Python 101 Documentation

Wydanie 0.5

Centrum Edukacji Obywatelskiej

October 17 2014

Spis treści

1 **Pobieranie tej dokumentacji** **3**

2 **Przygotowanie do szkolenia** **5**
- 2.1 Środowisko systemowe i wymagane oprogramowanie 5
- 2.2 Przygotowanie katalogu projektu 8

3 **Zaczynamy!** **11**
- 3.1 Python - Podstawy programowania 11
- 3.2 Mów mi Python – wprowadzenie do języka 17
- 3.3 Witaj Python! 18
- 3.4 Warunki i pętle 19
- 3.5 Listy, tuple i funkcje 21
- 3.6 Listy, zbiory, moduły i funkcje 23
- 3.7 Słowniki 27
- 3.8 Znam Pythona 29
- 3.9 Nie znam Pythona... jeszcze 30
- 3.10 Pojęcia i materiały 30
- 3.11 Gra w Ponga 31
- 3.12 Gra w życie Conwaya 36
- 3.13 Gra w Kółko i Krzyżyk 41
- 3.14 Quiz – aplikacja internetowa 48
- 3.15 ToDo – aplikacja internetowa 55
- 3.16 Chatter – aplikacja internetowa 64
- 3.17 Indices and tables 77

Niniejsze materiały to dokumentacja do szkolenia z języka Python realizowanego w ramach projektu Koduj z Klasą prowadzonego przez Fundację Centrum Edukacji Obywatelskiej.

Krótki link do tej strony: bit.ly/kzk-py

ROZDZIAŁ 1

Pobieranie tej dokumentacji

Materiały można pobrać do czytania w wersji offline. Poniższa komendy pobiorą dokumentację i rozpakują pliki na pulpicie w folderze `~/Pulpit/python-101-html`:

```
wget -O python-101-html.zip http://koduj-z-klasa.github.io/python101/python-101-html.zip
unzip python-101-html.zip -d ~/Pulpit/
```

Te materiały można także pobrać i zmodyfikować i przygotować według instrukcji w repozytorium

ROZDZIAŁ 2

Przygotowanie do szkolenia

Przed szkoleniem warto przygotować sprawdzić i przygotować swój komputer.

2.1 Środowisko systemowe i wymagane oprogramowanie

Postaraliśmy się by każdy z uczestników szkolenia Koduj z Klasą otrzymał gotowe środowisko, dzięki unifikacji minimalizujemy liczbę problemów konfiguracyjnych. Jednak przygotowanie swojego środowiska nie wymaga dużych nakładów pracy, wystarczy wykonać kilka krótkich instrukcji by dostosować swój komputer na potrzeby tego szkolenia.

2.1.1 Szkolny Remix Ucznia (SRU)

Nasze materiały zakładają wykorzystanie Szkolnego Remixu Ucznia (SRU) ze względu na jego gotowość do realizacji celów szkoleniowych niemal z pudełka.

http://sru.e-swoi.pl

Przykłady zakładają jednakową ścieżkę do katalogu domowego użytkownika, w przypadku innych instalacji należy uwzględniać własną ścieżkę:

`/home/sru`

LiveCD lub uruchomienie z Pendrive

Środowisko SRU jest przygotowane do uruchomienia bez instalacji na komputerach z wykorzystaniem tzw. Live CD. W ramach programu Koduj z Klasą przekazujemy pamięci USB zawierające SRU i umożliwiające uruchomienie systemu bez instalacji.

Informacja: Pamięci USB zwykle są tak przygotowane że pomiędzy uruchomieniami zapamiętywane są tylko zmiany w katalogu domowym. Dzięki temu użytkownicy nie mogą popsuć konfiguracji systemu, ale też nie są wstanie instalować nowego oprogramowania.

Dla komputerów z systemem Windows 8 i BIOS UEFI

Aby używać Szkolny Remiks Ucznia jako maszynę wirtualną bez konieczności startu z USB, zainstalujcie Virtualbox, a następnie pobierzcie plik OVA i po prostu kliknijcie na niego dwa razy. Następnie postępujcie wg wskazówek wyświetlanych przez VirtualBox. Po wykonanym imporcie, będziecie mogli po prostu uruchomić maszynę w Virtualbox.

Po imporcie plik OVA można skasować, aby nie zabierał już miejsca. Nie będzie więcej potrzebny. pamięci USB zawierające SRU i umożliwiające uruchomienie systemu bez instalacji.

Niezbędne pakiety:

- Virtualbox - wersja dla Windows
- Maszyna wirtualna SRU

Brakujące komponenty

Na starszych wersjach SRU może brakować oprogramowania przydatnego podczas szkolenia warto je do instalować

Pobranie przykładów na komputerze uruchomionego z pomocą LiveCD lub Pendrive SRU wymaga wcześniejszego instalacji narzędzia GIT, w terminalu (Win+T) uruchamiamy:

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

Inne systemy Linux

Wykorzystanie innych systemów Linux do celów szkoleniowych nie powinno sprawiać większych problemów pod warunkiem, że uzupełnimy brakujące oprogramowanie.

Do instalacji wykorzystujemy systemowy package manager, przykładowo dla Ubuntu i Debiana:

```
$ sudo apt-get build-dep python-pygame
$ sudo apt-get install python-dev python-pip python-virtualenv
```

2.1.2 Środowisko Windows

Nie zalecamy wykorzystania Windows ze względu na kompatybilność poleceń i ścieżek do plików używanych w scenariuszach. Jednak użycie Windows jako środowiska szkoleniowego nie jest wykluczone.

Informacja: Pamiętaj by zmieniać znaki / (slash) na \ (backslash) w ścieżkach, natomiast w miejscu komend systemu Linux użyj bliskich zamienników z Windows.

Instalacja Python 2.7 pod windows

Możemy szybko zainstalować Python z pomocą konsoli PowerShell (taka niebieska)

```
(new-object System.Net.WebClient).DownloadFile("https://www.python.org/ftp/python/2.7.8/python-2.7.8
msiexec /i python-2.7.8.msi TARGETDIR=C:\Python27
[Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:\Python27\Scripts\", "User")
(new-object System.Net.WebClient).DownloadFile("https://raw.github.com/pypa/pip/master/contrib/get-p
C:\Python27\python.exe get-pip.py virtualenv
```

Pozostałe biblioteki dystrybuowane w wersjach binarnych musimy zainstalować z katalogu /arch/ w repo, pozostałe instalujemy za pomocą pip:

```
pip install -r requirements.txt
```

Jak nie ma PowerShell

Jeśli nie mamy PowerShella to pozostaje ręcznie ściągnąć plik instalacyjny

https://www.python.org/downloads/

A następnie zainstalować pip przy użyciu świeżo zainstalowanego Pythona :)

```
python -c "exec('try: from urllib2 import urlopen \nexcept: from urllib.request import urlopen');f=u
```

Ponadto możemy ustawić zmienną systemową by za każdym razerm nie używać pełnej ścieżki.

```
set PATH=%PATH%;c:\Python27\;c:\Python27\Scripts\
```

2.1.3 Środowisko programistyczne Geany

Geany to profesjonalne środowisko środowisko OpenSource, a więc dostępne na licencji GNU General Public Licence. To oznacza, że każdy użytkownik posiada:

- wolność uruchamiania programu, w dowolnym celu (wolność 0)
- wolność analizowania programu oraz dostosowywania go do swoich potrzeb (wolność 1)
- wolność rozpowszechniania kopii programu (wolność 2)
- wolność udoskonalania programu i publicznego rozpowszechniania własnych ulepszeń, dzięki czemu może z nich skorzystać cała społeczność (wolność 3).

Geany oferuje kolorowanie składni dla najpopularniejszych języków, m.in. C, C++, C#, Java, PHP, HTML, Python, Perl i Pascal, wsparcie dla kodowania w ponad 50 standardach, mechanizm automatycznego zamykanie tagów dla HTMLXML, auto-wcięć, pracy na kartach i wiele, wiele więcej. Podczas pisania kodu przydatny okazuje się brudnopis, pozwalający tworzyć dowolne notatki, a także możliwość kompilacji plików źródłowych bezpośrednio z poziomu programu.

2.1.4 Środowisko programistyczne PyCharm

PyCharm to profesjonalne, komercyjne środowisko programistyczne dostępne za darmo do celów szkoleniowych.

To IDE doskonale wspiera proces uczenia się. Dzięki nawigacji po kodzie, podpowiedziom oraz wykrywaniu błędów niemal na bieżąco uczniowie mniej czasu będą spędzać na szukaniu problemów a więcej na poznawaniu tajników programowania.

Szybka instalacja na systemach linux

Instalacja wersji testowej na systemach Linux wymaga pobrania i rozpakowania archiwum:

```
wget http://download.jetbrains.com/python/pycharm-professional-3.4.1.tar.gz -O - | tar -xz
./pycharm-3.4.1/bin/pycharm.sh
```

Ręczna instalacja

Na systemach Windows możemy zainstalować PyCharm po pobraniu pliku instalacyjnego ze strony producenta z pomocą przeglądarki.

Jak zdobyć bezpłatną licencję

Każdy nauczyciel może wystąpić o klucz licencyjny przy pomocy formularza dostępnego na stronie producenta

Polski słownik ortograficzny

W programie możemy włączyć sprawdzanie polskiej pisowni, jednak potrzebne jest wskazanie pliku słownika. W ustawieniach `Ctrl+Alt+S` szukamy *spell* i dodajemy `custom dictionaries folder` wskazując na `/usr/share/hunspell/` (lokalizacja w SRU).

2.2 Przygotowanie katalogu projektu

Poszczególne zadania zakładają wykorzystanie wspólnego katalogu projektu `python101` znajdującego się w katalogu domowym użytkownika.

2.2.1 Pobieranie materiałów

Materiały szkoleniowe zostały umieszczone w repozytorium GIT 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 https://github.com/koduj-z-klasa/python101.git
```

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

Informacja: Przykłady zawierające znak zachęty `$` oznaczają komendy do wykonania w terminalu systemu operacyjnego (uruchom przez `Win+T`).

Oprócz znaku zachęty `$` przykłady mogą zawierać informację o lokalizacji w jakiej należy wykonać komendę. Np. `~/python101$` oznacza że komendę wykonujemy w folderze `python101` w katalogu domowym użytkownika, czyli `/home/sru/python101` w środowisku SRU.

Komendy należy kopiować i wklejać bez znaku zachęty `$` i poprzedzającego tekstu. Komendy można wklejać do terminala środkowym klawiszem myszki.

2.2.2 Korzystanie z kodu źródłowego w opisie przykładów

W materiałach będą pojawiać się przykłady kodu źródłowego jak ten poniżej. Te przykłady pokazują jak nasz kod może się rozwijać.

By wspierać uczenie się na błędach i zwracanie uwagi na niuanse składni języka programowania, warto by część przykładów uczestnicy próbowali odtworzyć samodzielnie.

Jednak dla większego tempa i w przypadku jasnych przykładów warto je zwyczajnie kopiować, omawiać ich działanie i ewentualnie modyfikować w ramach eksperymentów.

Niektóre przykłady starają się zachować numerację linii zgodną z oczekiwanym rezultatem. Przykładowo kod poniżej powinien zostać wklejony w linii `51` omawianego pliku.

Podczas przepisywania kodu można pominąć kawałki dokumentujące kod, to znaczy tzw. *komentarze*. Komentarzem są teksty zaczynające się od znaku `#` oraz teksty zamknięte pomiędzy potrójnymi cudzysłowami `"""`.

2.2.3 Synchronizacja kodu pomiędzy etapami przykładów

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

Zaczynamy!

3.1 Python - Podstawy programowania

Python jest dynamicznym językiem interpretowanym.

Interpretowany tzn. że kod, który napiszemy możemy natychmiast wykonać bez potrzeby tłumaczenia (kompilowania) kodu programistycznego na język maszynowy (czyli na formę zrozumiałą przez komputer).

Interpreter tłumaczy (kompiluje w locie) nasz kod oraz natychmiast go uruchamia. Interpreter może również być używany w trybie interaktywnym do testowania `kawałków` kodu (np. IPython omawiany w kolejnych scenariuszach).

Naukę zaczynamy od poznania interpretera. Interpreter uruchamiamy z konsoli poleceniem

```
$ python
```

Po uruchomieniu interpretera komputer powinien wypisać trzy linie tekstu a w czwartej tak zwany znak zachęty, po którym wpisujemy komendy. Standardowym znakiem zachęty w interpreterze Pythona jest >>>.

Informacja: Przykłady zawierające znak zachęty `$` oznaczają komendy do wykonania w terminalu systemu operacyjnego (uruchom przez `Win+T`).

Komendy należy kopiować i wklejać bez znaku zachęty `$` i poprzedzającego tekstu. Komendy można wklejać do terminala środkowym klawiszem myszki.

3.1.1 Zmienne

W interpreterze wpisz poniższy przykład:

```
>>> a = 3
>>> b = 5
>>> a + b
8
>>> c = -6.5
>>> a + c
-3.5
>>> 7/2
3.5
>>> 2*6
12
>>> 2**6
64
>>> kwiatek = 'stokrotka'
```

```
>>> kwiatek2 = "róża"
>>> kwiatek + ' i ' + kwiatek2
'stokrotka i róża'
```

Informacja: Przykłady zawierające znaki zachęty >>> elementy do wykonania w interpreterze języka Python. Komendy należy kopiować i wklejać bez znaku zachęty >>>.

W powyższym przykładzie wykonujemy operacje na zmiennych. Zmienne są "pudełkami" trzymającymi różne informacje. Zmienna posiada swoją nazwę i wartość. Nazwa zmiennej w Pythonie nie może zawierać:

- polskich znaków
- cyfry na pierwszym miejscu
- spacji
- znaków specjalnych za wyjątkiem podkreślnikiem (_)

Wielkość liter jest rozróżniana (zmienna x i zmienna X to dwie różne zmienne). Przykłady poprawnych:

- `i`
- `_chodnik`
- `nazwa_23`
- `a2c4`

Przykłady niepoprawnych:

- `1nazwa`
- `4_strony_świata`
- `z-myslnikiem`,
- `nazwa ze spacja`

Główne typy zmiennych, którymi będziemy się zajmować to:

- liczby całkowite (integer)
- liczby zmiennoprzecinkowe (float)
- napisy, czyli tzw. Łańcuchy znaków (string)

Istnieje wiele innych typów zmiennych. [odniesienie do zewn. Źródeł] Zróbmy proste zadanie - będąc w interpreterze wykonaj:

```
zmienna1 = raw_input(``Podaj imię: ``)
print(``Witaj``, zmienna1)
```

Stworzyliśmy właśnie pierwszy program w języku Python. Gratulacje!

Przy użyciu funkcji `raw_input` pobiera on do zmiennej to, co użytkownik wpisze z klawiatury a następnie używając funkcji `print` wyświetla na ekranie kawałek tekstu oraz zawartość wcześniej utworzonej zmiennej.

Aby zamienić podaną z klawiatury liczbę na zmienną liczbową (int lub float) musimy skorzystać z funkcji `int`.

```
zmienna1 = raw_input("Podaj 1 liczbę: ")
zmienna2 = raw_input("Podaj 2 liczbę: ")
wynik = int(zmienna1) + int(zmienna2)
print("Suma:", wynik)
```

Wskazówka: Grupy mniej zaawansowane mogą poświęcić czas na eksperymenty z operacjami na zmiennych podawanych z klawiatury.

Funkcje

Funkcje są to wcześniej zdefiniowane kawałki kodu, których możemy później użyć do wykonania określonej czynności, zamiast wpisywać ten sam kod po raz kolejny.

W okrągłych nawiasach po nazwach funkcji umieszczamy parametry lub argumenty funkcji (może być ich więcej niż jeden). Parametry i argumenty oddzielamy od siebie przecinkami. Jak widać na przykładzie funkcji `raw_input` niektóre funkcje pozostawiają `coś` po sobie. W tym przypadku funkcja `raw_input` pozostawia po sobie to, co użytkownik wpisał z klawiatury a my wrzucamy to do naszej zmiennej `zmienna1`. Kiedy funkcja pozostawia po sobie jakieś dane, mówimy, że funkcja zwraca dane.

Kolejny program zapiszemy już w pliku aby prościej było go zmieniać oraz wykonywać wiele razy. W tym celu należy otworzyć edytor tekstu, wpisać do niego instrukcje języka Python, a następnie zapisać z rozszerzeniem `.py`. Aby uruchomić tak zapisany program należy będąc w linii poleceń (konsola / terminal) w tym samym katalogu gdzie zapisaliśmy nasz plik wpisać:

```
$python nazwa-pliku.py
```

3.1.2 Wyrażenia warunkowe

Do podejmowania decyzji w programowaniu służy instrukcja warunkowa `if`.

Blok kodu podany po instrukcji if zostanie wykonany wtedy, gdy wyrażenie warunkowe będzie prawdziwe. W przeciwnym przypadku blok kodu zostanie zignorowany. Część `else` jest przydatna, jeśli chcemy, żeby nasz program sprawdził wyrażenie warunkowe i wykonał blok kodu jeśli wyrażenie warunkowe jest prawdziwe lub wykonał inny blok kodu jeśli wyrażenie warunkowe było fałszywe.

Python pozwala także na sprawdzenie większej liczby warunków w ramach jednej instrukcji `if`. Służy do tego instrukcja `elif` (skrót od else if).

```
if wyrażenie_warunkowe:
    blok kodu 1
elif:
    blok kodu 2
else:
    blok kodu 3
```

Wszystkie instrukcje w bloku kodu muszą być wcięte względem instrukcji `if`. W ten sposób Python rozpoznaje, które instrukcje ma wykonać po sprawdzeniu prawdziwości wyrażenia. Tak samo po instrukcjach `elif` i `else` musimy wstawić dwukropek a instrukcje muszą być wcięte.

Głębokość wcięcia nie ma znaczenia (dobry zwyczaj programowania w Pythonie mówi, żeby używać czterech spacji) ale musi być ono w całym programie zawsze tej samej głębokości. Pobawmy się instrukcjami `if`, `elif` i `else` na prostym przykładzie.

```
zmienna = raw_input('Podaj liczbę: ')
zmienna = int(zmienna)
if zmienna > 0:
    print('Wpisałeś liczbę dodatnią')
elif zmienna == 0:
    print('Wpisałeś zero')
else:
```

```
    print('Wpisałeś liczbę ujemną')
print('Koniec programu')
```

W programie na początku wczytywana jest wartość z klawiatury do zmiennej, a następnie dokonujemy zmiany jej typu na liczbę całkowitą. W dalszej części stosujemy instrukcję `if` sprawdzając czy wartość podanej liczby jest większa od 0. Jeśli wartość będzie większa od 0 na ekranie wyświetlony będzie napis `Wpisałeś liczbę dodatnią`, jeśli nie, program wykona kolejną instrukcję: `elif` sprawdzając czy liczba jest równa 0. Jeśli żaden z powyższych warunków nie będzie spełniony wykonane zostanie polecenie zawarte po instrukcji `else`. Program zakończy się wyświetlając: `Koniec programu`.

Jak również widać porównanie w Pythonie, wykonujemy poprzez podwójne użycie znaku równości: `==`. Matematyczne wyrażenie `nie równe` () w Pythonie zapisujemy jako `!=`.

3.1.3 Gra w "zgadnij liczbę"

Napisz program, w którym:

- do zmiennej `dana` przypiszesz pewną liczbę
- użytkownik będzie mógł podać z klawiatury dowolną liczbę całkowitą
- jeżeli użytkownik trafi program wyświetli komunikat: `Gratulacje!`, a jeśli nie, to wyświetli napis określający czy podana liczba jest większa od danej czy mniejsza.

```
dana = 18
strzal = int(raw_input("Wpisz liczbę całkowitą"))
if strzal == dana:
    print("Gratulacje! Zgadłeś")
elif strzal < dana:
    print("Nie! Szukana liczba jest większa!")
else:
    print("Nie! Szukana liczba jest mniejsza!")
print("Koniec programu.")
```

3.1.4 Zadania dodatkowe

1. Za pomocą poznanych narzędzi stwórz program będący kalkulatorem.
2. Napisz program rozwiązujący równania kwadratowe (*Przykład równania kwadratowego*).
3. Napisz program, który spyta użytkownika ile ma lat, a następnie wyświetli czy osoba ta jest młodzieżą, dzieckiem czy dorosłym (załóżmy, że dziecko ma mniej niż 12 lat, a dorosły więcej niż 18).
4. Napisz program, który będzie sortował trzy podane przez użytkownika liczby.
5. Napisz program, który w odpowiedzi na podaną przez użytkownika liczbę będzie wyświetlał komunikat czy jest to liczba parzysta, czy nieparzysta.
6. Napisz program, który będzie sprawdzał czy z podanych przez użytkownika trzech długości można zbudować trójkąt.

Przykład równania kwadratowego

```
1 print 'Dla równania kwadratowego ax2+bx+c=0'
2 a=int(raw_input('podaj wartość parametru a: '))
3 b=int(raw_input('podaj wartość parametru b: '))
4 c=int(raw_input('podaj wartość parametru c: '))
```

```
5  delta = b**2-4*a*c
6  if delta > 0:
7      x1 = (-b-delta**(1/2))/(2*a)
8      x2 = (-b+delta**(1/2))/(2*a)
9      print 'x1 = ', x1, ', x2= ', x2
10 elif delta == 0:
11     x0 = -b/(2*a)
12     print 'x0 = ', x0
13 else:
14     print 'brak rozwiązań'
```

3.1.5 Pętla WHILE

Pętla while służy do konstrukcji bloku instrukcji, które będą wykonywane warunkowo. W programie najpierw będzie sprawdzane czy warunek jest spełniony – jeśli tak, to wykonane będą wszystkie instrukcje zawarte w bloku. Następnie ponownie sprawdzany jest warunek, jeśli nadal jest spełniony to ponownie wykonuje wszystkie polecenia. Pętla jest wykonywana tak długo, jak długo warunek jest prawdziwy.

```
while wyrażenie_warunkowe:
    blok kodu
```

Zobaczmy działanie pętli `while` na poniższym przykładzie.

```
1  import random
2  dana = random.choice(range(10))
3  kontynuuj = True
4  while koniec:
5      strzal = int(raw_input("Wpisz liczbę całkowitą"))
6      if strzal == dana:
7          print("Gratulacje! Zgadłeś")
8          kontynuuj = False
9      elif strzal < dana:
10         print("Nie! Szukana liczba jest większa!")
11     else:
12         print("Nie! Szukana liczba jest mniejsza!")
13 print("Koniec programu.")
```

Program będzie wykonywany do momentu, w którym użytkownik poda właściwą liczbę. Zatem nie trzeba do każdego strzału ponownie uruchamiać programu. Zmienna `kontynuuj` ma ustawioną wartość logiczną `True` (z angielskiego prawda). W momencie, w którym użytkownik poda właściwą liczbę zmienna przyjmie wartość logiczną `False` (z angielskiego fałsz), co spowoduje zakończenie wykonywania pętli while.

3.1.6 Wyrażenia break i continue

Wyrażenie `break` powoduje natychmiastowe zakończenie wykonywania pętli.

```
1  import random
2  dana = random.choice(range(10))
3  while True:
4      strzal = int(raw_input("Wpisz liczbę całkowitą"))
5      if strzal == dana:
6          print("Gratulacje! Zgadłeś")
7          break
8      elif strzal < dana:
9          print("Nie! Szukana liczba jest większa!")
10     else:
```

```
        print("Nie! Szukana liczba jest mniejsza!")
print("Koniec programu.")
```

Wyrażenie `continue` powoduje ominięcie następujących po nim wyrażeń w bloku, a następnie rozpoczyna ponowne wykonanie pętli.

```
import random
dana = random.choice(range(10))
while True:
    strzal = int(raw_input("Wpisz liczbę całkowitą"))
    if strzal > dana:
        print("Nie! Szukana liczba jest mniejsza!")
        continue
    elif strzal < dana:
        print("Nie! Szukana liczba jest większa!")
        continue
    print("Gratulacje! Zgadłeś")
    break
print("Koniec programu.")
```

Zadania dodatkowe

1. Napisz program, który sumuje liczby dodatnie podawane przez użytkownika – pętla pozwala użytkownikowi podawać liczby dopóki nie poda liczby niedodatniej. Następnie obok podawanego wyniku będzie wyświetlana liczba określająca ilość podanych liczb.
2. Na podstawie wcześniejszego zadania napisz program obliczający średnią liczb dodatnich, a następnie zmodyfikuj go tak, aby obliczana była średnia również dla liczb ujemnych.

3.1.7 Pętla FOR

Pętla for służy do wykonywania tego samego bloku operacji dla każdego elementu z pewnej listy. Ilość wykonań tego bloku jest równa liczbie elementów tej listy. Wywoływana w pętli zmienna przyjmuje po kolei wartości każdego z elementów.

Przykłady list:

- lista liczb wpisanych ręcznie – elementy podane w nawiasach kwadratowych

  ```
  [2,3,4,5]
  ```

- funkcja range – wywoła kolejno liczby naturalne zaczynając od podanej w nawiasie na pierwszym miejscu, kończąc na liczbie mniejszej o 1 od liczby na miejscu drugim

  ```
  range(2,6)
  ```

Zobrazujmy działanie pętli `for` na prostym przykładzie, wymieniającym kolejno elementy z pewnej listy.

```
print("Mamy listę elementów: ", [5,6,7,8])
for liczba in [5,6,7,8]:
    print("element listy: ", liczba)
```

Zadania dodatkowe

1. Napisz dwa programy, które wypisują liczby naturalne od 1 do 15. W pierwszym programie wykonaj pętlę for, a w drugim while.

2. Zmodyfikuj powyższe zadanie, tak aby programy obliczały sumę liczb od 1 do 15.
3. Za pomocą pętli for, napisz program, który oblicza silnię liczby podanej przez użytkownika.
4. Oblicz sumę kwadratów liczb naturalnych z zakresu od 1 do 100.

3.1.8 Słowniczek

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

Interpreter program, który zajmuje się tłumaczeniem kodu języka programowania na język maszynowy i jego wykonywaniem.

Zmienne symbole zdefiniowane i nazwane przez programistę, które służą do przechowywania wartości, obliczeń na nich i odwoływanie się do wartości przez zdefiniowaną nazwę.

Funkcje fragmenty kodu zamknięte w określonym przez programistę symbolu, mogące przyjmować parametry oraz mogące zwracać wartości. Umożliwiają wielokrotne wywoływanie tego samego kodu, bez konieczności jego przepisywania za każdym razem, gdy zajdzie potrzeba jego wykonania.

Typ zmiennych rodzaj danych, który przypisany jest do zmiennej w momencie jej tworzenia.

3.2 Mów mi Python – wprowadzenie do języka

3.2.1 Jestem Python

Python jest dynamicznie typowanym językiem interpretowanym 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".

W systemach opartych na Linuksie interpreter Pythona jest standardowo zainstalowany, ponieważ duża część oprogramowania na nim bazuje. W systemach Microsoft Windows Pythona należy doinstalować. Funkcjonalność Pythona może być dowolnie rozszerzana dzięki licznym bibliotekom pozwalającym tworzyć aplikacje okienkowe (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 (Anaconda, Canopy).

Kodować można w dowolnym edytorze tekstowym, jednak ze względów praktycznych warto korzystać z programów ułatwiających pisanie kodu. Polecić można np. lekkie, szybkie i obsługujące wiele języków środowisko Geany, a także profesjonalne rozwiązanie, jakim jest aplikacja PyCharm. Obydwa programy działają na platformie Linux i Windows.

Zanim przystąpimy do pracy w katalogu domowym utworzymy podkatalog python, w którym będziemy zapisywali nasze skrypty:

```
~ $ mkdir python
```

Poznawanie Pythona zrealizujemy poprzez rozwiązywanie prostych zadań, które pozwolą zaprezentować elastyczność i łatwość tego języka. Nazwy kolejnych skryptów umieszczone są jako komentarz zawsze w czwartej linii kodu. Pliki zawierające skrypty Pythona mają zazwyczaj rozszerzenie .py. Bardzo przydatnym narzędziem podczas kodowania w Pythonie jest konsola interpretera, którą uruchomimy wydając w terminalu polecenie python lub ipython [1]. Można w niej testować i debugować wszystkie wyrażenia, warunki, polecenia itd., z których korzystamy w skryptach.

[1] Ipython to rozszerzona konsola Pythona przeznaczona do wszelkiego rodzaju interaktywnych obliczeń.

3.3 Witaj Python!

3.3.1 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."

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

# ~/python/01_hello.py

# inicjalizujemy zmienne (wartości)
curYear = 2014
pythonYear = 1989
wiekPythona = curYear - pythonYear # ile lat ma Python

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

# wyprowadzamy dane
print "Witaj w moim świecie ",imie
print "Mów mi Python, mam", wiekPythona, "lat."

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

3.3.2 JAK TO DZIAŁA

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

Deklaracja zmiennej w Pythonie nie jest wymagana, wystarczy podanej nazwie przypisać jakąś wartość za pomocą operatora przypisania "=" [2]. Zmiennym często przypisujemy wartości za pomocą wyrażeń, czyli działań arytmetycznych lub logicznych.

Funkcja `raw_input()` zwraca pobrane z klawiatury znaki jako napis, czyli typ **string**.

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

Funkcja `print` drukuje podane argumenty oddzielone przecinkami. Komunikaty tekstowe ujmujemy w cudzysłowy podwójne lub pojedyncze. Przecinek oddziela kolejne argumenty spacjami.

Instrukcja `if` (jeżeli) pozwala na warunkowe wykonanie 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. Warto zauważyć, że polecenia instrukcji warunkowej kończymy dwukropkiem.

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

[2] Dlatego niekiedy mówi się, że w Pythonie zmiennych nie ma, są natomiast wartości określonego typu.

3.3.3 POĆWICZ SAM

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

3.4 Warunki i pętle

3.4.1 ZADANIE

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

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

# ~/python/02_if.py

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

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

    if a < b:
        if a < c:
            print a
        else
            print c
    elif b < c:
        print b
    else:
        print c

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

print "Nie, to nie :-("
```

3.4.2 JAK TO DZIAŁA

Pojęcia: *pętla, obiekt, metoda, instrukcja warunkowa zagnieżdżona, formatowanie kodu.*

Pętla `while` umożliwia powtarzanie określonych operacji, czyli pozwala użytkownikowi wprowadzać kolejne serie liczb. Definiując pętle określamy warunek powtarzania kodu. Dopóki jest prawdziwy, czyli dopóki zmienna *op* ma wartość "t" pętla działa. Do wydzielania kodu przynależnego do pętli i innych instrukcji (np. `if`) stosujemy wcięcia. Formatując kod, możemy używać zarówno tabulatorów, jak i spacji, ważne aby w obrębie pliku było to konsekwentne [3].

W Pythonie wszystko jest obiektem, czyli typy wbudowane, np. napisy, posiadają metody (funkcje) wykonujące określone operacje na wartościach. W podanym kodzie funkcja `raw_input()` zwraca ciąg znaków wprowadzony przez użytkownika, z którego wydobywamy poszczególne słowa za pomocą metody `split()` typu string. Instrukcje warunkowe (`if`), jak i pętle, można zagnieżdżać stosując wcięcia. W jednej złożonej instrukcji warunkowej można sprawdzać wiele warunków (`elif:`).

[3] Dobry styl programowania sugeruje używanie do wcięć 4 spacji.

3.4.3 POĆWICZ SAM

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

3.4.4 ZADANIE

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

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

# ~/python/03_petle.py

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

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",
    print litera.upper(), "=>", litera,
```

3.4.5 JAK TO DZIAŁA

Pojęcia: *iteracja, pętla, kod ASCII, lista, inkrementacja, operatory arytmetyczne, logiczne, przypisania i zawierania.*

Pętla for wykorzystuje zmienną i, która przybiera wartości z listy liczb całkowitych zwróconej przez funkcję `range()`. Parametry tej funkcji określają wartość początkową i końcową listy, przy czym wartość końcowa nie wchodzi do listy. Kod `range(122,96,-1)` generuje listę wartości malejących od 122 do 97(!) z krokiem -1.

Funkcja `chr()` zwraca znak, którego kod ASCII, czyli liczbę całkowitą, przyjmuje jako argument. Metoda `lower()` typu string (napisu) zwraca małą literę, `upper()` – dużą. Wyrażenie przypisywane zmiennej *tmp* pokazuje, jak można łączyć napisy (konkatenacja).

Zmienna pomocnicza `x` jest zwiększana (inkrementacja) w pętlach o 1. Wyrażenie `x += 1` odpowiada wyrażeniu `x = x + 1`. Pierwszy warunek wykorzystuje operator logiczny `and` (koniunkcję) i operator modulo `%` (zwraca resztę z dzielenia), aby do ciągu znaków w zmiennej `tmp` dodać znak końca linii (`\n`) za pomocą operatora +=. W drugim warunku używamy operatora porównania ==.

Poniżej podano wybrane operatory dostępne w Pythonie.

- **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)

3.4.6 POĆWICZ SAM

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.

3.5 Listy, tuple i funkcje

3.5.1 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*.

Wszystkie poniższe przykłady proponujemy wykonać w konsoli Pythona. Nie umieszczaj w konsoli komentarzy, możesz też pominąć lub skrócić komunikaty funkcji `print`. Można również wpisać poniższy kod do pliku i go uruchomić.

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

# ~/python/04_1_listy.py

tupla = input("Podaj liczby oddzielone przecinkami: ")
lista = [] # deklaracja pustej listy
for i in range(len(tupla)):
    lista.append(int(tupla[i]))

print "Elementy i ich indeksy:"
for i, v in enumerate(lista):
    print v, "[",i,"]"

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

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

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

```
27 print lista
28
29 a, i = input("Podaj element do dodania i indeks, przed którym ma się on znaleźć: ")
30 lista.insert(i, a)
31 print lista
32
33 e = int(raw_input("Podaj liczbę, której wystąpienia w liście chcesz zliczyć? "))
34 print lista.count(e)
35 print "Pierwszy indeks, pod którym zapisana jest podana liczba to: "
36 print lista.index(e)
37
38 print "Pobieramy ostatni element z listy: "
39 print lista.pop()
40 print lista
41
42 i, j = input("Podaj indeks początkowy i końcowy, aby uzyskać frgament listy: ")
43 print lista[i:j]
```

3.5.2 JAK TO DZIAŁA

Pojęcia: *tupla, lista, metoda.*

Funkcja `input()` pobiera dane wprowadzone przez użytkownika (tak jak `raw_input()`), ale próbuje zinterpretować je jako kod Pythona. Podane na wejściu liczby oddzielone przecinkami zostają więc spakowane jako **tupla** (krotka). Jest to uporządkowana sekwencja poindeksowanych danych, przypominająca tablicę, której wartości nie można zmieniać. Gdybyśmy chcieli wpisać do tupli wartości od razu w kodzie, napisalibyśmy: `tupla = (4, 3, 5)` [4]. Listy to również uporządkowane sekwencje indeksowanych danych, zazwyczaj tego samego typu, które jednak możemy zmieniać.

Dostęp do elementów tupli lub listy uzyskujemy podając nazwę i indeks, np. `lista[0]`. Elementy indeksowane są od 0 (zera!). Funkcja `len()` zwraca ilość elementów w tupli/liście. Funkcja `enumerate()` zwraca obiekt zawierający indeksy i elementy sekwencji (np. tupli lub listy) podanej jako atrybut. Funkcja `reversed()` zwraca odwróconą sekwencję.

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. Funkcja `reversed(lista)` zwraca kopię listy w odwróconym porządku, natomiast `sorted(lista)` zwraca kopię listy posortowanej rosnąco. Jeżeli chcemy trwale odwrócić lub posortować elementy listy stosujemy metody: `.reverse()` i `.sort()`. Z każdej sekwencji (napisu, tupli czy listy) możemy wydobywać fragmenty dzięki notacji *slice* (wycinek). W najprostszym przypadku polega ona na podaniu początkowego i końcowego (wyłącznie) indeksu elementów, które chcemy wydobyć, np. `lista[1:4]`.

3.5.3 POĆWICZ SAM

> Utwórz w konsoli Pythona dowolną listę i przećwicz notację *slice*. Sprawdź działanie indeksów pustych i ujemnych, np. `lista[2:]`, `lista[:4]`, `lista[-2]`, `lista[-2:]`. Posortuj dowolną listę malejąco. Wskazówka: wykorzystaj metodę `.sort(reverse=True)`.

[4] W definicji tupli nawiasy są opcjonalne, można więc pisać tak: `tupla = 3, 2, 5, 8`.

3.5.4 ZADANIE

Wypisz ciąg Fibonacciego aż do *n*-ego 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

```
1  #! /usr/bin/env python
2  # -*- coding: UTF-8 -*-
3
4  # ~/python/04_2_fibonacci.py
5
6  def fibonacci(n): #definicja funkcji
7      pwyrazy = (0, 1) #dwa pierwsze wyrazy ciągu zapisane w tupli
8      a, b = pwyrazy #przypisanie wielokrotne, rozpakowanie tupli
9      while a < n:
10         print b
11         a, b = b, a+b #przypisanie wielokrotne
12
13 n = int(raw_input("Podaj numer wyrazu: "))
14 fibonacci(n) #wywołanie funkcji
15 print "" #pusta linia
16 print "=" * 25 #na koniec szlaczek
```

3.5.5 JAK TO DZIAŁA

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

Definicja funkcji w Pythonie polega na użyciu słowa kluczowego `def`, podaniu nazwy funkcji i w nawiasach okrągłych ewentualnej listy argumentów. Definicję kończymy znakiem dwukropka, po którym wpisujemy w następnych liniach, pamiętając o wcięciach, ciało funkcji. Funkcja może, ale nie musi zwracać wartości. Jeżeli chcemy zwrócić jakąś wartość używamy 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.

3.5.6 POĆWICZ SAM

Zmień funkcję `fibonnacci()` tak, aby zwracała wartość *n*-tego wyrazu. Wydrukuj tylko tę wartość w programie.

3.6 Listy, zbiory, moduły i funkcje

3.6.1 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.

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

# ~/python/05_oceny_03.py

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

przedmioty = set(['polski','angielski']) #definicja zbioru
drukuj(przedmioty, "Lista przedmiotów zawiera: ") #wywołanie funkcji z modułu ocenyfun

print "\nAby przerwać wprowadzanie przedmiotów, naciśnij Enter."
while True:
    przedmiot = raw_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 = raw_input("\nZ którego przedmiotu wprowadzisz oceny? ")
        if przedmiot not in przedmioty: #jeżeli przedmiotu nie ma w zbiorze
            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: #mechanizm obsługi błędów
        ocena = int(raw_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}\nMediana: {1:5.2f}\nOdchylenie: {2:5.2f}".format(s,m,o)
```

3.6.2 JAK TO DZIAŁA

Pojęcia: *import, moduł, zbiór, przechwytywanie wyjątków, formatowanie napisów i danych na wyjściu.*

Klauza `from moduł import funkcja` umożliwia wykorzystanie w programie funkcji zdefiniowanych w in-

nych 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 [5], aby można go było poprawnie dołączyć.

Instrukcja `set()` tworzy zbiór, czyli nieuporządkowany zestaw niepowtarzalnych (!) elementów. Instrukcje `if przedmiot in przedmioty` i `if przedmiot not in przedmioty` za pomocą operatorów zawierania `(not) in` sprawdzają, czy podany przedmiot już jest lub nie w zbiorze. Polecenie `przedmioty.add()` pozwala dodawać elementy do zbioru, przy czym jeżeli element jest już w zbiorze, nie zostanie dodany. Polecenie `przedmioty.remove()` usunnie podany jako argument element ze zbioru.

Oceny z wybranego przedmiotu pobieramy w pętli dopóty, dopóki użytkownik nie wprowadzi 0 (zera). Blok `try...except` pozwala przechwycić wyjątki, czyli w naszym przypadku niemożność przekształcenia wprowadzonej 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(...).format(s,m,o)` zastosowano formatowanie drukowanych wartości, do których odwołujemy się w specyfikacji `{0:5.2f}`. Pierwsza cyfra wskazuje, którą wartość z numerowanej od 0 (zera) listy, umieszczonej w funkcji `format()`, wydrukować; np. aby wydrukować drugą wartość, trzeba by użyć kodu `{1:}`.Po dwukropku podajemy szerokość pola przeznaczonego na wydruk, po kropce ilość miejsc po przecinku, symbol *f* oznacza natomiast liczbę zmiennoprzecinkową stałej precyzji.

3.6.3 POĆWICZ SAM

> 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: `print w1 - w2; print w1 | w2; print w1 & w2; print w1 ^ w2`. Przykłady te ilustrują użycie klasycznych operatorów na zbiorach, czyli: różnica (-) , suma (|), przecięcie (część wspólna, &) i elementy unikalne (^).

Funkcje wykorzystywane w programie umieszczamy w osobnym pliku.

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

# ~/python/ocenyfun.py

"""
    Moduł ocenyfun zawiera funkcje wykorzystywane w programie m01_oceny.
"""

import math # zaimportuj moduł matematyczny

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

def srednia(oceny):
    suma = sum(oceny)
    return suma/float(len(oceny))

def mediana(oceny):
```

[5] W przypadku prostych programów zapisuj moduły w tym samym katalogu co program główny.

```
22        oceny.sort();
23        if len(oceny) % 2 == 0: #parzysta ilość ocen
24            half = len(oceny)/2
25            #można tak:
26            #return float(oceny[half-1]+oceny[half]) / 2.0
27            #albo tak:
28            return sum(oceny[half-1:half+1]) / 2.0
29        else: #nieparzysta ilość ocen
30            return oceny[len(oceny)/2]
31
32    def wariancja(oceny,srednia):
33        """
34        Wariancja to suma kwadratów różnicy każdej oceny i średniej podzielona przez ilość ocen:
35        sigma = (o1-s)+(o2-s)+...+(on-s) / n, gdzie:
36        o1, o2, ..., on - kolejne oceny,
37        s - średnia ocen,
38        n - liczba ocen.
39        """
40        sigma = 0.0
41        for ocena in oceny:
42            sigma += (ocena-srednia)**2
43        return sigma/len(oceny)
44
45    def odchylenie(oceny,srednia): #pierwiastek kwadratowy z wariancji
46        w = wariancja(oceny,srednia)
47        return math.sqrt(w)
```

3.6.4 JAK TO DZIAŁA

Pojęcia: *funkcja, argumenty funkcji, zwracanie wartości, moduł.*

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 [6]. 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 elementy z listy, przy czym wyrażenie `half = len(oceny)/2` wylicza nam indeks drugiego ze środkowych elementów.

W funkcja `wariancja()` pętla for odczytuje kolejne oceny i w kodzie `sigma += (ocena-srednia)**2` korzysta z operatorów skróconego dodawania (+=) i potęgowania (**), aby wyliczyć sumę kwadratów różnic kolejnych ocen i średniej.

3.6.5 POĆWICZ SAM

Dopisz funkcję, która wyświetli wszystkie oceny oraz ich odchylenia od wartości średniej.

[6] Przypomnijmy: alternatywna funkcja `sorted(lista)` zwraca uporządkowaną rosnąco kopię listy.

3.7 Słowniki

3.7.1 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.

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

# ~/python/06_slownik_02.py

import os.path # moduł udostępniający funkcję isfile()

print """Podaj dane w formacie:
wyraz obcy: znaczenie1, znaczenie2
Aby zakończyć wprowadzanie danych, podaj 0.
"""

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

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

def zapisz(slownik):
    file1 = open(sFile,"w") #otwieramy plik do zapisu, istniejący plik zostanie nadpisany(!)
    for wobcy in slownik:
        znaczenia=",".join(slownik[wobcy]) # "sklejamy" znaczenia przecinkami w jeden napis
        linia = ":".join([wobcy,znaczenia])# wyraz_obcy:znaczenie1,znaczenie2,...
        print >>file1, linia # zapisujemy w pliku kolejne linie
    file1.close() #zamykamy plik

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

nowy = False #zmienna oznaczająca, że użytkownik uzupełnił lub zmienił słownik
ileWyrazow = otworz(sFile)
print "Wpisów w bazie:", ileWyrazow

#główna pętla programu
while True:
    dane = raw_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 wprowadzonych danych
        if wobcy in slownik:
            print "Wyraz", wobcy, " i jego znaczenia są już w słowniku."
            op = raw_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(",") #podane znaczenia zapisujemy w liście
            znaczenia = map(oczysc, znaczenia) #oczyszczamy elementy listy
            slownik[wobcy] = znaczenia
            nowy = True
    else:
        print "Błędny format!"

if nowy: zapisz(slownik)

print "="*50
print "{0: <15}{1: <40}".format("Wyraz obcy","Znaczenia")
print "="*50
for wobcy in slownik:
    print "{0: <15}{1: <40}".format(wobcy,",".join(slownik[wobcy]))
```

3.7.2 JAK TO DZIAŁA

Pojęcia: *słownik, odczyt i zapis plików, formatowanie napisów.*

Słownik to struktura nieposortowanych danych w formacie klucz:wartość. Kluczami są najczęściej napisy, które wskazują na wartości dowolnego typu, np. inne napisy, liczby, listy, tuple itd. Notacja `oceny = { 'polski':'1,4,2', 'fizyka':'4,3,1' }` utworzy nam słownik ocen z poszczególnych przedmiotów. Aby zapisać coś w słowniku stosujemy notację `oceny['biologia'] = 4,2,5`. Aby odczytać wartość używamy po prostu: `oceny['polski']`.

W programie wykorzystujemy słownik, którego kluczami są obce wyrazy, natomiast wartościami są listy możliwych znaczeń. Przykładowy element naszego słownika wygląda więc tak: `{ 'go':'iść,pojechać' }`. Natomiast ten sam element zapisany w pliku będzie miał format: *wyraz_obcy:znaczenie1,znaczeni2,...*. Dlatego funkcja `otworz()` przekształca format pliku na słownik, a funkcja `zapisz()` słownik na format pliku.

Funkcja `otworz(plik)` sprawdza za pomocą funkcji `isFile(plik)` z modułu *os.path*, czy podany plik istnieje na dysku. Polecenie `open("plik", "r")` otwiera podany plik w trybie do odczytu. Wyrażenie `with ... as sTxt` zapewnia obsługę błędów podczas dostępu do pliku (m. in. zadba o jego zamknięcie) i udostępnia zawartość pliku w zmiennej *sTxt*. Pętla `for line in sTxt:` odczytuje kolejne linie (czyli napisy). Metoda `.split()` zwraca listę zawierającą wydzielone według podanego znaku części ciągu, np.: `t = line.split(":")`. Operacją odwrotną jest "sklejanie" w jeden ciąg elementów listy za pomocą podanego znaku, np. `",".join(slownik[wobcy])`. Metoda `.replace("co","czym")` pozwala zastąpić w ciągu wszystkie wystąpienia *co – czym.*, np.: `znaczenia = t[1].replace("\n","")`.

Funkcja `zapisz()` otrzymuje słownik zawierający dane odczytane z pliku na dysku i dopisane przez użytkownika. W pętli odczytujemy klucze słownika, następnie tworzymy znaczenia oddzielone przecinkami i sklejamy je z wyrazem obcym za pomocą dwukropka. Kolejne linie za pisujemy do pliku `print >>file1, ":".join([wobcy,znaczenia])`, wykorzystując operator >> i nazwę uchwytu pliku (*file1*).

W pętli głównej programu pobrane dane rozbite na wyraz obcy i jego znaczenia zapisujemy w liście *t*. Oczyszczamy pierwszy element tej listy zawierający wyraz obcy (`t[0].strip().lower()`) i sprawdzamy czy nie jest to słowo "koniec", jeśli tak wychodzimy z pętli wprowadzanie danych (`break`). W przeciwnym wypadku sprawdzamy metodą `.count(":")`, czy dwukropek występuje we wprowadzonym ciągu tylko raz. Jeśli nie, format jest nieprawidłowy, w przeciwnym razie, o ile wyrazu nie ma w słowniku lub gdy chcemy go przedefiniować, tworzymy listę znaczeń.

Funkcja `map(funkcja, lista)` do każdego elementu listy stosuje podaną jako argument funkcję (mapowanie funkcji). W naszym przypadku każde znaczenie z listy zostaje oczyszczone przez funkcję `oczysc()`.

Na końcu drukujemy nasz słownik. Specyfikacja `{0: <15}{1: <40}` oznacza, że pierwszy argument umieszczony w funkcji `format()`, drukowany ma być wyrównany do lewej (<) w polu o szerokości 15 znaków, drugi argument, również wyrównany do lewej, w polu o szerokości 40 znaków.

3.7.3 POĆWICZ SAM

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.

3.8 Znam Pythona

3.8.1 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.

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

# ~/python/07_1_trojkat.py

import math

#a, b, c = input("Podaj 3 boki trójkąta (oddzielone przecinkami): ")
# można też tak:
#a, b, c = [int(x) for x in raw_input("Podaj 3 boki trójkąta (oddzielone spacjami): ").split()]
if a+b > c and a+c > b and b+c > a:
    print "Z podanych boków można zbudować trójkąt."
    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!"

    print "Obwód wynosi:", (a+b+c)
    p = 0.5 * (a + b + c) #współczynnik wzoru Herona
    P = math.sqrt(p*(p-a)*(p-b)*(p-c)) #pole ze wzoru Herona
    print "Pole wynosi:", P
else:
    print "Z podanych odcinków nie można utworzyć trójkąta."
```

3.8.2 POĆWICZ SAM

Zmień program tak, aby użytkownik w przypadku podania boków, z których trójkąta zbudować się nie da, mógł spróbować kolejny raz.

3.8.3 ZADANIE

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

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

# ~/python/07_2_szyfr_cezara.py

KLUCZ = 3

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

utxt = raw_input("Podaj ciąg do zaszyfrowania:\n")
stxt = szyfruj(utxt)
print "Ciąg zaszyfrowany:\n", stxt
```

3.8.4 POĆWICZ SAM

Napisz funkcję deszyfrującą `deszyfruj(txt)`. Dodaj do funkcji `szyfruj(), deszyfruj()` drugi parametr w postaci długości klucza podawanej przez użytkownika. Dodaj poprawne szyfrowanie dużych liter, obsługę białych znaków i znaków interpunkcyjnych.

3.9 Nie znam Pythona... jeszcze

3.9.1 ZADANIE

Wypróbuj w konsoli podane przykłady wyrażeń listowych (ang. *list comprehensions*) Pythona:

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

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

3.10 Pojęcia i materiały

3.10.1 Lista pojęć

- *zmienna, wartość, wyrażenie, wejście i wyjście danych, instrukcja warunkowa, komentarz*;

- *pętla, obiekt, metoda, instrukcja warunkowa zagnieżdżona, formatowanie kodu*;
- *iteracja, kod ASCII, lista, inkrementacja, operatory arytmetyczne, logiczne, przypisania, porównania i zawierania*;
- *tupla, lista, metoda, funkcja, zwracanie wartości, pakowanie i rozpakowanie tupli, przypisanie wielokrotne*;
- *import, moduł, zbiór, przechwytywanie wyjątków, formatowanie napisów i danych na wyjściu*;
- *funkcja, argumenty funkcji, zwracanie wartości*;
- *słownik, odczyt i zapis plików*.

3.10.2 Materiały pomocnicze

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

3.10.3 Źródło

- `basic_all.zip`

Metryka

Autorzy Robert Bednarz <rob@lo1.sandomierz.pl>

Utworzony 2014-10-17 o 17:02

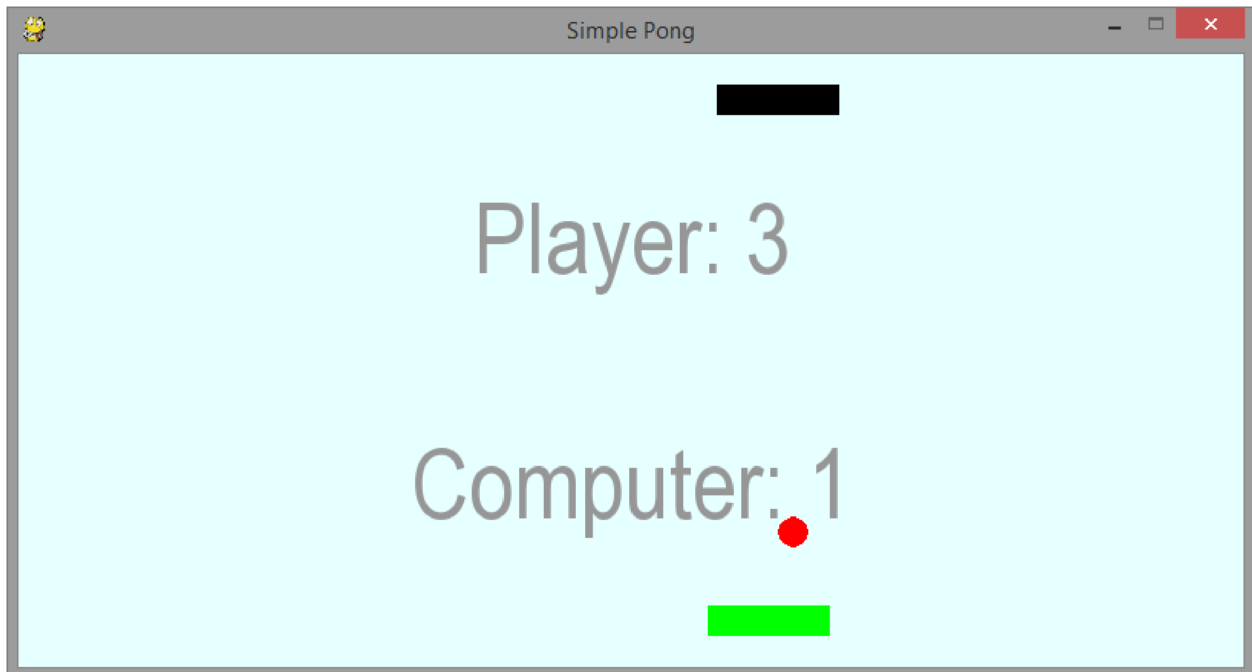
3.11 Gra w Ponga

Klasyczna gra w odbijanie piłeczki zrealizowana z użyciem biblioteki PyGame.

3.11.1 Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szczątkowy kod źródłowy:

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



3.11.2 Okienko gry

Na wstępie w pliku `~/python101/pong/pong.py` otrzymujemy kod który przygotuje okienko 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:
        :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:
```

```
31                drawable.draw_on(self.surface)
32
33            # dopiero w tym miejscu następuje fatyczne rysowanie
34            # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
35            pygame.display.update()
36
37
38    board = Board(800, 400)
39    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 pong/pong.py
```

Mrugnęło? Program się wykonał i zakończył działanie :). Żeby zobaczyć efekt na dłużej, możemy na końcu chwilkę uśpić nasz program:

Jednak zamiast tego, dla lepszej kontroli powinniśmy zadeklarować klasę kontrolera gry, usuńmy kod o linii 37 do końca i dodajmy klasę kontrolera:

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 został zaimportowany gdzieś indziej instrukcją `import pong`, deklaracje klas zostały by wykonane, ale sama gra nie będzie uruchomiona.

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z2
```

3.11.3 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:

Następnie dodajmy klasę samej piłeczki dziedzicząc z `Drawable`:

W przykładzie powyżej wykonaliśmy *dziedziczenie* oraz *przesłanianie* konstruktora, ponieważ rozszerzamy `Drawable` i chcemy zachować efekt działania konstruktora na początku konstruktora `Ball` wywołujemy konstruktorr klasy bazowej:

```
super(Ball, self).__init__(width, height, x, y, color)
```

Teraz musimy naszą piłeczkę zintegrować z resztą gry:

Informacja: Metoda `Board.draw` oczekuje wielu opcjonalnych argumentów, chodź na razie przekazujemy tylko jeden. By zwiększyć czytelność potencjalnie dużej listy argumentów — kto wie co jeszcze dodamy :) — podajemy każdy argument w swojej 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.

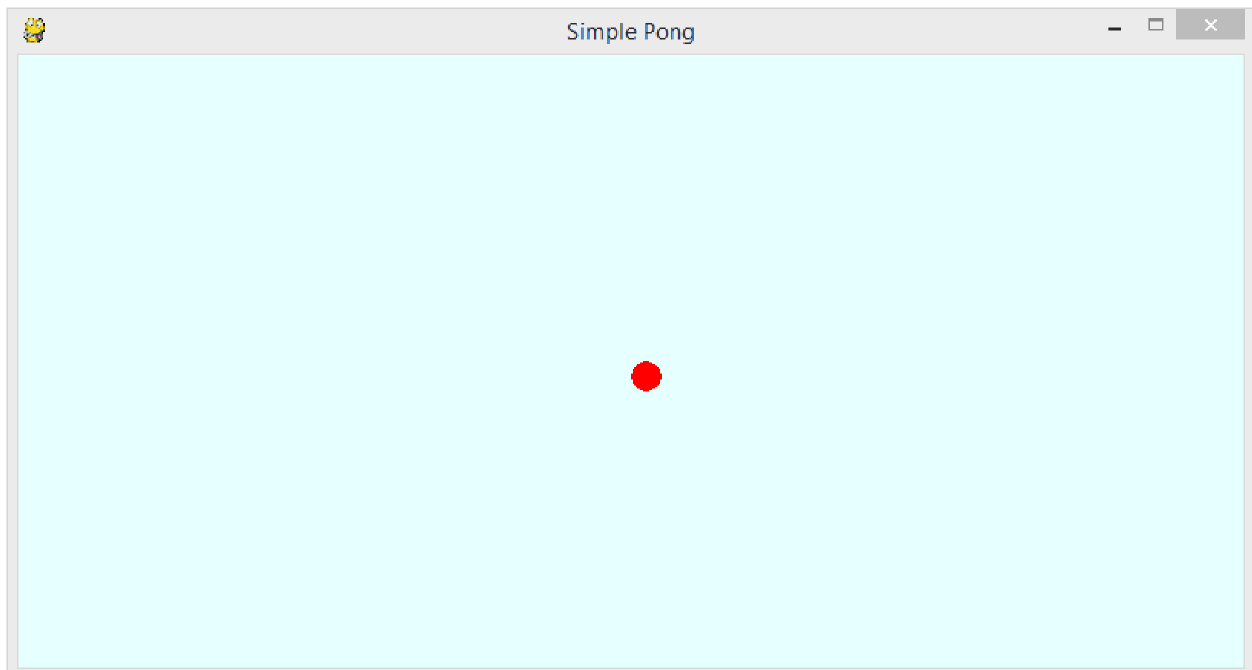
Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z3
```

3.11.4 Odbijanie piłeczki

Uruchommy naszą "grę" ;)

```
~/python101$ python pong/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:

Jeszcze zmodyfikujmy wywołanie metody `move` w naszej pętli głównej:

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 piłka się odbija, uruchamiamy nasz program:

```
~/python101$ python pong/pong.py
```

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z4
```

3.11.5 Odbijamy piłeczkę rakietką

Dodajmy "rakietkę" od przy pomocy której będziemy mogli odbijać piłeczkę. Dodajmy zwykły prostokąt, który będziemy przesuwać przy pomocy myszki.

Informacja: W tym przykładzie zastosowaliśmy operator warunkowy, za jego pomocą ograniczamy 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:

Tak jak w przypadku dodawania piłeczki, rakietkę też trzeba dodać do "gry", dodatkowo musimy ją pokazać piłeczce:

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f pong/z5
```

Informacja: W tym miejscu można się pobawić naszą grą, zmodyfikuj ją według uznania i pochwal się rezultatem z innymi. Jeśli kod przestanie działać, można szybko porzucić zmiany poniższą komendą.

```
~/python101$ git reset --hard
```

3.11.6 Gramy przeciwko komputerowi

Dodajemy przeciwnika, nasz przeciwnik będzie mistrzem, będzie dokładnie śledził piłeczkę i zawsze starał się utrzymać rakietkę gotową do odbicia piłeczki.

Tak jak w przypadku piłeczki i rakietki dodajemy nasze `Ai` do gry, a wraz nią wraz dodajemy drugą rakietkę. Dwie rakietki ustawiamy na przeciwległych brzegach planszy.

Trzeba pamiętać by pokazać drugą rakietkę piłeczce, tak by mogła się od niej odbijać.

3.11.7 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.

Tradycyjnie dodajemy instancję nowej klasy do gry:

3.11.8 Zadania dodatkowe i rzeczy które można poprawić

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 powinen 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.

3.11.9 Słowniczek

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

Metryka

Autorzy Janusz Skonieczny <js@bravelabs.pl>, Robert Bednarz

Utworzony 2014-10-06 o 15:12

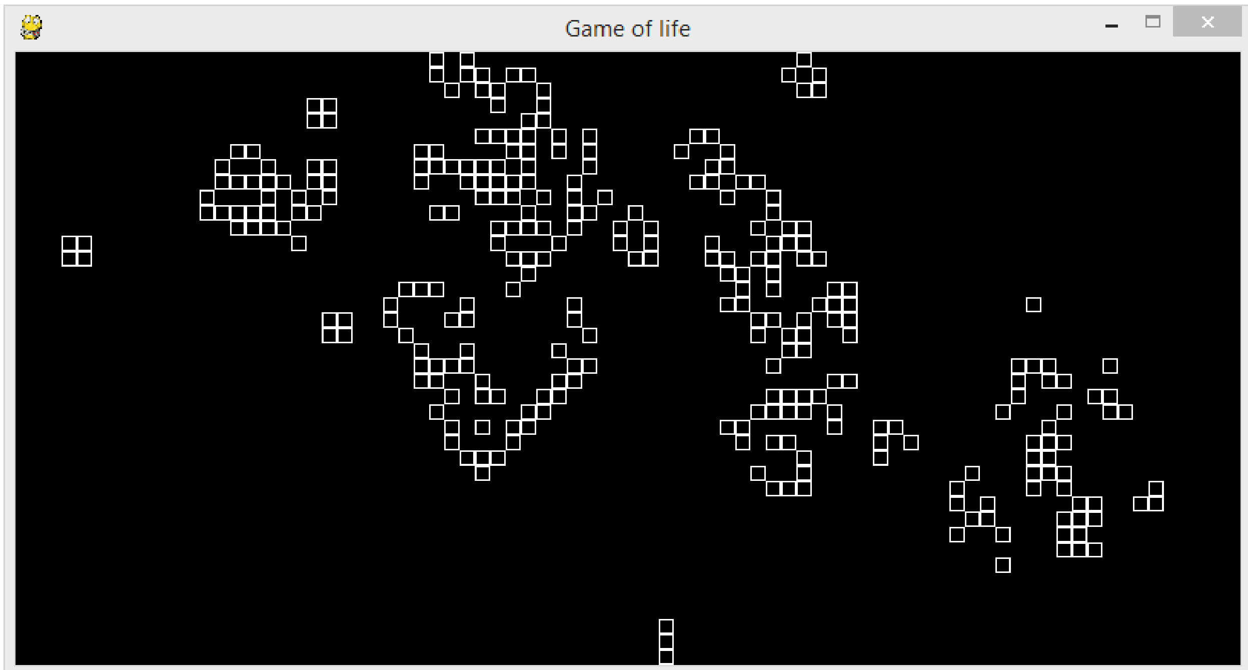
3.12 Gra w życie Conwaya

Gra w życie zrealizowana z użyciem biblioteki PyGame.

3.12.1 Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szczątkowy kod źródłowy:

```
~/python101$ git checkout -f life/z1
```



3.12.2 Okienko gry

Na wstępie w pliku ~/python101/games/life.py otrzymujemy kod który przygotuje okienko naszej gry:

Informacja: Ten przykład zakłada wcześniejsze zrealizowanie przykładu: *Gra w Ponga*, 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

```

```
26          :param args: lista obiektów do narysowania
27          """
28          background = (0, 0, 0)
29          self.surface.fill(background)
30          for drawable in args:
31              drawable.draw_on(self.surface)
32
33          # dopiero w tym miejscu następuje fatyczne rysowanie
34          # w oknie gry, wcześniej tylko ustalaliśmy co i jak ma zostać narysowane
35          pygame.display.update()
36
37
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
56      def run(self):
57          """
58          Główna pętla gry
59          """
60          while not self.handle_events():
61              # działaj w pętli do momentu otrzymania sygnału do wyjścia
62              self.board.draw()
63              self.fps_clock.tick(15)
64
65      def handle_events(self):
66          """
67          Obsługa zdarzeń systemowych, tutaj zinterpretujemy np. ruchy myszką
68
69          :return True jeżeli pygame przekazał zdarzenie wyjścia z gry
70          """
71          for event in pygame.event.get():
72              if event.type == pygame.locals.QUIT:
73                  pygame.quit()
74                  return True
75
76
77  # Ta część powinna być zawsze na końcu modułu (ten plik jest modułem)
78  # chcemy uruchomić naszą grę dopiero po tym jak wszystkie klasy zostaną zadeklarowane
79  if __name__ == "__main__":
80      game = GameOfLife(80, 40)
81      game.run()
```

W powyższym kodzie mamy podstawy potrzebne do uruchomienia gry:

```
~/python101$ python games/life.py
```

3.12.3 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*.

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.

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],
4     [DEAD, DEAD, DEAD, DEAD],
5 ]
```

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.

```
1 def reset_generation(self)
2     generation = []
3     for x in xrange(self.width):
4         column = []
5         for y in xrange(self.height)
6             column.append(DEAD)
7         generation.append(column)
8     return generation
```

Powyżej wykorzystaliśmy 2 pętle (jedna zagnieżdżona w drugiej) oraz funkcję `xrange` 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:

```
1 [DEAD for y in xrange(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`

```
1 [nowa_kolumna for x in xrange(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:

```
1 [[DEAD for y in xrange(self.height)] for x in xrange(self.width)]
```

3.12.4 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.

Następnie dodajmy metodę `draw_on` która będzie rysować żywe komórki na planszy. Tą metodę wywołamy w metodzie `GameOfLife.draw`.

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.

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.

3.12.5 Dodajemy populację do kontrolera gry

Czas by rozwinąć nasz kontroler gry, klasę `GameOfLife` o instancję klasy `Population`

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z2
```

3.12.6 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:

Następnie przygotujmy funkcję która będzie tworzyć nową populację

Jeszcze ostatnie modyfikacje kontrolera gry tak by komórki zaczęły żyć po wciśnięciu klawisza enter.

Gotowy kod możemy wyciągnąć komendą:

```
~/python101$ git checkout -f life/z3
```

3.12.7 Zadania dodatkowe i rzeczy które można poprawić

1. TODO

3.12.8 Słowniczek

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).

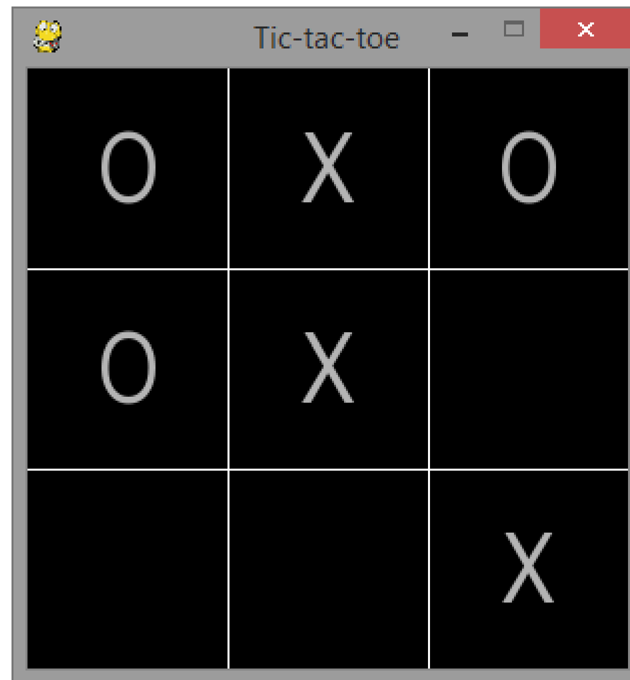
Metryka

Autorzy Janusz Skonieczny <js@bravelabs.pl>, Robert Bednarz

Utworzony 2014-10-03 o 15:29

3.13 Gra w Kółko i Krzyżyk

Klasyczna gra w kółko i krzyżyk zrealizowana przy pomocy PyGame.



3.13.1 Przygotowanie

Do rozpoczęcia pracy z przykładem pobieramy szczątkowy kod źródłowy:

```
~/python101$ git checkout -f tic_tac_toe/z1
```

3.13.2 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: *Gra w życie Conwaya*, 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 rozbicia i uzupełnienia opisów

```
1  # coding=utf-8
2  # Copyright 2014 Janusz Skonieczny
3
4  """
5  Gra w kółko i krzyżyk
6  """
7
8  import pygame
9  import pygame.locals
10 import logging
11
12 # Konfiguracja modułu logowania, element dla zaawansowanych
13 logging_format = '%(asctime)s %(levelname)-7s | %(module)s.%(funcName)s - %(message)s'
```

```
logging.basicConfig(level=logging.DEBUG, format=logging_format, datefmt='%H:%M:%S')
logging.getLogger().setLevel(logging.INFO)


class Board(object):
    """
    Plansza do gry. Odpowiada za rysowanie okna gry.
    """

    def __init__(self, width):
        """
        Konstruktor planszy do gry. Przygotowuje okienko gry.

        :param width: szerokość w pikselach
        """
        self.surface = pygame.display.set_mode((width, width), 0, 32)
        pygame.display.set_caption('Tic-tac-toe')

        # 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, 48)

        # tablica znaczników 3x3 w formie listy
        self.markers = [None] * 9

    def draw(self, *args):
        """
        Rysuje okno gry

        :param args: lista obiektów do narysowania
        """
        background = (0, 0, 0)
        self.surface.fill(background)
        self.draw_net()
        self.draw_markers()
        self.draw_score()
        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()

    def draw_net(self):
        """
        Rysuje siatkę linii na planszy
        """
        color = (255, 255, 255)
        width = self.surface.get_width()
        for i in range(1, 3):
            pos = width / 3 * i
            # linia pozioma
            pygame.draw.line(self.surface, color, (0, pos), (width, pos), 1)
            # linia pionowa
            pygame.draw.line(self.surface, color, (pos, 0), (pos, width), 1)

    def player_move(self, x, y):
```

```
72          """
73          Ustawia na planszy znacznik gracza X na podstawie współrzędnych w pikselach
74          """
75          cell_size = self.surface.get_width() / 3
76          x /= cell_size
77          y /= cell_size
78          self.markers[x + y * 3] = player_marker(True)
79
80      def draw_markers(self):
81          """
82          Rysuje znaczniki graczy
83          """
84          box_side = self.surface.get_width() / 3
85          for x in range(3):
86              for y in range(3):
87                  marker = self.markers[x + y * 3]
88                  if not marker:
89                      continue
90                  # zmieniamy współrzędne znacznika
91                  # na współrzędne w pikselach dla centrum pola
92                  center_x = x * box_side + box_side / 2
93                  center_y = y * box_side + box_side / 2
94
95                  self.draw_text(self.surface, marker, (center_x, center_y))
96
97      def draw_text(self, surface,  text, center, color=(180, 180, 180)):
98          """
99          Rysuje wskazany tekst we wskazanym miejscu
100         """
101         text = self.font.render(text, True, color)
102         rect = text.get_rect()
103         rect.center = center
104         surface.blit(text, rect)
105
106     def draw_score(self):
107         """
108         Sprawdza czy gra została skończona i rysuje właściwy komunikat
109         """
110         if check_win(self.markers, True):
111             score = u"Wygrałeś(aś)"
112         elif check_win(self.markers, True):
113             score = u"Przegrałeś(aś)"
114         elif None not in self.markers:
115             score = u"Remis!"
116         else:
117             return
118
119         i = self.surface.get_width() / 2
120         self.draw_text(self.surface, score, center=(i, i), color=(255, 26, 26))
121
122
123 class TicTacToeGame(object):
124     """
125     Łączy wszystkie elementy gry w całość.
126     """
127
128     def __init__(self, width, ai_turn=False):
129         """
```

```
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
```

```
188                 return
189             logging.debug("Plansza: %s" % self.board.markers)
190             move = self.next_move(self.board.markers)
191             self.board.markers[move] = player_marker(False)
192 
193     @classmethod
194     def next_move(cls, markers):
195         """
196         Wybierz następny ruch komputera na podstawie wskazanej planszy
197         :param markers: plansza gry
198         :return: index tablicy jednowymiarowe w której należy ustawić znacznik kółka
199         """
200         # pobierz dostępne ruchy wraz z oceną
201         moves = cls.score_moves(markers, False)
202         # wybierz najlepiej oceniony ruch
203         score, move = max(moves, key=lambda m: m[0])
204         logging.info("Dostępne ruchy: %s", moves)
205         logging.info("Wybrany ruch: %s %s", move, score)
206         return move
207 
208     @classmethod
209     def score_moves(cls, markers, x_player):
210         """
211         Ocenia rekurencyjne możliwe ruchy
212 
213         Jeśli ruch jest zwycięstwem otrzymuje +1, jeśli przegraną -1
214         lub 0 jeśli nie nie ma zwycięscy. Dla ruchów bez zwycięscy rekreacyjnie
215         analizowane są kolejne ruchy a suma ich punktów jest wynikiem aktualnego
216         ruchu.
217 
218         :param markers: plansza na podstawie której analizowane są następne ruchy
219         :param x_player: True jeśli ruch dotyczy gracza X, False dla gracza O
220         """
221         # wybieramy wszystkie możliwe ruchy na podstawie wolnych pól
222         available_moves = (i for i, m in enumerate(markers) if m is None)
223         for move in available_moves:
224             from copy import copy
225             # tworzymy kopię planszy która na której testowo zostanie
226             # wykonany ruch w celu jego późniejszej oceny
227             proposal = copy(markers)
228             proposal[move] = player_marker(x_player)
229 
230             # sprawdzamy czy ktoś wygrywa gracz którego ruch testujemy
231             if check_win(proposal, x_player):
232                 # dodajemy punkty jeśli to my wygrywamy
233                 # czyli nie x_player
234                 score = -1 if x_player else 1
235                 yield score, move
236                 continue
237 
238             # ruch jest neutralny,
239             # sprawdzamy rekurencyjne kolejne ruchy zmieniając gracza
240             next_moves = list(cls.score_moves(proposal, not x_player))
241             if not next_moves:
242                 yield 0, move
243                 continue
244 
245             # 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/life.py
```

3.14 Quiz – aplikacja internetowa

Realizacja aplikacji internetowej Quiz w oparciu o mikro-framework Flask.

3.14.1 Struktura katalogów

W katalogu użytkownika tworzymy nowy katalog dla aplikacji `quiz`, a w nim plik główny `quiz.py`:

```
~$ mkdir quiz; cd quiz; touch quiz.py
```

Aplikacja na przykładzie quizu – użytkownik zaznacza w formularzu poprawne odpowiedzi na pytania i otrzymuje ocenę – ma pokazać podstawową strukturę frameworka Flask.

3.14.2 Szkielet aplikacji

Utworzenie minimalnej aplikacji Flask pozwoli na uruchomienie serwera deweloperskiego, umożliwiającego wygodne rozwijanie kodu. W pliku `quiz.py` wpisujemy:

```
1  # -*- coding: utf-8 -*-
2  # todo/todo.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:

```
~/python101/modul3/zadanie1$ python todo.py
```

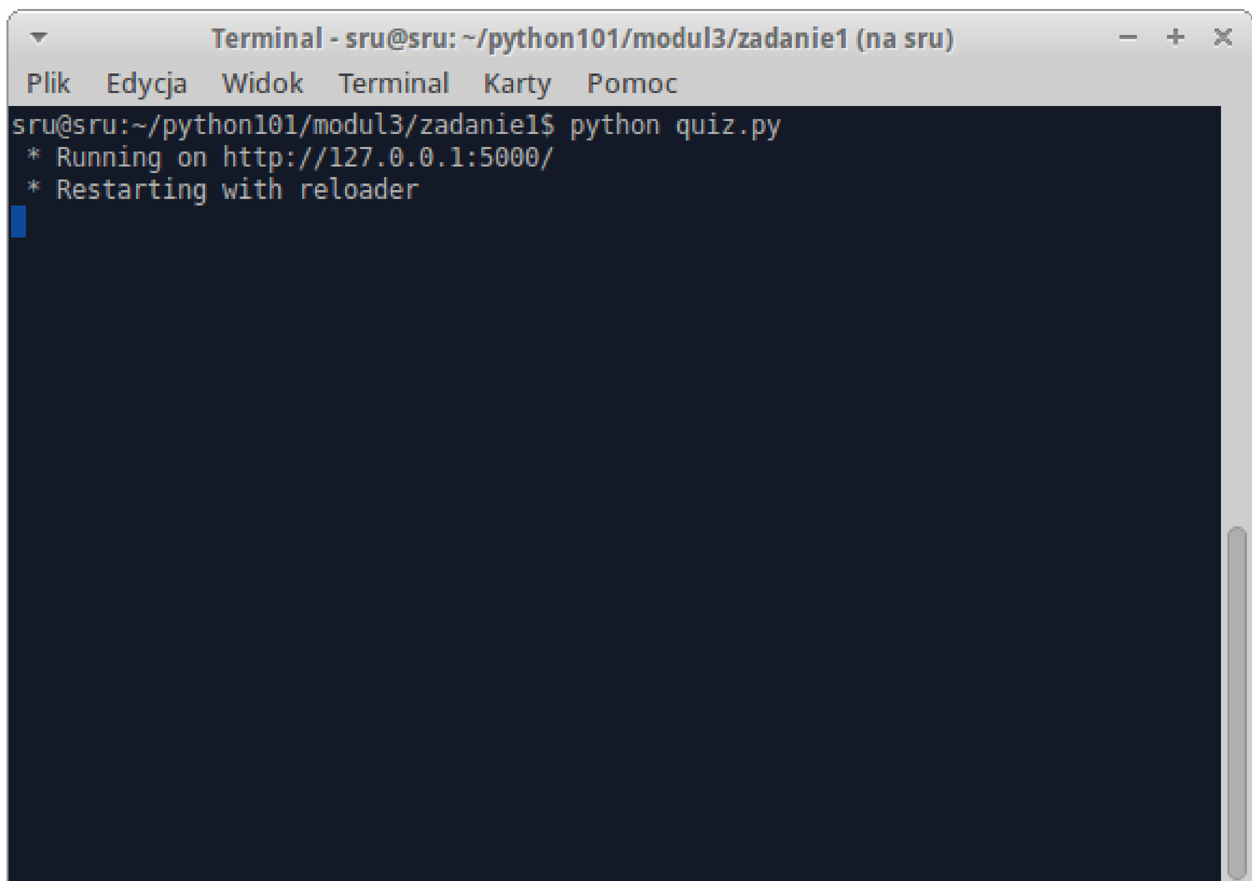
Domyślnie serwer uruchamia się pod adresem *http://127.0.0.1:5000*. Po wpisaniu adresu do przeglądarki internetowej otrzymamy stronę z błędem HTTP 404, co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego zachowania (widoku) dla tego adresu. W uproszczeniu możemy widok utożsamiać z pojedynczą stroną w ramach aplikacji internetowej.

3.14.3 Definiowanie widoków – strona główna

Widoki to funkcje Pythona powiązane z określonymi adresami URL za pomocą tzw. dekoratorów. Widoki pozwalają nam obsługiwać żądania GET i POST, a także, przy wykorzystaniu szablonów, generować i zwracać żądane przez klienta strony WWW. W szablonach oprócz znaczników HTML możemy umieszczać różne dane. Flask renderuje (łączy) kod HTML z danymi i odsyła do przeglądarki.

W pliku `todo.py` umieścimy funkcję `index()`, widok naszej strony głównej:

```
1  # -*- coding: utf-8 -*-
2  # todo/todo.py
3
4  from flask import Flask
```


Terminal - sru@sru: ~/python101/modul3/zadanie1 (na sru)
Plik Edycja Widok Terminal Karty Pomoc
sru@sru:~/python101/modul3/zadanie1$ python quiz.py
* Running on http://127.0.0.1:5000/
* Restarting with reloader

```
5  from flask import render_template
6
7  app = Flask(__name__)
8
9  # dekorator laczacy adres glowny z widokiem index
10 @app.route('/')
11 def index():
12     # gdybyśmy chcieli wyświetlić prosty tekst, użyjemy funkcji poniżej
13     #return 'Hello, SWOI'
14     # zwracamy wyrenderowany szablon index.html:
15     return render_templpate('index.html')
16
17 if __name__ == '__main__':
18     app.run(debug=True)
```

Zauważmy, że widok `index()` za pomocą dekoratora `@app.route('/')` związaliśmy z adresem głównym (/). Dalej w katalogu `quiz` tworzymy podkatalog `templates`, a w nim szablon `index.html`, wydajemy polecenia w terminalu:

```
~/python101/modul3/zadanie1$ mkdir templates; cd templates; touch index.html
```

Do pliku `index.html` wstawiamy przykładowy kod HTML:

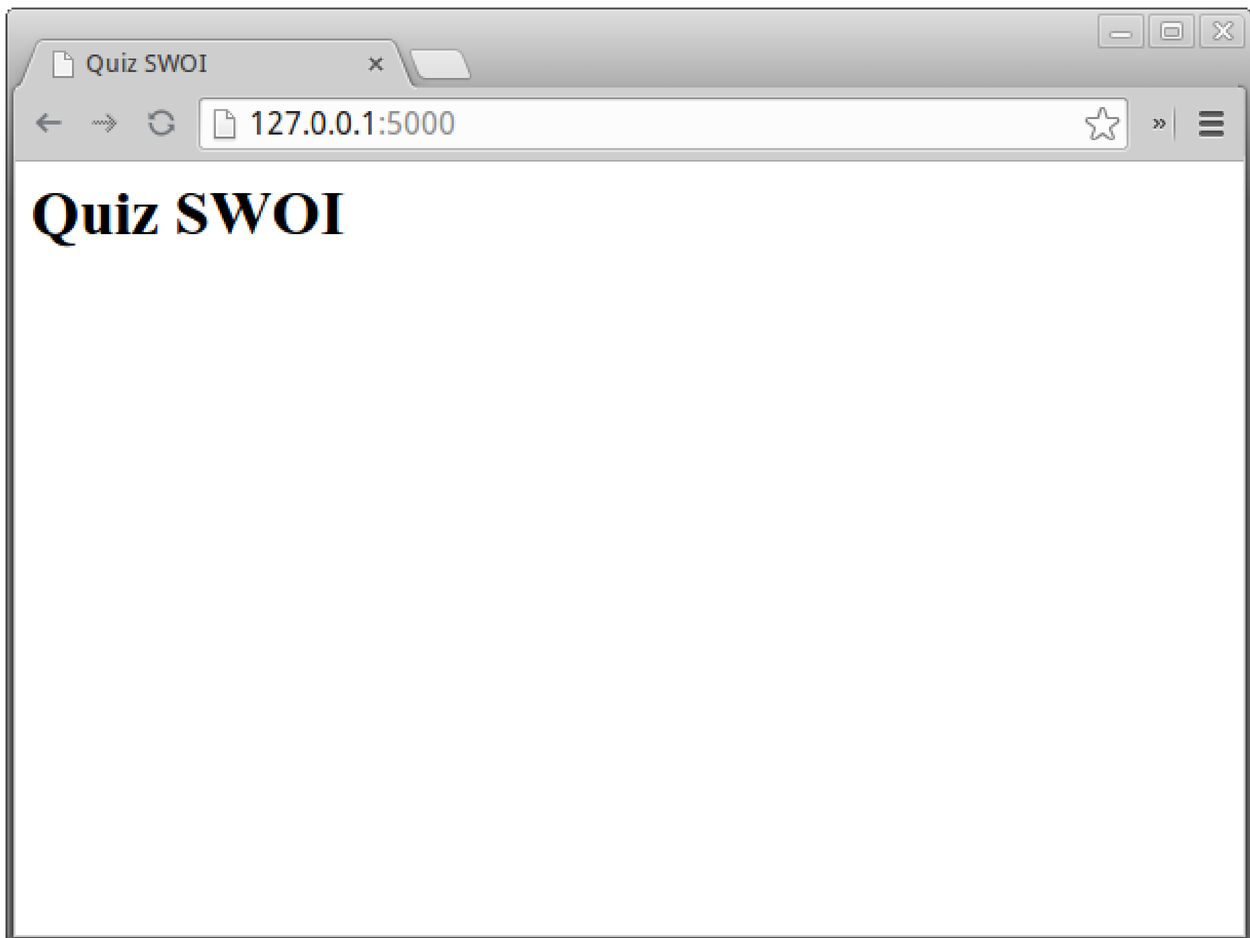
```
1 <!-- quiz/templates/index.html -->
2 <html>
3     <head>
4         <title>Quiz SWOi</title>
5     </head>
6 <body>
7     <h1>Witaj na serwerze!</h1>
8 </body>
9 </html>
```

Po odwiedzeniu adresu *http://127.0.0.1:5000*, otrzymamy stronę HTML.

3.14.4 Pokaż dane aplikacji – pytania i odpowiedzi

Dane naszej aplikacji, a więc pytania i odpowiedzi, umieścimy w liście `QUESTIONS` w postaci słowników zawierających: treść pytania, listę możliwych odpowiedzi oraz poprawną odpowiedź. W pliku `quiz.py` wstawiamy listę pytań, aktualizujemy widok `index()`, przekazując do szablonu pytania w zmiennej `questions`.

```
1  # -*- coding: utf-8 -*-
2  # quiz/quiz.py
3
4  from flask import Flask
5  from flask import render_template
6
7  app = Flask(__name__)
8
9  # konfiguracja aplikacji, sekret potrzebny do obsługi sesji HTTP wymaganej przez funkcję flash
10 app.config.update(dict(
11     SECRET_KEY='bardzosekretnawartosc',
12 ))
13
14 # lista pytan
15 QUESTIONS = [
16     {
```


Quiz SWOI
127.0.0.1:5000
Quiz SWOI

```
        'question': u'Stolica Hiszpani, to:',# pytanie
        'answers': [u'Madryt', u'Warszawa', u'Barcelona'], # mozliwe odpowiedzi
        'correct_answer': u'Madryt', # poprawna odpowiedz
    },
    {
        'question': u'Objętość sześcianu o boku 6 cm, wynosi:', # pytanie
        'answers': [u'36', u'216', u'18'], # mozliwe odpowiedzi
        'correct_answer': u'216', # poprawna odpowiedz
    },
    {
        'question': u'Symbol pierwiastka Helu, to:', # pytanie
        'answers': [u'Fe', u'H', u'He'], # mozlowe odpowiedzi
        'correct_answer': u'He', # poprawna odpowiedz
    }
]


@app.route('/')
def index():
    # do templatki index.html przekazujemy liste pytan jako zmienna questions
    return render_template('index.html', questions=QUESTIONS)


if __name__ == '__main__':
    app.run(debug=True)
```

Dodatkowo dodaliśmy konfigurację aplikacji, ustalając sekretny klucz, który przyda nam się w późniejszej części. Aktualizujemy szablon `index.html`, aby wyświetlić listę pytań w postaci formularza HTML.

```
<!-- quiz/templates/index.html -->
<html>
    <head>
        <title>Quiz SWOI</title>
    </head>
    <body>
        <h1>Quiz SWOI</h1>

        <!-- formularz z quizem -->
        <form method="POST">
            <!-- iterujemy po liscie pytan -->
            {% for entry in questions %}
                <p>
                    <!-- dla kazdego pytania wypisujemy pytanie (pole question) -->
                    {{ entry.question }}
                    <br>
                    <!-- zapamietujemy numer pytania liczac od zera -->
                    {% set question_number = loop.index0 %}
                    <!-- iterujemy po mozliwych odpowiedziach dla danego pytania -->
                    {% for answer in entry.answers %}
                        <label>
                            <!-- odpowiedzi zamieniamy na radio buttony -->
                            <input type="radio" value="{{ answer }}" name="{{ question_number }}">
                            {{ answer }}
                        </label>
                        <br>
                    {% endfor %}
                </p>
            {% endfor %}
```

```

            <!-- button wysylajacy wypelniony formularz -->
            <button type="submit">Sprawdź odpowiedzi</button>
        </form>

    </body>
</html>
```

Wewnątrz szablonu przeglądamy pytania zawarte w zmiennej questions za pomocą instrukcji `{% for entry in questions %}`, tworzymy formularz HTML składający się z treści pytania `{{ entry.question }}` i listy odpowiedzi (kolejna pętla `{% for answer in entry.answers %}`) w postaci grupy opcji nazywanych dla odróżnienia kolejnymi indeksami pytań liczonymi od 0 (`{% set question_number = loop.index0 %}`).

W efekcie powinniśmy otrzymać następującą stronę internetową:

3.14.5 Oceniamy odpowiedzi

Mechanizm sprawdzana liczby poprawnych odpowiedzi umieścimy w pliku `quiz.py`, modyfikując widok `index()`:

```
# uzupelniamy importy
from flask import request
from flask import redirect, url_for
```

```python
from flask import flash


# rozszerzamy widok
@app.route('/', methods=['GET', 'POST'])
def index():
    # jezeli zadanie jest typu POST, to znaczy, ze ktos przeslal odpowiedzi do sprawdzenia
    if request.method == 'POST':
        score = 0 # liczba poprawnych odpowiedzi
        answers = request.form # zapamietujemy slownik z odpowiedziami
        # sprawdzamy odpowiedzi:
        for question_number, user_answer in answers.items():
            # pobieramy z listy informacje o poprawnej odpowiedzi
            correct_answer = QUESTIONS[int(question_number)]['correct_answer']
            if user_answer == correct_answer: # porownujemy odpowiedzi
                score += 1 # zwiekszamy wynik
        # przygotowujemy informacje o wyniku
        flash(u'Liczba poprawnych odpowiedzi, to: {0}'.format(score))
        # po POST przekierowujemy na strone glowna
        return redirect(url_for('index'))

    # jezeli zadanie jest typu GET, renderujemy index.html
    return render_template('index.html', questions=QUESTIONS)
```

W szablonie `index.html` po znaczniku `<h1>` wstawiamy instrukcje wyświetlające wynik:

```html
<!-- umieszczamy informacje ustawiona za pomoca funkcji flash -->
<p>
    {% for message in get_flashed_messages() %}
        <strong class="success">{{ message }}</strong>
    {% endfor %}
</p>
```

Jak to działa

Uzupełniliśmy dekorator app.route, aby obsługiwał zarówno żądania *GET* (wejście na stronę główną po wpisaniu adresu => pokazujemy pytania), jak i *POST* (przesłanie odpowiedzi z formularza pytań => oceniamy odpowiedzi).

W widoku `index()` dodaliśmy instrukcję warunkową `if request.method == 'POST':`, która wykrywa żądania POST i wykonuje blok kodu zliczający poprawne odpowiedzi. Zliczanie wykonywane jest w pętli `for question_number, user_answer in answers.items()`.

W tym celu iterujemy po przesłanych odpowiedziach i sprawdzamy, czy nadesłana odpowiedź jest zgodna z tą, którą przechowujemy w polu `correct_answer` konkretnego pytania. Dzięki temu, że w szablonie dodaliśmy do każdego pytania jego numer (zmienna `question_number`), to możemy teraz po tym numerze odwołać się do konkretnego pytania na naszej liście.

Jeżeli nadesłana odpowiedź jest zgodna z tym, co mamy zapisane w pytaniu, to naliczamy punkt. Informacje o wyniku przekazujemy do użytkownika za pomocą funkcji `flash`, która korzysta z sesji HTTP (właśnie dlatego musieliśmy ustalić `SECRET_KEY` dla naszej aplikacji).

W efekcie otrzymujemy aplikację Quiz.

Materiały

1. Strona projektu Flask http://flask.pocoo.org/

2. Co to jest framework? http://pl.wikipedia.org/wiki/Framework
3. Co nieco o HTTP i żądaniach GET i POST http://pl.wikipedia.org/wiki/Http

Pojęcia

Aplikacja program komputerowy.

Framework zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji.

GET typ żądania HTTP, służący do pobierania zasobów z serwera WWW.

HTML język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.

HTTP protokół przesyłania dokumentów WWW.

POST typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW.

Serwer deweloperski serwer używany w czasie prac nad oprogramowaniem.

Serwer WWW serwer obsługujący protokół HTTP.

Templatka szablon strony WWW wykorzystywany przez Flask do renderowania widoków.

URL ustandaryzowany format adresowania zasobów w internecie (przykład: http://pl.wikipedia.org/wiki/Uniform_Resource_Locator).

Widok fragment danych, który jest reprezentowany użytkownikowi.

Źródła

- `quiz_all.zip`

Metryka

Autorzy Tomasz Nowacki, Robert Bednarz, Janusz Skonieczny

Utworzony 2014-10-17 o 17:02

3.15 ToDo – aplikacja internetowa

Realizacja prostej listy ToDo (lista zadań do zrobienia) jako aplikacji internetowej, z wykorzystaniem Pythona i frameworka Flask w wersji 0.10.1.

3.15.1 Katalog, plik i przeznaczenie aplikacji

Zaczynamy od utworzenia katalogu projektu ToDo w katalogu domowym użytkownika, a w nim pliku `todo.py`:

```
~ $ mkdir todo; cd todo; touch todo.py
```

Aplikacja ma pozwalać na dodawanie z określoną datą, przeglądanie i oznaczanie jako wykonane różnych zadań, które zapisywane będą w bazie danych SQLite.

3.15.2 Szkielet aplikacji

Utworzenie minimalnej aplikacji Flask pozwoli na uruchomienie serwera deweloperskiego, umożliwiającego wygodne rozwijanie kodu. W pliku `todo.py` wpisujemy:

```
# -*- coding: utf-8 -*-
# todo/todo.py

from flask import Flask

app = Flask(__name__)

if __name__ == '__main__':
    app.run(debug=True)
```

Serwer uruchamiamy komendą: `python todo.py`

Domyślnie serwer uruchamia się pod adresem *127.0.0.1:5000*. Po wpisaniu adresu do przeglądarki internetowej otrzymamy stronę z błędem HTTP 404, co wynika z faktu, że nasza aplikacja nie ma jeszcze zdefiniowanego żadnego zachowania (widoku) dla tego adresu. *Widok* to funkcja obsługująca wywołania powiązanego z nim adresu. Widok (funkcja) zwraca najczęściej użytkownikowi wyrenderowaną z szablonu stronę internetową.

Not Found

The requested URL was not found on the server. If you entered the URL manually please check your spelling and try again.

3.15.3 Definiowanie widoków

W pliku `todo.py` umieścimy funkcję `index()`, domyślny widok naszej strony:

```
# -*- coding: utf-8 -*-
# todo/todo.py

from flask import Flask
app = Flask(__name__)

# dekorator laczacy adres glowny z widokiem index
@app.route('/')
def index():
    return 'Witaj na moim serwerze!'

if __name__ == '__main__':
    app.run(debug=True)
```

Widok `index()` za pomocą dekoratora związaliśmy z adresem głównym (/). Po odświeżeniu adresu *127.0.0.1:5000* zamiast błędu powinniśmy zobaczyć napis: "Witaj na moim serwerze!"

3.15.4 Model bazy danych

W katalogu aplikacji tworzymy plik `schema.sql`, który zawiera opis struktury tabeli z zadaniami. Do tabeli wprowadzimy przykładowe dane.

```
-- todo/schema.sql

-- tabela z zadaniami
drop table if exists entries;
create table entries (
    id integer primary key autoincrement, -- unikalny indentyfikator
    title text not null, -- opis zadania do wykonania
    is_done boolean not null, -- informacja czy zadania zostalo juz wykonane
    created_at datetime not null -- data dodania zadania
);

-- pierwsze dane
insert into entries (id, title, is_done, created_at)
values (null, 'Wyrzucić śmieci', 0, datetime(current_timestamp));
insert into entries (id, title, is_done, created_at)
values (null, 'Nakarmić psa', 0, datetime(current_timestamp));
```

Tworzymy bazę danych w pliku `db.sqlite`, łączymy się z nią i próbujemy wyświetlić dane, które powinny były zostać zapisane w tabeli *entries*:

```
sqlite3 db.sqlite < schema.sql
sqlite3 db.sqlite
select \* from entries;
```

```
SQLite version 3.8.2 2013-12-06 14:53:30
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select * from entries;
1|Wyrzucić śmieci|0|2014-08-13 10:39:58
2|Nakarmić psa|0|2014-08-13 10:39:58
sqlite>
```

Pracę z bazą kończymy poleceniem `.quit`.

3.15.5 Połączenie z bazą danych

Bazę danych już mamy, teraz pora napisać funkcje umożiwiające łączenie się z nią z poziomu naszej aplikacji. W pliku `todo.py` dodajemy:

```python
# -*- coding: utf-8 -*-
# todo/todo.py

# importujemy biblioteki potrzebne do nawiązania połączenia z baza
import os
import sqlite3

from flask import Flask, g

app = Flask(__name__)

# konfiguracja aplikacji
```

```
13  app.config.update(dict(
14      # nieznany nikomu sekret dla mechanizmu sesji
15      SECRET_KEY = 'bardzosekretnawartosc',
16      # polozenie naszej bazy
17      DATABASE = os.path.join(app.root_path, 'db.sqlite'),
18      # nazwa aplikacji
19      SITE_NAME = 'Moja lista ToDo'
20  ))
21
22  def connect_db():
23      """Nawiazywanie połaczenia z bazą danych określoną w konfiguracji."""
24      """http://flask.pocoo.org/docs/0.10/patterns/sqlite3/"""
25      rv = sqlite3.connect(app.config['DATABASE'])
26      rv.row_factory = sqlite3.Row
27      return rv
28
29  def get_db():
30      """Funkcja pomocnicza, ktora tworzy połączenia z bazą przy pierwszym
31      wywołaniu i umieszcza ja w kontekście aplikacji (obiekt g). W kolejnych
32      wywołaniach zwraca połączenie z kontekstu."""
33      if not hasattr(g, 'db'):
34          g.db = connect_db() # jezeli kontekst nie zawiera informacji o polaczeniu to je tworzymy
35      return g.db # zwracamy polaczenie z baza
36
37  # dekorator wykonujacy funkcje po wyslaniu odpowiedzi do klienta
38  @app.teardown_request
39  def close_db(error):
40      """Zamykanie polaczenia z baza."""
41      if hasattr(g, 'db'):
42          g.db.close()
43
44  # dekorator laczacy adres glowny z widokiem index
45  @app.route('/')
46  def index():
47      return 'Hello, SWOI'
48
49  if __name__ == '__main__':
50      app.run(debug=True)
```

Dodaliśmy sekretny klucz zabezpieczający mechanizm sesji, ustawiliśmy ścieżkę do pliku bazy danych w katalogu aplikacji (stąd użycie funkcji `app.root_path`) oraz nazwę aplikacji. Utworzyliśmy trzy funkcje odpowiedzialne za nawiązywanie (`connect_db`, `get_db`) i kończenie (`close_db`) połączenia z bazą danych.

3.15.6 Pobieranie i wyświetlanie danych

Wyświetlanie danych umożliwia wbudowany we Flask system szablonów (templatek), czyli mechanizm renderowania kodu HTML na podstawie plików zawierających instrukcje wstawiające wybrane dane z aplikacji oraz znaczniki HTML. Modyfikujemy funkcję `index()` w pliku `todo.py`:

```
# dodajemy nowe importy do pozostalych
from flask import render_template

# dekorator laczacy adres glowny z widokiem index
@app.route('/')
def index():
    db = get_db() # laczymy sie z baza
    # pobieramy wszystkie wpisy z bazy:
```

```
    cur = db.execute('select id, title, is_done, created_at from entries order by created_at desc;')
    entries = cur.fetchall()
    # renderujemy tempaltke i zwracamy ja do klienta
    return render_template('show_entries.html', entries=entries)
```

W widoku `index()` pobieramy obiekt bazy danych (*db*) i wykonujemy zapytanie (*select...*), by wyciągnąć wszystkie zapisane zadania. Na koniec renderujemy szablon przekazując do niego pobrane zadania (`entries`). Szablon, czyli plik `show_entries.html`, umieszczamy w podkatalogu `templates` aplikacji. Poniżej jego zawartość:

```
1  <!-- todo/templates/show_entries.html -->
2  <html>
3      <head>
4      <!-- automatycznie przekazana informacja z konfiguracji aplikacji -->
5          <title>{{ config.SITE_NAME }}</title>
6      </head>
7      <body>
8          <h1>{{ config.SITE_NAME }}:</h1>
9          <ol>
10             <!-- wypisujemy kolejno wszystkie zdania -->
11             {% for entry in entries %}
12                 <li>
13                     {{ entry.title }}<em>{{ entry.created_at }}</em>
14                 </li>
15             {% endfor %}
16         </ol>
17     </body>
18 </html>
```

Wewnątrz szablonu przeglądamy wszystkie wpisy (*entries*) i umieszczamy je na liście HTML. Do szablonu automatycznie przekazywany jest obiekt `config` (dane konfiguracyjne), z którego pobieramy tytuł strony (*SITE_NAME*). Po odwiedzeniu strony *127.0.0.1:5000* powinniśmy zobaczyć listę zadań.

Moja lista ToDo:

1. Wyrzucić śmieci*2014-08-13 10:39:58*
2. Nakarmić psa*2014-08-13 10:39:58*

3.15.7 Formularz dodawania zadań

Aby umożliwić dodawanie i zapisywanie w bazie nowych zadań, modyfikujemy widok `index()`, tak aby obsługiwał żądania POST, które zawierają dane wysłane z formularza na serwer.

```
# dodajemy importy
from datetime import datetime
from flask import flash, redirect, url_for, request

# dekorator laczacy adres glowny z widokiem index
# poza powiazaniem adresu / z funkcja index, dodajemy mozliwosc akcpetacji
# zadan HTTP POST (domyslnie dozwolone sa tyko zadania GET)
@app.route('/', methods=['GET', 'POST'])
def index():
    """Glowny widok strony. Obsluguje wyswietlanie i dodawanie zadan."""
```

```
    # zmienna przechowujaca informacje o ewentualnych bledach
    error = None

    # jezeli otrzymujemy dane POST z formularza, dodajemy nowe zadanie
    if request.method == 'POST':
        if len(request.form['entry']) > 0: # sprawdzamy poprawnosc przeslanych danych
            db = get_db() # nawiazujemy polaczenie z baza danych
            new_entry = request.form['entry'] # wyciagamy tresc zadania z przeslanego formularza
            is_done = '0' # ustalamy, ze nowo dodane zadanie nie jest jeszcze wykonane
            created_at = datetime.now() # data dodania
            # zapytanie do bazy, ktore wstawia nowy wiersz z danymi
            db.execute('insert into entries (title, is_done, created_at) values (?, ?, ?);', [new_ent
            db.commit() # wykonujemy zapytanie
            flash('Dodano nowe zadanie') # informacja o pomyslnym dodaniu nowego zadania
            return redirect(url_for('index')) # przekierowujemy na strone glowna
        error = u'Nie możesz dodać pustego zadania' # komunikat o bledzie

    db = get_db() # laczymy sie z baza
    # pobieramy wszystkie wpisy z bazy:
    cur = db.execute('select id, title, is_done, created_at from entries order by created_at desc;')
    entries = cur.fetchall()
    # renderujemy tempaltke i zwracamy do klienta:
    return render_template('show_entries.html', entries=entries, error=error)
```

Wpisując adres w polu adresu przeglądarki, wysyłamy do serwera żądanie typu GET, które obsługujemy zwracając klientowi odpowiednie dane (listę zadań). Natomiast żądania typu POST są wykorzystywane do zmiany informacji na serwerze (np. dodania nowego wpisu). Dlatego widok `index()` rozszerzyliśmy o sprawdzanie typu żądania, w razie wykrycia danych POST, sprawdzamy poprawność danych przesłanych z formularza i jeżeli są poprawne, dodajemy nowe zadanie do bazy. W przeciwnym razie zwracamy użytkownikowi informację o błędzie. Szablon `show_entries.html` aktualizujemy, dodając odpowiedni formularz:

```
1  <!-- todo/templates/show_entries.html -->
2  <html>
3      <head>
4      <!-- automatycznie przekazana informacja z konfiguracji aplikacji -->
5          <title>{{ config.SITE_NAME }}</title>
6      </head>
7      <body>
8          <h1>{{ config.SITE_NAME }}:</h1>
9          <!-- formularz dodawania zadania -->
10         <form class="add-form" method="POST" action="{{ url_for('index') }}">
11             <input name="entry" value=""/>
12             <button type="submit">Dodaj zadanie</button>
13         </form>
14
15         <!-- informacje o sukcesie lub bledzie -->
16         <p>
17             {% if error %}
18                 <strong class="error">Błąd: {{ error }}</strong>
19             {% endif %}
20
21             {% for message in get_flashed_messages() %}
22                 <strong class="success">{{ message }}</strong>
23             {% endfor %}
24         </p>
25         <ol>
26             <!-- wypisujemy kolejno wszystkie zdania -->
27             {% for entry in entries %}
```

```
                <li>
                    {{ entry.title }}<em>{{ entry.created_at }}</em>
                </li>
            {% endfor %}
        </ol>
    </body>
</html>
```

W szablonie dodaliśmy formularz oraz informację o błędzie lub sukcesie przy próbie dodawania zadania. Określając atrybut *action* w formularzu, skorzystaliśmy z wbudowanej funkcji `url_for`, która zamienia nazwę widoku (w tym wypadku `index`) na odpowiadający jej adres URL (w tym wypadku `/`). W ten sposób łączymy formularz z konkretną funkcją Pythonową (widokiem), która obsługuje dany adres.

3.15.8 Wygląd aplikacji (opcja)

Wygląd aplikacji możemy zdefiniować w arkuszu stylów CSS o nazwie `style.css`, który zapisujemy w podkatalogu `static` aplikacji:

```
/* todo/static/style.css */

body { margin-top: 20px; }
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="entry"] { width: 300px; }
input[name="entry"]:focus {
    border-color: blue;
    border-radius: 5px;
}
li { margin-bottom: 5px; }
button {
    padding: 0;
    cursor: pointer;
    font-size: 11px;
    background: white;
    border: none;
    color: blue;
}
```

```
.error { color: red; }
.success { color: green; }
.done { text-decoration: line-through; }
```

Zdefiniowane style podpinamy do pliku `show_entries.html`, dodając w sekcji head wpis <link... >:

```
<head>
    <title>{{ config.SITE_NAME }}</title>
    <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='style.css') }}">
</head>
```

Dzięki temu nasza aplikacja nabierze nieco lepszego wyglądu.

Moja lista ToDo:

Dodaj zadanie

Dodano nowe zadanie

1. kolejne ważne zadanie *2014-08-13 14:32:51.449445*
2. nowe zadanie *2014-08-13 14:29:45.798655*
3. Wyrzucić śmieci *2014-08-13 12:29:32*
4. Nakarmić psa *2014-08-13 12:29:32*

3.15.9 Oznaczanie zadań jako wykonane (opcja)

Do każdego zadania dodamy formularz, którego wysłanie będzie oznaczało, że wykonaliśmy dane zadanie, czyli zmienimy atrybut `is_done` wpisu z *0* (niewykonane) na *1* (wykonane). Odpowiednie żądanie typu POST obsłuży nowy widok w pliku `todo.py`. W szablonie `show_entries.html` dodamy kod wyróżniający zadania wykonane.

```
# dodajemy ponad definicja if __main__(...)

# nadajemy osobny adres, oraz zezwalamy jedynie na zadania typu POST
@app.route('/mark_as_done', methods=['POST'])
def mark_as_done():
    """Zmiana statusu zadania na wykonane."""
    # z przeslanego formularza pobieramy identyfikator zadania
    entry_id = request.form['id']
    db = get_db() # laczymy sie z baza danych
    # przygotowujemy zapytanie aktualizujace pole is_done zadania o danym identyfikatorze
    db.execute('update entries set is_done=1 where id=?', [entry_id,])
    db.commit() # zapisujemy nowe dane
    return redirect(url_for('index')) # na koniec przekierowujemy na liste wszystkich zadan
```

W szablonie `show_entries.html` modyfikujemy fragment wyświetlający listę zadań i dodajemy formularz:

```
<ol>
    {% for entry in entries %}
```

```
        <li>
            <!-- dodatkowe dekoracje dla zadan zakonczonych -->
            {% if entry.is_done %}
                <span class="done">
            {% endif %}

            {{ entry.title }}<em>{{ entry.created_at }}</em>

            <!-- dodatkowe dekoracje dla zadan zakonczonych -->
            {% if entry.is_done %}
                </span>
            {% endif %}

            <!-- formularz zmiany statusu zadania -->
            {% if not entry.is_done %}
                <form method="POST" action="{{ url_for('mark_as_done') }}">
                    <!-- wysylamy jedynie informacje o id zadania -->
                    <input type="hidden" name="id" value="{{ entry.id }}"/>
                    <button type="submit">Wykonane</button>
                </form>
            {% endif %}
        </li>
    {% endfor %}
</ol>
```

Aplikację można uznać za skończoną. Możemy dodawać zadania oraz zmieniać ich status.

3.15.10 POĆWICZ SAM

Dodaj możliwość usuwania zadań. Dodaj mechanizm logowania użytkownika tak, aby użytkownik mógł dodawać i edytować tylko swoją listę zadań. Wprowadź osobne listy zadań dla każdego użytkownika.

Pojęcia

Aplikacja program komputerowy.

Baza danych program przeznaczony do przechowywania i przetwarzania danych.

CSS język służący do opisu formy prezentacji stron WWW.

Framework zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji.

GET typ żądania HTTP, służący do pobierania zasobów z serwera WWW.

HTML język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.

HTTP protokół przesyłania dokumentów WWW.

POST typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW.

Serwer deweloperski serwer używany w czasie prac nad oprogramowaniem.

Serwer WWW serwer obsługujący protokół HTTP.

Templatka szablon strony WWW wykorzystywany przez Flask do renderowania widoków.

URL ustandaryzowany format adresowania zasobów w internecie (przykład: http://pl.wikipedia.org/wiki/Uniform_Resource_Locator).

- Widok – funkcja obsługująca żądania przychodzące na powiązany z nią adres, zazwyczaj zwraca użytkownikowi żądaną stronę html wyrenderowaną ze wskazanego szablonu.

Materiały

1. Strona projektu Flask http://flask.pocoo.org/
2. Informacje o SQLite http://pl.wikipedia.org/wiki/SQLite
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

- `todo_all.zip`

Metryka

Autorzy Tomasz Nowacki, Robert Bednarz

Utworzony 2014-10-17 o 17:02

3.16 Chatter – aplikacja internetowa

Zastosowanie Pythona i frameworka Django (wersja 1.6.5) do stworzenia aplikacji internetowej Chatter; prostego czata, w którym zarejestrowani użytkownicy będą mogli wymieniać się krótkimi wiadomościami.

3.16.1 Projekt i aplikacja

Tworzymy nowy projekt Django, a następnie uruchamiamy lokalny serwer, który pozwoli śledzić postęp pracy. W katalogu domowym wydajemy polecenia w terminalu:

```
~ $ django-admin.py startproject chatter
~ $ cd chatter
~/chatter $ python manage.py runserver 127.0.0.1:8080
```

Powstanie katalog projektu `chatter` i aplikacja o nazwie `chatter`. Pod adresem *127.0.0.1:8080* w przeglądarce zobaczymy stronę powitalną.

It worked!
Congratulations on your first Django-powered page.

Of course, you haven't actually done any work yet. Next, start your first app by running `python manage.py startapp [appname]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

Informacja: Jeden projekt może zawierać wiele aplikacji zapisywanych w osobnych podkatalogach katalogu projektu. Lokalny serwer deweloperski można zatrzymać za pomocą skrótu Ctrl+C.

Teraz zmodyfikujemy ustawienia projektu, aby korzystał z polskiej wersji językowej oraz lokalnych ustawień daty i czasu. Musimy również zarejestrować naszą aplikację w projekcie. W pliku `setting.py` zmieniamy następujące linie:

```
# chatter/chatter/settnigs.py

# rejestrujemy aplikacje
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    'chatter', # nasza aplikacja
)

LANGUAGE_CODE = 'pl' # ustawienia jezyka

TIME_ZONE = 'Europe/Warsaw' # ustawienia daty i czasu
```

3.16.2 Model – Widok – Kontroler

W projektowaniu aplikacji internetowych za pomocą Django odwołujemy się do wzorca M(odel)V(iew)C(ontroller), czyli Model–Widok–Kontroler [7], co pozwala na oddzielenie danych od ich prezentacji oraz logiki aplikacji. Funkcje kolejnych elementów są następujące:

- *Modele* – w Django reprezentują źródło informacji, są to klasy Pythona, które zawierają pola, właściwości i zachowania danych, odwzorowują pojedyncze tabele w bazie danych [8]. Definiowane są w pliku `models.py`.
- *Widoki* – w Django są to funkcje Pythona, które na podstawie żądań www (dla danych adresów URL) zwracają odpowiedź w postaci kodu HTML generowanego w szablonach (templates), przekierowania, dokumentu XML czy obrazka. Definiowane są w pliku `views.py`.
- *Kontroler* – to mechanizm kierujący kolejne żądania do odpowiednich widoków na podstawie konfiguracji adresów URL zawartej w pliku `urls.py`.

3.16.3 Model danych i baza

Model, jak zostało powiedziane, jest klasą Pythona opisującą dane naszej aplikacji, czyli wiadomości. Instancje tej klasy będą konkretnymi wiadomościami napisanymi przez użytkowników systemu. Każda wiadomość będzie zwierała treść, datę dodania oraz autora wiadomości (użytkownika).

W katalogu `chatter/chatter` w pliku `models.py` wpisujemy:

```
1  # -*- coding: utf-8 -*-
2
3  # chatter/chatter/models.py
4
5  from django.db import models
6  from django.contrib.auth.models import User
7
8  class Message(models.Model):
9      """Klasa reprezentujaca wiadomosc w systemie."""
10     text = models.CharField(u'wiadomość', max_length=250)
11     pub_date = models.DateTimeField(u'data publikacji')
12     user = models.ForeignKey(User)
13
14     class Meta: # to jest klasa w klasie
15         verbose_name = u'wiadomość' # nazwa modelu w jezyku polskim
16         verbose_name_plural = u'wiadomości' #  nazwe modelu w l. mnogiej
17         ordering = ['pub_date'] # porzadkowanie danych wzgledem daty
```

Po skonfigurowaniu projektu i zdefiniowaniu modelu danych możemy utworzyć bazę danych [9] dla naszej aplikacji, czyli wszystkie potrzebne tabele. Podczas tworzenia bazy Django pyta o nazwę, email i hasło administratora. Podajemy te dane po wydaniu w katalogu projektu w terminalu polecenia:

```
~/chatter $ python manage.py syncdb
```

3.16.4 Panel administracyjny

Django pozwala szybko utworzyć panel administratora dla naszego projektu. Rejestrujemy więc model danych jako element panelu w nowo utworzonym pliku `admin.py` w katalogu `chatter/chatter`:

[7] Twórcy Django traktują jednak ten wzorzec elastycznie, mówiąc że ich framework wykorzystuje wzorzec MTV, czyli model (model), szablon (template), widok (view).

[8] Takie odwzorowanie nosi nazwę mapowania obiektowo-relacyjnego (ORM). ORM odwzorowuje strukturę bazy na obiekty Pythona.

[9] Domyślnie Django korzysta z bazy SQLite, która przechowywana jest w jednym pliku `db.sqlite3` w katalogu aplikacji.

```
# -*- coding: utf-8 -*-

# chatter/chatter/admin.py

from django.contrib import admin
from chatter.models import Message # importujemy nasz model

admin.site.register(Message) # rejestrujemy nasz model w panelu administracyjnym
```

Po ewentualnym ponownum uruchomieniu serwera wchodzimy na adres *127.0.0.1:8080/admin/*. Otrzymamy dostęp do panelu administracyjnego, w którym możemy dodawać nowych użytkowników i wiadomości [10].

Administracja Django Witaj, root.

Administracja stroną

Auth		
Grupy	Dodaj	Zmień
Użytkownicy	Dodaj	Zmień
Chatter		
Wiadomości	Dodaj	Zmień

Ostatnie ak

Moje akcj

Message
Message
Message
Message

Administracja Django Witaj, root.

Początek › Chatter › Wiadomości › Dodaj wiadomość

Dodaj wiadomość

Wiadomość: nowa wiadomość

Data publikacji: Data: 11.08.2014 Dzisiaj | Czas: 14:39:40 Teraz |

User: root

Zapisz i dodaj nowe Zapisz i ko

[10] Bezpieczna aplikacja powinna dysponować osobnym mechanizmem rejestracji użytkowników i dodawania wiadomości, tak by nie trzeba było udostępniać panelu administracyjnego osobom postronnym.

3.16.5 Strona główna – widoki i szablony

Dodawanie stron w Django polega na tworzeniu widoków, czyli funkcji Pythona powiązanych z określonymi adresami url. Widoki najczęściej zwracały będą kod HTML wyrenderowany na podstawie szablonów, do których możemy przekazywać dodatkowe dane [11], np. z bazy. Dla przejrzystości przyjęto, że w katalogu aplikacji (`chatter/chatter`):

1. plik `views.py` zawiera definicję widoków, w tym wywołania szablonów,
2. plik `url.py` zawiera reguły łączące widoki z adresami url,
3. w katalogu `chatter/chatter/templates/chatter` zapisujemy szablony (templatki) pod nazwami określonymi w wywołujących je widokach, np. `index.html`.

Aby utworzyć stronę główną, stworzymy pierwszy widok, czyli funkcję `index()` [12], którą powiążemy z adres URL głównej strony (/). Widok zwracał będzie kod wyrenderowany na podsatwie szablonu `index.html`. W pliku `views.py` umieszczamy:

```
1  # -*- coding: utf-8 -*-
2  # chatter/chatter/views.py
3
4  # HttpResponse pozwala zwracac proste wiadomości tekstowe
5  from django.http import HttpResponse
6  # render pozwala zwracac szablony
7  from django.shortcuts import render
8
9  def index(request):
10     """Strona glowna aplikacji."""
11     # aby wyświetlić (zwrócić) prosta wiadomosc tekstowa wystarczyłaby instrukcja poniżej:
12     #return HttpResponse("Hello, SWOI")
13     # my zwrócimy szablon index.html, uwaga (!) ścieżkę podajemy względną do katalogu templates :
14     return render(request, 'chatter/index.html')
```

Widok `index()` łączymy z adresem URL strony głównej: *127.0.0.1:8000/* w pliku `urls.py`:

```
1  # -*- coding: utf-8 -*-
2  # chatter/chatter/views.py
3
4  from django.conf.urls import patterns, include, url
5  from django.contrib import admin
6
7  from chatter import views # importujemy zdefiniowane w pliku views.py widoki
8
9  admin.autodiscover()
10
11 urlpatterns = patterns('',
12     # glowny adres (/) o nazwie index laczymy z widokiem index
13     url(r'^$', views.index, name='index'),
14
15     url(r'^admin/', include(admin.site.urls)),
16 )
```

Tworzymy katalog dla szablonów wydając polecenie:

```
~/chatter/chatter $ mkdir -p templates/chatter
```

Tworzymy szablon, plik `chatter/chatter/templates/chatter/index.html`, który zawiera:

[11] Danych 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.

[12] Nazwa `index()` jest przykładowa, funkcja mogłaby się nazywać inaczej.

```
<!-- chatter/chatter/templates/chatter/index.html -->

<html>
    <head></head>
    <body>
        <h1>Witaj w systemie Chatter</h1>
    </body>
</html>
```

Po wpisaniu adresu *127.0.0.1:8080/* zobaczymy tekst, który zwróciliśmy z widoku, czyli "Witaj w systemie Chatter".

Witaj w systemie Chatter

3.16.6 Logowanie użytkowników

Dodanie formularza logowania dla użytkowników polega na:

1. dodaniu w pliku `views.py` nowego widoku `my_login()`, który wywoływać będzie szablon zapisany w pliku `templates/chatter/login.html`,
2. powiązaniu w pliku urls.py nowego widoku z adresem `/login`.

Django upraszcza zadanie, ponieważ zawiera odpowiednie formularze i model reprezentujący użytkowników w systemie, z którego – nota bene – skorzystaliśmy już podczas tworzenia bazy danych.

Widok `my_login()` wyświetli formularz logowania i obsłuży żądania typu POST (wysłanie danych z formularza na serwer), sprawdzi więc poprawność przesłanych danych (nazwa użytkownika i hasło). Jeżeli dane będą poprawne, zaloguje użytkownika i wyświetli spersonalizowaną stronę główną (`index.html`), w przeciwnym wypadku zwrócona zostanie informacja o błędzie.

Importujemy potrzebne moduły, tworzymy widok `my_login()` i uzupełniamy widok `index()` w pliku `views.py`:

```python
# -*- coding: utf-8 -*-
# chatter/chatter/views.py

# HttpResponse pozwala zwracac proste wiadomości tekstowe
from django.http import HttpResponse
# render pozwala zwracac szablony
from django.shortcuts import render
# dodajemy nowe importy
from django.shortcuts import render, redirect
from django.contrib.auth import forms, authenticate, login, logout
from django.contrib.auth.models import User
from django.core.urlresolvers import reverse

def index(request):
    """Strona glowna aplikacji."""
    # tworzymy zmienną (słownik), zawierającą informacje o użytkowniku
    context = {'user': request.user}
    # zmienna context przekaujemy do szablonu index.html
    return render(request, 'chatter/index.html', context)

```

```
def my_login(request):
    """Logowanie uzytkownika w sytemie."""
    form = forms.AuthenticationForm() # ustawiamy formularz logowania

    if request.method == 'POST': # sprawdzamy, czy ktos probuje sie zalogowac
        # przypisujemy nadeslane dane do formularza logowania
        form = forms.AuthenticationForm(request, request.POST)
        # sprawdzamy poprawnosc formularza lub zwracamy informacje o bledzie
        if form.is_valid(): # jezeli wszystko jest ok - logujemy uzytkownika
            user = form.get_user()
            login(request, user)
            return redirect(reverse('index')) # przekierowujemy uzytkownika na strone glowna

    context = {'form': form} # ustawiamy zmienne przekazywane do templatki
    # renderujemy templatke logowania
    return render(request, 'chatter/login.html', context)
```

W pliku `urls.py` dopisujemy regułę łączącą url */login* z widokiem `my_login()`:

```
# adres logowania (/login) o nazwie login powiazany z widokiem my_login
url(r'^login/$', views.my_login, name='login'),
```

Tworzymy nowy szablon `login.html` w katalogu `templates/chatter/`:

```
<!-- chatter/chatter/templates/login.html -->
<html>
    <body>
        <h1>Zaloguj się w systemie Chatter</h1>

        <form method="POST">
            {% csrf_token %}
            {{ form.as_p }}
            <button type="submit">Zaloguj</button>
        </form>
    </body>
</html>
```

Zmieniamy również szablon `index.html` głównego widoku, aby uwzględniał status użytkownika (zalogowany/niezalogowany):

```
<! -- chatter/chatter/templates/chatter/index.html -->

<html>
    <head></head>
    <body>
        {% if not user.is_authenticated %}
            <h1>Witaj w systemie Chatter</h1>
            <p><a href="{% url 'login' %}">Zaloguj się</a></p>
        {% else %}
            <h1>Witaj, {{ user.username }}</h1>
        {% endif %}
    </body>
</html>
```

Zwróćmy uwagę, jak umieszczamy linki w szablonach. Mianowicie kod `{% url 'login' %}` wykorzystuje wbudowaną funkcję `url()`, która na podstawie nazwy adresu określanej w regułach pliku `urls.py` (parametr `name`) generuje skojarzony z nią adres.

JAK TO DZIAŁA: Po przejściu pod adres *127.0.0.1:8080/login/*, powiązany z widokiem `my_login()`, przeglądarka wysyła żądanie GET do serwera. Widok `my_login()` przygotowuje formularz autoryzacji (AuthenticationForm),

przekazuje go do szablonu `login.html` i zwraca do klienta. Efekt jest taki:

Zaloguj się w systemie Chatter

Użytkownik:

Hasło:

Zaloguj

Po wypełnieniu formularza danymi i kliknięciu przycisku "Zaloguj", do serwera zostanie wysłane żądanie typu POST. W widoku `my_login()` obsługujemy taki przypadek za pomocą instrukcji `if`. Sprawdzamy poprawność przesłanych danych (walidacja), logujemy użytkownika w systemie i zwracamy przekierowanie na stronę główną, która wyświetla nazwę zalogowanego użytkownika. Jeżeli dane nie są poprawne, zwracana jest informacja o błędach. Przetestuj!

3.16.7 Dodawanie i wyświetlanie wiadomości

Chcemy, by zalogowani użytkownicy mogli przeglądać wiadomości od innych użytkowników i dodawać własne. Utworzymy widok `messages()`, który wyświetli wszystkie wiadomości (żądanie GET) i ewentualnie zapisze nową wiadomość nadesłaną przez użytkownika (żądanie POST). Widok skorzysta z nowego szablonu `messages.html` i powiązany zostanie z adresem */messages*. Zaczynamy od zmian w `views.py`.

```
# -*- coding: utf 8 -*-

# chatter/chatter/views.py

# dodajemy nowe importy
from chatter.models import Message
from django.utils import timezone
from django.contrib.auth.decorators import login_required


# pozostale widoki

# dekorator, ktory "chroni" nasz widok przed dostepem przez osoby niezalogowane, jezeli uzytkownik n
# bedzie probowal odwiedzic ten widok, to zostanie przekierowany na strone logowania
@login_required(login_url='/login')
def messages(request):
    """Widok wiadomosci."""
    error = None

    # zadanie POST oznacza, ze ktos probuje dodac nowa wiadomosc w systemie
    if request.method == 'POST':
        text = request.POST.get('text', '') # pobieramy tresc przeslanej wiadomosci
        # sprawdzamy, czy nie jest ona dluzsza od 250 znakow:
        # - jezeli jest dluzsza, to zwracamy blad, jezeli jest krotsza lub rowna, to zapisujemy ja w
        if not 0 < len(text) <= 250:
            error = u'Wiadomość nie może być pusta i musi mieć co najwyżej 250 znaków'
```

```
        else:
            # ustawiamy dane dla modelu Message
            msg = Message(text=text, pub_date=timezone.now(), user=request.user)
            msg.save() # zapisujemy nowa widomosc
            return redirect(reverse('messages')) # przekierowujemy na strone wiadomosci

    user = request.user # informacje o aktualnie zalogowanym uzytkowniku
    messages = Message.objects.all() # pobieramy wszystkie wiadomosci
    # ustawiamy zmienne przekazywane do szablonu
    context = {'user': user, 'messages': messages, 'error': error}
    # renderujemy templatke wiadomosci
    return render(request, 'chatter/messages.html', context)
```

Teraz tworzymy nowy szablon `messages.html` w katalogu `templates/chatter/`.

```
<!-- chatter/chatter/templates/messages.html -->
<html>
    <body>
        <h1>Witaj, {{ user.username }}</h1>

        <!-- w razie potrzeby wyswietlamy bledy -->
        {% if error %}
            <p>{{ error }}</p>
        {% endif %}

        <form method="POST">
            {% csrf_token %}
            <input name="text"/>
            <button type="submit">Dodaj wiadomość</button>
        </form>

        <!-- wyswietlamy wszystkie wiadomosci -->
        <h2>Wiadomości:</h2>
        <ol>
            {% for message in messages %}
                <li>
                    <strong>{{ message.user.username }}</strong> ({{ message.pub_date }}):
                    <br>
                    {{ message.text }}
                </li>
            {% endfor %}
        </ol>
    </body>
</html>
```

Uzupełniamy szablon widoku głównego, aby zalogowanym użytkownikom wyświetlał się link prowadzący do strony z wiadomościami. W pliku `index.html` po klauzuli `{% else %}`, poniżej znacznika `<h1>` wstawiamy:

```
<p><a href="{% url 'messages' %}">Zobacz wiadomości</a></p>
```

Na koniec dodajemy nową regułę do `urls.py`:

```
url(r'^messages/$', views.messages, name='messages'),
```

Jeżeli uruchomimy serwer deweloperski, zalogujemy się do aplikacji i odwiedzimy adres *127.0.0.1:8080/messages/*, zobaczymy listę wiadomości dodanych przez użytkowników [13].

JAK TO DZIAŁA: W widoku `messages()`, podobnie jak w widoku `login()`, mamy dwie ścieżki

[13] Jeżeli w panelu administracyjnym nie dodałeś żadnej wiadomości, lista będzie pusta.

Witaj, SWOI

Dodaj wiadomość

Wiadomości:

1. **root** (11 sierpnia 2014 10:37:06):
   nowa wiadomość

postępowania, w zależności od użytej metody HTTP. GET pobiera wszystkie wiadomości (`messages = Message.objects.all()`), przekazuje je do szablonu i renderuje. Django konstruuje odpowiednie zapytanie i mapuje dane z bazy na obiekty klasy `Message` (mapowanie obiektowo-relacyjne (ORM)).

POST zawiera z kolei treść nowej wiadomości, której długość sprawdzamy i jeżeli wszystko jest w porządku, tworzymy nową wiadomość (instancję klasy *Message*, czyli obiekt `msg`) i zapisujemy ją w bazie danych (wywołujemy metodę obiektu: `msg.save()`).

3.16.8 Rejestrowanie użytkowników

Utworzymy nowy widok `my_register()`, szablon `register.html` i nowy adres URL */register*, który skieruje użytkownika do formularza rejestracji, wymagającego podania nazwy i hasła. Zaczynamy od dodania widoku w pliku `views.py`.

```
# chatter/chatter/views.py

# pozostale widoki

def my_register(request):
    """Rejestracja nowego uzytkownika."""
    form = forms.UserCreationForm() # ustawiamy formularz rejestracji

    # POST oznacza, ze ktos probuje utworzyc nowego uzytkownika
    if request.method == 'POST':
        # przypisujemy nadeslane dane do formularza tworzenia uzytkownika
        form = forms.UserCreationForm(request.POST)
        # sprawdzamy poprawnosc nadeslanych danych:
        # - jezeli wszystko jest wporzadku to tworzymy uzytkownika
        # - w przeciwnym razie zwracamy formularz wraz z informacja o bledach
        if form.is_valid():
            # zapamietujemy podana nazwe uzytkownika i haslo
            username = form.data['username']
            password = form.data['password1']
            # zapisujemy formularz tworzac nowego uzytkownika
            form.save()
            # uwierzytelniamy uzytkownika
            user = authenticate(username=username, password=password)
            login(request, user)
```

```
            # po udanej rejestracji, przekierowujemy go na strone glowna
            return redirect(reverse('index'))

    # ustawiamy zmienne przekazywane do szablonu
    context = {'form': form}
    # renderujemy templatke rejestracji
    return render(request, 'chatter/register.html', context)
```

Tworzymy nowy szablon `register.html` w katalogu `templates/chatter`:

```
1  <!-- chatter/chatter/templates/register.html -->
2
3  <html>
4      <body>
5          <h1>Zarejestruj nowego użytkownika w systemie Chatter</h1>
6
7          <form method="POST">
8              {% csrf_token %}
9              {{ form.as_p }}
10             <button type="submit">Zarejestruj</button>
11         </form>
12     </body>
13 </html>
```

W szablonie widoku głównego `index.html` po linku "Zaloguj się" wstawiamy kolejny:

```
<p><a href="{% url 'register' %}">Zarejestruj się</a></p>
```

Na koniec uzupełniamy plik `urls.py`:

```
url(r'^register/$', views.my_register, name='register'),
```

JAK TO DZIAŁA: Zasada działania jest taka sama jak w przypadku pozostałych widoków. Po wpisaniu adresu *127.0.0.1:8080/register/* otrzymujemy formularz rejestracji nowego użytkownika, który podobnie jak formularz logowania, jest wbudowany w Django, więc wystarczy przekazać go do szablonu. Po wypełnieniu i zatwierdzeniu formularza wysyłamy żądanie POST, widok `my_register()` odbiera przekazane dane (nazwę użytkownika, hasło i powtórzone hasło), sprawdza ich poprawność (poprawność i unikalność nazwy użytkownika oraz hasło) oraz tworzy i zapisuje nowego użytkownika. Po rejestracji użytkownik przekierowywany jest na stronę główną.

3.16.9 Wylogowywanie użytkowników

Django ma wbudowaną również funkcję wylogowującą. Utworzymy zatem nowy widok `my_logut()` i powiążemy go z adresem `/logout`. Do pliku `views.py` dodajemy:

```
def my_logout(request):
    """Wylogowywanie uzytkownika z systemu"""
    logout(request)
    # przekierowujemy na strone glowna
    return redirect(reverse('index'))
```

3.16.10 POĆWICZ SAM

> Powiąż widok `my_logut` z adresem *logout/* dopisując regułę w odpowiednim pliku. Powiązanie nazwij "logout". Wylogowywanie nie wymaga osobnego szablonu, dodaj jednak link wylogowujący do 1) szablonu `index.html` po linku "Zobacz wiadomości" oraz do 2) szablonu `messages.html` po nagłówku `<h1>`.

Zarejestruj nowego użytkownika w systemie Chatter

Nazwa użytkownika: [ ] Wymagane. 30 znaków lub mniej. Tylko litery, cyfry i znaki @/./+/-/_.

Hasło: [ ]

Potwierdzenie hasła: [ ] Podaj powyższe hasło w celu weryfikacji.

[Zarejestruj]

Witaj, SWOI

Wyloguj

[ ] [Dodaj wiadomość]

Wiadomości:

1. **sru** (13 września 2014 12:35:34): Pierwsza wiadomość
2. **SWOI** (14 września 2014 10:46:23): Zrobiłem pierwszą aplikację w Django!

Pojęcia

Aplikacja program komputerowy.

Framework zestaw komponentów i bibliotek wykorzystywany do budowy aplikacji.

GET typ żądania HTTP, służący do pobierania zasobów z serwera WWW.

HTML język znaczników wykorzystywany do formatowania dokumentów, zwłaszcza stron WWW.

HTTP protokół przesyłania dokumentów WWW.

Kontroler logika aplikacji, w Django zawarta w funkcji obsługującej widok.

Logowanie proces autoryzacji i uwierzytelniania użytkownika w systemie.

Model schematy i źródła danych aplikacji.

ORM mapowanie obiektowo-relacyjne, oprogramowanie służące do przekształcania struktur bazy danych na obiekty klasy danego języka oprogramowania.

POST typ żądania HTTP, służący do umieszczania zasobów na serwerze WWW.

Serwer deweloperski serwer używany w czasie prac nad oprogramowaniem.

Serwer WWW serwer obsługujący protokół HTTP.

Templatka szablon strony WWW wykorzystywany przez Django do renderowania widoków.

URL ustandaryzowany format adresowania zasobów w internecie (przykład: http://pl.wikipedia.org/wiki/Uniform_Resource_Locator).

Widok funkcja obsługująca żądania przychodzące na powiązany z nią adres, zazwyczaj zwraca użytkownikowi żądaną stronę html wyrenderowaną ze wskazanego szablonu.

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

- `chatter_all.zip`

Metryka

Autorzy Tomasz Nowacki, Robert Bednarz

Utworzony 2014-10-17 o 17:02

3.17 Indices and tables

* *genindex*
* *modindex*
* *search*

A

Aplikacja, 55, 63, 76

B

Baza danych, 64

C

CSS, 64

D

dziedziczenie, 36

F

Framework, 55, 64, 76
Funkcje, 17

G

generator, 41
GET, 55, 64, 76

H

HTML, 55, 64, 76
HTTP, 55, 64, 76

I

Interpreter, 17

J

Język interpretowany, 17

K

Kontroler, 76

L

Logowanie, 76

M

magiczne liczby, 41
Model, 76

O

ORM, 76

P

POST, 55, 64, 76
przesłanianie, 36

S

Serwer deweloperski, 55, 64, 76
Serwer WWW, 55, 64, 76
stała, 41

T

Templatka, 55, 64, 76
Typ zmiennych, 17

U

URL, 55, 64, 76

W

Widok, 55, 76

Z

Zmienne, 17