazwę komputera
6 mc = minecraft.Minecraft.create("192.168.1.8")
```

Świat Minecrafta Pi

Świat Minecrafta Pi opisujemy za pomocą trójwymiarowego układu współrzędnych:



Obserwując położenie bohatera gry Steve'a zauważymy, że zmiany współrzędnej *x* (klawisze `A` i `D`) i *z* (klawisze `W` i `S`) przesuwają postać w lewo/prawo, do przodu/tyłu, czyli horyzontalnie, natomiast zmiany współrzędnej *y* do góry/w dół - wertykalnie.

Informacja: W Pi Edition wartości *x* i *y* ograniczono do przedziału [-128, 127].

Ćwiczenie 1

Uruchamiamy rozszerzoną konsolę Pythona i wchodzimy do katalogu `mcpi-sim`:

```
    ~$ ipython qtconsole
In [1]: cd /root/mcpi-sim
```

Wskazówka: Podane polecenie można wpisać również w okienko "Uruchom" wywoływane w środowiskach linuksowych zazwyczaj przez skrót `ALT+F2`.

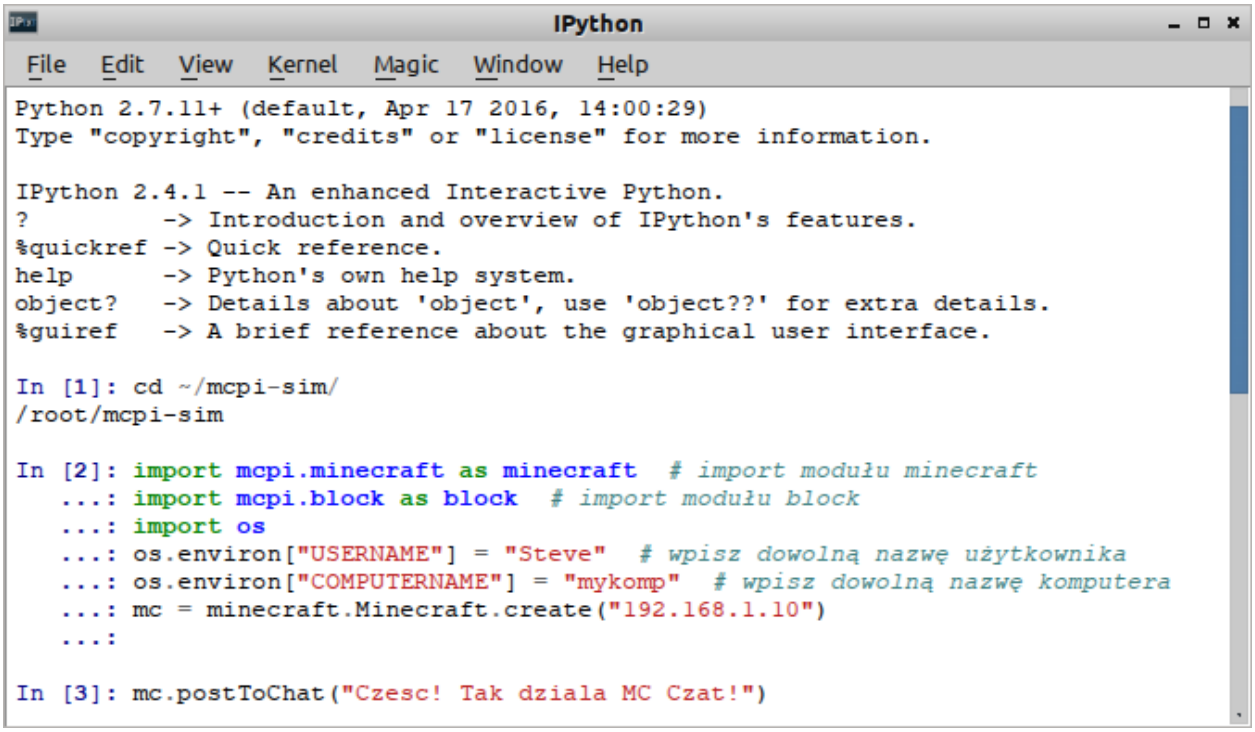
Zamiast rozszerzonej konsoli qt możemy użyć zwykłej konsoli `ipython` lub podstawowego interpretera `python` uruchamianych w terminalu. Uwaga: jeżeli skorzystamy z interpretera podstawowego kod kopiujemy i wklejamy linia po linii.

Kopiujemy do okna konsoli, uruchamiamy omówiony powyżej "Kod 2", służący nawiązaniu połączenia z serwerem, i wysyłamy wiadomość na czat:

Poznamy teraz kilka podstawowych metod pozwalających na manipulowanie światem Minecrafta.

Orientuj się Steve!

Wpisz w konsoli poniższy kod:




```
>>> mc.player.getPos()
>>> x, y, z = mc.player.getPos()
>>> print x, y, z
>>> x, y, z = mc.player.getTilePos()
>>> print x, y, z
```

Metoda `getPos()` obiektu `player` zwraca nam obiekt zawierający współrzędne określające pozycję bohatera. Metoda `getTitlePos()` zwraca z kolei współrzędne bloku, na którym stoi bohater. Instrukcje typu `x, y, z = mc.player.getPos()` rozpakowują kolejne współrzędne do zmiennych *x*, *y* i *z*. Możemy wykorzystać je do zmiany położenia bohatera:

```
>>> mc.player.setPos(x+10, y+20, z)
```

Powyższy kod przesunie bohatera w bok o 10 bloków i do góry na wysokość 20 bloków. Podobnie zadziała kod `mc.player.setTilePos(x+10, y+20, z)`, który przeniesie postać na blok, którego pozycję podamy.

Idź i przesuń się

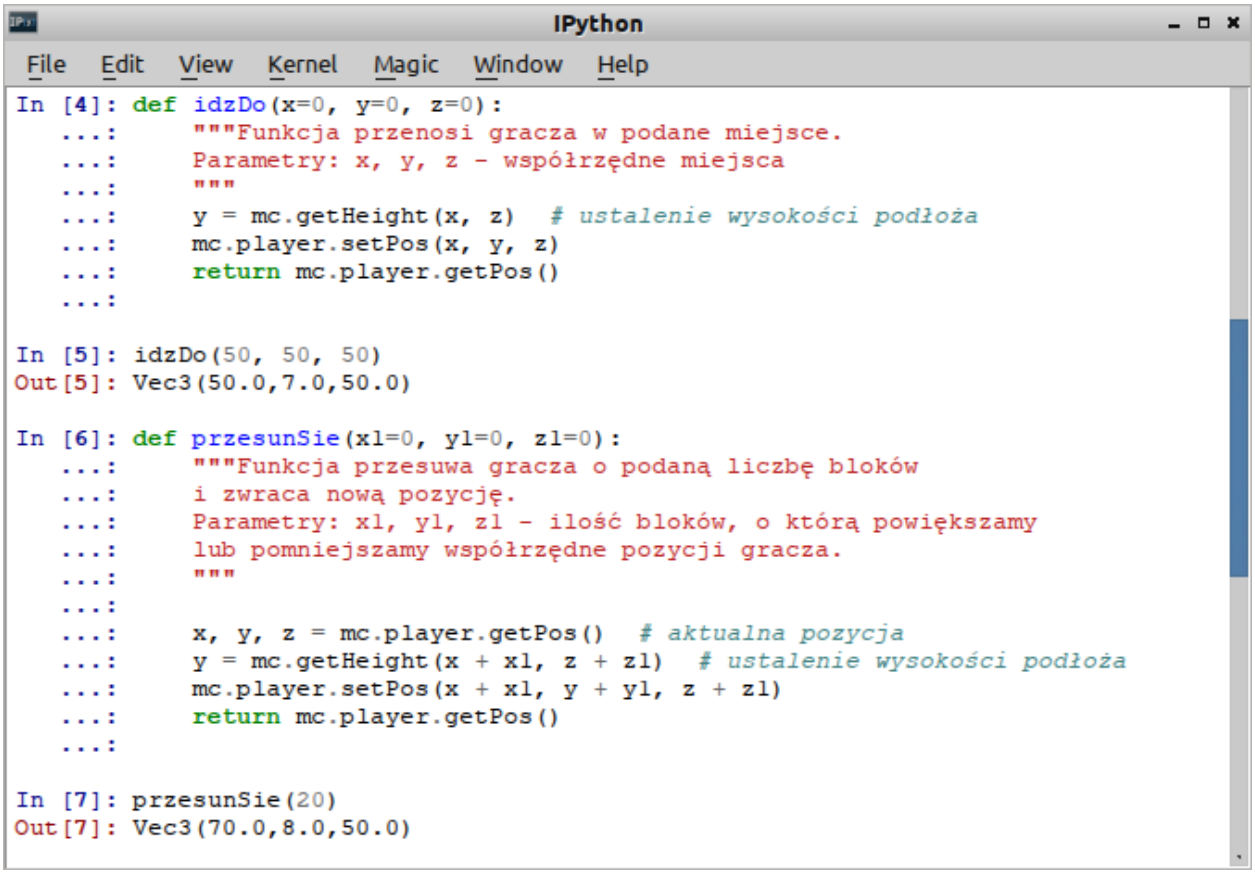
Zadania takie możemy realizować za pomocą funkcji, które dodatkowo zwrócą nam nową pozycję. W pliku `mcpi-podst.py` umieszczamy kod:

```
16 def idzDo(x=0, y=0, z=0):
17     """Funkcja przenosi gracza w podane miejsce.
18     Parametry: x, y, z - współrzędne miejsca
19     """
20     y = mc.getHeight(x, z)  # ustalenie wysokości podłoża
21     mc.player.setPos(x, y, z)
22     return mc.player.getPos()
23
24
```

```
25 def przesunSie(x1=0, y1=0, z1=0):
26     """Funkcja przesuwa gracza o podaną liczbę bloków
27     i zwraca nową pozycję.
28     Parametry: x1, y1, z1 - ilość bloków, o którą powiększamy
29     lub pomniejszamy współrzędne pozycji gracza.
30     """
31
32     x, y, z = mc.player.getPos()  # aktualna pozycja
33     y = mc.getHeight(x + x1, z + z1)  # ustalenie wysokości podłoża
34     mc.player.setPos(x + x1, y + y1, z + z1)
35     return mc.player.getPos()
```

W pierwszej funkcji `idzDo()` warto zwrócić uwagę na metodę `getHeight()`, która pozwala ustalić wysokość świata w punkcie *x, z*, czyli współrzędną *y* najwyższego bloku nie będącego powietrzem. Dzięki temu umieścimy bohatera zawsze na jakiejś powierzchni, a nie np. pod ziemią ;-). Druga funkcja `przesunSie()` nie tyle umieszcza, co przesuwa postać, stąd dodatkowe instrukcje.

Dopisz wywołanie `print idzDo(50, 0, 50)` w funkcji `main()` przed instrukcją `return` i przetestuj kod uruchamiając skrypt `mcpi-podst.py` lub w konsoli. Później dopisz również drugą funkcję `print przesunSie(20)` i sprawdź jej działanie.



Ćwiczenie 2

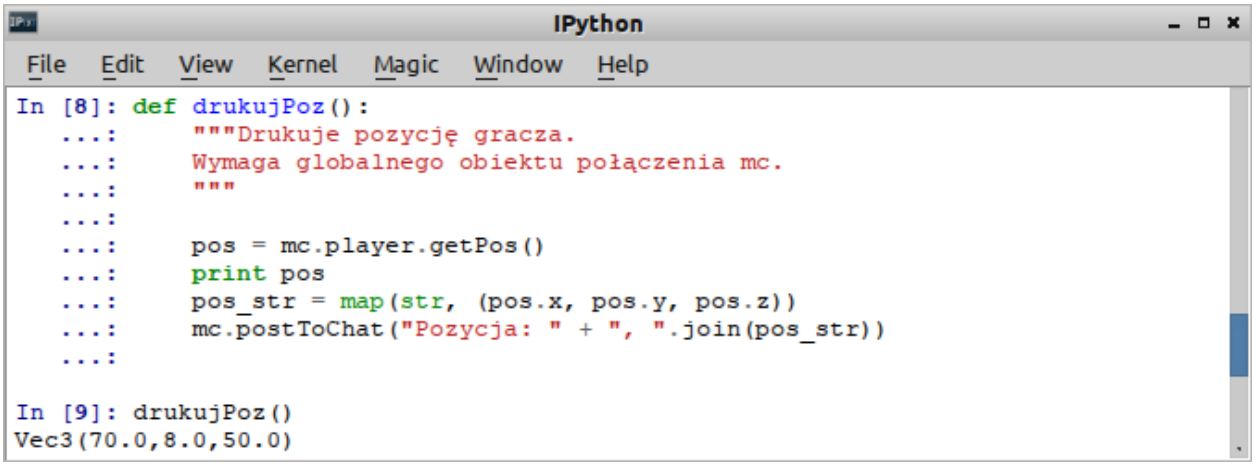
Sprawdź, co się stanie, kiedy podasz współrzędne większe niż świat Minecrafta. Zmień kod obydwu funkcji na "bezpieczny dla życia" ;-)

Gdzie jestem?

Aby odczytywać i drukować pozycję bohatera dodamy kolejną funkcję do pliku `mcpi-podst.py`:

```
38 def drukujPoz():
39     """Drukuje pozycję gracza.
40     Wymaga globalnego obiektu połączenia mc.
41     """
42
43     pos = mc.player.getPos()
44     print pos
45     pos_str = map(str, (pos.x, pos.y, pos.z))
46     mc.postToChat("Pozycja: " + ", ".join(pos_str))
```

Funkcja nie tylko drukuje koordynaty w konsoli (`print x, y, z`), ale również – po przekształceniu ich na listę wartości typu string `pos_str = map(str, pos_list)` – wysyła jako komunikat na czat Minecrafta. Wywołanie funkcji dopisujemy do funkcji głównej i testujemy kod:



Więcej ruchu

Teraz możemy trochę pochodzić, ale będziemy obserwować to z lotu ptaka. Dopiszmy kod poniższej funkcji do pliku `mcpi-podst.py`:

```
49 def ruszajSie():
50     from time import sleep
51
52     krok = 10
53     # ustawienie pozycji gracza w środku świata na odpowiedniej wysokości
54     przesunSie(0, 0, 0)
55
56     mc.postToChat("Latam...")
57     przesunSie(0, krok, 0)  # idź krok bloków do góry - latamy :-)
58     sleep(2)
59
60     mc.camera.setFollow()  # ustawienie kamery z góry
61
62     mc.postToChat("Ide w bok...")
63     for i in range(krok):
64         przesunSie(1)  # idź krok bloków w bok
```

```
    sleep(2)

    mc.postToChat("Ide w drugi bok...")
    for i in range(krok):
        przesunSie(-1)  # idź krok bloków w drugi bok
    sleep(2)

    mc.postToChat("Ide do przodu...")
    for i in range(krok):
        przesunSie(0, 0, 1)  # idź krok bloków do przodu
    sleep(2)

    mc.postToChat("Ide do tylu...")
    for i in range(krok):
        przesunSie(0, 0, -1)  # idź krok bloków do tyłu
    sleep(2)

    drukujPoz()
    mc.camera.setNormal()  # ustawienie kamery normalnie
```

Warto zauważyć, jak pętla `for i in range(krok)` umożliwia symulowanie ruchu postaci. Wywołanie funkcji dodajemy do funkcji głównej. Kod testujemy uruchamiając skrypt lub w konsoli.

Po czym chodzę?

Teraz spróbujemy dowiedzieć się, po jakich blokach chodzimy. Definiujemy jeszcze jedną funkcję:

```
def jakiBlok():
    x, y, z = mc.player.getPos()
    return mc.getBlock(x, y - 1, z)
```

Dopisujemy jej wywołanie: `print "Typ bloku:  ", jakiBlok()` – w funkcji głównej i testujemy.

Plac budowy

Skoro orientujemy się już w przestrzeni, możemy zacząć budować. Na początku wykorzystamy **symulator**. Rozpoczniemy od przygotowania placu budowy. Posłuży nam do tego odpowiednia funkcja, którą umieścimy w pliku `mcsim.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import sys
import os
import local.minecraft as minecraft  # import modułu minecraft
import local.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

# utworzenie połaczenia z symulatorem
mc = minecraft.Minecraft.create("")


def plac(x, y, z, roz=10, gracz=False):
```

IPython

File Edit View Kernel Magic Window Help

```
In [10]: def ruszajSie():
    ...:     from time import sleep
    ...:
    ...:     krok = 10
    ...:     # ustawienie pozycji gracza w środku świata na odpowiedniej
wysokości
    ...:     przesunSie(0, 0, 0)
    ...:
    ...:     mc.postToChat("Latam...")
    ...:     przesunSie(0, krok, 0)  # idź krok bloków do góry - latamy :-)
    ...:     sleep(2)
    ...:
    ...:     mc.camera.setFollow()  # ustawienie kamery z góry
    ...:
    ...:     mc.postToChat("Ide w bok...")
    ...:     for i in range(krok):
    ...:         przesunSie(1)  # idź krok bloków w bok
    ...:     sleep(2)
    ...:
    ...:     mc.postToChat("Ide w drugi bok...")
    ...:     for i in range(krok):
    ...:         przesunSie(-1)  # idź krok bloków w drugi bok
    ...:     sleep(2)
    ...:
    ...:     mc.postToChat("Ide do przodu...")
    ...:     for i in range(krok):
    ...:         przesunSie(0, 0, 1)  # idź krok bloków do przodu
    ...:     sleep(2)
    ...:
    ...:     mc.postToChat("Ide do tylu...")
    ...:     for i in range(krok):
    ...:         przesunSie(0, 0, -1)  # idź krok bloków do tyłu
    ...:     sleep(2)
    ...:
    ...:     drukujPoz()
    ...:     mc.camera.setNormal()  # ustawienie kamery normalnie
    ...:

In [11]: ruszajSie()
Vec3(70.0,8.0,50.0)
```

IPython

File Edit View Kernel Magic Window Help

```
In [12]: def jakiBlok():
    ...:     x, y, z = mc.player.getPos()
    ...:     return mc.getBlock(x, y - 1, z)
    ...:

In [13]: print "Typ bloku:", jakiBlok()
Typ bloku: 35

In [14]:
```

```
    """Funkcja wypełnia sześcienny obszar od podanej pozycji
    powietrzem i opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    kamien = block.STONE
    powietrze = block.AIR

    # kamienna podłoże
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, kamien)
    # czyszczenie
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, powietrze)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def main(args):
    mc.postToChat("Cześć! Tak działa MC chat!")  # wysłanie komunikatu do mc
    plac(0, 0, 0, 18)
    return 0


if __name__ == '__main__':
    sys.exit(main(sys.argv))
```

Funkcja `plac()` korzysta z metody `setBlocks(x0,y0,z0,x1,y1,z1,blockType, blockData)`, która wypełnia obszar w przedziałach [*x0-x1*], [*y0-y1*], [*z0-z1*] blokiem podanego typu o opcjonalnych właściwościach. Na początku tworzymy "podłogę" z kamienia, później wypełniamy sześcian o podanym rozmiarze powietrzem. W symulatorze nie jest to przydatne, ale bardzo przydaje się do "wyczyszczenia" miejsca w świecie Minecrafta. Opcjonalnie możemy umieścić gracza w środku utworzonego obszaru.

Kod testujemy uruchamiając skrypt `mcsim.py`:

```
~/mcpi-sim$ python mcsim.py
```

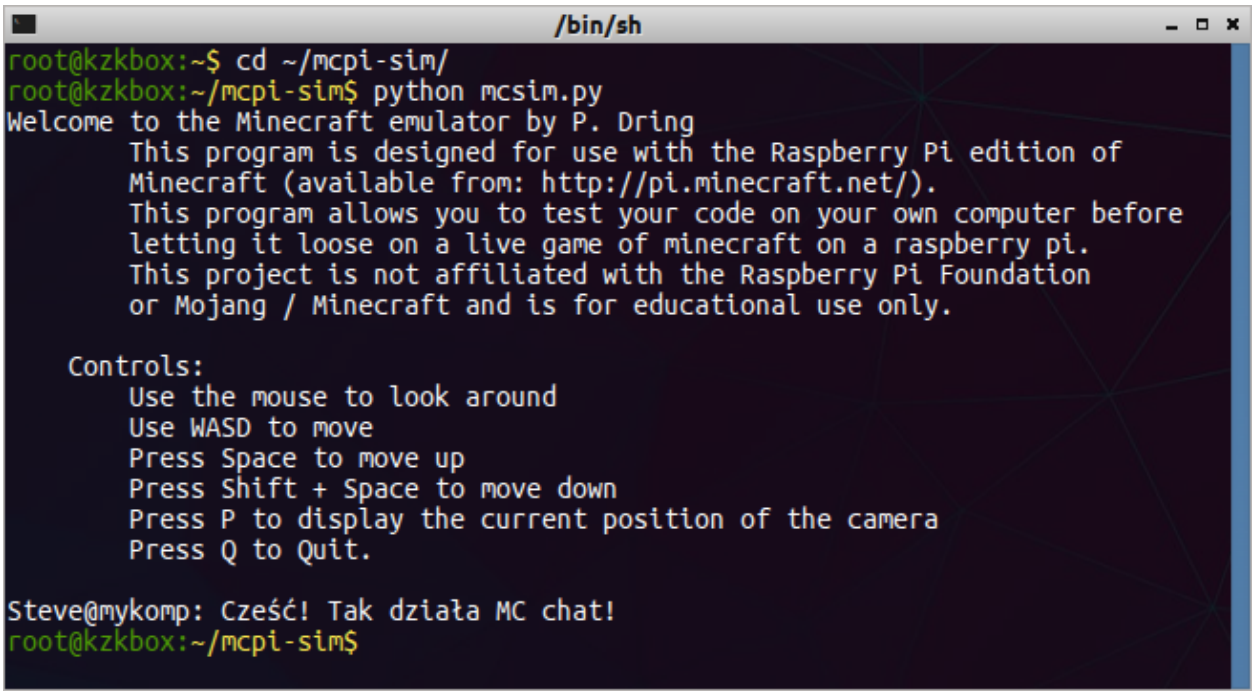
Ostrzeżenie: Skrypt `mcsim.py` musi znajdować się w katalogu `mcpi-sim` ze źródłami symulatora, który wykorzystuje specjalne wersje bibliotek *minecraft* i *block* z podkatalogu `local`.

Klawisze sterujące podglądem symulacji widoczne są w terminalu:

Umieszczanie bloków

W pliku `mcsim.py` przed funkcją główną (`main()`) umieszczamy funkcję `buduj()`:

```
def buduj():
    """
    Funkcja do testowania umieszczania bloków.
    Wymaga: globalnych obiektów mc i block.
    """
    mc.setBlock(0, 0, 18, block.CACTUS)
```



Używamy podstawowej metody `setBlock(x, y, z, blockType)`, która w podanych koordynatach umieszcza określony blok. Wywołanie funkcji `buduj()` dodajemy do `main()` po funkcji `plac()` i testujemy. Ponad "podłogą" powinien znaleźć się zielony blok.

Do rysowania bloków można użyć pętli. Zmieniamy funkcję `buduj()` następująco:

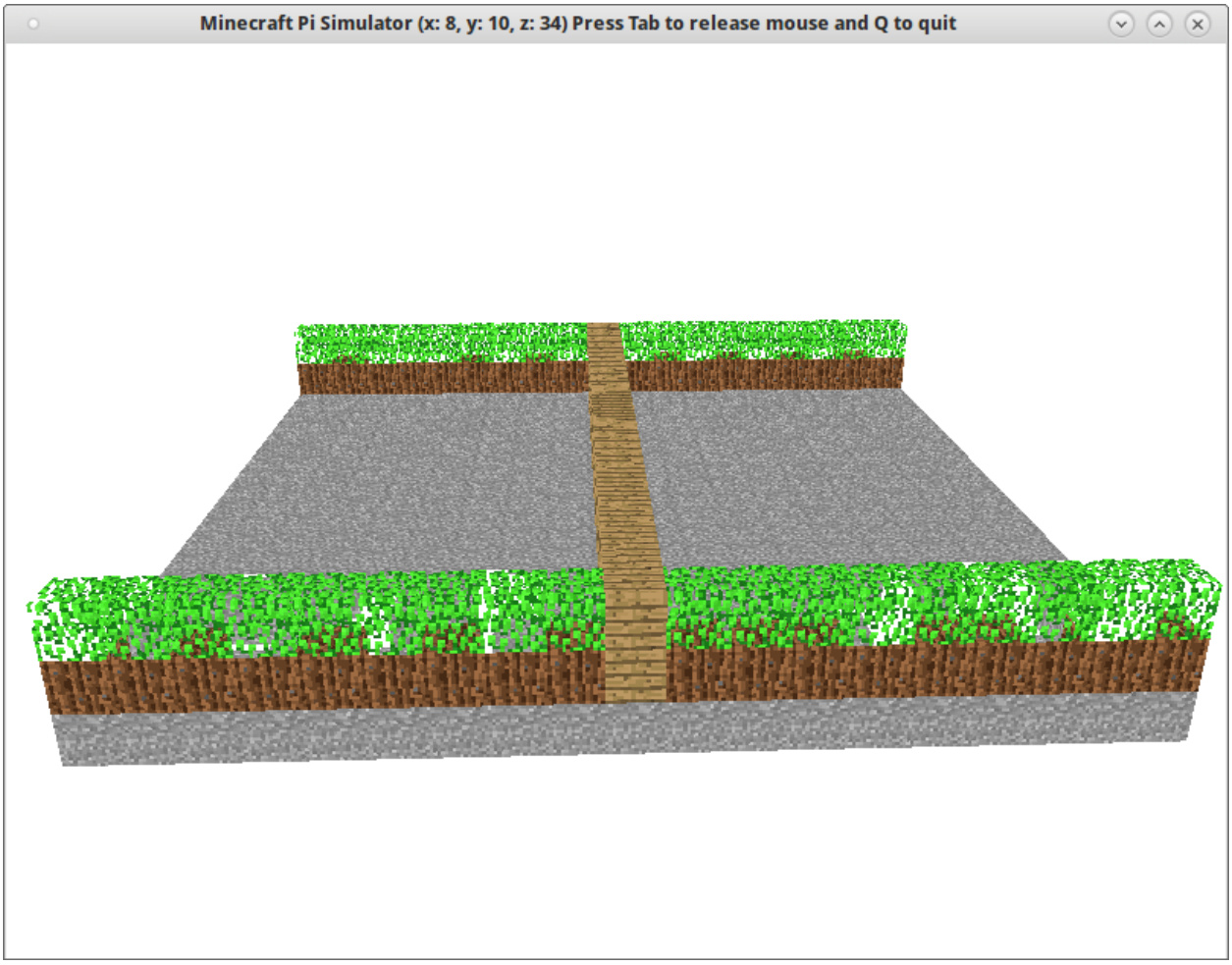
```
37 def buduj():
38     """
39     Funkcja do testowania umieszczania bloków.
40     Wymaga: globalnych obiektów mc i block.
41     """
42     for i in range(19):
43         mc.setBlock(0 + i, 0, 0, block.WOOD)
44         mc.setBlock(0 + i, 1, 0, block.LEAVES)
45         mc.setBlock(0 + i, 0, 18, block.WOOD)
46         mc.setBlock(0 + i, 1, 18, block.LEAVES)
47
48     for i in range(19):
49         mc.setBlock(9, 0, 18 - i, block.BRICK_BLOCK)
50         mc.setBlock(9, 1, 18 - i, block.BRICK_BLOCK)
```

Teraz plac powinien wyglądać, jak poniżej:

Ćwiczenie 3

Odpowiednio modyfikując funkcję `buduj()` skonstruuj:

- kwadrat 2D
- prostokąt 2D
- słup
- bramę, czyli prostokąt 3D
- sześcian


Minecraft Pi Simulator (x: 8, y: 10, z: 34) Press Tab to release mouse and Q to quit

Przykłady

Zapisz skrypt `mcsim.py` pod nazwą `mcpi-test.py` i dostosuj go do uruchomienia na serwerze *MC Pi*. W tym celu zamień ciąg "local" w importach na "mcpi" oraz podaj adres IP serwera *MC Pi* w poleceniu tworzącym połączenie. Następnie umieść w pliku kody poniższych funkcji i po kolei je przetestuj dodając ich wywołania w funkcji głównej.

Zostawiam ślady

```
def jakiBlok():
    while True:
        x, y, z = mc.player.getPos()
        blok_pod = mc.getBlock(x, y - 1, z)
        print(blok_pod)
        sleep(1)


def slad(blok=38):
    while True:
        x, y, z = mc.player.getPos()
        mc.setBlock(x, y, z, blok)
        sleep(0.1)


def slad_jezeli(pod=2, blok=38):
    while True:
        x, y, z = mc.player.getPos()
        blok_pod = mc.getBlock(x, y - 1, z)  # blok pod graczem

        if blok_pod == pod:
            mc.setBlock(x, y, z, blok)
        sleep(0.1)
```

Buduję pomnik

```
def pomnik():
    """
    Funkcja ustawia blok lawy, nad nim blok wody, a później powietrza.
    """
    x, y, z = mc.player.getPos()

    lawa = 10
    woda = 8
    powietrze = 0

    mc.setBlock(x + 5, y + 3, z, lawa)
    sleep(10)
    mc.setBlock(x + 5, y + 5, z, woda)
    sleep(4)
    mc.setBlock(x + 5, y + 5, z, powietrze)
```

Piramida

```
def kwadrat(bok, x, y, z):
    """
    Fukcja buduje kwadrat, którego środek to punkt x, y, z
    """
    pol = bok // 2
    piaskowiec = block.SANDSTONE
    mc.setBlocks(x - pol, y, z - pol, x + pol, y, z + pol, piaskowiec, 2)


def piramida(podstawa, x, y, z):
    """
    Buduje piramidę z piasku, której środek wypada w punkcie x, y, z
    """
    bok = podstawa
    wysokosc = y
    while bok >= 1:
        kwadrat(bok, x, wysokosc, z)
        bok -= 2
        wysokosc += 1
```

Źródła:

- `Skrypty mcpi podstawy`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik “Autorzy”

Figury 2D i 3D

Możliwość programowego umieszczania różnych bloków w Minecraft Pi Edition można wykorzystać jako atrakcyjny sposób wizualizacji różnych figur. Jednak o ile budowanie prostych kształtów, jak np. kwadrat czy sześcian, nie stanowi raczej problemu, o tyle trójkąty, koła i bardziej skomplikowane budowle nie są trywialnym zadaniem. Tworzenie 2- i 3-wymiarowych konstrukcji ułatwi nam biblioteka minecraftstuff.

Instalacja

Symulator `mcpi-sim` domyślnie nie działa z omawianą biblioteką i wymaga modyfikacji. Zmienione pliki oraz omawianą bibliotekę umieściliśmy w archiwum `mcpi-sim-fix.zip`, które po ściągnięciu należy rozpakować do katalogu `~/mcpi-sim/local` nadpisując oryginalne pliki.

Linia

W pustym pliku `mcsim-fig.py` umieszczamy kod:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import local.minecraft as minecraft  # import modułu minecraft
import local.block as block  # import modułu block
import local.minecraftstuff as mcstuff  # import biblioteki do rysowania figur
from local.vec3 import Vec3  # klasa reprezentująca punkt w MC

```

```
os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

mc = minecraft.Minecraft.create("")  # połaczenie z symulatorem
figura = mcstuff.MinecraftDrawing(mc)  # obiekt do rysowania kształtów


def plac(x, y, z, roz=10, gracz=False):
    """
    Funkcja tworzy podłoże i wypełnia sześcienny obszar od podanej pozycji,
    opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    podloga = block.STONE
    wypelniacz = block.AIR

    # podloga i czyszczenie
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def linie():
    # Funkcja rysuje linie
    # tuple z współrzędnymi punktów
    punkty1 = ((-10, 0, -10), (10, 0, -10), (10, 0, 10), (-10, 0, 10))
    punkty2 = ((-15, 5, 0), (15, 5, 0), (0, 5, 15), (0, 5, -15))
    p1 = Vec3(0, 0, 0)  # punkt początkowy
    for punkt in punkty1:
        x, y, z = punkt
        p2 = Vec3(x, y, z)  # punkt końcowy
        figura.drawLine(p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, block.WOOL, 14)
    for punkt in punkty2:
        x, y, z = punkt
        p2 = Vec3(x, y, z)  # punkt końcowy
        figura.drawLine(p1.x, p1.y, p1.z, p2.x, p2.y, p2.z, block.OBSIDIAN)


def main():
    mc.postToChat("Biblioteka minecraftstuff")  # wysłanie komunikatu do mc
    plac(-15, 0, -15, 30)
    linie()  # wywołanie funkcji

    return 0


if __name__ == '__main__':
    main()
```

Większość kodu omówiona została w *Podstawach*. W nowym kodzie, który został podświetlony, importujemy bibliotekę *minecraftstuff* oraz klasę *Vec3*. Reprezentuje ona punkty o podanych współrzędnych w trójwymiarowym świecie MC. Polecenie `figura = mcstuff.MinecraftDrawing(mc)` tworzy instancję głównej klasy biblio-

teki, która udostępni jej metody.

Do rysowania linii wykorzystujemy metodę `drawLine()`, której przekazujemy jako argumenty współrzędne punktu początkowego i końcowego, a także typ bloku i ewentualnie podtyp. Ponieważ chcemy narysować kilka linii wychodzących z tego samego punktu, współrzędne punktów końcowych umieszczamy w dwóch tuplach (niemodyfikowalnych listach) jako... tuple. W pętlach odczytujemy je (`for punkt in punkty1:`), rozpakowujemy (`x, y, z = punkt`) i przekazujemy do konstruktora omówionej wyżej klasy `Vec3` (`p2 = Vec3(x, y, z)`).

Całość omówionego kodu dla przejrzystości umieszczamy w funkcji `linie()`, którą wywołujemy w funkcji głównej i testujemy.

Koło

Przed funkcją główną `main()` wstawiamy kod:

```
54 def kolo(x, y, z, r):
55     # Funkcja rysuje koło pionowe i poziome o środku x, y, z i promieniu r
56     figura.drawCircle(x, y, z, r, block.LEAVES, 2)
57     figura.drawHorizontalCircle(x, y, z, r, block.LEAVES, 2)
```

Funkcja `kolo(x, y, z, r)` wykorzystuje metodę `drawCircle()` do rysowania koła w pionie oraz `drawHorizontalCircle()` do rysowania koła w poziomie. Obydwie metody pobierają współrzędne środka koła, jego promień oraz typ i podtyp bloku, służącego do rysowania.

Umieść wywołanie funkcji, np. `kolo(0, 10, 0, 10)`, w funkcji głównej i przetestuj.

Kula

Do skryptu wstawiamy kolejną funkcję przed funkcją `main()`:

```
60 def kula(x, y, z, r):
61     # Funkcja rysuje kulę o środku x, y, z i promieniu r
62     figura.drawSphere(x, y, z, r, block.WOOD, 2)
```

Metoda `drawSphere()` buduje kulę. Pierwsze trzy argumenty to współrzędne środka, kolejne to: promień, typ i ewentualny podtyp bloku. Umieść wywołanie funkcji, np. `kula(0, 10, 0, 9)`, w funkcji głównej i przetestuj.

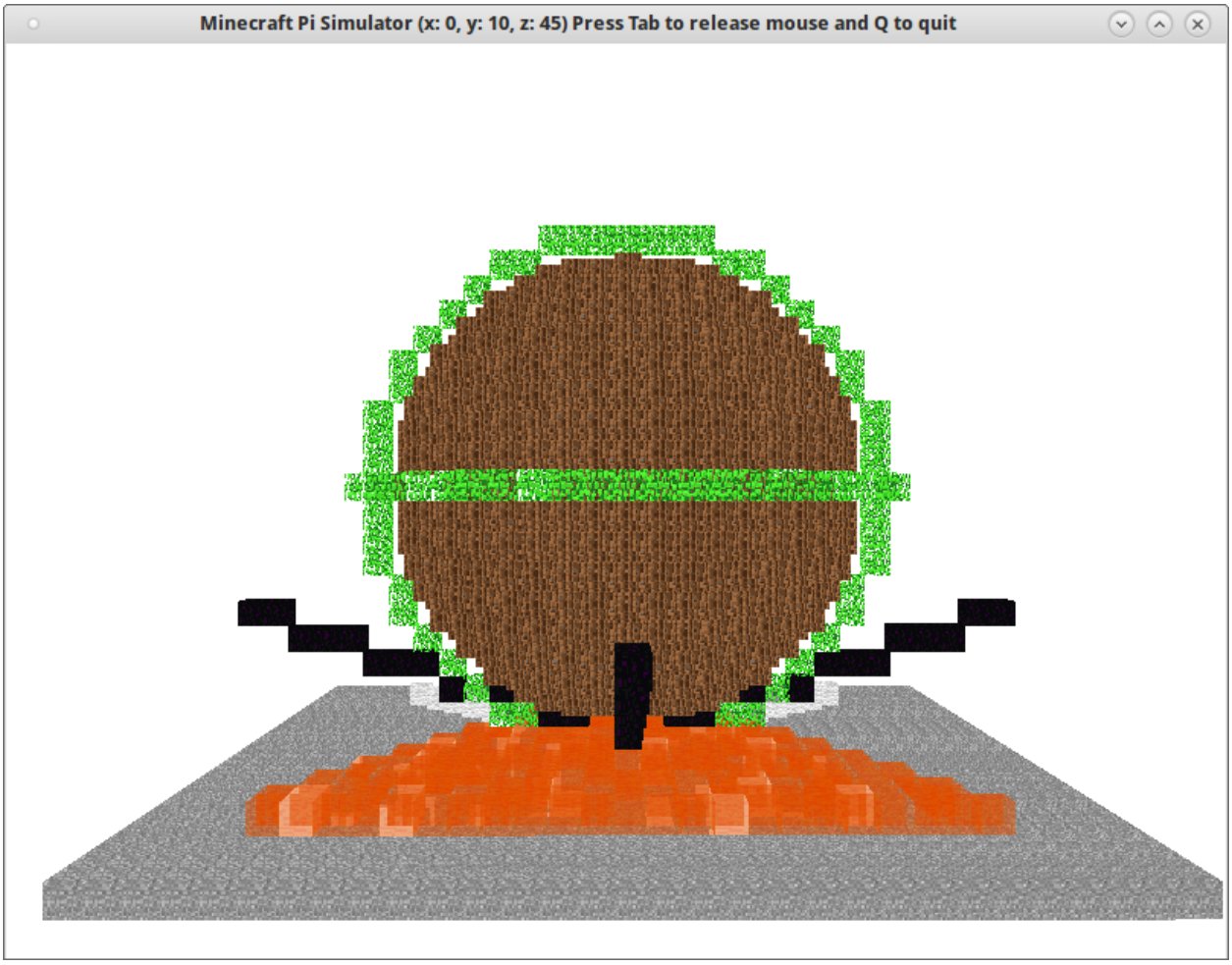
Kształt

Przed funkcją `main()` wstawiamy:

```
65 def ksztalt():
66     # Funkcja łączy podane w liście wierzchołki i opcjonalnie wypełnia figurę
67     ksztalt = []  # lista punktów
68     ksztalt.append(Vec3(-11, 0, 11))  # współrzędne 1 wierzchołka
69     ksztalt.append(Vec3(11, 0, 11))  # współrzędne 2 wierzchołka
70     ksztalt.append(Vec3(0, 0, -11))  # współrzędne 3 wierzchołka
71     figura.drawFace(ksztalt, True, block.SANDSTONE, 2)
```

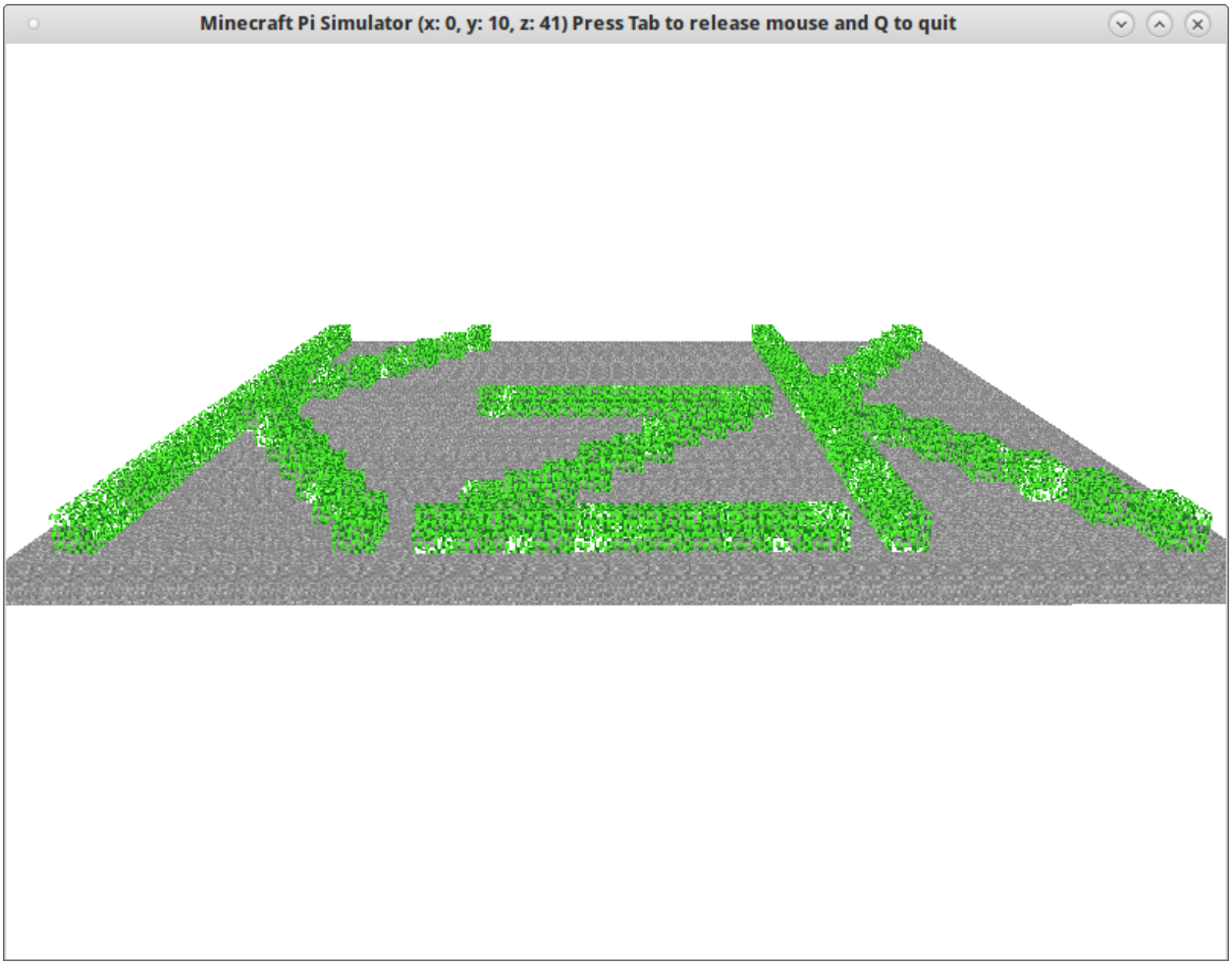
Chcąc narysować trójkąt do listy do listy `ksztalt` dodajemy trzy instancje klasy *Vec3* definiujące kolejne wierzchołki: `ksztalt.append(Vec3(-11, 0, 11))`. Do rysowania dowolnych kształtów służy metoda `drawFace()`, która punkty przekazane w liście łączy liniami budowanymi z podanego bloku. Drugi argument, logiczny, decyduje o tym, czy figura ma zostać wypełniona (`True`), czy nie (`False`).

Po wywołaniu wszystkich omówionych funkcji możemy zobaczyć w symulatorze poniższą budowlę:


Minecraft Pi Simulator (x: 0, y: 10, z: 45) Press Tab to release mouse and Q to quit

Ćwiczenie 1

Wykorzystując odpowiednią metodę biblioteki *minecraftstuff*, spróbuj zbudować napis "KzK" podobny do pokazanego poniżej. Przetestuj swój kod w symulatorze i w Minecrafcie Pi.



Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Żółw w przestrzeni

Biblioteka minecraftturtle implementuje tzw. grafikę żółwia (ang. *turtle graphics*) w trzech wymiarach. W praktyce ułatwia więc budowanie konstrukcji przestrzennych. Inspirowana jest wbudowaną w Pythona biblioteką turtle, często wykorzystywaną do nauki programowania najmłodszych. Poniżej pokażemy, jak poruszać się "żółwiem" w przestrzeni.

Instalacja

Symulator `mcpi-sim` domyślnie nie działa z omawianą biblioteką i wymaga modyfikacji. Zmienione pliki oraz omawianą bibliotekę umieściliśmy w archiwum `mcpi-sim-fix.zip`, które po ściągnięciu należy rozpakować do katalogu `~/mcpi-sim/local` nadpisując oryginalne pliki.

Kwadraty

W pustym pliku `mcsim-turtle.py` umieszczamy kod:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import local.minecraft as minecraft  # import modułu minecraft
import local.block as block  # import modułu block
import local.minecraftturtle as mcturtle
from local.vec3 import Vec3  # klasa reprezentująca punkt w MC

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

mc = minecraft.Minecraft.create("")  # połaczenie z symulatorem
start = Vec3(0, 1, 0)  # pozycja początkowa
turtle = mcturtle.MinecraftTurtle(mc, start)  # obiekt "żółwia"


def plac(x, y, z, roz=10, gracz=False):
    """
    Funkcja tworzy podłoże i wypełnia sześcienny obszar od podanej pozycji,
    opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    podloga = block.STONE
    wypelniacz = block.AIR

    # podloga i czyszczenie
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def kwadraty():
    # Funkcja rysuje dwa kwadraty w poziomie
    turtle.speed(0)  # szybkość budowania
    turtle.penblock(block.SAND)  # typ bloku
    for i in range(4):
        turtle.forward(10)  # do przodu 10 "kroków"
        turtle.right(90)  # w prawo o 90 stopni
    turtle.left(90)  # w lewo o 90 stopni
    for i in range(4):
        turtle.forward(10)
        turtle.left(90)


def main():
    mc.postToChat("Biblioteka minecraftturtle")  # wysłanie komunikatu do mc
    plac(-15, 0, -15, 30)
```

```
55      kwadraty()
56
57      return 0
58
59
60  if __name__ == '__main__':
61      main()
```

Początek kodu omawialiśmy już w *Podstawach*. W podświetlonym fragmencie przede wszystkim importujemy omawianą bibliotekę oraz klasę *Vec3* reprezentującą położenie w MC. Polecenie `turtle = mcturtle.MinecraftTurtle(mc, start)` tworzy obiekt "żółwia" w podanym położeniu (`start = Vec3(0, 1, 0)`).

Żółwiem sterujemy za pomocą m.in. następujących metod:

- `speed()` – ustawia prędkość budowania: 0 – brak animacji, 1 – b. wolno, 10 – najszybciej;
- `penblock()` – określamy blok, którym rysujemy ślad;
- `forward(x)` – idź do przodu o *x* "kroków";
- `right(x)`, `left(x)` – obróć się w prawo/lewo o *x* stopni;

Wywołanie przykładowej funkcji `kwadraty()` umieszczamy w funkcji głównej i testujemy kod.

Okna

Przed funkcją główną `main()` dopisujemy kolejną przykładową funkcję:

```
52  def okna():
53      # Funkcja rysuje kształt okien w pionie
54      turtle.penblock(block.WOOD)
55      turtle.setposition(10, 2, 0)
56      turtle.up(90)
57      turtle.forward(14)
58      turtle.down(90)
59      turtle.setposition(-10, 2, 0)
60      turtle.up(90)
61      turtle.forward(14)
62      turtle.down(90)
63      turtle.right(90)
64      turtle.forward(19)
65      turtle.setposition(0, 2, 0)
66      turtle.up(90)
67      turtle.forward(13)
68      turtle.setposition(9, 10, 0)
69      turtle.down(90)
70      turtle.left(180)
71      turtle.forward(19)
```

W podanym kodzie mamy kilka nowych metod:

- `setposition(x, y, z)` – ustawia "żółwia" na podanej pozycji;
- `up(x)` – obróć się do góry o *x* stopni;
- `down(x)` – obróć się w dół o *x* stopni;

Dopisz wywołanie funkcji `okna()` do funkcji głównej i wykonaj skrypt.

Szlaczek

Jeszcze jedna funkcja przed funkcją `main()`:

```
74  def szlaczek():
75      # Funkcja rysuje przerywaną linię
76      turtle.penblock(block.MELON)
77      turtle.setx(-15)
78      turtle.sety(2)
79      turtle.setz(15)
80      turtle.left(180)
81      for i in range(8):
82          if (i % 2 == 0):
83              turtle.forward(1)
84          else:
85              turtle.forward(3)
86          turtle.penup()
87          turtle.forward(2)
88          turtle.pendown()
```

Nowe metody to:

- `setx(x)`, `setx(y)`, `setx(z)` – metody ustawiają składowe pozycji; jest też metoda `position()`, która zwraca pozycję;
- `penup()`, `pendown()` – podniesienie/opuszczenie "pędzla", dodatkowo funkcja `isdown()` sprawdza, czy pędzel jest opuszczony.

Po wywołaniu kolejno w funkcji głównej wszystkich powyższych funkcji otrzymamy następującą budowlę:

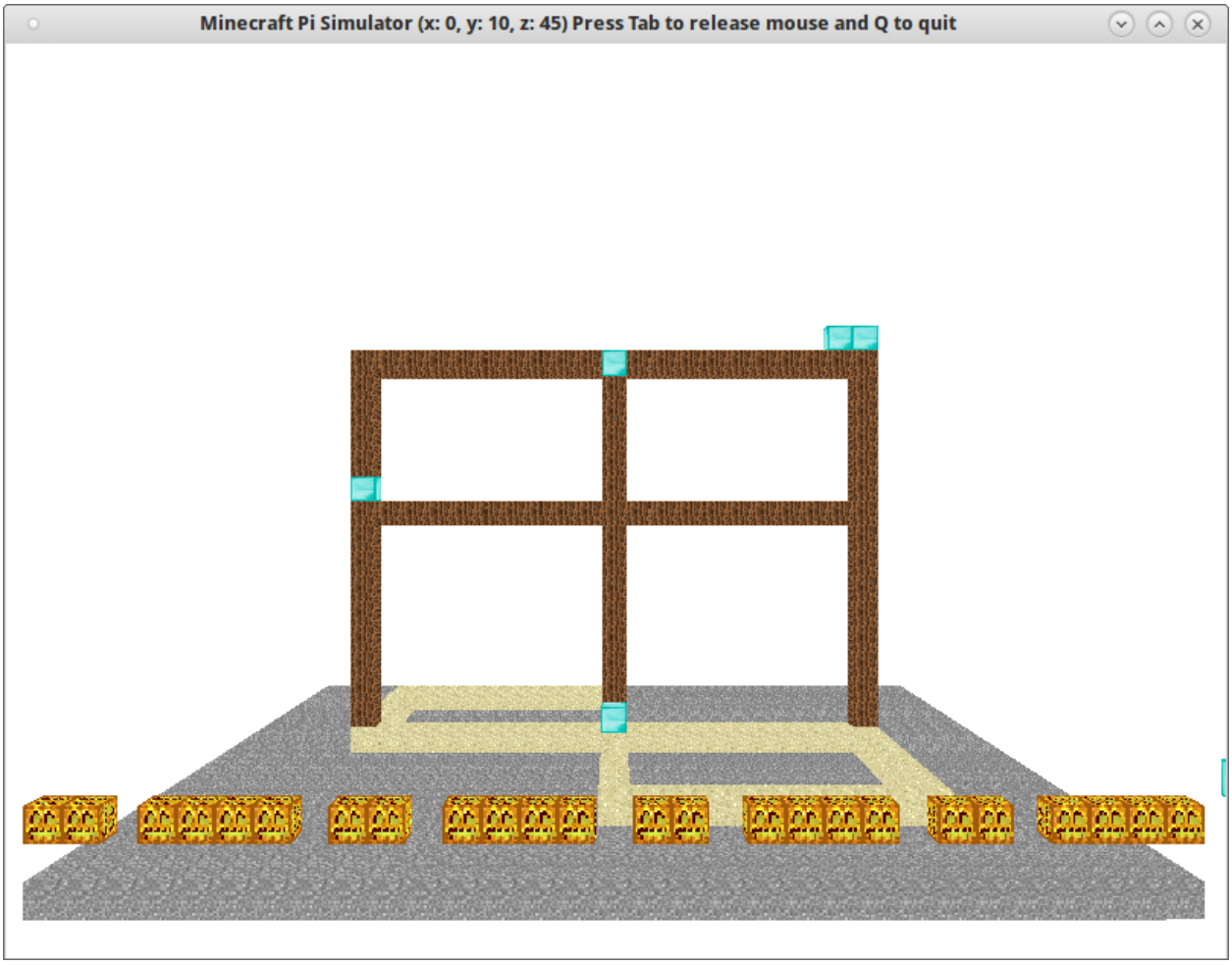
Ćwiczenia

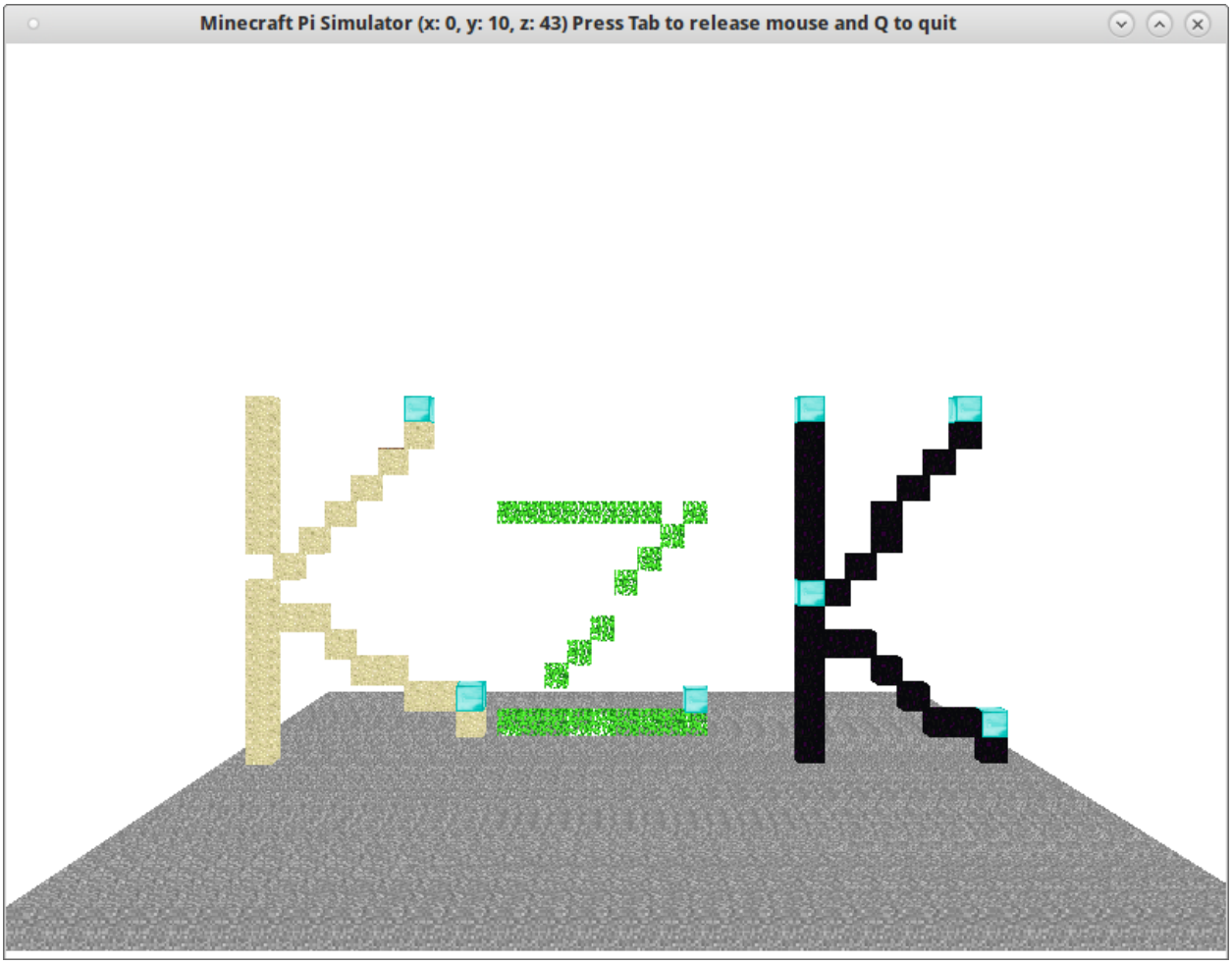
1. Napisz kod, który zbuduje napis "KzK" podobny do pokazanego niżej.
2. Napisz kod, który zbuduje sześcian. Przekształć go w funkcję, która buduje sześcian o podanej długości boku z podanego punktu.

Przykłady

Prawdziwie widowiskowe efekty uzyskamy przy wykorzystaniu pętli. Zapisz skrypt `mcsim-turtle.py` pod nazwą `mcpi-turtle.py` i dostosuj go do uruchomienia na serwerze *MC Pi*. W tym celu zamień ciąg "local" w importach na "mcpi" oraz podaj adres IP serwera *MC Pi* w poleceniu tworzącym połączenie. Następnie umieść w pliku kody poniższych funkcji i po kolei je przetestuj dodając ich wywołania w funkcji głównej.

```
91  def slonce():
92      turtle.setposition(-20, 3, 0)
93      turtle.speed(0)
94      turtle.penblock(block.GOLD_BLOCK)
95      while True:
96          turtle.forward(80)
97          turtle.left(165)
98          x, y, z = turtle.position
99          print max(x, z)
100         if abs(max(x, z)) < 1:
101             break
102
103
104 def wielokat(n):
```


Minecraft Pi Simulator (x: 0, y: 10, z: 45) Press Tab to release mouse and Q to quit


Minecraft Pi Simulator (x: 0, y: 10, z: 43) Press Tab to release mouse and Q to quit

```
105     turtle.setposition(15, 3, -18)
106     turtle.speed(0)
107     turtle.penblock(block.OBSIDIAN)
108     for i in range(n):
109         turtle.forward(10)
110         turtle.right(360 / n)
111
112
113 def main():
114     mc.postToChat("Biblioteka minecraftturtle")  # wysłanie komunikatu do mc
115     # plac(-15, 0, -15, 30)
116     # kwadraty()
117     # okna()
118     # szlaczek()
119     plac(-80, 0, -80, 160)
120     slonce()
121     wielokat(10)
122     return 0
```

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Funkcje w mcpi

O Minecrafcie w wersji na Raspberry Pi myśleć można jak o atrakcyjnej formie wizualizacji tego co można przedstawić w grafice dwu- lub trójwymiarowej. Zobaczmy zatem jakie budowle otrzymamy, wyliczając współrzędne bloków za pomocą funkcji matematycznych. Przy okazji niejako przypomnimy sobie użycie opisywanej już w naszych scenariuszach biblioteki *matplotlib*, która jest dedykowanym dla Pythona środowiskiem tworzenia wykresów 2D.

Funkcja liniowa

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu `mcpi-sim` pod nazwą `mcpi-funkcje.py`:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import numpy as np  # import biblioteki do obliczeń naukowych
import matplotlib.pyplot as plt  # import biblioteki do tworzenia wykresów
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # wpisz dowolną nazwę użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # wpisz dowolną nazwę komputera

mc = minecraft.Minecraft.create("192.168.1.10")  # połaczenie z mc


def plac(x, y, z, roz=10, gracz=False):
    """
    Funkcja tworzy podłoże i wypełnia sześcienny obszar od podanej pozycji,
    opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    podloga = block.STONE
    wypelniacz = block.AIR

    # podloga i czyszczenie
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def wykres(x, y, tytul="Wykres funkcji", *extra):
    """
    Funkcja wizualizuje wykres funkcji, której argumenty zawiera lista x
    a wartości lista y i ew. dodatkowe listy w parametrze *extra
    """
    if len(extra):
        plt.plot(x, y, extra[0], extra[1])  # dwa wykresy na raz
    else:
        plt.plot(x, y)
    plt.title(tytul)
    # plt.xlabel(podpis)
    plt.grid(True)
    plt.show()


def fun1(blok=block.IRON_BLOCK):
    """
```

```
    Funkcja f(x) = a*x + b
    """
    a = int(raw_input('Podaj współczynnik a: '))
    b = int(raw_input('Podaj współczynnik b: '))
    x = range(-10, 11)  # lista argumentów x = <-10;10> z krokiem 1
    y = [a * i + b for i in x]  # wyrażenie listowe
    print x, "\n", y
    wykres(x, y, "f(x) = a*x + b")
    for i in range(len(x)):
        mc.setBlock(x[i], 1, y[i], blok)


def main():
    mc.postToChat("Funkcje w Minecrafcie")  # wysłanie komunikatu do mc
    plac(-80, 0, -80, 160)
    mc.player.setPos(22, 10, 10)
    fun1()
    return 0


if __name__ == '__main__':
    main()
```

Większość kodu powinna być już zrozumiała, czyli importy bibliotek, nawiązywania połączenia z serwerem MC Pi, czy funkcja `plac()` tworząca przestrzeń do testów. Podobnie funkcja `wykres()`, która pokazuje nam graficzną reprezentację funkcji za pomocą biblioteki *matblotlib*. Na uwagę zasługuje w niej tylko parametr `*extra`, który pozwala przekazać argumenty i wartości dodatkowej funkcji.

Funkcja `fun1()` pobiera od użytkownika dwa współczynniki i odwzorowuje argumenty z dziedziny <-10;10> na wartości wg liniowego równania: `f(x) = a * x + b`. Przeciwdziedzinę można byłoby uzyskać "na piechotę" za pomocą kodu:

```
y = []
for i in x:
    y.append(a * i + b)
```

– ale efektywniejsze jest *wyrażenie listowe*: `y = [a * i + b for i in x]`. Po zobrazowaniu wykresu za pomocą funkcji funkcji `wykres()` i biblioteki *matplotlib* "budujemy" ją w MC Pi w pętli odczytującej wyliczone pary argumentów i wartości funkcji, stanowiących współrzędne kolejnych bloków umieszczanych poziomo.

Uruchom i przetestuj omówiony kod podając współczynniki np. *4* i *6*.

Układ współrzędnych

Spróbujmy pokazać w Mc Pi układ współrzędnych oraz ułatwić "budowanie" wykresów za pomocą osobnej funkcji. Po funkcji `wykres()` umieszczamy w pliku `mcpi-funkcje.py` nowy kod:

```
def uklad(blok=block.OBSIDIAN):
    """
    Funkcja rysuje układ współrzędnych
    """
    for i in range(-80, 81, 2):
        mc.setBlock(i, -1, 0, blok)
        mc.setBlock(0, -1, i, blok)
        mc.setBlock(0, i, 0, blok)

```

```
61
62 def rysuj(x, y, z, blok=block.IRON_BLOCK):
63     """
64     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
65     w punktach wyznaczonych przez pary elementów list x, y lub x, z
66     """
67     czylista = True if len(y) > 1 else False
68     for i in range(len(x)):
69         if czylista:
70             print(x[i], y[i])
71             mc.setBlock(x[i], y[i], z[0], blok)
72         else:
73             print(x[i], z[i])
74             mc.setBlock(x[i], y[0], z[i], blok)
```

– a pętlę tworzącą wykres w funkcji `fun1()` zastępujemy wywołaniem:

```
rysuj(x, y, [1], blok)
```

Funkcja `rysuj()` potrafi zbudować bloki zarówno w poziomie, jak i w pionie w zależności od tego, czy lista wartości funkcji przekazana zostanie jako parametr *y* czy też *z*. Do rozpoznania tego wykorzystujemy zmienną sterującą ustawianą w instrukcji: `czylista = True if len(y) > 1 else False`.

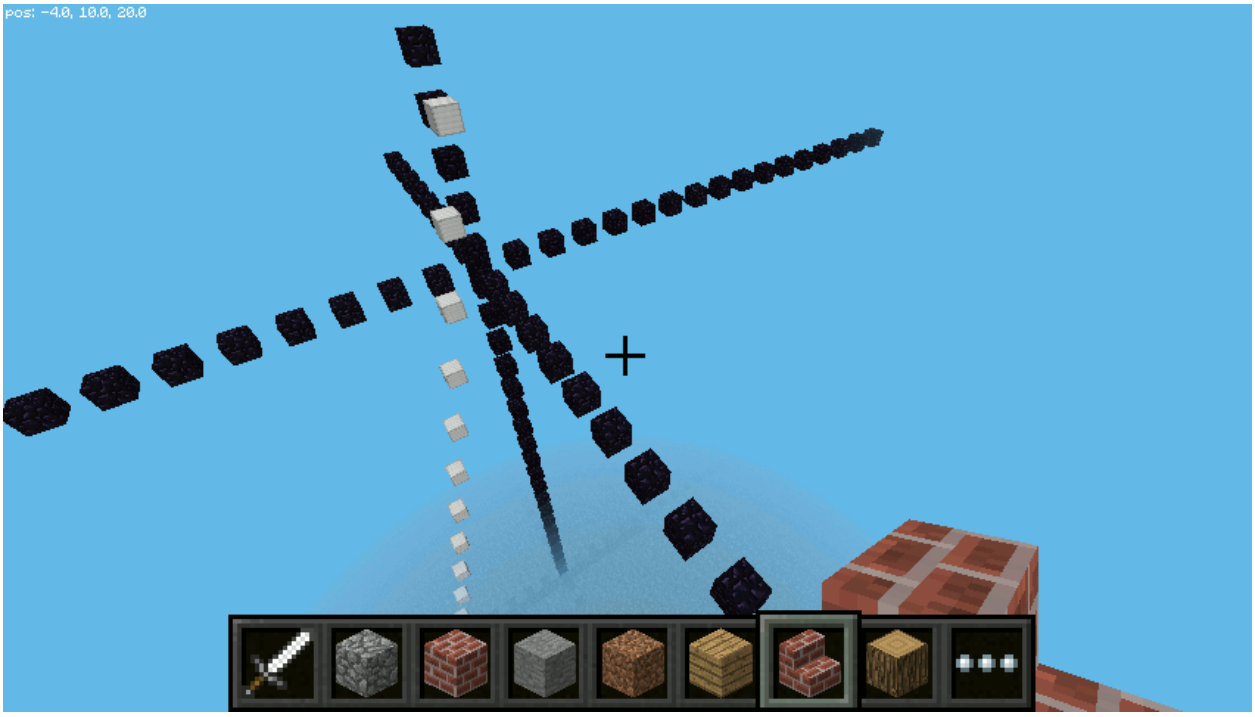
Zawartość funkcji `main()` zmieniamy na:

```
90 def main():
91     mc.postToChat("Funkcje w Minecrafcie")  # wysłanie komunikatu do mc
92     plac(-80, -40, -80, 160)
93     mc.player.setPos(-4, 10, 20)
94     uklad()
95     fun1()
96     return 0
```

Po uruchomieniu zmienionego kodu powinniśmy zobaczyć wykres naszej funkcji w pionie.

Kod "budujący" wykresy funkcji możemy urozmaicić wykorzystując poznaną wcześniej bibliotekę *minecraftstuff*. Poniżej funkcji `rysuj()` dodajemy:

```
77 def rysuj_linie(x, y, z, blok=block.IRON_BLOCK):
78     """
79     Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
80     w punktach wyznaczonych przez pary elementów list x, y lub x, z
81     przy użyciu metody drawLine()
82     """
83     import local.minecraftstuff as mcstuff
84     mcfig = mcstuff.MinecraftDrawing(mc)
85     czylista = True if len(y) > 1 else False
86     for i in range(len(x) - 1):
87         x1 = int(x[i])
88         x2 = int(x[i + 1])
89         if czylista:
90             y1 = int(y[i])
91             y2 = int(y[i + 1])
92             print (x1, y1, z[0], x2, y2, z[0])
93             mcfig.drawLine(x1, y1, z[0], x2, y2, z[0], blok)
94         else:
95             z1 = int(z[i])
96             z2 = int(z[i + 1])
```




```
97              print (x1, y[0], z1, x2, y[0], z2)
98              mcfig.drawLine(x1, y[0], z1, x2, y[0], z2, blok)
```

– a wywołanie `rysuj()` w funkcji `fun1()` zmieniamy na `rysuj_linie()`. Sprawdź rezultat.

Kolejne funkcje

W pliku `mcpi-funkcje.py` tuż nad funkcją główną `main()` umieszczamy kod wyliczający dziedziny i przeciwdziedziny dwóch kolejnych funkcji:

```
114 def fun2(blok=block.REDSTONE_ORE):
115     """
116     Wykres funkcji f(x), gdzie x = <-1;2> z krokiem 0.15, przy czym
117     f(x) = x/(x+2) dla x >= 1
118     f(x) = x*x/3 dla x < 1 i x > 0
119     f(x) = x/(-3) dla x <= 0
120     """
121     x = np.arange(-1, 2.15, 0.15)  # lista argumentów x
122     y = []  # lista wartości f(x)
123 
124     for i in x:
125         if i <= 0:
126             y.append(i / -3)
127         elif i < 1:
128             y.append(i ** 2 / 3)
129         else:
130             y.append(i / (i + 2))
131     wykres(x, y, "Funkcja mieszana")
132     x = [round(i * 20, 2) for i in x]
133     y = [round(i * 20, 2) for i in y]
```

```
    print x, "\n", y
    rysuj(x, y, [1], blok)


def fun3(blok=block.LAPIS_LAZULI_BLOCK):
    """
    Funkcja f(x) = log2(x)
    """
    x = np.arange(0.1, 41, 1)  # lista argumentów x
    y = [np.log2(i) for i in x]
    y = [round(i, 2) * 2 for i in y]
    print x, "\n", y
    wykres(x, y, "Funkcja logarytmiczna")
    rysuj(x, y, [1], blok)


def main():
    mc.postToChat("Funkcje w Minecrafcie")  # wysłanie komunikatu do mc
    plac(-80, -20, -80, 160)
    mc.player.setPos(-8, 10, 26)
    uklad(block.DIAMOND_BLOCK)
    fun1()
    fun2()
    fun3()
    return 0
```

W funkcji `fun2()` wartości dziedziny uzyskujemy dzięki metodzie `arange(start, stop, step)` z biblioteki *numpy*. Potrafi ona generować listę wartości zmiennopozycyjnych w podanym zakresie <*start;stop*) z określonym krokiem *step*.

Przeciwdziedzinę wyliczamy w pętli w zależności od przedziałów, w których znajdują się argumenty, za pomocą złożonej instrukcji warunkowej. Następnie wartości zarówno dziedziny, jak i przeciwdziedziny przeskalowujemy w wyrażeniach listowych, mnożąc przez stały współczynnik, aby wykres w MC Pi był większy i wyraźniejszy. Przy okazji współrzędne zaokrąglamy do dwóch miejsc po przecinku, np.: `x = [round(i * 20, 2) for i in x]`.

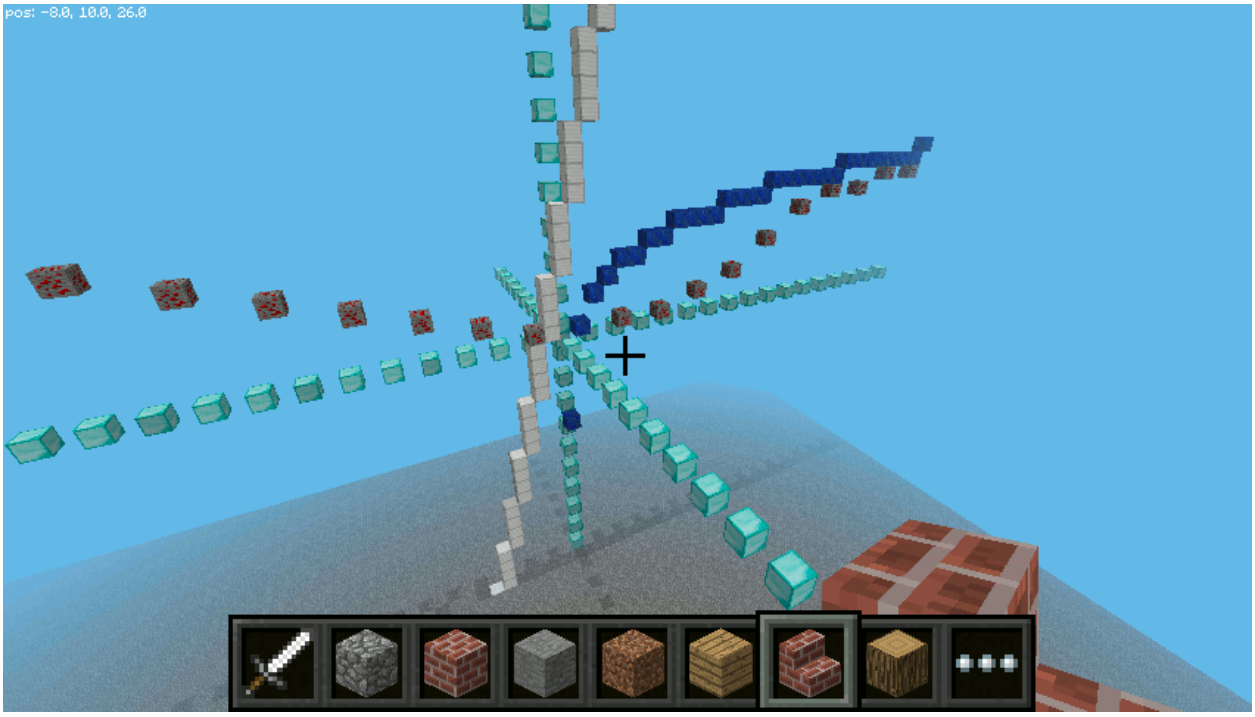
W funkcji `fun3()` w podobny jak powyżej sposób obliczamy argumenty i wartości funkcji logarytmicznej.

Na koniec zmieniamy też nieco wywołania w funkcji głównej. Przetestuj podany kod.

Funkcja kwadratowa

Przygotujemy wykres funkcji kwadratowej. Przed funkcją główną umieszczamy następujący kod:

```
def fkw(x, a=0.3, b=0.1, c=0):
    return a * x**2 + b * x + c


def fkwadratowa():
    """
    Funkcja przygotowuje dziedzinę funkcji kwadratowej
    oraz dwie przeciwdziedziny, druga z odwróconym znakiem. Następnie
    buduje ich wykresy w poziomie i w pionie.
    """
    while True:
        lewy = float(raw_input("Podaj lewy kraniec przedziału: "))
        prawy = float(raw_input("Podaj prawy kraniec przedziału: "))
```



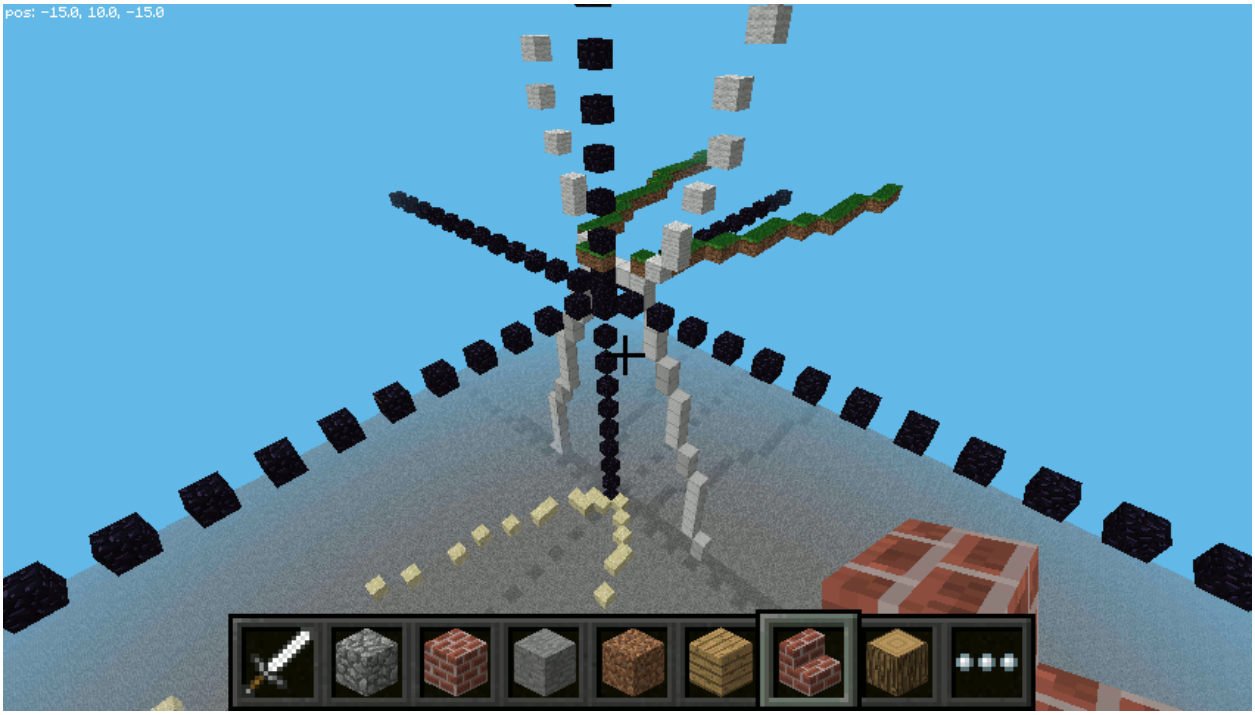

```
163         if lewy * prawy < 1 and lewy <= prawy:
164             break
165     print lewy, prawy
166
167     # x = np.arange(lewy, prawy, 0.2)
168     x = np.linspace(lewy, prawy, 60, True)
169     x = [round(i, 2) for i in x]
170     y1 = [fkw(i) for i in x]
171     y1 = [round(i, 2) for i in y1]
172     y2 = [-fkw(i) for i in x]
173     y2 = [round(i, 2) for i in y2]
174     print x, "\n", y1, "\n", y2
175     wykres(x, y1, "Funkcja kwadratowa", x, y2)
176     rysuj_linie(x, [1], y1, block.GRASS)
177     rysuj(x, [1], y2, block.SAND)
178     rysuj(x, y1, [1], block.WOOL)
179     rysuj_linie(x, y2, [1], block.IRON_BLOCK)
180
181
182 def main():
183     mc.postToChat("Funkcje w Minecrafcie")  # wysłanie komunikatu do mc
184     plac(-80, -20, -80, 160)
185     mc.player.setPos(-15, 10, -15)
186     uklad(block.OBSIDIAN)
187     fkwadratowa()
188     return 0
```

Na początku w funkcji `fkwadratowa()` pobieramy od użytkownika przedział, w którym budować będziemy funkcję. Wymuszamy przy tym w pętli `while`, aby lewa i prawa granica miały inne znaki. Dalej używamy funkcji `linspace(start, stop, num, endpoint)`, która generuje listę *num* wartości od punktu początkowego do końcowego, który uwzględniany jest, jeżeli argument *endpoint* ma wartość *True*. Kolejne wyrażenia listowe wyliczają przeciwdziedziny i zaokrąglają wartości do 2 miejsc po przecinku.

Sama funkcja kwadratowa `a*x^2 + b*x + c` zdefiniowana jest w funkcji `fkw()`, do której przekazujemy kolejne argumenty dziedziny i opcjonalnie współczynniki.

Instrukcje `rysuj()` i `rysuj_linie()` dzięki przekazywaniu przeciwdziedziny jako 2. lub 3. argumentu budują wykresy w poziomie lub w pionie za pomocą pojedynczych lub połączonych bloków.

Po przygotowaniu w funkcji głównej miejsca, ustawieniu gracza, narysowaniu układu i podaniu przedziału <-20, 20> otrzymamy konstrukcję podobną do poniższej.



Po zmianie funkcji na *x**2 / 3* można otrzymać:

Zwróciłeś uwagę na to, że jeden z wykresów opada?

Funkcje trygonometryczne

Na koniec zobrazujemy funkcje trygonometryczne. Przed funkcją główną dopisujemy kod:

```
182 def trygon():
183     x1 = np.arange(-50.0, 50.0, 1)
184     y1 = 5 * np.sin(0.1 * np.pi * x1)
185     y1 = [round(i, 2) for i in y1]
186     print x1, "\n", y1
187
188     x2 = range(0, 361, 10)  # lista argumentów x
189     y2 = [None if i == 90 or i == 270 else np.tan(i * np.pi / 180) for i in x2]
190     x2 = [i // 10 for i in x2]
191     y2 = [round(i * 3, 2) if i is not None else None for i in y2]
192     print x2, "\n", y2
193     wykres(x1, y1, "Funkcje sinus i tangens", x2, y2)
194
195     del x2[9]  # usuń 10 element listy
196     del y2[9]  # usuń 10 element listy
```


```
197     del x2[x2.index(27)]  # usuń element o wartości 27
198     del y2[y2.index(None)]  # usuń element None
199     print x2, "\n", y2
200     rysuj(x1, [1], y1, block.GOLD_BLOCK)
201     rysuj(x2, y2, [1], block.OBSIDIAN)
202
203
204 def main():
205     mc.postToChat("Funkcje w Minecrafcie")  # wysłanie komunikatu do mc
206     plac(-80, -20, -80, 160)
207     mc.player.setPos(17, 17, 24)
208     uklad(block.DIAMOND_BLOCK)
209     trygon()
210     return 0
```

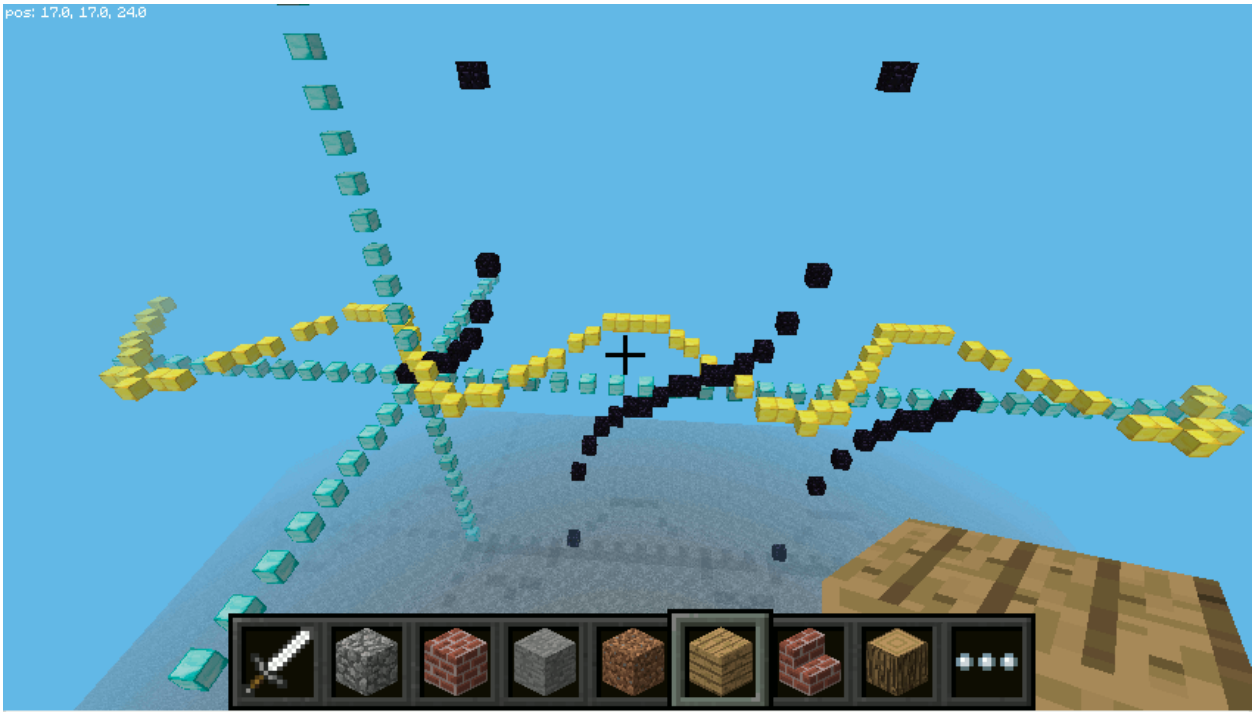
W funkcji `trygon()` na początku wyliczamy dziedzinę i przeciwdziedzinę funkcji *5 * sin(0.1 * Pi * x)*, przy czym wartości *y* zaokrąglamy.

Dalej generujemy argumenty *x* dla funkcji tangens w przedziale od 0 do 360 co 10 stopni. Obliczając wartości *y* za pomocą wyrażenia listowego `y2 = [None if i == 90 or i == 270 else np.tan(i * np.pi / 180) for i in x2]` dla argumentów *90* i *270* wstawiamy *None* (czyli nic), ponieważ dla tych argumentów funkcja nie przyjmuje wartości. Dzięki temu uzyskamy poprawny wykres w *matplotlib*.

Aby wykresy obydwu funkcji nałożyły się na siebie, używając wyrażenia listowego, skalujemy argumenty i wartości funkcji tangens. Pierwsze dzielimy przez 10, drugie mnożymy przez 3 (i przy okazji zaokrąglamy). Konstrukcja `if i is not None else None` zapobiega wykonywaniu operacji dla wartości `None`, co generowałoby błędy.

Przygotowanie danych do zwizualizowania w Minecrafcie wymaga usunięcia 2 argumentów z listy *x2* oraz odpowiadających im wartości `None` z listy *y2*, ponieważ nie tworzą one poprawnych współrzędnych. Pierwszą parę usuwamy podając wprost odpowiedni indeks w instrukcjach `del x2[9]` i `del y2[9]`. Indeksy elementów drugiej pary najpierw wyszukujemy `x2.index(27)` i `y2.index(None)`, a później przekazujemy do instrukcji usuwającej `del()`.

Po wywołaniu z ustawieniami w funkcji głównej takimi jak w powyższym kodzie powinniśmy zobaczyć obraz podobny do poniższego.



Ćwiczenia

Warto poeksperymentować z wzorami funkcji, ich współczynnikami, wartościami przedziałów i ilością argumentów, aby zbadać jak te zmiany wpływają na ich reprezentację graficzną.

Można też rysować mieszać metody rysujące wykresy (`rysuj()`, `rysuj_linie()`), kolejność przekazywania im parametrów, rodzaje bloków itp. Spróbuj np. budować wykresy z piasku (`block.STONE`) ponad powierzchnią.

Źródła:

- `Skrypty mcpi-funkcje`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Algorytmy

W tym scenariuszu spróbujemy pokazać w Minecrafcie Pi algorytm symulujący ruchy Browna oraz algorytm stosujący metodę Monte Carlo do wyliczenia przybliżonej wartości liczby Pi.

Ruchy Browna

Za pomocą wybranego edytora utwórz pusty plik, umieść w nim podany niżej kod i zapisz w katalogu `mcpi-sim` pod nazwą `mcpi-rbrowna.py`:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
```

```
import os
import numpy as np  # import biblioteki do obliczeń naukowych
import matplotlib.pyplot as plt  # import biblioteki do tworzenia wykresow
from random import randint
from time import sleep
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # wpisz dowolną nazwę użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # wpisz dowolną nazwę komputera

mc = minecraft.Minecraft.create("192.168.1.10")  # połączenie z serwerem


def plac(x, y, z, roz=10, gracz=False):
    """
    Funkcja tworzy podłoże i wypełnia sześcienny obszar od podanej pozycji,
    opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    podloga = block.SAND
    wypelniacz = block.AIR

    # podloga i czyszczenie
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def wykres(x, y, tytul="Wykres funkcji", *extra):
    """
    Funkcja wizualizuje wykres funkcji, której argumenty zawiera lista x
    a wartości lista y i ew. dodatkowe listy w parametrze *extra
    """
    if len(extra):
        plt.plot(x, y, extra[0], extra[1])  # dwa wykresy na raz
    else:
        plt.plot(x, y, "o:", color="blue", linewidth="3", alpha=0.8)
    plt.title(tytul)
    plt.grid(True)
    plt.show()


def rysuj(x, y, z, blok=block.IRON_BLOCK):
    """
    Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
    w punktach wyznaczonych przez pary elementów list x, y lub x, z
    """
    czylista = True if len(y) > 1 else False
    for i in range(len(x)):
        if czylista:
            print(x[i], y[i])
```

```
            mc.setBlock(x[i], y[i], z[0], blok)
        else:
            print(x[i], z[i])
            mc.setBlock(x[i], y[0], z[i], blok)


def ruchyBrowna():

    n = int(raw_input("Ile ruchów? "))
    r = int(raw_input("Krok przesunięcia? "))

    x = y = 0
    lx = [0]  # lista odciętych
    ly = [0]  # lista rzędnych

    for i in range(0, n):
        # losujemy kąt i zamieniamy na radiany
        rad = float(randint(0, 360)) * np.pi / 180
        x = x + r * np.cos(rad)  # wylicz współrzędną x
        y = y + r * np.sin(rad)  # wylicz współrzędną y
        x = int(round(x, 2))  # zaokrągl
        y = int(round(y, 2))  # zaokrągl
        print(x, y)
        lx.append(x)
        ly.append(y)

    # oblicz wektor końcowego przesunięcia
    s = np.fabs(np.sqrt(x**2 + y**2))
    print "Wektor przesunięcia: {:.2f}".format(s)

    wykres(lx, ly, "Ruchy Browna")
    rysuj(lx, [1], ly, block.WOOL)


def main():
    mc.postToChat("Ruchy Browna")  # wysłanie komunikatu do mc
    plac(-80, -20, -80, 160)
    plac(-80, 0, -80, 160)
    ruchyBrowna()
    return 0


if __name__ == '__main__':
    main()
```

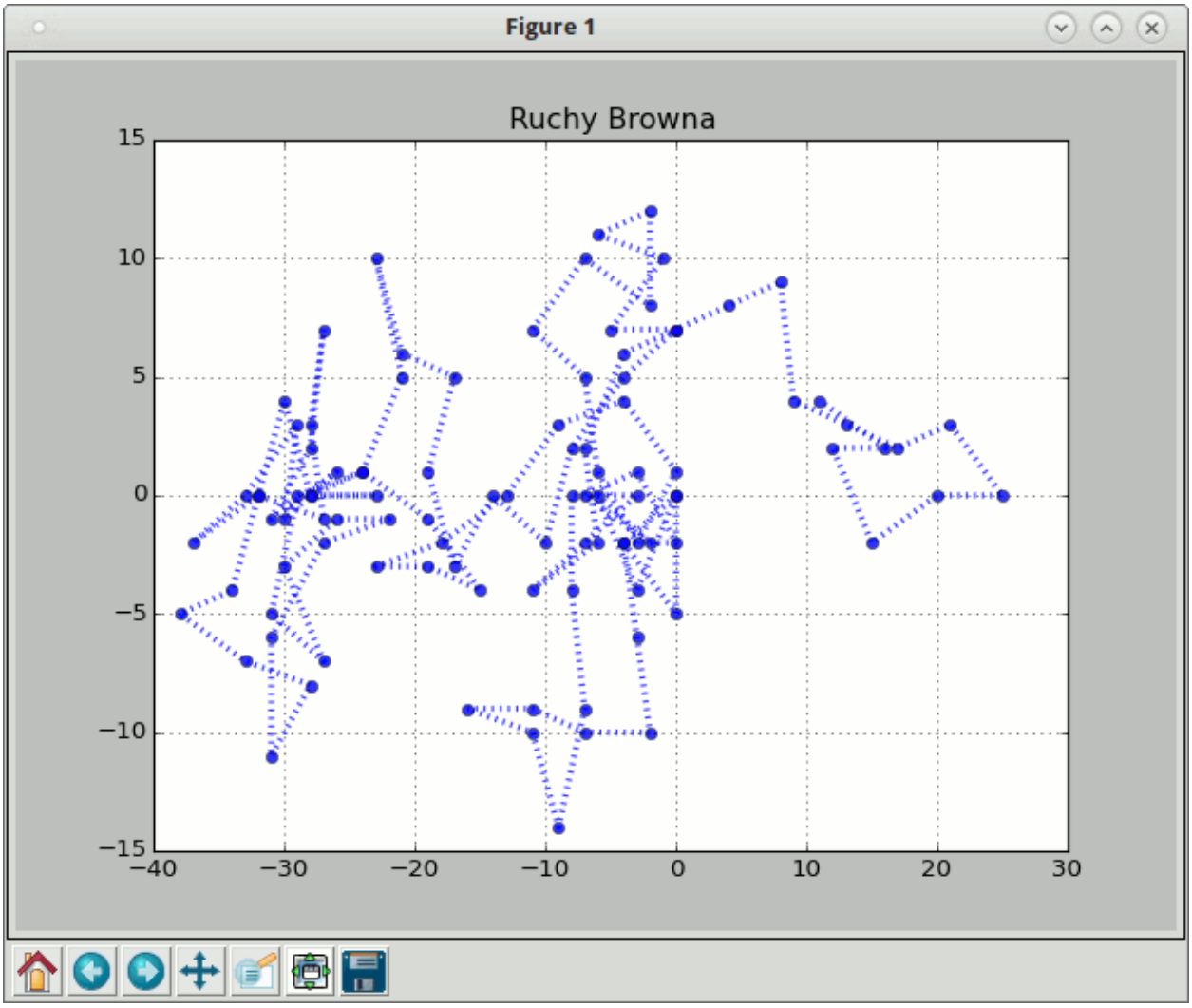
Większość kodu powinna być już zrozumiała. Importy bibliotek, nawiązywanie połączenia z serwerem MC Pi, funkcje `plac()`, `wykres()` i `rysuj()` omówione zostały w poprzednim scenariuszu *Funkcje w mcpi*.

W funkcji `ruchyBrowna()` na początku pobieramy od użytkownika ilość ruchów cząsteczki do wygenerowania oraz ich długość, co ma znaczenie podczas ich odwzorowywania w świecie MC Pi. Następnie w pętli:

- losujemy kąt wskazujący kierunek ruchu cząsteczki,
- wyliczamy współrzędne kolejnego punktu korzystając z funkcji *cos()* i *sin()* (np. `x = x + r * np.cos(rad)`),
- zaokrąglamy wyniki do 2 miejsc po przecinku (np. `x = int(round(x, 2))`) i drukujemy,
- na koniec dodajemy obliczone współrzędne do list odciętych i rzędnych (np. `lx.append(x)`).

Po wyjściu z pętli obliczamy długość wektora przesunięcia, korzystając z twierdzenia Pitagorasa, i drukujemy wynik z dokładnością do dwóch miejsc po przecinku (wyrażenie formatujące: `{:.2f}`).

Po tych operacjach pozostaje wykreślenie ruchu cząsteczki w *matplotlib* i wyznaczenie go w Minecrafcie.



Wskazówka: Przed uruchomieniem wizualizacji warto ustawić Steve'a w tryb lotu (dwukrotne naciśnięcie spacji).

(Nie)powtarzalność

Kilkukrotne uruchomienie dotychczasowego kodu pokazuje, że za każdym razem generowany jest inny tor ruchu cząsteczki. Z jednej strony to dobrze, bo to potwierdza przypadkowość symulowanych ruchów, z drugiej strony przydatna byłaby możliwość zapamiętania wyjątkowo malowniczych sekwencji.

Zmienimy więc funkcję `ruchyBrowna()` tak, aby zapisywała i ewentualnie odczytywała wygenerowany i zapisany ruch cząsteczki. Musimy też dodać dwie funkcje narzędziowe zapisujące i czytające dane.

```
68 def ruchyBrowna(dane=[]):
69
```

```
70      if len(dane):
71          lx, ly = dane  # rozpakowanie listy
72          x = lx[-1]  # ostatni element lx
73          y = ly[-1]  # ostatni element ly
74      else:
75          n = int(raw_input("Ile ruchów? "))
76          r = int(raw_input("Krok przesunięcia? "))
77
78          x = y = 0
79          lx = [0]  # lista odciętych
80          ly = [0]  # lista rzędnych
81
82          for i in range(0, n):
83              # losujemy kąt i zamieniamy na radiany
84              rad = float(randint(0, 360)) * np.pi / 180
85              x = x + r * np.cos(rad)  # wylicz współrzędną x
86              y = y + r * np.sin(rad)  # wylicz współrzędną y
87              x = int(round(x, 2))  # zaokrągl
88              y = int(round(y, 2))  # zaokrągl
89              print(x, y)
90              lx.append(x)
91              ly.append(y)
92
93      # oblicz wektor końcowego przesunięcia
94      s = np.fabs(np.sqrt(x**2 + y**2))
95      print "Wektor przesunięcia: {:.2f}".format(s)
96
97      wykres(lx, ly, "Ruchy Browna")
98      rysuj(lx, [1], ly, block.WOOL)
99      if not len(dane):
100         zapisz_dane((lx, ly))
101
102
103 def zapisz_dane(dane):
104     """Funkcja zapisuje dane w formacie json w pliku"""
105     import json
106     plik = open('rbrowna.log', 'w')
107     json.dump(dane, plik)
108     plik.close()
109
110
111 def czytaj_dane():
112     """Funkcja odczytuje dane w formacie json z pliku"""
113     import json
114     dane = []
115     nazwapliku = raw_input("Podaj nazwę pliku z danymi lub naciśnij ENTER: ")
116     if os.path.isfile(nazwapliku):
117         with open(nazwapliku, "r") as plik:
118             dane = json.load(plik)
119     else:
120         print "Podany plik nie istnieje!"
121     return dane
122
123
124 def main():
125     mc.postToChat("Ruchy Browna")  # wysłanie komunikatu do mc
126     plac(-80, -20, -80, 160)
127     plac(-80, 0, -80, 160)
```

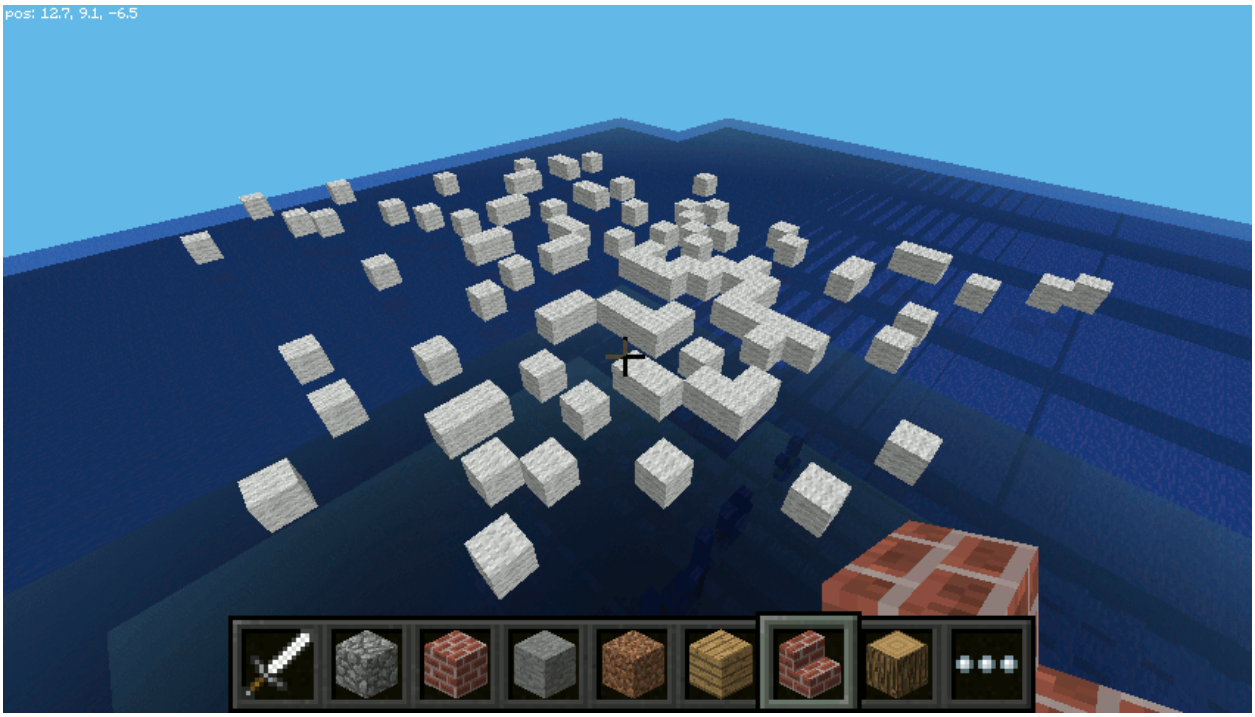
```
128     ruchyBrowna(czytaj_dane())
129     return 0
```

Z powyższego kodu wynika, że jeżeli funkcja `ruchyBrowna()` otrzyma niepustą listę danych (`if len(dane):`), wczyta z niej dane współrzędnych *x* i *y*. W przeciwnym wypadku generowane będą nowe, które zostaną zapisane: `zapisz_dane((lx, ly))`.

Funkcja `zapisz_dane()`, pobiera tuplę zawierającą listę współrzędnych *x* i *y*, otwiera plik o podanej nazwie do zapisu (`open('rbrowna.log', 'w')`) i zapisuje w nim dane w formacie json.

Funkcja `czytaj_dane()` prosi o podanie nazwy pliku z danymi, jeśli istnieje, zwraca dane zapisane w formacie *json*, które w funkcji `ruchyBrowna()` rozpakowywane są jako listy wartości *x* i *y*: `lx, ly = dane`. Jeżeli podany plik z danymi nie istnieje, zwracana jest pusta lista, a w funkcji `ruchyBrowna()` generowane są nowe dane.

W funkcji głównej zmieniamy wywołanie funkcji na `ruchyBrowna(czytaj_dane())` i testujemy zmieniony kod. Za pierwszym razem wciskamy `Enter`, generujemy i zapisujemy dane, za drugim razem podajemy nazwę pliku `rbrowna.log`.



Ruch cząsteczki

Do tej pory ruch cząsteczki wizualizowane był jako pojedyncze punkty. Możemy jednak pokazać pokonaną trasę liniowo, używając omawianej już biblioteki *minecraftstaff*. Pod funkcją `rysuj()` umieszczamy następującą funkcję:

```
68  def rysuj_linie(x, y, z, blok=block.IRON_BLOCK):
69      """
70      Funkcja wizualizuje wykres funkcji, umieszczając bloki w pionie/poziomie
71      w punktach wyznaczonych przez pary elementów list x, y lub x, z
72      przy użyciu metody drawLine()
73      """
74      import local.minecraftstuff as mcstuff
75      mcfig = mcstuff.MinecraftDrawing(mc)
```

```
    czylista = True if len(y) > 1 else False
    for i in range(len(x) - 1):
        x1 = int(x[i])
        x2 = int(x[i + 1])
        if czylista:
            y1 = int(y[i])
            y2 = int(y[i + 1])
            mc.setBlock(x2, y2, z[0], block.GRASS)
            mc.setBlock(x1, y1, z[0], block.GRASS)
            mcfig.drawLine(x1, y1, z[0], x2, y2, z[0], blok)
            mc.setBlock(x2, y2, z[0], block.GRASS)
            mc.setBlock(x1, y1, z[0], block.GRASS)
            print (x1, y1, z[0], x2, y2, z[0])
        else:
            z1 = int(z[i])
            z2 = int(z[i + 1])
            mc.setBlock(x1, y[0], z1, block.GRASS)
            mc.setBlock(x2, y[0], z2, block.GRASS)
            mcfig.drawLine(x1, y[0], z1, x2, y[0], z2, blok)
            mc.setBlock(x1, y[0], z1, block.GRASS)
            mc.setBlock(x2, y[0], z2, block.GRASS)
            print (x1, y[0], z1, x2, y[0], z2)
        sleep(1)  # przerwa na reklamę :-)
    mc.setBlock(0, 1, 0, block.OBSIDIAN)
    if czylista:
        mc.setBlock(x2, y2, z[0], block.OBSIDIAN)
    else:
        mc.setBlock(x2, y[0], z2, block.OBSIDIAN)
```

Jak widać, jest to zmodyfikowana funkcja, której użyliśmy po raz pierwszy w scenariuszu *Funkcje*. Zmiany dotyczą dodatkowych instrukcji typu `mc.setBlock(x2, y2, z[0], block.GRASS)`, których zadaniem jest zaznaczenie innymi blokami wylosowanych punktów reprezentujących ruch cząsteczki. Instrukcja `sleep(1)` wstrzymując budowanie na 1 sekundę wywołuje wrażenie animacji i pozwala śledzić na bieżąco budowany tor. Końcowe instrukcje służą zaznaczeniu początku i końca ruchu blokami obsydianu.

Na koniec trzeba w funkcji `ruchyBrowna()` zmienić wywołanie `rysuj()` na `rysuj_linie()`.

Eksperymenty

Uruchamiamy kod i eksperymentujemy. Dla 100 ruchów z krokiem przesunięcia 5 możemy uzyskać np. takie rezultaty:

Nic nie stoina przeszkodzie, żeby cząsteczka "ruszała się" w pionie nad i... pod wodą:

Liczba Pi

Mamy koło o promieniu *r*, którego środek umieszczamy w początku układu współrzędnych (0, 0). Na kole opisany jest kwadrat o boku *2r*. Spróbujmy to zbudować w MC Pi. W pliku `mcpi-lpi.py` umieszczamy kod:

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import os
import random
from time import sleep
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block
```


pos: 4.7, 20.0, 3.8


pos: 7.1, 20.0, -22.8


pos: 2.1, 5.5, 22.0


pos: 2.1, -5.7, 22.6

```
import local.minecraftstuff as mcstuff

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

mc = minecraft.Minecraft.create("192.168.1.10")  # połączenie z serwerem


def plac(x, y, z, roz=10, gracz=False):
    """Funkcja wypełnia sześcienny obszar od podanej pozycji
    powietrzem i opcjonalnie umieszcza gracza w środku.
    Parametry: x, y, z - współrzędne pozycji początkowej,
    roz - rozmiar wypełnianej przestrzeni,
    gracz - czy umieścić gracza w środku
    Wymaga: globalnych obiektów mc i block.
    """

    podloga = block.STONE
    wypelniacz = block.AIR

    # kamienna podłoże
    mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
    # czyszczenie
    mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
    # umieść gracza w środku
    if gracz:
        mc.player.setPos(x + roz / 2, y + roz / 2, z + roz / 2)


def model(promien, x, y, z):
    """
    Fukcja buduje obrys kwadratu, którego środek to punkt x, y, z
    oraz koło wpisane w ten kwadrat
    """

    mcfig = mcstuff.MinecraftDrawing(mc)
    obrys = block.SANDSTONE
    wypelniacz = block.AIR

    mc.setBlocks(x - promien, y, z - promien, x +
                 promien, y, z + promien, obrys)
    mc.setBlocks(x - promien + 1, y, z - promien + 1, x +
                 promien - 1, y, z + promien - 1, wypelniacz)
    mcfig.drawHorizontalCircle(0, 0, 0, promien, block.GRASS)


def liczbaPi():
    r = float(raw_input("Podaj promień koła: "))
    model(r, 0, 0, 0)


def main():
    mc.postToChat("LiczbaPi")  # wysłanie komunikatu do mc
    plac(-50, 0, -50, 100)
    mc.player.setPos(20, 20, 0)
    liczbaPi()
    return 0

```

```
67
68 if __name__ == '__main__':
69     main()
```

Funkcja `model()` działa podobnie do funkcji `plac()`, czyli na początku budujemy wokół środka układu współrzędnych płytę z bloku, który będzie zarysem kwadratu. Później budujemy drugą płytę o blok mniejszą z powietrza. Na koniec rysujemy koło.

Deszcz punktów

Teraz wyobraźmy sobie, że pada deszcz. Część kropel upada w obrębie kwadratu, ich ilość oznaczymy zmienną `ileKw`, a część również w obrębie koła – oznaczymy je zmienną `ileKo`. Ponieważ znamy promień koła, możemy ułożyć proporcję, zakładając, że stosunek pola koła do pola kwadratu równy będzie stosunkowi kropel w kole do kropel w kwadracie:

$$\frac{\Pi * r^2}{(2 * r)^2} = \frac{ileKo}{ileKw}$$

Z prostego przekształcenia tej równości możemy wyznaczyć liczbę Pi:

$$\Pi = \frac{4 * ileKo}{ileKw}$$

Uzupełniamy więc kod funkcji `liczbaPi()`:

```
55 def liczbaPi():
56     r = float(raw_input("Podaj promień koła: "))
57     model(r, 0, 0, 0)
58
59     # pobieramy ilość punktów w kwadracie
```

```
60      ileKw = int(raw_input("Podaj ilość losowanych punktów: "))
61      ileKo = 0  # ilość punktów w kole
62
63      blok = block.SAND
64      for i in range(ileKw):
65          x = round(random.uniform(-r, r))
66          y = round(random.uniform(-r, r))
67          print x, y
68          if abs(x)**2 + abs(y)**2 <= r**2:
69              ileKo += 1
70          # umieść blok w MC Pi
71          mc.setBlock(x, 10, y, blok)
72
73      mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
74      pi = 4 * ileKo / float(ileKw)
75      mc.postToChat("Pi w przyblizeniu: {:.10f}".format(pi))
76
```

Jak widać w nowym kodzie, na początku pobieramy od użytkownika ilość "kropel" deszczu, czyli punktów do wylosowania. Następnie w pętli losujemy ich współrzędne w przedziale <-*r;r*> w instrukcji typu: `x = round(random.uniform(-r, r), 10)`. Funkcja `uniform()` zwraca wartości zmiennoprzecinkowe, które zaokrąglamy do 10 miejsca po przecinku.

Korzystając z twierdzenia Pitagorasa układamy warunek pozwalający sprawdzić, które punkty "wpadły" do koła: `if abs(x)**2 + abs(y)**2 <= r**2:` – i zliczamy je.

Instrukcja `mc.setBlock(x, 10, y, blok)` rysuje punkty w MC Pi za pomocą bloków piasku (*SAND*), dzięki czemu uzyskujemy efekt spadania.

Wyliczenie wartości *Pi* i wydrukowanie jej jest prostą formalnością.

Uruchomienie powyższego kodu dla promienia 30 i 1000 punktów dało następujący efekt:

Jak widać, niektóre punkty po zaokrągleniu ich współrzędnych w MC Pi nakładają się na siebie.

Podkolorowanie

Punkty wpadające do koła mogłyby wyglądać inaczej niż poza nim. Można by to osiągnąć przez ustawienie różnych typów bloków w pętli `for`, ale tylko blok piaskowy daje efekt spadania. Zrobimy więc inaczej. Zmieniamy funkcję `liczbaPi()`:

```
55 def liczbaPi():
56     r = float(raw_input("Podaj promień koła: "))
57     model(r, 0, 0, 0)
58
59     # pobieramy ilość punktów w kwadracie
60     ileKw = int(raw_input("Podaj ilość losowanych punktów: "))
61     ileKo = 0  # ilość punktów w kole
62     wKwadrat = []  # pomocnicza lista punktów w kwadracie
63     wKolo = []  # pomocnicza lista punktów w kole
64
65     blok = block.SAND
66     for i in range(ileKw):
67         x = round(random.uniform(-r, r))
68         y = round(random.uniform(-r, r))
69         wKwadrat.append((x, y))
70         print x, y
71         if abs(x)**2 + abs(y)**2 <= r**2:
72             ileKo += 1
73             wKolo.append((x, y))
74
75         mc.setBlock(x, 10, y, blok)
76
77     sleep(5)
78     for pkt in set(wKwadrat) - set(wKolo):
79         x, y = pkt
80         mc.setBlock(x, i, y, block.OBSIDIAN)
81         for i in range(1, 3):
82             print x, i, y
83             if mc.getBlock(x, i, y) == 12:
84                 mc.setBlock(x, i, y, block.OBSIDIAN)
85
86     mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
87     pi = 4 * ileKo / float(ileKw)
88     mc.postToChat("Pi w przyblizeniu: {:.10f}".format(pi))
```

Deklarujemy dwie pomocnicze listy, do których zapisujemy w pętli współrzędne punktów należących do kwadratu i koła, np. `wKwadrat.append((x, y))`. Następnie wstrzymujemy wykonanie kodu na 5 sekund, aby bloki piasku zdążyły opaść. W wyrażeniu `set(wKwadrat) - set(wKolo)` każda lista zostaje przekształcona na zbiór, a następnie zostaje obliczona ich różnica. W efekcie otrzymujemy współrzędne punktów należących do kwadratu, ale nie do koła. Ponieważ niektóre bloki piasku układają się jeden na drugim, wychwytujemy je w pętli wewnętrznej `if mc.getBlock(x, i, y) == 12:` – i zmieniamy na obsydian.

Trzeci wymiar

Siła MC Pi tkwi w 3 wymiarze. Możemy bez większych problemów go wykorzystać. Na początku warto zauważyć, że w algorytmie wyliczania wartości liczby Pi nic się nie zmieni. Stosunek pola koła do pola kwadratu zastępujemy bowiem stosunkiem objętości walca, którego podstawa ma promień *r*, do objętości sześcianu o boku *2r*. Otrzymamy

zatem:

$$\frac{\Pi * r^2 * 2 * r}{(2 * r)^3} = \frac{ileKo}{ileKw}$$

Po przekształceniu skończymy na takim samym jak wcześniej wzorze, czyli:

$$\Pi = \frac{4 * ileKo}{ileKw}$$

Aby to wykreślić, zmienimy funkcje `model()`, `liczbaPi()` i `main()`. Sugerujemy, żeby dotychczasowy plik zapisać pod inną nazwą, np. `mcpi-lpi3D.py`, i wprowadzić następujące zmiany:

```
def model(r, x, y, z, klatka=False):
    """
    Fukcja buduje obrys kwadratu, którego środek to punkt x, y, z
    oraz koło wpisane w ten kwadrat
    """

    mcfig = mcstuff.MinecraftDrawing(mc)
    obrys = block.OBSIDIAN
    wypelniacz = block.AIR

    mc.setBlocks(x - r - 10, y - r, z - r - 10, x +
                 r + 10, y + r, z + r + 10, wypelniacz)
    mcfig.drawLine(x + r, y + r, z + r, x - r, y + r, z + r, obrys)
    mcfig.drawLine(x - r, y + r, z + r, x - r, y + r, z - r, obrys)
    mcfig.drawLine(x - r, y + r, z - r, x + r, y + r, z - r, obrys)
    mcfig.drawLine(x + r, y + r, z - r, x + r, y + r, z + r, obrys)

    mcfig.drawLine(x + r, y - r, z + r, x - r, y - r, z + r, obrys)
    mcfig.drawLine(x - r, y - r, z + r, x - r, y - r, z - r, obrys)
    mcfig.drawLine(x - r, y - r, z - r, x + r, y - r, z - r, obrys)
    mcfig.drawLine(x + r, y - r, z - r, x + r, y - r, z + r, obrys)

    mcfig.drawLine(x + r, y + r, z + r, x + r, y - r, z + r, obrys)
    mcfig.drawLine(x - r, y + r, z + r, x - r, y - r, z + r, obrys)
    mcfig.drawLine(x - r, y + r, z - r, x - r, y - r, z - r, obrys)
    mcfig.drawLine(x + r, y + r, z - r, x + r, y - r, z - r, obrys)

    mc.player.setPos(x + r, y + r + 1, z + r)

    if klatka:
        mc.setBlocks(x - r, y - r, z - r, x + r, y + r, z + r, block.GLASS)
        mc.setBlocks(x - r + 1, y - r + 1, z - r + 1, x +
                     r - 1, y + r - 1, z + r - 1, wypelniacz)
        mc.player.setPos(0, 0, 0)

    for i in range(-r, r + 1, 5):
        mcfig.drawHorizontalCircle(0, i, 0, r, block.GRASS)


def liczbaPi(klatka=False):
    r = int(raw_input("Podaj promień koła: "))
    model(r, 0, 0, 0, klatka)

    # pobieramy ilość punktów w kwadracie
    ileKw = int(raw_input("Podaj ilość losowanych punktów: "))
    ileKo = 0  # ilość punktów w kole
    wKwadrat = []  # pomocnicza lista punktów w kwadracie
```

```
    wKolo = []  # pomocnicza lista punktów w kole

    for i in range(ileKw):
        blok = block.OBSIDIAN
        x = round(random.uniform(-r, r))
        y = round(random.uniform(-r, r))
        z = round(random.uniform(-r, r))
        wKwadrat.append((x, y, z))
        print x, y, z
        if abs(x)**2 + abs(z)**2 <= r**2:
            blok = block.DIAMOND_BLOCK
            ileKo += 1
            wKolo.append((x, y, z))

        mc.setBlock(x, y, z, blok)

    mc.postToChat("W kole = " + str(ileKo) + " W Kwadracie = " + str(ileKw))
    pi = 4 * ileKo / float(ileKw)
    mc.postToChat("Pi w przyblizeniu: {:.10f}".format(pi))
    mc.postToChat("Stan na kamieniu!")

    while True:
        poz = mc.player.getPos()
        x, y, z = poz
        if mc.getBlock(x, y - 1, z) == block.STONE.id:
            for pkt in wKolo:
                x, y, z = pkt
                mc.setBlock(x, y, z, block.SAND)
            sleep(3)
            mc.player.setPos(0, r - 1, 0)
            break


def main():
    mc.postToChat("LiczbaPi")  # wysłanie komunikatu do mc
    plac(-50, 0, -50, 100)
    liczbaPi(False)
    return 0
```

Zadaniem funkcji `model()` jest stworzenie przestrzeni dla obrysu sześcianu i jego szkieletu. Opcjonalnie, jeżeli przekażemy do funkcji parametr `klatka` równy `True`, ściany mogą zostać wypełnione szkłem. Walec wizualizujemy w pętli `for` rysując kilka okręgów blokami trawy.

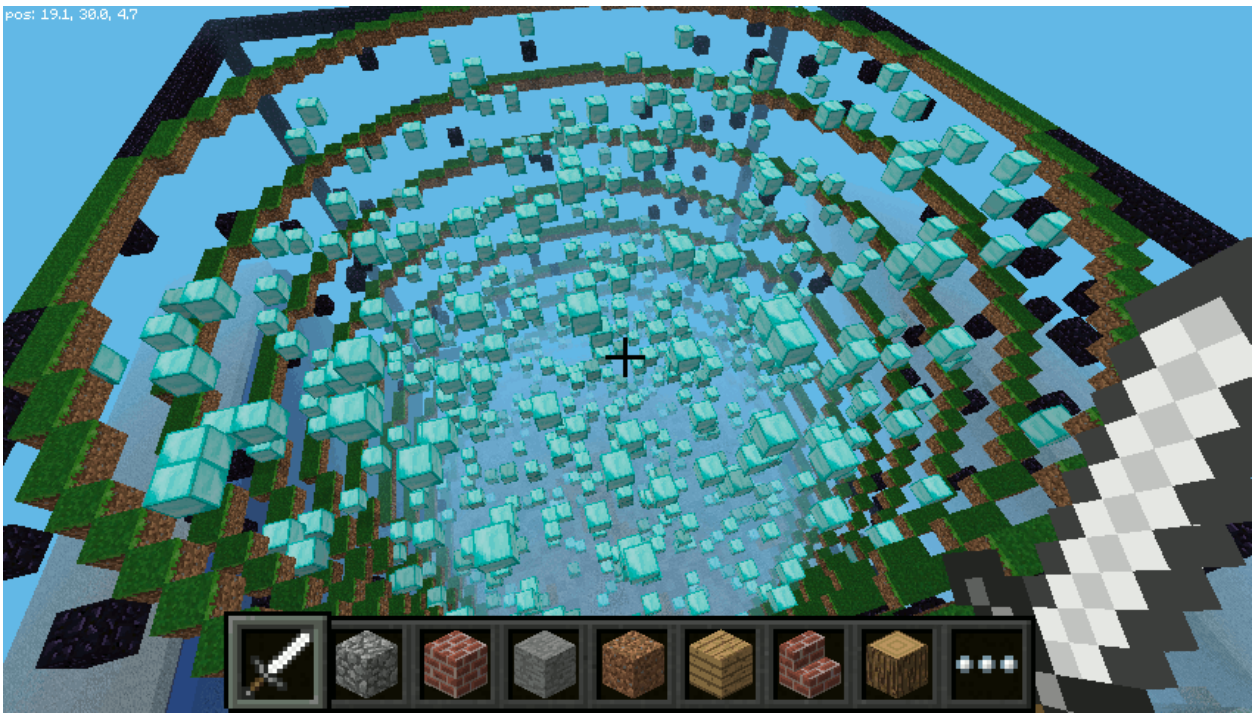
W funkcji `liczbaPi()` najważniejszą zmianą jest dodanie trzeciej zmiennej. Wartości wszystkich trzech współrzędnych losowane są w takim samym zakresie, ponieważ za środek całego układu przyjmujemy początek układu współrzędnych. Ważna zmiana zachodzi w funkcji warunkowej: `if abs(x)**2 + abs(z)**2 <= r**2:`. Do sprawdzenia, czy punkt należy do koła wykorzystujemy zmienne x i z, uwzględniając fakt, że w MC Pi wyznaczają one położenie w poziomie.

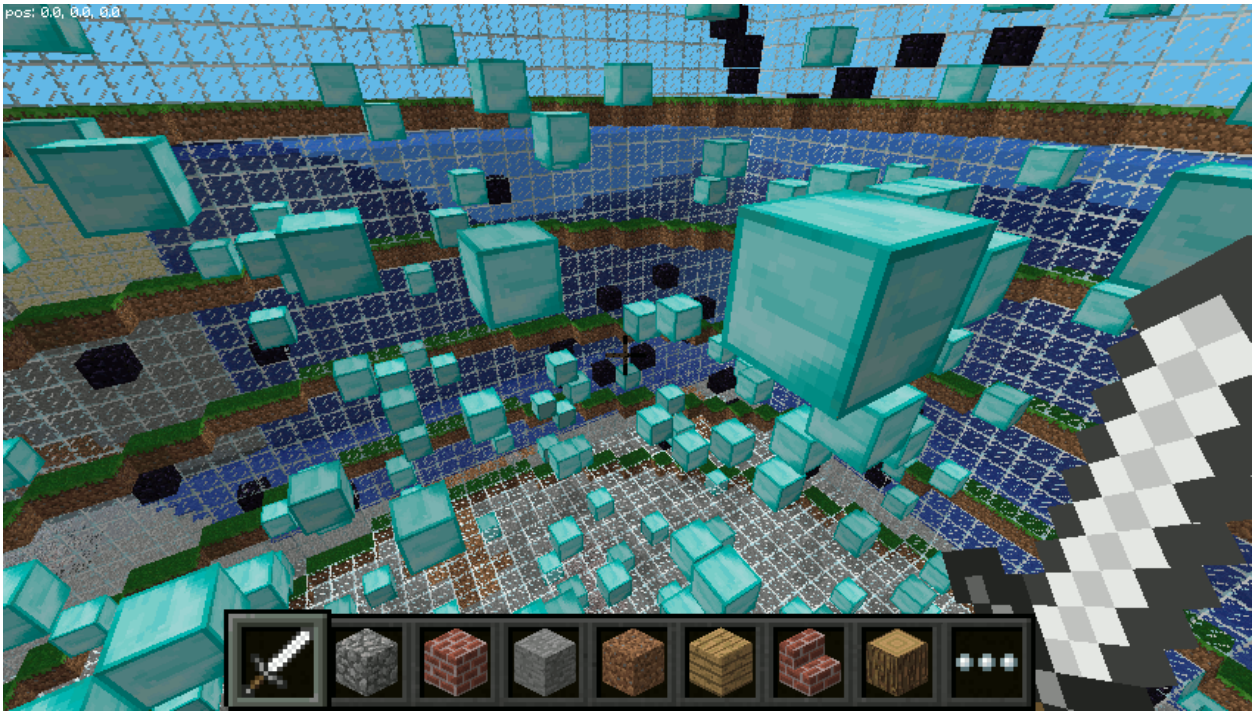
Bloki należące do sześcianu rysujemy za pomocą obsydianu, te w walcu – za pomocą diamentów.

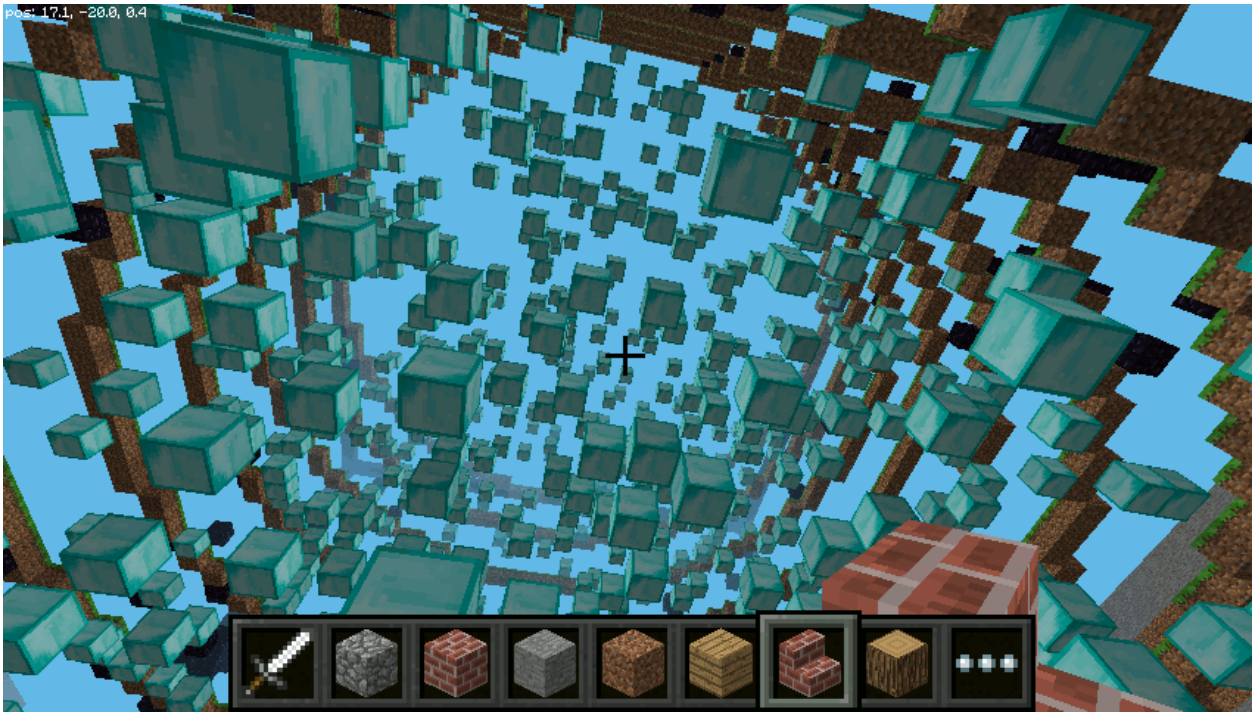
Na końcu funkcji dodajemy nieskończoną pętlę (`while True:`), której zadaniem jest sprawdzanie, na jakim bloku znajduje się gracz: `if mc.getBlock(x, y - 1, z) == block.STONE.id:`. Jeżeli stanie on na kamieniu, wszystkie bloki należące do walca zamieniamy w pętli `for pkt in wKolo:` w piasek, a gracza teleportujemy do środka sześcianu.

Dla promienia o wielkości 20 i 1000 bloków uzyskać można poniższe budowle:

Pozostaje eksperymentować z rozmiarami, typami bloków czy parametrem `klatka` określanym w wywołaniu funkcji


pos: 19.1, 30.0, 4.7


pos: 0.0, 0.0, 0.0



`liczbaPi()` w funkcji głównej.

Źródła:

- `Skrypty mcpi-algorytmy`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Gra w życie

Gra w życie jest najbardziej znaną implementacją automatu komórkowego, wymyśloną przez brytyjskiego matematyka Johna Conwaya. Cały pomysł polega na symulowaniu rozwoju populacji komórek, które umieszczone w wyznaczonym obszarze tworzą różne zaskakujące układy.

Grę zaimplementujemy przy użyciu programowania obiektowego, którego podstawowym elementem są klasy. Można je rozumieć jako definicje obiektów odwzorowujących mniej lub bardziej dokładniej jakieś elementy rzeczywistości, niekoniecznie materialne. Obiekty łączą dane, czy też właściwości, oraz metody na nich operujące. Obiekt tworzymy na podstawie klas i nazywamy je wtedy instancjami danej klasy.

Plansza gry

Zaczniemy od przygotowania obszaru, w którym będziemy obserwować kolejne populacje komórek. Tworzymy pusty plik w katalogu `mcpi-sim` i zapisujemy pod nazwą `mcpi-glife.py`. Wstawiamy do niego poniższy kod:

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3
4 # import sys
5 import os
```

```
from random import randint
from time import sleep
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

mc = minecraft.Minecraft.create("192.168.1.10")  # połączenie z MCPi


class GraWZycie(object):
    """
    Łączy wszystkie elementy gry w całość.
    """

    def __init__(self, mc, szer, wys, ile=40):
        """
        Przygotowanie ustawień gry
        :param szer: szerokość planszy mierzona liczbą komórek
        :param wys: wysokość planszy mierzona liczbą komórek
        """
        self.mc = mc
        mc.postToChat('Gra o zycie')
        self.szer = szer
        self.wys = wys

    def uruchom(self):
        """
        Główna pętla gry
        """
        self.plac(0, 0, 0, self.szer, self.wys)  # narysuj pole gry

    def plac(self, x, y, z, szer=20, wys=10):
        """
        Funkcja tworzy plac gry
        """
        podloga = block.STONE
        wypelniacz = block.AIR
        granica = block.OBSIDIAN

        # granica, podłoże, czyszczenie
        self.mc.setBlocks(
            x - 5, y, z - 5,
            x + szer + 5, y + max(szer, wys), z + wys + 5, wypelniacz)
        self.mc.setBlocks(
            x - 1, y - 1, z - 1, x + szer + 1, y - 1, z + wys + 1, granica)
        self.mc.setBlocks(x, y - 1, z, x + szer, y - 1, z + wys, podloga)
        self.mc.setBlocks(
            x, y, z, x + szer, y + max(szer, wys), z + wys, wypelniacz)


if __name__ == "__main__":
    gra = GraWZycie(mc, 20, 10, 40)  # instancja klasy GraWZycie
    mc.player.setPos(10, 20, -5)
    gra.uruchom()  # wywołanie metody uruchom()
```

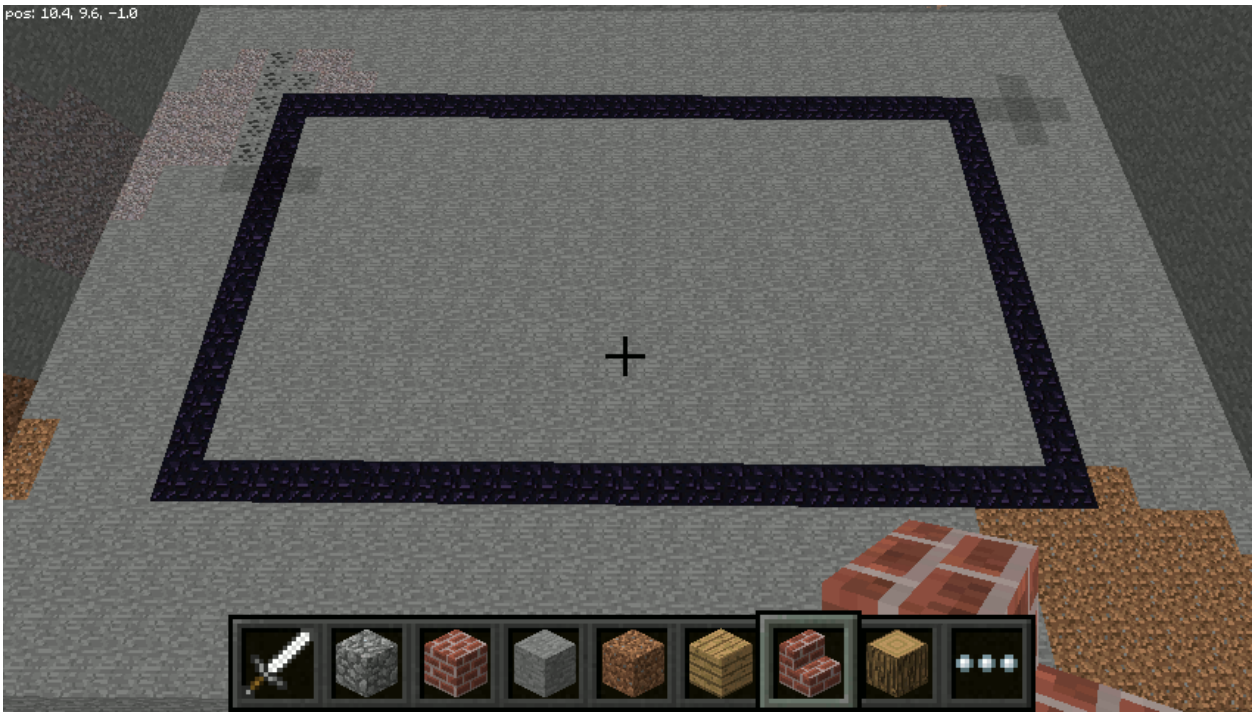
Główna klasa w programie nazywa się `GraWZycie`, jej definicja rozpoczyna się słowem kluczowym `class`, a

nazwa obowiązkową dużą literą. Pierwsza zdefiniowana metoda o nazwie `__init__()` to konstruktor klasy, wywoływany w momencie tworzenia jej instancji. Dzieje się tak w głównej funkcji `main()` w instrukcji: `gra = GraWZycie(mc, 20, 10, 40)`. Tworząc instancję klasy, czyli obiekt `gra`, przekazujemy do konstruktora parametry: obiekt `mc` reprezentujący grę Minecraft, szerokość i wysokość pola gry, a także ilość tworzonych na wstępie komórek.

Konstruktor z przekazanych parametrów tworzy właściwości klasy w instrukcjach typu `self.mc = mc`. Do właściwości klasy odwołujemy się w innych metodach za pomocą słowa `self` – np. w wywołanej w funkcji głównej metodzie `uruchom()`. Jej zadaniem jest wykonanie metody `plac()`, która buduje planszę gry. Przekazujemy jej współrzędne punktu początkowego, a także szerokość i wysokość planszy.

Informacja: Warto zauważyć i zapamiętać, że każda metoda w klasie jako pierwszy parametr przyjmuje zawsze wskaźnik do instancji obiektu, na którym będzie działać, czyli konwencjonalne słowo `self`.

W wyniku uruchomienia i przetestowania kodu powinniśmy zobaczyć zbudowaną planszę do gry, czyli prostokąt, o podanych w funkcji głównej wymiarach.



Populacja

Utworzymy klasę `Populacja`, a w niej strukturę danych reprezentującą układ żywych i martwych komórek. Przed funkcją główną `main()` wstawiamy kod:

```
61 # magiczne liczby używane do określenia czy komórka jest żywa
62 DEAD = 0
63 ALIVE = 1
64 BLOK_ALIVE = 35  # block.WOOL
65
66
67 class Populacja(object):
```

```
68      """
69      Populacja komórek
70      """
71
72      def __init__(self, mc, ilex, iley):
73          """
74          Przygotowuje ustawienia populacji
75          :param mc: obiekt Minecrafta
76          :param ilex: rozmiar x macierzy komórek (wiersze)
77          :param iley: rozmiar y macierzy komórek (kolumny)
78          """
79          self.mc = mc
80          self.iley = iley
81          self.ilex = ilex
82          self.generacja = self.reset_generacja()
83
84      def reset_generacja(self):
85          """
86          Tworzy i zwraca macierz pustej populacji
87          """
88          # wyrażenie listowe tworzy x kolumn o y komórkach
89          # wypełnionych wartością 0 (DEAD)
90          return [[DEAD for y in xrange(self.iley)] for x in xrange(self.ilex)]
91
92      def losuj(self, ile=50):
93          """
94          Losowo wypełnia macierz żywymi komórkami, czyli wartością 1 (ALIVE)
95          """
96          for i in range(ile):
97              x = randint(0, self.ilex - 1)
98              y = randint(0, self.iley - 1)
99              self.generacja[x][y] = ALIVE
100         print self.generacja
```

Konstruktor klasy `Populacja` pobiera obiekt Minecrafta (`mc`) oraz rozmiary dwuwymiarowej macierzy (`ilex`, `iley`), czyli tablicy, która reprezentować będzie układy komórek. Po przypisaniu właściwościom klasy przekazanych parametrów tworzymy początkowy stan populacji, tj. macierz wypełnioną zerami. W metodzie `reset_generacja()` wykorzystujemy wyrażenie listowe, które – ujmując rzecz w terminologii Pythona – zwraca listę *ilex* list zawierających *iley* komórek z wartościami zero. To właśnie wspomniana wcześniej macierz dwuwymiarowa.

Ćwiczenie 1

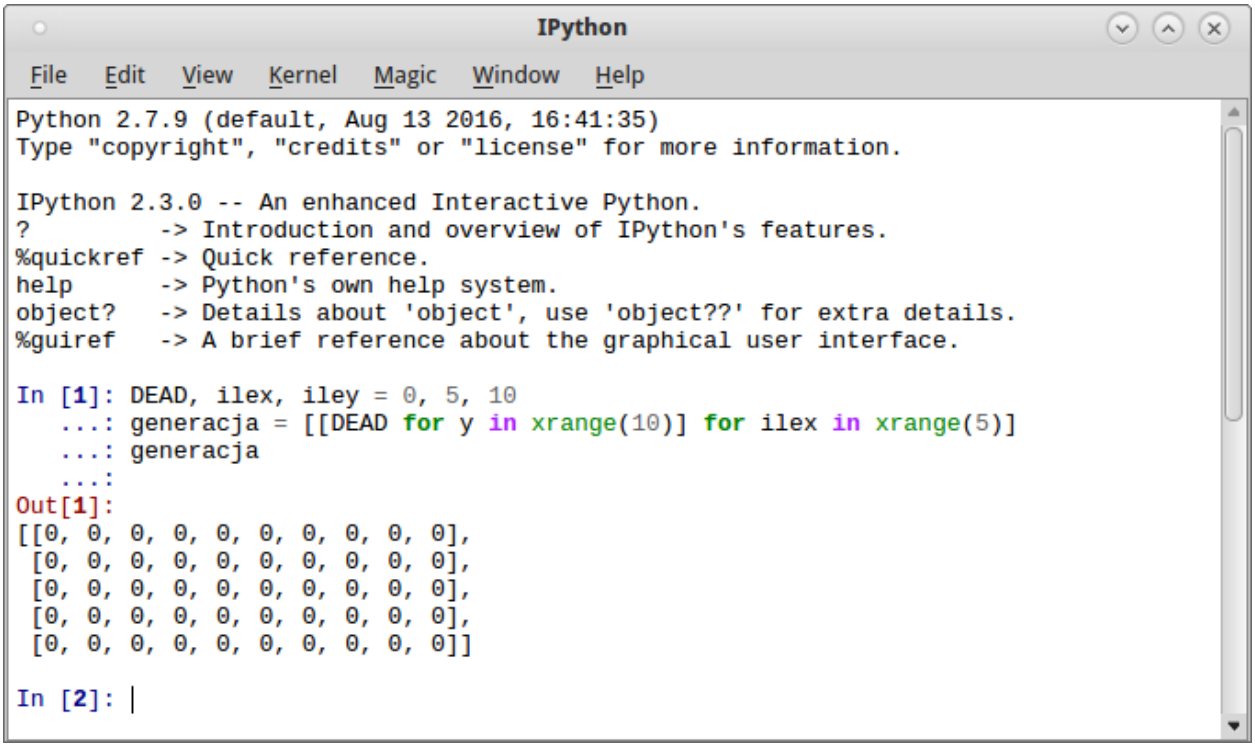
Uruchom konsolę IPython Qt Console i wklej do niej polecenia:

```
DEAD, ilex, iley = 0, 5, 10
generacja = [[DEAD for y in xrange(10)] for ilex in xrange(5)]
generacja
```

Zobacz efekt (nie zamykaj konsoli, jeszcze się przyda):

Komórki mogą być martwe (`DEAD`– wartość 0) i tak jest na początku, ale aby populacja mogła ewoluować, trzeba niektóre z nich ożywić (`ALIVE` – wartość 1). Odpowiada za to metoda `losuj()`, która przyjmuje jeden argument określający, ile komórek ma być początkowo żywych. Następnie w pętli losowana jest wymagana ilość par indeksów wskazujących wiersz i kolumnę, czyli komórkę, która ma być żywa (`ALIVE`). Na końcu drukujemy w terminalu początkowy układ komórek.

Ćwiczenie 2



Spróbuj w kilku komórkach macierzy utworzonej w konsoli, zapisać wartość ALIVE, czyli 1.

W konstruktorze klasy głównej `GraWZycie` tworzymy instancję klasy `Populacja` – to powoduje wykonanie jej konstruktora. Potem wywołujemy metodę tworzącą układ początkowy. Tak więc na końcu konstruktora klasy `GraWZycie` (`__init__()`)dodajemy poniższy kod:

```
32          self.populacja = Populacja(mc, szer, wys)  # instancja klasy Populacja
33          if ile:
34              self.populacja.losuj(ile)
```

Przetestuj kod.

Rysowanie macierzy

Skoro mamy przygotowany plac gry oraz początkowy układ populacji, trzeba ją narysować, czyli umieścić określone bloki we współrzędnych Minecrafta odpowiadających indeksom ożywionych komórek macierzy. Na końcu klasy `Populacja` dodajemy dwie nowe metody `rysyj()` i `zywe_komorki()`:

```
103     def rysuj(self):
104         """
105         Rysuje komórki na planszy, czyli umieszcza odpowiednie bloki
106         """
107         print "Rysowanie macierzy..."
108         for x, z in self.zywe_komorki():
109             podtyp = randint(0, 15)
110             mc.setBlock(x, 0, z, BLOK_ALIVE, podtyp)
111
112     def zywe_komorki(self):
113         """
114         Generator zwracający współrzędne żywych komórek.
```

```
        """
        for x in range(len(self.generacja)):
            kolumna = self.generacja[x]
            for y in range(len(kolumna)):
                if kolumna[y] == ALIVE:
                    yield x, y  # zwracamy współrzędne, jeśli komórka jest żywa
```

– a rysowanie wywołujemy w metodzie `uruchom()` klasy `GraWZycie`, dopisując:

```
        self.populacja.rysuj()
```

Wyjaśnienia wymaga funkcja `rysuj()`. W pętli pobieramy współrzędne żywych komórek, które rozpakowywane są z 2-elementowej listy do zmiennych: `for x, z in self.zywe_komorki():`. Dalej losujemy podtyp bloku bawełny i umieszczamy go we wskazanym miejscu.

Funkcja `zywe_komorki()` to tzw. *generator*, co poznajemy po tym, że zwraca wartości za pomocą słowa kluczowego `yield`. Jej działanie polega na przeglądaniu macierzy za pomocą zagnieżdżonych pętli i zwracaniu współrzędnych "żywych"komórek.

Ćwiczenie 3

Odwołując się do utworzonej wcześniej przykładowej macierzy, przetestuj w konsoli poniższy kod:

```
for x in range(len(generacja)):
    kolumna = generacja[x]
    for y in range(len(kolumna)):
        print x, y, " = ", generacja[x][y]
```

Różnica pomiędzy generatorem a zwykłą funkcją polega na tym, że zwykła funkcja po przeglądnięciu całej macierzy zwróciłaby od razu kompletną listę żywych komórek, a generator robi to "na żądanie". Po napotkaniu żywej komórki zwraca jej współrzędne, zapamiętuje stan lokalnych pętli i czeka na następne wywołanie. Dzięki temu oszczędzamy pamięć, a dla dużych struktur także zwiększamy wydajność.

Uruchom kod, oprócz pola gry, powinieneś zobaczyć bloki reprezentujące pierwszą generację komórek.

Ewolucja – zasady gry

Jak można było zauważyć, rozgrywka toczy się na placu podzielonym na kwadratowe komórki, którego reprezentacją algorytmiczną jest macierz. Każda komórka ma maksymalnie ośmiu sąsiadów. To czy komórka przetrwa, zależy od ich ilości. Reguły są następujące:

- Martwa komórka, która ma dokładnie 3 sąsiadów, staje się żywa w następnej generacji.
- Żywa komórka z 2 lub 3 sąsiadami zachowuje swój stan, w innym przypadku umiera z powodu "samotności" lub "zatłoczenia".

Kolejne generacje obliczamy w umownych jednostkach czasu. Do kodu klasy `Populacja` dodajemy dwie metody zawierające logikę gry:

```
    def sasiedzi(self, x, y):
        """
        Generator zwracający wszystkich okolicznych sąsiadów
        """
        for nx in range(x - 1, x + 2):
            for ny in range(y - 1, y + 2):
                if nx == x and ny == y:
                    continue  # pomiń współrzędne centrum
                if nx >= self.ilex:
```


```
137                     # sąsiad poza końcem planszy, bierzemy pierwszego w danym
138                     # rzędzie
139                     nx = 0
140                 elif nx < 0:
141                     # sąsiad przed początkiem planszy, bierzemy ostatniego w
142                     # danym rzędzie
143                     nx = self.ilex - 1
144                 if ny >= self.iley:
145                     # sąsiad poza końcem planszy, bierzemy pierwszego w danej
146                     # kolumnie
147                     ny = 0
148                 elif ny < 0:
149                     # sąsiad przed początkiem planszy, bierzemy ostatniego w
150                     # danej kolumnie
151                     ny = self.iley - 1
152
153                 # zwróć stan komórki w podanych współrzędnych
154                 yield self.generacja[nx][ny]
155
156     def nast_generacja(self):
157         """
158         Generuje następną generację populacji komórek
159         """
160         print "Obliczanie generacji..."
161         nast_gen = self.reset_generacja()
162         for x in range(len(self.generacja)):
163             kolumna = self.generacja[x]
164             for y in range(len(kolumna)):
165                 # pobieramy wartości sąsiadów
166                 # dla żywej komórki dostaniemy wartość 1 (ALIVE)
167                 # dla martwej otrzymamy wartość 0 (DEAD)
168                 # zwykła suma pozwala nam określić liczbę żywych sąsiadów
```

```
169                 iluS = sum(self.sasiedzi(x, y))
170                 if iluS == 3:
171                     # rozmnażamy się
172                     nast_gen[x][y] = ALIVE
173                 elif iluS == 2:
174                     # przechodzi do kolejnej generacji bez zmian
175                     nast_gen[x][y] = kolumna[y]
176                 else:
177                     # za dużo lub za mało sąsiadów by przeżyć
178                     nast_gen[x][y] = DEAD
179 
180         # nowa generacja staje się aktualną generacją
181         self.generacja = nast_gen
```

Metoda `nast_generacja()` wylicza kolejny stan populacji. Na początku tworzymy pustą macierz `naste_gen` wypełnioną zerami – tak jak w konstruktorze klasy. Następnie przy użyciu dwóch zagnieżdżonych pętli `for` – takich samych jak w generatorze `zywe_komorki()` – przeglądamy wiersze, wydobywając z nich kolejne komórki i badamy ich otoczenie.

Najważniejszy krok algorytmu to określenie ilości żywych sąsiednich komórek, co ma miejsce w instrukcji: `iluS = sum(self.sasiedzi(x, y))`. Funkcja `sum()` sumuje zapisane w sąsiednich komórkach wartości, zwracane przez generator `sasiedzi()`. Generator ten wykorzystuje zagnieżdżone pętle `for`, aby uzyskać współrzędne sąsiednich komórek, następnie w instrukcjach warunkowych `if` sprawdza, czy nie wychodzą one poza planszę.

Uwaga: "Gra w życie" zakłada, że symulacja toczy się na nieograniczonej planszy, jednak dla celów wizualizacji w MC Pi musimy przyjąć jakieś jej wymiary, a także podjąć decyzję, co ma się dziać, kiedy je przekraczamy. W naszej implementacji, kiedy badając stan sąsiada przekraczamy planszę, bierzemy pod uwagę stan komórki z przeciwległego końca wiersza lub kolumny.

Ćwiczenie 4

Na przykładzie utworzonej wcześniej macierzy przetestuj w konsoli kod:

```
x, y = 2, 2
for nx in range(x - 1, x + 2):
    for ny in range(y - 1, y + 2):
        print nx, ny, "=", generacja[nx][ny]
```

Jak widzisz, zwraca on wartości zapisane w komórkach otaczających wyznaczoną współrzędnymi *x, y*.

Wróćmy do metody `nast_generacja()`. Po wywołaniu `iluS = sum(self.sasiedzi(x, y))`, wiemy już, ilu mamy wokół siebie sąsiadów. Dalej za pomocą instrukcji warunkowych, np. `if iluS == 3:`, sprawdzamy więc ich ilość i – zgodnie z regułami – ożywiamy badaną komórkę, zachowujemy jej stan lub ją uśmiercamy. Uzyskany stan zapisujemy w nowej macierzy `nast_gen`. Po zbadaniu wszystkich komórek nowa macierz reprezentująca nową generację nadpisuje poprzednią: `self.generacja = nast_gen`. Pozostaje ją narysować. Zmieniamy metodę `uruchom()` klasy `GraWZycie`:

```
36     def uruchom(self):
37         """
38         Główna pętla gry
39         """
40         i = 0
41         while True:  # działaj w pętli do momentu otrzymania sygnału do wyjścia
42             print("Generacja: " + str(i))
43             self.plac(0, 0, 0, self.szer, self.wys)  # narysuj pole gry
44             self.populacja.rysuj()
```

```
45              self.populacja.nast_generacja()
46              i += 1
47              sleep(1)
```

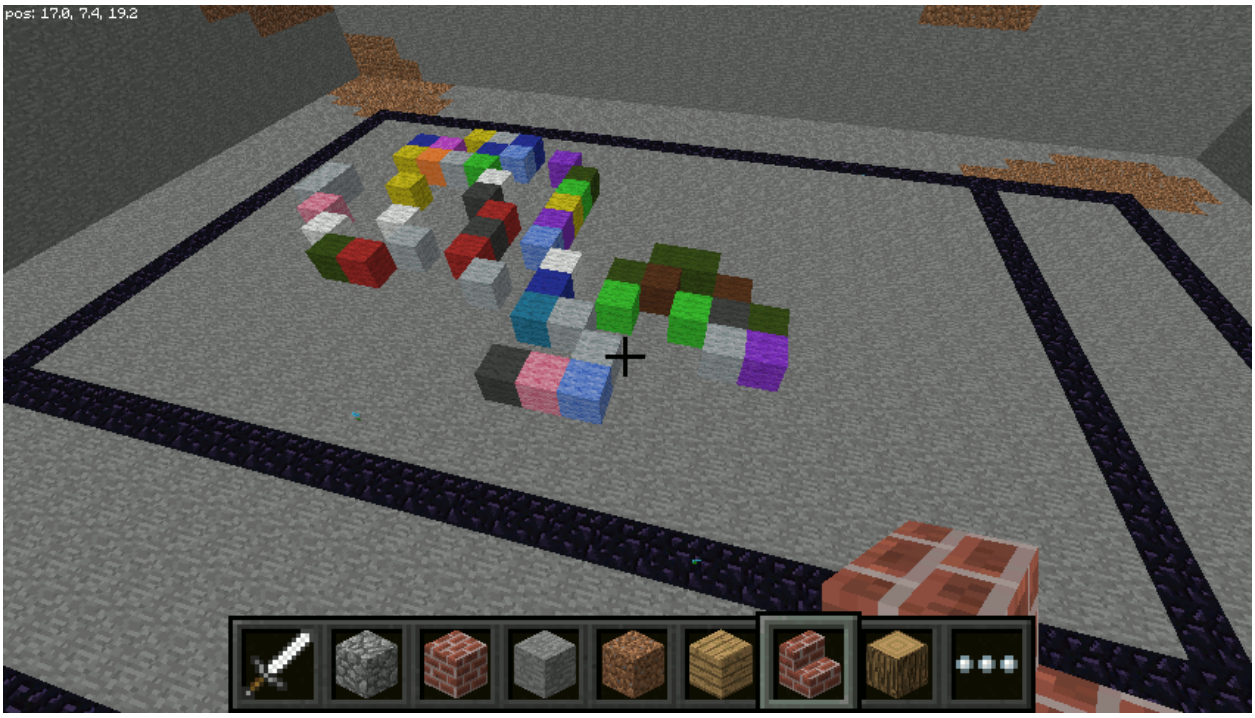
Proces generowania i rysowania kolejnych generacji komórek dokonuje się w zmienionej metodzie `uruchom()` głównej klasy naszego skryptu. Wykorzystujemy nieskończoną pętlę `while True:`, w której:

- rysujemy plac gry,
- rysujemy aktualną populację,
- wyliczamy następną generację,
- wstrzymujemy działanie na sekundę
- i wszystko powtarzamy.

Tak uruchomiony program możemy przerwać tylko "ręcznie" przerywając działanie skryptu.

Wskazówka: Uwaga: metoda zakończenia działania skryptu zależy od sposobu jego uruchomienia i systemu operacyjnego. Np. w Linuksie skrypt uruchomiony w terminalu poleceniem `python skrypt.py` przerwiemy naciskając `CTRL+C` lub bardziej radykalnie `ALT+F4` (zamknięcie okna z terminalem).

Przetestuj skrypt!



Początek zabawy

Śledzenie ewolucji losowo przygotowanego układu komórek nie jest zazwyczaj zbyt widowiskowe, zwłaszcza kiedy symulację przeprowadzamy na dużej planszy. O wiele ciekawsza jest możliwość śledzenia zmian samodzielnie zaprojektowanego układu początkowego. Dodajmy więc możliwość wczytywania takiego układu bezpośrednio z Minecrafta. Do klasy `Populacja` poniżej metody `losuj()` dodajemy kod:

```
111     def wczytaj(self):
112         """
113         Funkcja wczytuje populację komórek z MC RPi
114         """
115         ileKom = 0
116         print "Proszę czekać, aktuzalizacja macierzy..."
117         for x in range(self.ilex):
118             for z in range(self.iley):
119                 blok = self.mc.getBlock(x, 0, z)
120                 if blok != block.AIR:
121                     self.generacja[x][z] = ALIVE
122                     ileKom += 1
123         print self.generacja
124         print "Żywych:", str(ileKom)
125         sleep(3)
```

Działanie metody `wczytaj()` jest proste: za pomocą zagnieżdżonych pętli pobieramy typ bloku z każdego miejsca placu gry: `blok = self.mc.getBlock(x, 0, z)`. Jeżeli na placu znajduje się jakikolwiek blok inny niż powietrze, oznaczamy odpowiednią komórkę początkowej generacji, wskazywaną przez współrzędną bloku jako żywą: `self.generacja[x][z] = ALIVE`. Przy okazji zliczamy ilość takich komórek.

Wywołanie funkcji trzeba dopisać do konstruktora klasy `GraWZycie` w następujący sposób:

```
33         if ile:
34             self.populacja.losuj(ile)
35         else:
36             self.populacja.wczytaj()
```

Jak widać wykonanie metody `wczytaj()` zależne jest od wartości parametru `ile`. Tak więc jeżeli chcesz przetestować nową możliwość, w wywołaniu konstruktora w funkcji głównej ustaw ten parametr na 0 (zero), np: `gra = GraWZycie(mc, 30, 20, 0)`.

Informacja: Uwaga: przy dużych rozmiarach pola gry odczytywanie wszystkich bloków zajmuje dużo czasu! Przed testowaniem wczytywania własnych układów warto uruchomić skrypt przynajmniej raz, aby zbudować w MC Pi plac gry.

Nie pozostaje nic innego, jak zacząć się bawić. Można np. urządzić zawody: czyja populacja komórek utrzyma się dłużej – oczywiście warto wykluczyć budowanie znanych i udokumentowanych układów stałych.

Ćwiczenie 5

Dodaj do skryptu mechanizm kończący symulacji, kiedy na planszy nie ma już żadnych żywych komórek.

Źródła:

- `Skrypty mcsim-glife`

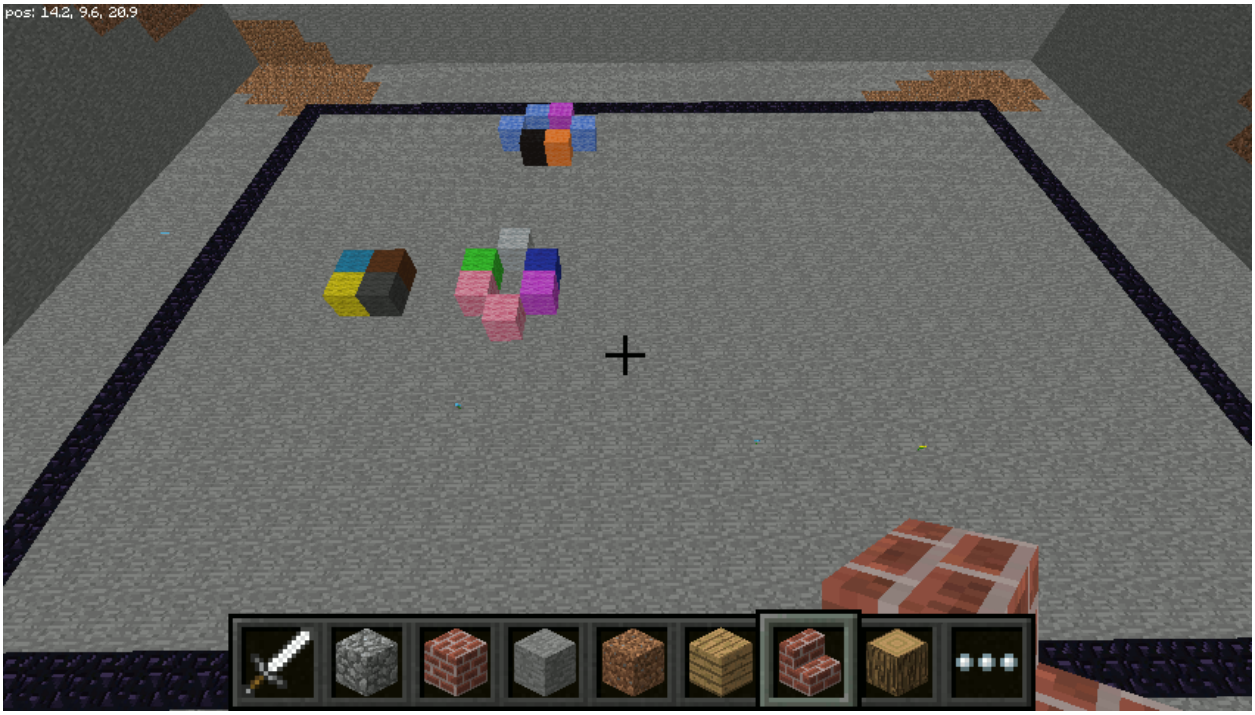
  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Gra robotów

Pole gry

Spróbujemy teraz pokazać rozgrywkę z *gry robotów*. Zaczniemy od zbudowania areny wykorzystywanej w grze. W pliku `mcpi-rg.py` umieszczamy następujący kod:


pos: 14.2, 9.6, 20.9


pos: 30.4, 8.4, 7.7

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import os
import json
from time import sleep
import mcpi.minecraft as minecraft  # import modułu minecraft
import mcpi.block as block  # import modułu block

os.environ["USERNAME"] = "Steve"  # nazwa użytkownika
os.environ["COMPUTERNAME"] = "mykomp"  # nazwa komputera

mc = minecraft.Minecraft.create("192.168.1.10")  # połączenie z MCPi


class GraRobotow(object):
    """Główna klasa gry"""

    obstacle = [(0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,0),(7,0),(8,0),(9,0),
        (10,0),(11,0),(12,0),(13,0),(14,0),(15,0),(16,0),(17,0),(18,0),(0,1),
        (1,1),(2,1),(3,1),(4,1),(5,1),(6,1),(12,1),(13,1),(14,1),(15,1),
        (16,1),(17,1),(18,1),(0,2),(1,2),(2,2),(3,2),(4,2),(14,2),(15,2),
        (16,2),(17,2),(18,2),(0,3),(1,3),(2,3),(16,3),(17,3),(18,3),(0,4),
        (1,4),(2,4),(16,4),(17,4),(18,4),(0,5),(1,5),(17,5),(18,5),(0,6),
        (1,6),(17,6),(18,6),(0,7),(18,7),(0,8),(18,8),(0,9),(18,9),(0,10),
        (18,10),(0,11),(18,11),(0,12),(1,12),(17,12),(18,12),(0,13),(1,13),
        (17,13),(18,13),(0,14),(1,14),(2,14),(16,14),(17,14),(18,14),(0,15),
        (1,15),(2,15),(16,15),(17,15),(18,15),(0,16),(1,16),(2,16),(3,16),
        (4,16),(14,16),(15,16),(16,16),(17,16),(18,16),(0,17),(1,17),(2,17),
        (3,17),(4,17),(5,17),(6,17),(12,17),(13,17),(14,17),(15,17),(16,17),
        (17,17),(18,17),(0,18),(1,18),(2,18),(3,18),(4,18),(5,18),(6,18),
        (7,18),(8,18),(9,18),(10,18),(11,18),(12,18),(13,18),(14,18),(15,18),
        (16,18),(17,18),(18,18)]

    plansza = []  # współrzędne dozwolonych pól gry

    def __init__(self, mc):
        """Konstruktor klasy"""
        self.mc = mc
        self.poleGry(0, 0, 0, 18)
        # self.mc.player.setPos(19, 20, 19)

    def poleGry(self, x, y, z, roz=10):
        """Funkcja tworzy pole gry"""

        podloga = block.STONE
        wypelniacz = block.AIR

        # podloga i czyszczenie
        self.mc.setBlocks(x, y - 1, z, x + roz, y - 1, z + roz, podloga)
        self.mc.setBlocks(x, y, z, x + roz, y + roz, z + roz, wypelniacz)
        # granice pola
        x = y = z = 0
        for i in range(19):
            for j in range(19):
                if (i, j) in self.obstacle:
                    self.mc.setBlock(x + i, y, z + j, block.GRASS)
                else:  # tworzenie listy współrzędnych dozwolonych pól gry
```

```
                    self.plansza.append((x + i, z + j))


def main(args):
    gra = GraRobotow(mc)  # instancja klasy GraRobotow
    print gra.plansza  # pokaż w konsoli listę współrzędnych pól gry
    return 0


if __name__ == '__main__':
    import sys
    sys.exit(main(sys.argv))
```

Zaczynamy od definicji klasy *GraRobotow*, której instancję tworzymy w funkcji głównej `main()` i przypisujemy do zmiennej: `gra = GraRobotow(mc)`. Konstruktor klasy wywołuje metodę `poleGry()`, która buduje pusty plac i arenę, na której walczą roboty.

Pole gry wpisane jest w kwadrat o boku 19 jednostek. Część pól kwadratu wyłączona jest z rozgrywki, ich współrzędne zawiera lista `obstacle`. Funkcja `poleGry()` wykorzystuje dwie zagnieżdżone pętle, w których zmienne iteracyjne *i*, *j* przyjmują wartości od 0 do 18, wyznaczając wszystkie pola kwadratu. Jeżeli dane pole zawarte jest w liście pól wyłączonych `if (i, j) in obstacle`, umieszczamy w nim blok trawy – wyznaczą one granice planszy. W przeciwnym wypadku dołączamy współrzędne pola w postaci tupli do listy pól dozwolonych: `self.plansza.append((x + i, z + j))`. Wykorzystamy tę listę później do "czyszczenia" pola gry.

Po uruchomieniu powinniśmy zobaczyć plac gry, a w konsoli listę pól, na których będą walczyć roboty.

Dane gry

Dane gry, czyli zapis 100 rund rozgrywki zawierający m. in. informacje o położeniu robotów oraz ich sile (punkty *hp*) musimy wygenerować uruchamiając walkę gotowych lub napisanych przez nas robotów.

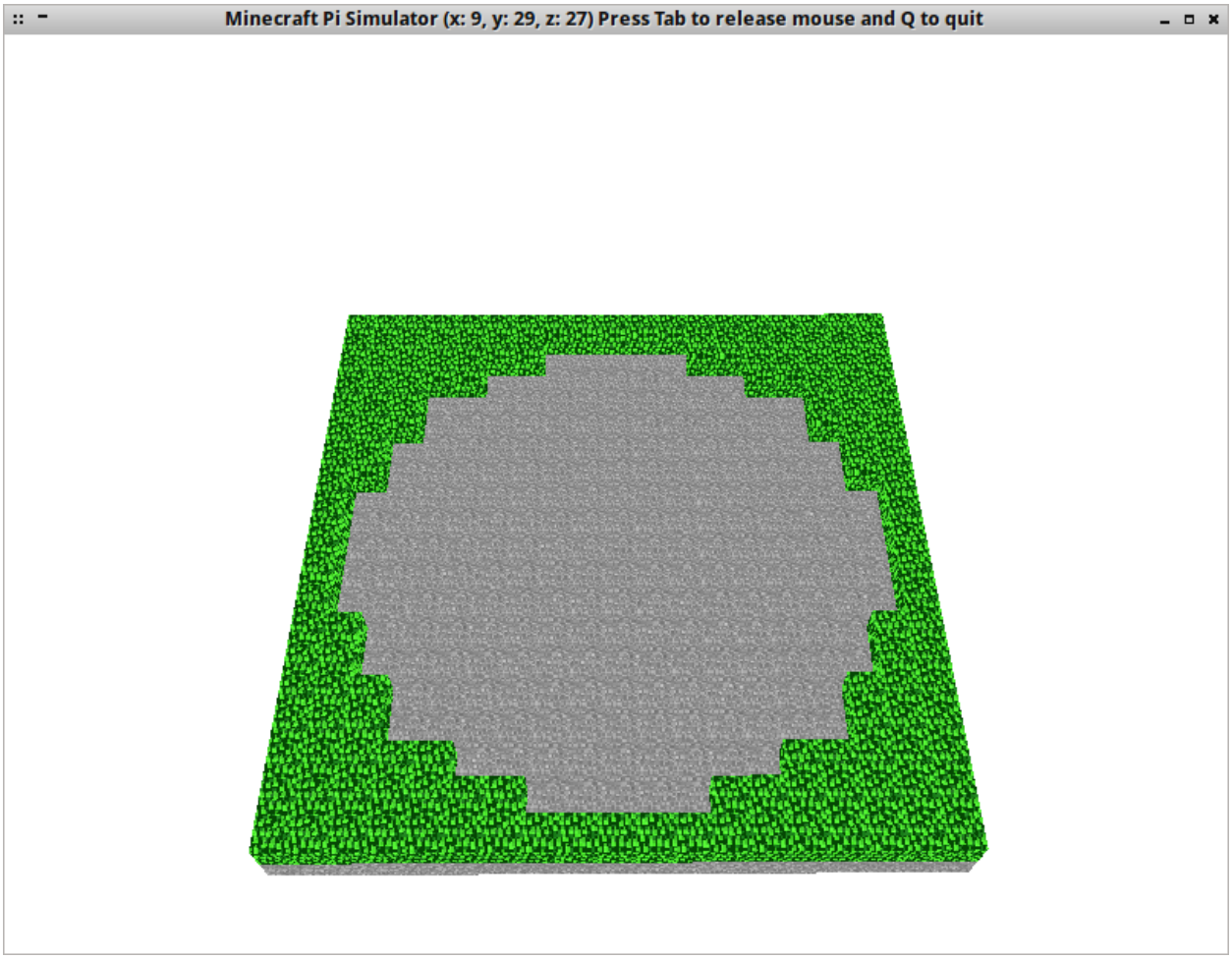
W tym celu trzeba zmodyfikować bibliotekę `game.py` z pakietu `rgkit`. Jeżeli korzystałeś z naszego *scenariusza* i zainstalowałeś `rgkit` w *wirtualnym środowisku* `~/robot/env`, plik ten znajdziesz w ścieżce `~/robot/env/lib/python2.7/site-packages/rgkit/game.py`. Na końcu funkcji `run_all_turns()` po linii nr 386 wstawiamy podany niżej kod:

```
# BEGIN DODANE na potrzeby Kzk
import json
plik = open('lastgame.log', 'w')
json.dump(self.history, plik)
plik.close()
# END OF DODANE
```

Następnie po wywołaniu przykładowej walki: `(env) root@kzk:~/robot$ rgrun bots/stupid26.py bots/Wall-E.py` w katalogu `~/robot` znajdziemy plik `lastgame.log`, który musimy umieścić w katalogu ze skryptem `mcpi-rg.py`.

Do definicji klasy `GraRobotow` w pliku `mcpi-rg.py` dodajemy metodę `uruchom()`:

```
    def uruchom(self, plik, ile=100):
        """Funkcja odczytuje z pliku i wizualizuje rundy gry robotów."""

        if not os.path.exists(plik):
            print "Podany plik nie istnieje!"
            return

        plik = open(plik, "r")  # otwórz plik w trybie tylko do odczytu
```


Minecraft Pi Simulator (x: 9, y: 29, z: 27) Press Tab to release mouse and Q to quit

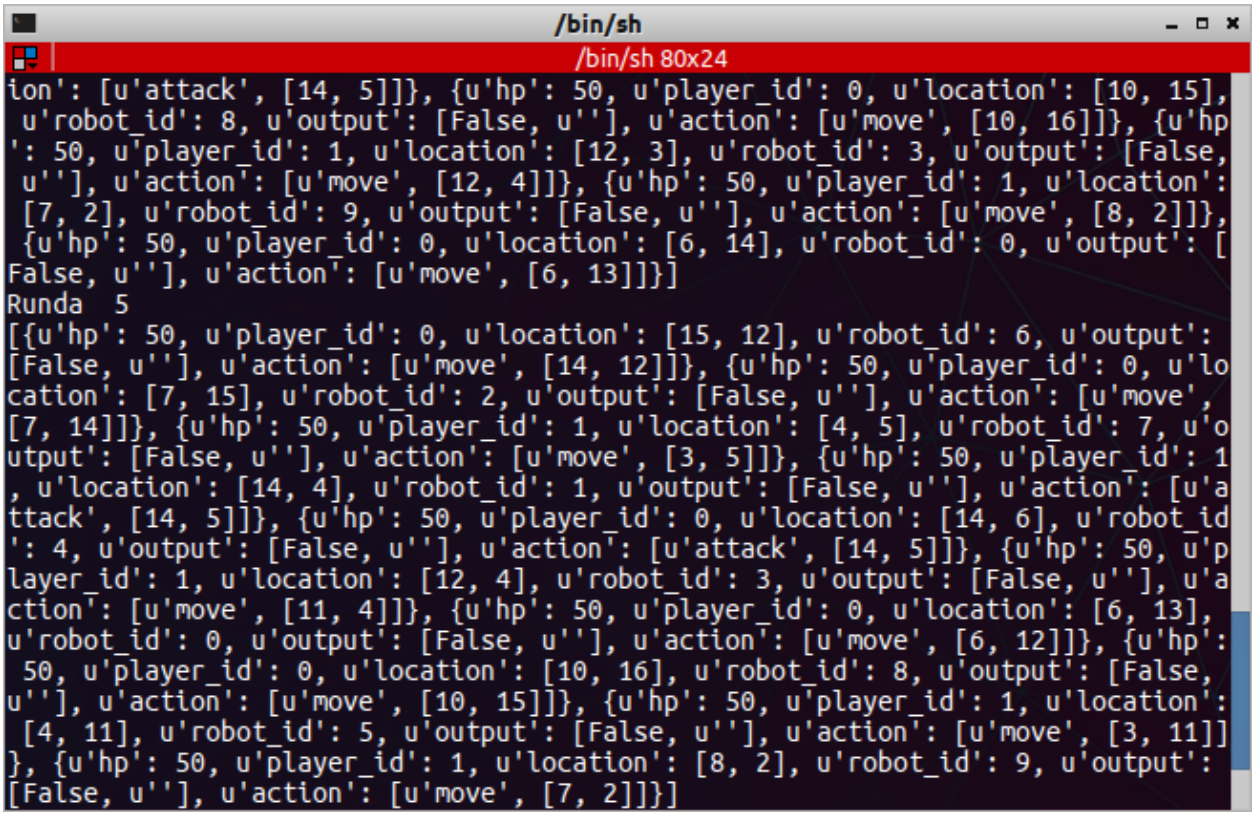
```
69          runda_nr = 0
70          for runda in json.load(plik):
71              print "Runda ", runda_nr
72              print runda  # pokaż dane rundy w konsoli
73              runda_nr = runda_nr + 1
74              if runda_nr > ile:
75                  break
76
77
78  def main(args):
79      gra = GraRobotow(mc)  # instancja klasy GraRobotow
80      gra.uruchom("lastgame.log", 10)
81      return 0
```

Omawianą metodę wywołujemy w funkcji głównej `main()` przekazując jej jako parametry nazwę pliku z zapisem rozgrywki oraz ilość rund do pokazania: `gra.uruchom(łastgame.log", 10)`.

W samej metodzie zaczynamy od sprawdzenia, czy podany plik istnieje w katalogu ze skryptem. Jeżeli nie istnieje (`if not os.path.exists(plik):`) drukujemy komunikat i wychodzimy z funkcji.

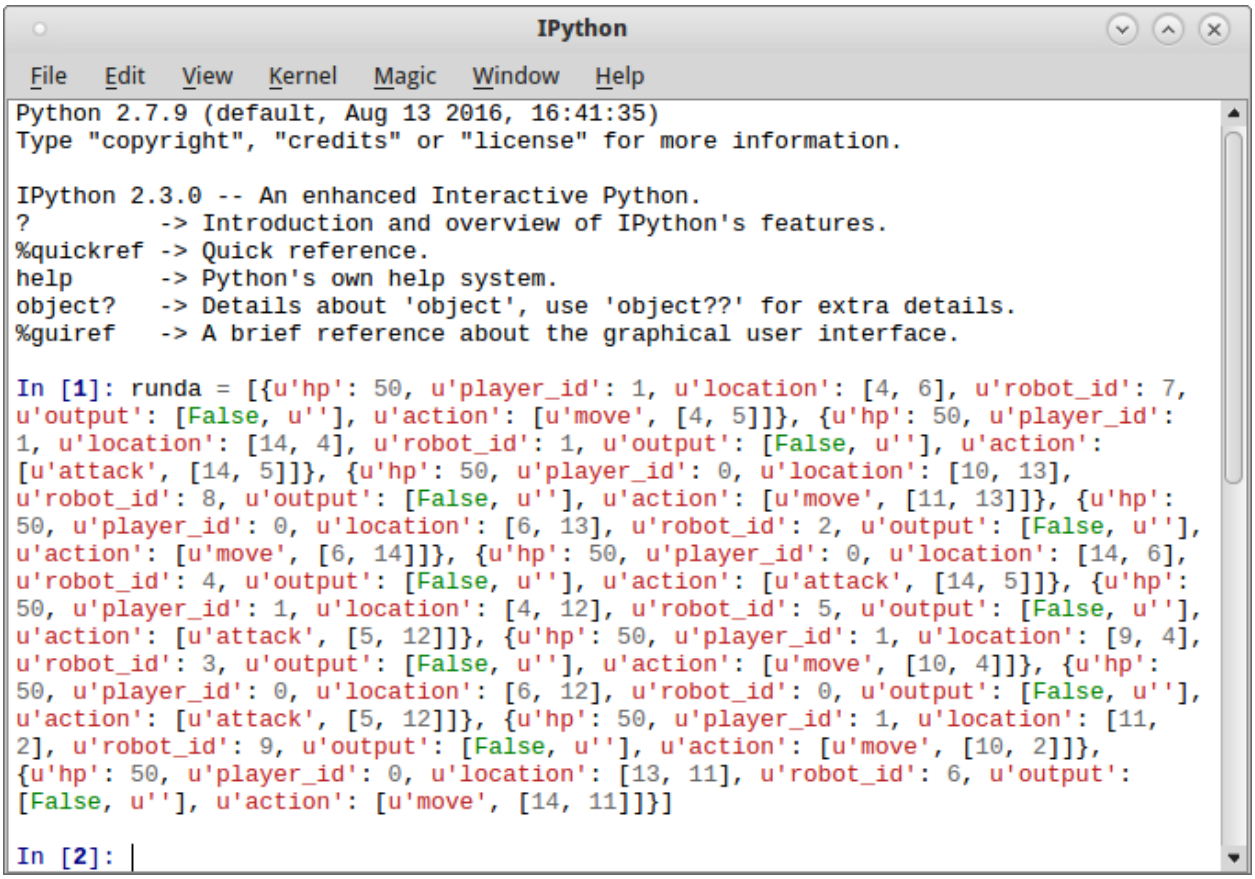
Jeżeli plik istnieje, otwieramy go w trybie tylko do odczytu. Dalej, ponieważ dane gry zapisane są w formacie *json*, w pętli `for runda in json.load(plik):` dekodujemy jego zawartość wykorzystując metodę `load()` modułu *json*. Instrukcja `print runda` pokaże nam w konsoli format danych kolejnych rund.

Po uruchomieniu kodu widzimy, że każda runda to lista zawierająca słowniki określające właściwości poszczególnych robotów.

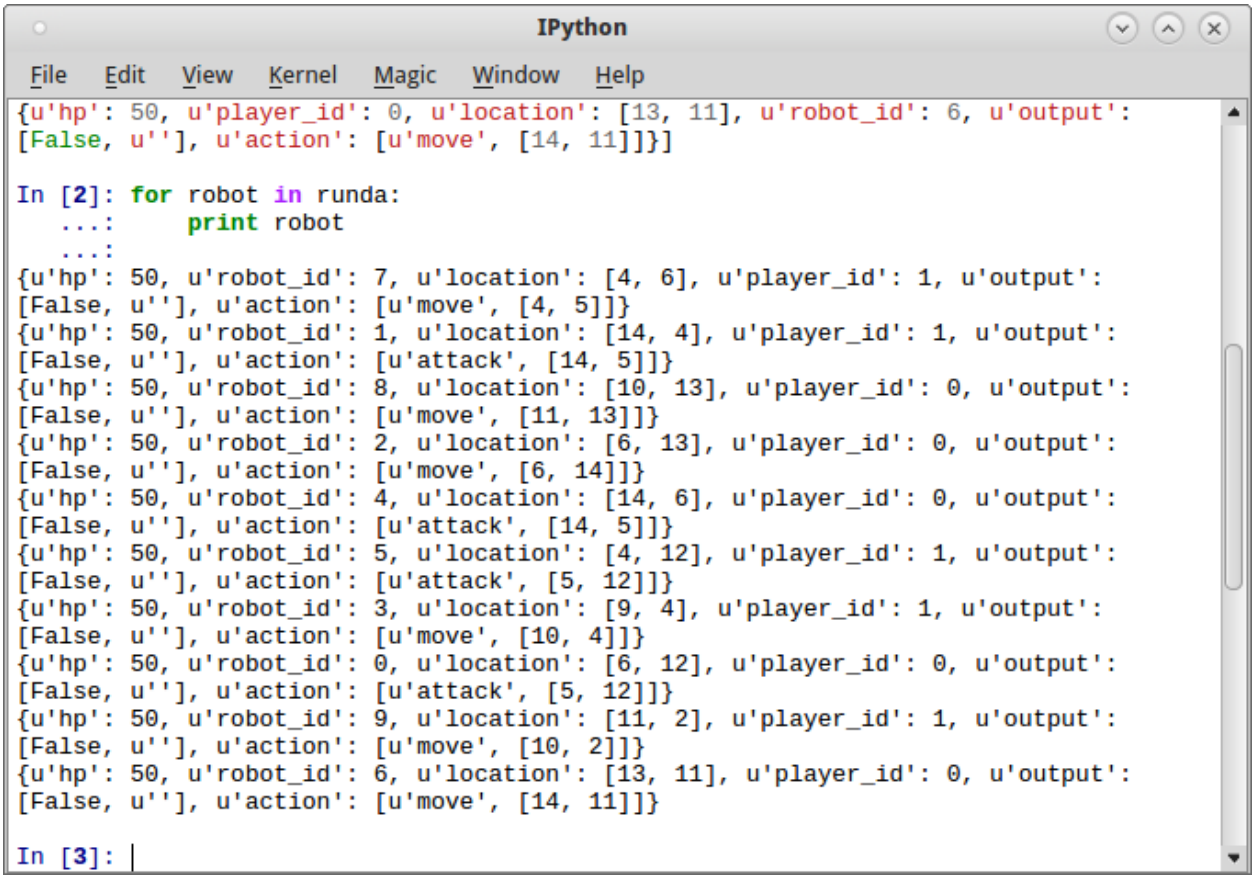


Ćwiczenie 1

Skopiuj z konsoli dane jednej z rund, uruchom konsolę IPython Qt i wklej do niej.


IPython
File Edit View Kernel Magic Window Help
Python 2.7.9 (default, Aug 13 2016, 16:41:35)
Type "copyright", "credits" or "license" for more information.
IPython 2.3.0 -- An enhanced Interactive Python.
? -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.
%guiref -> A brief reference about the graphical user interface.
In [1]: runda = [{u'hp': 50, u'player_id': 1, u'location': [4, 6], u'robot_id': 7, u'output': [False, u''], u'action': [u'move', [4, 5]]}, {u'hp': 50, u'player_id': 1, u'location': [14, 4], u'robot_id': 1, u'output': [False, u''], u'action': [u'attack', [14, 5]]}, {u'hp': 50, u'player_id': 0, u'location': [10, 13], u'robot_id': 8, u'output': [False, u''], u'action': [u'move', [11, 13]]}, {u'hp': 50, u'player_id': 0, u'location': [6, 13], u'robot_id': 2, u'output': [False, u''], u'action': [u'move', [6, 14]]}, {u'hp': 50, u'player_id': 0, u'location': [14, 6], u'robot_id': 4, u'output': [False, u''], u'action': [u'attack', [14, 5]]}, {u'hp': 50, u'player_id': 1, u'location': [4, 12], u'robot_id': 5, u'output': [False, u''], u'action': [u'attack', [5, 12]]}, {u'hp': 50, u'player_id': 1, u'location': [9, 4], u'robot_id': 3, u'output': [False, u''], u'action': [u'move', [10, 4]]}, {u'hp': 50, u'player_id': 0, u'location': [6, 12], u'robot_id': 0, u'output': [False, u''], u'action': [u'attack', [5, 12]]}, {u'hp': 50, u'player_id': 1, u'location': [11, 2], u'robot_id': 9, u'output': [False, u''], u'action': [u'move', [10, 2]]}, {u'hp': 50, u'player_id': 0, u'location': [13, 11], u'robot_id': 6, u'output': [False, u''], u'action': [u'move', [14, 11]]}]
In [2]:

Następnie przećwicz wydobywanie słowników z listy:



– oraz wydobywanie konkretnych danych ze słowników, a także rozpakowywanie tupli (`robot['location']`) określających położenie robota:

Pokaż rundę

Słowniki opisujące roboty walczące w danej rundzie zawierają m.in. identyfikatory gracza, położenie robota oraz jego ilość punktów *hp*. Wykorzystamy te informacje w funkcji `pokazRunde()`.

Klasę *GraRobotow* w pliku `mcpi-rg.py` uzupełniamy dwoma metodami:

```
77      def pokazRunde(self, runda):
78          """Funkcja buduje układ robotów na planszy w przekazanej rundzie."""
79          self.czyscPole()
80          for robot in runda:
81              blok = block.WOOL if robot['player_id'] else block.WOOD
82              x, z = robot['location']
83              print robot['player_id'], blok, x, z
84              self.mc.setBlock(x, 0, z, blok)
85          sleep(1)
86          print
87
88      def czyscPole(self):
89          """Funkcja wypelnia blokami powietrza pole gry."""
90          for xz in self.plansza:
```


IPython
File Edit View Kernel Magic Window Help
In [3]: for robot in runda:
...: print robot['player_id'], robot['hp'], robot['location']
...: x, z = robot['location']
...: print x, z
...:
1 50 [4, 6]
4 6
1 50 [14, 4]
14 4
0 50 [10, 13]
10 13
0 50 [6, 13]
6 13
0 50 [14, 6]
14 6
1 50 [4, 12]
4 12
1 50 [9, 4]
9 4
0 50 [6, 12]
6 12
1 50 [11, 2]
11 2
0 50 [13, 11]
13 11
In [4]:

```
91              x, z = xz
92              self.mc.setBlock(x, 0, z, block.AIR)
```

W metodzie `pokazRunde()` na początku czyścimy pole gry, czyli wypełniamy je blokami powietrza – to zadanie funkcji `czyscPole()`. Jak widać, wykorzystuje ona stworzoną wcześniej listę dozwolonych pól. Kolejne tuple współrzędnych odczytujemy w pętli `for xz in self.plansza:` i rozpakowujemy `x, z = xz`.

Po wyczyszczeniu pola gry, z danych rundy przekazanych do metody `pokazRunde()` odczytujemy w pętli `for robot in runda:` słowniki opisujące kolejne roboty.

W skróconej instrukcji warunkowej sprawdzamy identyfikator gracza: `if robot['player_id']`. Jeżeli wynosi 1 (jeden), roboty będą oznaczane blokami bawełny, jeżeli 0 (zero) – blokami drewna.

Następnie z każdego słownika rozpakowujemy tuplę określającą położenie robota: `x, z = robot['location']`. W uzyskanych współrzędnych umieszczamy ustalony dla gracza typ bloku.

Dodatkowo drukujemy kolejne dane w konsoli `print robot['player_id'], blok, x, z`.

Zanim uruchomimy kod, musimy jeszcze zamienić instrukcję `print runda` w metodzie `uruchom()` na wywołanie omówionej funkcji:

```
70          for runda in json.load(plik):
71              print "Runda ", runda_nr
72              self.pokazRunde(runda)
73              runda_nr = runda_nr + 1
74              if runda_nr > ile:
75                  break
```

Po uruchomieniu kodu powinniśmy zobaczyć już rozgrywkę:

Kolory

Takie same bloki wykorzystywane do pokazywania ruchów robotów obydwu graczy nie wyglądają zbyt dobrze. Spróbujemy odróżnić od siebie obydwie drużyny i pokazać, że roboty w starciach tracą siłę, czyli punkty życia *hp*.

Do definicji klasy *GraRobotow* dodajemy jeszcze jedną metodę o nazwie `wybierzBlok()`:

```
94      def wybierzBlok(self, player_id, hp):
95          """Funkcja dobiera kolor bloku w zależności od gracza i hp robota."""
96          player1_bloki = (block.GRAVEL, block.SANDSTONE, block.BRICK_BLOCK,
97                           block.FARMLAND, block.OBSIDIAN, block.OBSIDIAN)
98          player2_bloki = (block.WOOL, block.LEAVES, block.CACTUS,
99                           block.MELON, block.WOOD, block.WOOD)
100         return player1_bloki[hp / 10] if player_id else player2_bloki[hp / 10]
```

Metoda definiuje dwie tuple, po jednej dla każdego gracza, zawierające zestawy bloków używane do wyświetlenia robotów danej drużyny. Dobór typów w tuplach jest oczywiście czysto umowny.

Siła robotów (*hp*) przyjmuje wartości od 0 do 50, dzieląc tę wartość całkowicie przez 10, otrzymujemy liczby od 0 do 5, które wykorzystamy jako indeksy wskazujące typ bloku przeznaczony do wyświetlenia robota danego zawodnika.

Skrócona instrukcja warunkowa `player1_bloki[hp / 10] if player_id else player2_bloki[hp / 10]` bada wartość identyfikatora gracza `if player_id` i zwraca `player1_bloki[hp / 10]`, jeżeli wynosi on 1 (jeden) oraz `player2_bloki[hp / 10]` jeżeli równa się 0 (zero).

Pozostaje jeszcze zastąpienie instrukcji `blok = block.WOOL if robot['player_id'] else block.WOOD` w metodzie `pokazRunde()` wywołaniem omówionej funkcji, czyli:


Minecraft Pi Simulator (x: 9, y: 10, z: 28) Press Tab to release mouse and Q to quit

```
80          for robot in runda:
81              blok = self.wybierzBlok(robot['player_id'], robot['hp'])
82              x, z = robot['location']
83              print robot['player_id'], blok, x, z
84              self.mc.setBlock(x, 0, z, blok)
```

Trzeci wymiar

Ćwiczenia

Warto poeksperymentować z wizualizacją gry wykorzystując trójwymiarowość Minecrafta. Można uzyskać spektakularne rezulaty. Poniżej kilka sugestii.

- Stosunkowo łatwo urozmaicić wizualizację gry używając wartości *hp* (siła robota) jako współrzędnej określającej położenie bloku w pionie. Wystarczy zmienić instrukcję `self.mc.setBlock(x, 0, z, blok)` w funkcji `pokazRunde()`.
- Jeżeli udało ci się wprowadzić powyższą poprawkę i bloki umieszczame są na różnej wysokości, można zmienić typ umieszczanych bloków na piasek (`SAND`).
- Można spróbować wykorzystać omawianą w scenariuszu *Figury 2D i 3D* bibliotekę minecraftstuff. Wykorzystując funkcję `drawLine()` oraz wartość siły robotów `robot['hp']` jako współrzędną określającą położenie bloku w pionie, można rysować kolejne rundy w postaci słupków.


pos: 10.0, 10.0, -2.2


pos: 27.0, 54.1, 31.1


pos: 0.5, 43.4, 0.5


pos: 6.3, 8.7, -1.9

Informacja: Dziękujemy uczestnikom szkolenia przeprowadzonego w ramach programu "Koduj z Klasą" w Krakowie (03.12.2016 r.), którzy zgłosili powyższe pomysły i sugestie.

Źródła:

- `Skrypty mcpi-rg`
- `Log RG`

  Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

  Autorzy Patrz plik "Autorzy"

Słownik Minecraft Pi

API interfejs programistyczny aplikacji (ang. Application Programming Interface) – zestaw struktur danych, klas obiektów i metod umożliwiających komunikację z aplikacją, biblioteką lub systemem.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Materiały

1. Minecraft Pi Edition
2. Dokumentacja Minecraft API
3. Getting started with Minecraft Pi

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

   Autorzy Patrz plik "Autorzy"

2.5 Dodatkowe informacje

2.5.1 FAQ

1. Jak utworzyć rozruchowy nośnik USB z dystrybucją Linux? »»»
2. ...

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

   Autorzy Patrz plik "Autorzy"

2.5.2 Scenariusze

Poniżej zamieszczamy propozycje scenariuszy zajęć wykorzystujących materiały zgromadzone w repozytorium "Python 101 – materiały Koduj z Klasą".

Cele, materiały i metody

Mów mi Python! – czyli programowanie w języku Python w ramach projektu "Koduj z klasą" organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: http://www.ceo.org.pl/pl/koduj.

Po co, czyli cele

Celem projektu jest zachęcanie nauczycieli i uczniów do programowania z wykorzystaniem języka Python. Przygotowane materiały prezentują zarówno zalety języka, jak i podstawowe pojęcia związane z tworzeniem programów i algorytmiką.

Ogólnym celem projektu jest propagowanie myślenia komputacyjnego, natomiast praktycznym rezultatem szkoleń ma być wyposażenie uczestników w minimum wiedzy i umiejętności umożliwiających samodzielne kodowanie w Pythonie.

Materiały szkoleniowe

1. **Podstawy Pythona**
   - **Toto Lotek** – rozbudowany przykład wprowadzający podstawowe elementy języka, jak i programowania: zmienna, pobieranie i wyprowadzanie tekstu, proste typy danych, instrukcja warunkowa if, wyrażenie logiczne, pętla for, pętla while, break, continue, złożone typy danych, lista, zbiór, tupla, algorytm, poprawność algorytmu, obsługa wyjątków, funkcja, moduł.
   - **Python kreśli** (*Matplotlib*) – materiał prezentujący tworzenie wykresów oraz operacje matematyczne w Pythonie. Zagadnienia: listy, notacja wycinkowa, wyrażenia listowe, wizualizacja danych.
   - **Python w przykładach** – zestaw przykładów prezentujących praktyczne wykorzystanie wprowadzonych zagadnień
2. **Gra robotów** (Robot Game, rgkit*)

   Przykład gry planszowej, w której zadaniem gracza-programisty jest tworzenie strategii walki robotów. Na podstawie przykładowych zasad działania robota oraz odpowiadającego im kodu, gracz "buduje" i testuje swojego robota. Zagadnienia: klasa, metoda, biblioteka, wyrażenia listowe, zbiory, listy, tuple, instrukcje warunkowe.

3. **Gry w Pythonie** (*Pygame*)

   Przykłady multimedialne prezentujące tworzenie i manipulowanie prostymi obiektami graficznymi (Pong, Kółko i krzyżyk) oraz graficzną wizualizację struktur danych (Życie Conwaya).

   - **Pong** (wersja strukturalna i obiektowa)
   - **Kółko i krzyżyk** (wersja strukturalna i obiektowa)
   - **Życie Conwaya** (wersja strukturalna i obiektowa)

5. **Bazy danych w Pythonie**

   Przykłady wykorzystania bazy danych na przykładzie *SQLite3*: model bazy, tabela, pole, rekord, klucz podstawowy, klucz obcy, relacje, połączenie z bazą, operacje CRUD (Create, Read, Update, Delete), podstawy języka SQL, kwerenda, system ORM, klasa, obiekt, właściwości.

   - **Moduł SQL**
   - **Systemy ORM** (*Peewee* i *SQLAlchemy*)
   - **SQL v. ORM**

6. **Aplikacje internetowe**

   Przykłady zastosowania frameworków Flask i Django do tworzenia aplikacji działających w architekturze klient – serwer przy wykorzystaniu protokołu HTTP. Zagadnienia: żądania GET, POST, formularze, renderowanie widoków, szablony, tagi, treści dynamiczne i statyczne, arkusze stylów CSS

   - **Quiz** (Flask)
   - **ToDo** (Flask, SQLite)
   - **Quiz ORM** (Flask)
   - **Czat** (Django)

Materiały online

- Wersja HTML: http://python101.readthedocs.org lub http://python101.rtfd.org
- Wersje źródłowe: https://github.com/koduj-z-klasa/python101
- Forum Koduj z Klasą: http://discourse.kodujzklasa.pl

Oprogramowanie

1. Interpreter Pythona w wersji **2.7.x**.
2. System operacyjny:
   - **Linux** w wersji *live USB* lub *desktop*, np. *LxPupTahr*, (X)Ubuntu lub Debian. Python jest domyślnym składnikiem systemu.
   - lub **Windows** 7/8/10. *Interpreter Pythona należy doinstalować*.
7. **Edytor kodu**, np. *Geany*, *PyCharm*, *Sublime Text*, *Atom* (działają w obu systemach).
8. Narzędzia dodatkowe: *pip, virtualenv, git*.
9. Biblioteki i frameworki Pythona wykorzystywane w przykładach: *Matplotlib, Pygame, Peewee, SQLAlchemy, Flask, Django, Rgkit, RobotGame-bots, Rgsimulator*.

Uwaga: W ramach projektu przygotowano specjalną dystrybucję systemu *Linux Live LxPupTahr* przeznaczoną do instalacji na kluczach USB w trybie live. System zawiera wszystkie wymagane narzędzia i biblioteki Pythona, umożliwia realizację wszystkich scenariuszy oraz zapis plików tworzonych przez uczestników szkoleń.

Metody realizacji

Cechy języka Python przedstawiane są na przykładach, których realizacja może przyjąć różne formy w zależności od dostępnego czasu. Zasada ogólna jest prosta: im więcej mamy czasu, tym więcej metod aktywizujących (kodowanie, testowanie, ćwiczenia, konsola Pythona, konsola Django itp.); im mniej, tym więcej metod podających (pokaz, wyjaśnienia najważniejszych fragmentów kodu, kopiuj-wklej). W niektórych materiałach (np. Robot Game, gry w Pygame) po skopiowaniu i wklejeniu kodu warto stosować zasadę uruchom-zmodyfikuj-uruchom.

1. Prezentacja, czyli uruchamianie gotowych przykładów wraz z omówieniem najważniejszych fragmentów kodu.
2. Wspólne budowanie programów od podstaw: kodowanie w edytorze, wklejanie bardziej skomplikowanych fragmentów kodu.
3. Ćwiczenia w interpreterze Pythona – niezbędne m. in. podczas wyjaśnianiu elementów języka oraz konstrukcji wykorzystywanych w przykładach.
4. Ćwiczenia i zadania wykonywane samodzielnie przez uczestników.

   Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

   Autorzy Patrz plik "Autorzy"

Warsztaty 4 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu "Koduj z klasą" organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: http://www.ceo.org.pl/pl/koduj.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla każdego nauczyciela i ucznia, co oznacza, że materiał zawiera moduły o różnym stopniu trudności. Scenariusze zajęć oraz zakres przykładów można dostosować do poziomu uczestników.

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody* . Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest wersja HTML . Wersje źródłowe dostępne są w repozytorium Python101 .

Materiał zajęć

Podstawy Pythona

Czas realizacji: 1 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Mały Lotek*, punkty *1.2.1 – 1.2.5*, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja:: Na początku zapoznajemy użytkowników ze środowiskiem i narzędziami, tj. menedżer plików, edytor i jego konfiguracja, terminal znakowy, konsola Pythona, uruchamianie skryptu w terminalu, uruchamianie z edytora.

Omawiamy założenia aplikacji *Mały lotek*: losowanie pojedynczej liczby i próba jej odgadnięcia przez użytkownika. Następnie rozpoczynamy wspólne kodowanie wg materiału.

Po ukończeniu pierwszej części można urządzić mini-konkurs: zgadnij wylosowaną liczbę.

Budując program *można* reżyserować podstawowe błędy składniowe i logiczne, aby uczestnicy nauczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity, niewłaściwe wcięcia, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.1

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne.

Materiały i środki: Python 2.7.x, biblioteka Matplotlib, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Python kreśli*. Projektor, dostęp do internetu nie jest konieczny.

Realizacja:: Zaczynamy od prostego przykładu w konsoli Pythona, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia *wyrażeń listowych* zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Nie ma potrzeby ani czasu na dokładne wyjaśnienia tych technik. Celem ich użycia jest zaprezentowanie jednej z zalet Pythona: zwięzłości. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem "klocków" (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka rgkit, przykładowe roboty z repozytorium robotgame-bots oraz skrypt rgsimulator, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Gra robotów*, końcowy kod przykładowego robota w wersji *A* i *B*, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja:: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie *virtualenv*, instalację biblioteki *rgkit, rgbots* i *rgsimulator*, polecenie *rgrun*. Uwaga: jeżeli korzystamy z *LxPupTahr*, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał *Klocki 1* . Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy "prawdziwe" walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – *Klocki 2A* i/lub zbiory – *Klocki 2B* . W zależności od poziomu grupy można przećwiczyć wersje: tylko A, $A + B$, $A + B$ równolegle z porównywaniem kodu. Uwaga: nie mamy czasu na wgłębianie się w szczegóły implementacji.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki *rg* w dodawanych "klockach". Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Warsztaty 8 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu "Koduj z klasą" organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: http://www.ceo.org.pl/pl/koduj.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla nauczycieli, którzy brali udział w pierwszej edycji programu "Koduj z klasą".

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody* . Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest wersja HTML . Wersje źródłowe dostępne są w repozytorium Python101 .

Materiał zajęć

Toto Lotek

Czas realizacji: 2 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Duży lotek*, punkty *1.2.6 – 1.2.14*, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Jako punkt wyjścia prosimy każdego o skopiowanie i uruchomienie *Małego Lotka* . Przypominamy podstawy programowania w Pythonie (zmienna, pobieranie i wyprowadzanie danych, instrukcja warunkowa). Następnie omawiamy założenia aplikacji *Duży lotek*: losowanie i zgadywanie wielu liczb i rozpoczynamy wspólne kodowanie wg materiału.

W zależności od poziomu grupy dbamy o mniej lub bardziej samodzielne wykonywanie przewidzianych w materiale ćwiczeń.

Po ukończeniu można urządzić mini-konkurs, np. zgadnij 5 wylosowanych z 20 liczb.

Budując program *można* reżyserować błędy składniowe i logiczne, aby uczestnicy uczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity,

niewłaściwe wcięcia, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne

Materiały i środki: Python 2.7.x, biblioteka *Matplotlib*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza Python kreśli. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Zaczynamy od prostego przykładu w konsoli Pythona, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia *wyrażeń listowych* zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Wyjaśniamy i ćwiczymy w interpreterze charakterystyczne dla Pythona konstrukcje. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej bądź innej.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem "klocków" (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka rgkit, przykładowe roboty z repozytorium robotgame-bots oraz skrypt rgsimulator, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Gra robotów*, końcowy kod przykładowego robota w wersji *A* i *B*, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja:: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie *virtualenv*, instalację biblioteki *rgkit, rgbots* i *rgsimulator*, polecenie *rgrun*. Uwaga: jeżeli korzystamy z *LxPupTahr*, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał *Klocki 1* . Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy "prawdziwe" walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – *Klocki 2A* i/lub zbiory – *Klocki 2B* . W zależności od poziomu grupy można przećwiczyć wersje: tylko A, $A + B$, $A + B$ równolegle z porównywaniem kodu. W grupach zaawansowanych warto pokazać klocki z zestawu B i omówić działanie *wyrażeń zbiorów* i *funkcji lambda*.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki *rg* w dodawanych "klockach". Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Bazy danych w Pythonie

Czas realizacji: 2*45 min.

Metody: równoległe kodowanie dwóch skryptów w edytorze, uruchamianie i testowanie wersji pośrednich, ćwiczenia z użyciem interpretera *SQLite*.

Materiały i środki: Python 2.7.x, biblioteka *SQLite3 DB-API* oraz framework *Peewee*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *SQL v. ORM* oraz interpreter *SQLite*, kody pełnych wersji obu skryptów. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku pokazujemy przydatność poznawanych zagadnień: wszechobecność baz danych w projektowaniu aplikacji desktopowych i internetowych (tu odesłanie do materiałów prezentujących *Flask* i *Django*); obsługa bazy i podstawy języka SQL to treści nauczania informatyki w szkole ponadgimnazjalnej; zadania maturalne wymagają umiejętności projektowania i obsługi baz danych.

Na podstawie materiału równolegle budujemy oba skrypty metodą kopiuj-wklej. Wyjaśniamy podstawy składni SQL-a, z drugiej eksponując założenia i korzystanie z systemów ORM. Pokazujemy, jak ORM-y skracają i usprawniają wykonywanie operacji CRUD oraz wpisują się w paradygmat projektowania obiektowego. Uwaga: ORM-y nie zastępują znajomości SQL-a, zwłaszcza w zastosowaniach profesjonalnych, mają również swoje wady, np. narzuty w wydajności.

Interpreter *SQLite* wykorzystujemy do pokazania struktury utworzonych tabel (polecenia *.table*, *.schema*), później można (warto) przećwiczyć w nim polecenia CRUD w SQL-u.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Warsztaty 16 godz.

Mów mi Python! – czyli programowanie w języku Python w ramach projektu "Koduj z klasą" organizowanego przez Centrum Edukacji Obywatelskiej. Szczegóły pod adresem: http://www.ceo.org.pl/pl/koduj.

Dla kogo, czyli co musi wiedzieć uczestnik

Dla nauczycieli pragnących wziąć udział w programie „Koduj z klasą" a w pierwszej edycji programu nie mieli takiej możliwości.

Cele, treści i metody

Cele projektu, spis wszystkich materiałów oraz zalecane metody ich realizacji dostępne są w dokumencie *Cele, materiały i metody* . Umieszczono tam również *listę oprogramowania* wymaganego do realizacji wszystkich materiałów. Podstawą szkoleń jest wersja HTML . Wersje źródłowe dostępne są w repozytorium Python101 .

Materiał zajęć

Środowisko programistyczne

Czas realizacji: 1 * 45 min.

Metody: uruchamianie menedżera plików, terminala, edytora, konfigurowanie edytora, uruchamianie interpretera i praca w konsoli Pythona.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Na początku zapoznajemy użytkowników z narzędziami. Uruchamiamy menedżer plików i wykonujemy kilka podstawowych operacji. Omawiamy działanie terminala (zwłaszcza dopełnianie i powtarzanie poleceń). Uruchamiamy i konfigurujemy (np. wcięcia 4 spacje) wybrany edytor kodu. Wspominamy o alternatywach. Uruchamiamy konsolę Pythona i wykonujemy kilka przykładów działań. Omawiamy uruchamianie skryptów w terminalu i z edytora (jeśli jest taka możliwość). Omawiamy instalowanie Pythona i bibliotek za pomocą narzędzi systemowych, jak i programu pip. Wyjaśniamy, w jaki sposób można przygotować bootowalny pendrive (odsyłamy do materiału *Linux Live*).

Toto Lotek

Czas realizacji: 3 * 45 min.

Metody: kodowanie programu w edytorze od podstaw, wprowadzanie elementów języka w konsoli interpretera, ćwiczenia samodzielne w zależności od poziomu grupy.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Toto Lotek*, punkty *1.2.1 – 1.2.14*, kod pełnego programu oraz ewentualne wersje pośrednie. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Omawiamy założenia każdej z części aplikacji, tj.: *Mały lotek* – losowanie pojedynczej liczby i próba jej odgadnięcia przez użytkownika; *Duży lotek* – rozwinięcie, losowanie i zgadywanie wielu liczb. Wspólnie kodujemy wg materiału. Nie stosujemy metody kopiuj-wklej (!). Uczestnicy samodzielnie wpisują kod, uruchamiają go i poprawiają błędy.

Budując program *można* reżyserować podstawowe błędy składniowe i logiczne, aby uczestnicy nauczyli się je dostrzegać i usuwać. Np.: próba użycia liczby pobranej od użytkownika bez przekształcenia jej na typ całkowity, niewłaściwe wcięcia, brak inkrementacji zmiennej iteracyjnej (nieskończona pętla), itp. Uczymy dobrych praktyk programowania: przejrzystość kodu (odstępy) i komentarze.

Po ukończeniu pierwszej części można urządzić mini-konkurs: zgadnij wylosowaną liczbę.

Większość kodu (zgodnie z materiałem) ćwiczymy w konsoli, ucząc jej obsługi i wykorzystania.

Wykresy w Pythonie

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, wspólnie tworzenie i rozwijanie skryptów generujących wykresy, ćwiczenie samodzielne

Materiały i środki: Python 2.7.x, biblioteka *Matplotlib*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza Python kreśli. Projektor, dostęp do internetu nie jest konieczny.

Realizacja: Zaczynamy od *prostego przykładu w konsoli Pythona*, z której cały czas korzystamy. Stopniowo kodujemy przykłady wykorzystując je do praktycznego (!) wprowadzenia *wyrażeń listowych* zastępujących pętle *for*. Pokazujemy również mechanizmy związane z indeksowaniem list, m. in. *notację wycinkową* (ang. *slice*). Wyjaśniamy i ćwiczymy w interpreterze charakterystyczne dla Pythona konstrukcje. Jeżeli wystarczy czasu, zachęcamy do samodzielnego sporządzenia wykresu funkcji kwadratowej bądź innej.

Python w przykładach

Czas realizacji: 1 * 45 min.

Metody: ćwiczenia w konsoli Pythona, samodzielne wspólnie tworzenie i rozwijanie skryptów, ćwiczenia samodzielne.

Materiały i środki: Python 2.7.x, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Python w przykładach* i *Pythonimzów*. Projektor, zalecany dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: W zależności od zainteresowań grupy wybieramy jeden przykład spośród *1.4.5-1.4.9* do wspólnej realizacji, koncentrujemy się na utrwaleniu poznanych rzeczy, pokazaniu nowych. Jeśli się da, wprowadzamy "pythonizmy", pokazując ich użycie w praktyce.

W przykładzie *Ciąg Fibonacciego* można pokazać rozwiązanie rekurencyjne. Przykłady *Słownik słówek* oraz *Szyfr Cezara* pozwalają wyeksponować operacje na tekstach i znakach, bardzo przydatne w rozwiązywaniu zadań typu maturalnego. *Oceny z przedmiotów* ilustrują operacje matematyczne, *Trójkąt* – przykładowe implementowanie algorytmu.

Gry w Pythonie

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki, kodowanie wykorzystaniem "klocków" (gotowego kodu), poprawianie błędów, optymalizacja.

Materiały i środki: Python 2.7.x, biblioteka *Pygame*, czcionka `freesansbold.ttf`, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersje HTML scenariuszy *Pong (str)* i *Pong (obj)*, kody pośrednie i końcowy kod gry. Projektor, dostęp do internetu, jeżeli planujemy wykorzystanie serwisu GitHub do synchronizacji kodu lub scenariusze offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku omawiamy zasady gry w *Ponga*, pierwszej gry komputerowej (sic!). Kodowanie zaczynamy od wersji strukturalnej, wyjaśniając sposób tworzenia obiektów graficznych i manipulowania nimi. Posługujemy się metodą kopiuj-wklej. Zachęcamy uczestników do manipulowania właściwościami obiektów typu kolor, rozmiar itp.

Wyjaśniamy istotę działania programu z interfejsem graficznym opartego na pętli obsługującej zdarzenia (ang. event driven apps).

Następnie przechodzimy do wersji obiektowej, którą realizujemy krokowo metodą kopiuj-wklej wg scenariusza lub omawiamy kod końcowy. Wprowadzamy pojęcia klasa, obiekt (instancja), pole (atrybut) i metoda, konstruktor, pokazując naturalność traktowania graficznych elementów gry jako obiektów mających swoje właściwości (kolor, rozmiar, położenie) i zachowania (rysowanie, ruch), które można modyfikować.

Odtwarzamy logikę i interakcje między obiektami: m. in. zastosowanie operatora ***** do przekazywania argumentów. Pokazujemy elegancję podejścia obiektowego, które wykorzystane zostanie w *Grze robotów* (sic!).

Jako ćwiczenie można zaproponować dodanie drugiej piłeczki i/lub zmianę orientacji pola gry: paletki po bokach.

Gra robotów

Czas realizacji: 2 * 45 min.

Metody: omówienie zasad gry, pokaz rozgrywki między przykładowymi robotami, kodowanie klasy robota z wykorzystaniem "klocków" (gotowego kodu), uruchamianie kolejnych walk.

Materiały i środki: Python 2.7.x, biblioteka rgkit, przykładowe roboty z repozytorium robotgame-bots oraz skrypt rgsimulator, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Gra robotów*, końcowy kod przykładowego robota w wersji *A* i *B*, koniecznie (!) kody wersji pośrednich. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja:: Na początku omawiamy przygotowanie środowiska testowego, czyli użycie *virtualenv*, instalację biblioteki *rgkit, rgbots* i *rgsimulator*, polecenie *rgrun*. Uwaga: jeżeli korzystamy z *LxPupTahr*, w katalogu `~/robot` mamy kompletne wirtualne środowisko pracy.

Podstawą jest zrozumienie reguł. Po wyjaśnieniu najważniejszych zasad gry, konstruujemy robota podstawowego w oparciu o materiał *Klocki 1* . Kolejne implementowane zasady działania robota sprawdzamy w symulatorze, ucząc jednocześnie jego wykorzystania. W symulatorze reżyserujemy również przykładowe układy, wyjaśniając szczegółowe zasady rozgrywki. Później uruchomiamy "prawdziwe" walki, w tym z robotami open source (np. `stupid26.py`).

Dalej rozwijamy strategię działania robota w oparciu o funkcje – *Klocki 2A* i/lub zbiory – *Klocki 2B* . W zależności od poziomu grupy można przećwiczyć wersje: tylko A, $A + B$, $A + B$ równolegle z porównywaniem kodu. W grupach zaawansowanych warto pokazać klocki z zestawu B i omówić działanie *wyrażeń zbiorów* i *funkcji lambda*.

Wprowadzając kolejne zasady, wyjaśniamy odwołania do API biblioteki *rg* w dodawanych "klockach". Kolejne wersje robota zapisujemy w osobnych plikach, aby można je było konfrontować ze sobą.

Zachęcamy uczestników do analizy kodu i zachowań robotów: co nam dało wprowadzenie danej zasady? jak można zmienić kolejność ich stosowania w kodzie? jak zachowują się roboty open source? jak można ulepszyć działanie robota?

Bazy danych w Pythonie

Czas realizacji: 2*45 min.

Metody: równoległe kodowanie dwóch skryptów w edytorze, uruchamianie i testowanie wersji pośrednich, ćwiczenia z użyciem interpretera *SQLite*.

Materiały i środki: Python 2.7.x, biblioteka *SQLite3 DB-API* oraz framework *Peewee*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *SQL v. ORM* oraz interpreter *SQLite*, kody pełnych wersji obu skryptów. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Na początku pokazujemy przydatność poznawanych zagadnień: wszechobecność baz danych w projektowaniu aplikacji desktopowych i internetowych (tu odesłanie do materiałów prezentujących *Flask* i *Django*); obsługa bazy i podstawy języka SQL to treści nauczania informatyki w szkole ponadgimnazjalnej; zadania maturalne wymagają umiejętności projektowania i obsługi baz danych.

Na podstawie materiału równolegle budujemy oba skrypty metodą kopiuj-wklej. Wyjaśniamy podstawy składni SQL-a, z drugiej eksponując założenia i korzystanie z systemów ORM. Pokazujemy, jak ORM-y skracają i usprawniają wykonywanie operacji CRUD oraz wpisują się w paradygmat projektowania obiektowego. Uwaga: ORM-y nie zastępują znajomości SQL-a, zwłaszcza w zastosowaniach profesjonalnych, mają również swoje wady, np. narzuty w wydajności.

Interpreter *SQLite* wykorzystujemy do pokazania struktury utworzonych tabel (polecenia *.table*, *.schema*), później można (warto) przećwiczyć w nim polecenia CRUD w SQL-u.

Aplikacje internetowe

Czas realizacji: 4*45 min.

Metody: kodowanie wybranych aplikacji internetowych, uruchamianie i testowanie kolejnych, ćwiczenia samodzielne.

Materiały i środki: Python 2.7.x, framework *Flask* i/lub *Django*, edytor kodu, terminal, zalecany system *Linux Live LxPupTahr*, wersja HTML scenariusza *Quiz* i *Czat*, kody wersji pośrednich i końcowych aplikacji. Projektor, dostęp do internetu lub scenariusz offline w wersji HTML dla każdego uczestnika.

Realizacja: Omówienie architektury klient-serwer jako podstawy działania aplikacji internetowych. Zaczynamy od scenariusza *Quiz*, który kodujemy metodą kopiuj-wklej. Wprowadzamy i wyjaśniamy pojęcia: protokół HTTP, żądanie GET i POST, kody odpowiedzi HTTP. Po uruchomieniu i przetestowaniu aplikacji pokazujemy jej prostotę, ale wskazujemy też ograniczenia: brak bazy danych, brak możliwości zarządzania użytkownikami, brak możliwości zmiany danych na serwerze.

Następnie realizujemy aplikację "Czat" wg scenariusza, stosując zasadę od znanego do nowego i nawiązując do wcześniejszych materiałów (*SQL v. ORM* i *Quiz*). Pokazujemy modułowość projektowania aplikacji, wynikającą z założeń wzorca MVC. Omawiamy projektowanie modelu bazy jako przykład zastosowania ORM w praktyce. Eksponujemy schemat dodawania stron: widok w `views.py` → szablon html → powiązanie z adresem w `urls.py`. Omawiamy dwa sposoby obsługi żądań: sprawdzanie w funkcji typu żądania i ręczne przygotowanie odpowiedzi oraz oparte na klasach widoki wbudowane automatyzujące większość czynności.

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik "Autorzy"

2.6 Autorzy

- Robert Bednarz: "System i oprogramowanie", "Podstawy Pythona", "Matplotlib", "Gra robotów", "Gry w Pythonie" (wersje strukturalne), "Bazy danych w Pythonie", "Aplikacje okienkowe Qt5", "Aplikacje WWW (Flask)", "Aplikacje WWW (Django)", "Minecraft Pi", "Scenariusze"
- Dorota Rybicka ("Wprowadzenie do języka Python")
- Adam Jurkiewicz ("IDE - edytory kodu")
- Grzegorz Wilczek ("Wprowadzenie do języka Python")
- Janusz Skonieczny: "System i oprogramowanie", "Przygotowanie katalogu projektu", "Gry w Pythonie" (wersje obiektowe), "Git – wersjonowanie kodów źródłowych"
- Paweł Świeczka: "Scenariusze"
- Rafał Brzychcy, Tomasz Nowacki, Łukasz Zarzecki – pomysłodawcy i autorzy wyjściowych wersji materiałów: "Wprowadzenia do języka Python", "Gry w Pythonie" (wersja strukturalna), aplikacji internetowych: Quiz, ToDo, Chatter.

2.6.1 Indices and tables

- genindex
- modindex

- search

Utworzony 2022-05-22 o 19:52 w Sphinx 1.5.3

Autorzy Patrz plik “Autorzy”

Indeks

R

S

T

U

W

Z