Wprowadzenie
do programowania w Pythonie

Uruchamianie skryptów Pythona

Przez skrypt rozumiemy plik tekstowy o rozszerzeniu **.py**. By go wykonać wydajemy polecenie:

python nazwa_pliku.py

lub po prostu

./nazwa_pliku.py

jeżeli plik zawiera odwołanie do interpretera - w pierwszej linijce plik zawiera:
#!/usr/bin/env python
Jeżeli wykonujemy skrypty pythona bezpośrednio (Linux/Unix) to pliki te muszą mieć atrybut wykonywalności (**chmod 755 nazwa_pliku.py**)

Witaj świecie

Zapisz podany kod jako skrypt1.py i uruchom go jedną z opisanych wcześniej metod:

```
#!/usr/bin/env python
print """Witaj w swiecie Pythona"""
print "Jeszcze raz"
print 'I po raz trzeci'
```

Powinieneś zobaczyć trzy wiersze z tekstem ze skryptu. Pierwszy skrypt prezentuje **napisy** (łańcuchy). W Pythonie napisy zamykane są poprzez pojedynczy lub podwójny cudzysłów, z tym że pojedynczy może być od jednego do trzech a podwójnego jeden lub trzy. Jeżeli tekst ograniczymy z obu stron potrójnymi cudzysłowami (dowolnego typu) to tekst może zajmować kilka wierszy. W pozostałych przypadkach musi być zawarty w jednym wierszu. Zastosowane polecenie **print** służy w Pythonie do wyświetlania tekstu (w konsoli)

Oto kolejny skrypt:

```
#!/usr/bin/env python
a = 'Witaj w swiecie Pythona'
print a[0]
print a[0:5]
```

Gdy go wykonamy zobaczymy w pierwszym wierszu "W" a w drugim "Witaj". W skrypcie tym przypisaliśmy napis "Witaj w swiecie Pythona" do zmiennej "**a**" a następnie skorzystaliśmy z **operatora indeksowania** by wyświetlić określony fragment napisu. Operator indeksowania ma postać: **nazwa_zmiennej[od:do]** gdzie **od:do** oznacza numery porządkowe w napisie (od którego wyświetlać / do którego wyświetlać). Jeżeli podamy tylko jedną liczbę to zostanie wyświetlony znak z pozycji o podanym numerze.
Do łączenia napisów służy znak "**+**":

```
#!/usr/bin/env python
a = 'Witaj w swiecie Pythona'
b = ' w 2007 roku'
print a + b
```

Co wyświetli "Witaj w swiecie Pythona w 2007 roku" Do łączenia napisów ze zmiennymi innych typów służą dwie funkcje:

```
#!/usr/bin/env python
a = 'Witaj w swiecie Pythona'
b = ' w 2006 roku. Szczesliwy numerek to: '
c = 13
print a + b + str(c)
print a + b + repr(c)
```

str zamienia typ zmiennej na typ napisowy, co umożliwia dołączenie np. liczby do napisu.

Liczby

Operacje matematyczne są proste i intuicyjne. Oto przykład:

```
#!/usr/bin/env python
a = 2
b = 5
print a+b
print a*b
print a/b
print a%b
```

Znak % oznacza dzielenie modulo - zwróci resztę z dzielenia. Wynik będzie następujący:

```
7
10
0
2
```

Trzeci wynik, dzielenie zwróciło wynik zero. Dlaczego? Gdyż operujemy na liczbach całkowitych (integer, int). Liczby zmiennoprzecinkowe zapisalibyśmy w Pythonie tak:

```
a = 2.0
b = 5.0
```

Przez co otrzymalibyśmy wynik:

```
7.0
10.0
0.4
2.0
```

Pisałem już o łączeniu napisów z innymi typami danych. Dość często stosuje się inne, wygodniejsze rozwiązanie:

```
#!/usr/bin/env python
a = 2.0
b = 5.0
wynik = a/b
poczatek = "Wynik dzielenia wynosi:"
koniec = "co bylo oczekiwane"
print "%s %f %s" % (poczatek, wynik, koniec)
```

Co da

Wynik dzielenia wynosi: 0.400000 co bylo oczekiwane

%s, **%d** i **%f** to odpowiednio - napis, liczba całkowita i liczba zmiennoprzecinkowa. Znaki te wstawione w napis zostaną zamienione wartościami zmiennych podanych na końcu wiersza (**% (zmienna, zmienna, itd)**)

Instrukcje Warunkowe

Do wykonywania testów "jeżeli" służy składnia:

```
#!/usr/bin/env python
b = 2
a = 1
if a > b:
 print b
else:
 print a

c = 2

if a > b and a > c:
 print a
elif c == b:
 print "C i B rowne"
else:
 print b
```

Po wykonaniu skryptu zobaczymy "1" i "C i B rowne" Struktura składni ma postać:

```
if warunek:
      instrukcje
else:
      instrukcje
```

Należy zwrócić uwagę na **wcięcia** - są one obowiązkowe! Struktura blokowa jest elementem składni pythona (w C/C++ czy np. PHP odpowiednikiem są nawiasy klamrowe). Jako warunki podajemy zdarzenia zwracające wartości prawda/fałsz, np.:

== - jest równe

!= - nie jest równe

< > - mniejsze, większe niż

Należy pamiętać że a = b to przypisanie, natomiast a == b to porównanie. Pominięcie jednego znaku równości jest częstym błędem.

Polskie znaki w skryptach Pythona

Domyślnie interpreter przyjmuje kodowanie ASCII dla plików *py. Jeżeli stosujemy polskie znaki to musimy podać odpowiednie kodowanie. Zaleca się stosowanie kodowania utf-8 i zapisywania plików z tym kodowaniem. Po zapisaniu pliku z tym kodowaniem należy dodać na początku pliku:
-*- coding: utf-8 -*-

Listy i tuple

Listy i tuple są sekwencjami/zbiorami różnych obiektów. W innych językach nazywane tablicami. Oto przykład:

```
#!/usr/bin/env python
imiona = [ "zbychu", "rychu", "zdzisiu" ]
print imiona
imiona[1] = "rychu2"
print imiona[1]
print len(imiona)
imiona.append("Renata")
print imiona
```

Na początku tworzymy **listę** imiona. By wyświetlić wpis z danej pozycji korzystamy z opisanego wcześniej operatora indeksowania. Polecenie **imiona[1] = "rychu2"** zmienia wartość drugiego (numeracja elementów rozpoczyna się od zera) elementu listy. Funkcja **len** zwraca ilość elementów listy. Oprócz tego listy i tuple mają sporo innych opcji, o których powiemy później. Mogą one zawierać również np. liczby czy inne listy/tuple. Za pomocą operatora "+" można łączyć kilka list/tupli w jedną. Tuple różnią się tym od list że nie można zmieniać wartości elementów tupli po jej utworzeniu. Tuplę tworzymy tak:
tupla = (1, 3, "jurek")

Metody Listy

- **list(s)** - konwertuje sekwencję s na listę
- **s.append(x)** - dodaje nowy element x na końcu s
- **s.extend(t)** - dodaje nową listę t na końcu s
- **s.count(x)** - zlicza wystąpienie x w s
- **s.index(x)** - zwraca najmniejszy indeks i, gdzie s[i] == x
- **s.pop([i])** - zwraca i-ty element i usuwa go z listy. Jeżli nie podamy parametru to usunięty zostanie ostatni element
- **s.remove(x)** - odnajduje x i usuwa go z listy s
- **s.reverse()** - odwraca w miejscu kolejność elementów s
- **s.sort([funkcja])** - Sortuje w miejscu elementy. "funkcja" to funkcja porównawcza

Metody Napisowe

- **s.capitalize()** - zmienia pierwszą literę na dużą
- **s.center(długość)** - Centruje napis w polu o podanej długości
- **s.count(sub)** - zlicza wystąpienie podciągu sub w napisie s
- **s.encode(kodowanie)** - zwraca zakodowaną wersję napisu ('utf-8', 'ascii', 'utf-16')
- **s.isalnum()** - sprawdza czy wszystkie znaki są znakami alfanumerycznymi
- **s.isdigit()** - sprawdza czy wszystkie znaki są cyframi
- **s.islower()** - sprawdza czy wszystkie litery są małe
- **s.isspace()** - sprawdza czy wszystkie znaki są białymi znakami
- **s.isupper()** - sprawdza czy wszystkie litery są duże
- **s.join(t)** - łączy wszystkie napisy na liście t używając s jako separatora

```
l = ['a', 'b', 'c']
s = '.'
print s.join(l)
```

- **s.lstrip()** - usuwa początkowe białe znaki
- **s.replace(old, new)** - zastępuje stary podciąg nowym
- **s.rstrip()** - usuwa końcowe białe znaki
- **s.split(separator)** - dzieli napis używają podanego separatora
- **s.strip()** - usuwa początkowe i końcowe białe znaki

Pętle

W Pythonie mamy też kilka rodzajów pętli. Pierwsza z nich to **while**:

```
#!/usr/bin/env python
licznik = 10
wartosc = 15

while licznik <= wartosc:
        licznik += 1
        print "Jestem w while."
```

Po wykonaniu kodu zobaczymy sześć "**Jestem w while.**". Pętla ta ma postać ogólną:

```
while WARUNEK:
  ZDARZENIA_DLA_PĘTLI
```

Kolejna pętla wygląda następująco:

```
#!/usr/bin/env python
for i in range(10):
  print i
print "Petla 2:"
for i in range(3, 5):
  print i
print "Petla 3:"
for i in range(10, 100, 10):
  print i
```

Funkcja **range** tworzy listę wartości całkowitych od zera do podanej wartości (jeżeli podamy 1 argument) lub od - do (jeżeli podamy 2 argumenty). Możemy podać też trzeci parametr określający przyrost (normalnie +1). **For** może też iterować inne typy danych, np napisy, listy:

```
#!/usr/bin/env python
bar = "Zdrabniamy literki"
for i in bar:
 print i

bar = ["foo", "bar", "yaz"]
for i in bar:
 print i
```

Jeżeli zakres ma być duży to lepiej użyć funkcji xrange, której efekt działania jest taki sam lecz nie tworzy ona listy od - do.

Słowniki

Słownik to w innych językach takich jak PHP tablica asocjacyjna (haszująca) zawierająca obiekty poindeksowane za pomocą kluczy.

```
#!/usr/bin/env python
bar = {"imie" : "jurek", "nazwisko" : "lepper"}
print bar["imie"]

for i in bar:
 print i + " - " +bar[i]
nazwa = { "klucz" : "wartość",  "klucz" : "wartość" }
```

Słowniki podobne są do list, by wyświetlić określony wpis wystarczy podać jego klucz - **nazwa_słownika["nazwa_klucza"]**. Podany przykład iteracji zwraca pod zmienną i nazwę klucza tak więc by wyświetlić i klucze i wartości musieliśmy użyć również **nazwa_słownika["nazwa_klucza"]**.
Próba odwołania się do nieistniejącego klucza spowoduje błąd wykonywania skryptu. Jeżeli może zdarzyć się sytuacja że klucz może nie istnieć to warto skorzystać z metody słownika **has_key**:

```
if bar.has_key("imie"):
      print bar["imie"]
```

lub

```
print bar.get("imie", "Brak klucza")
```

Pierwszy parametr metody get to nazwa klucza a drugi (opcjonalny, domyślnie - None) to reakcja w przypadku braku klucza. By uzyskać listę kluczy wystarczy użyć metody **keys()**. Do usuwania elementów słownika służy instrukcja **del**.

```
# -*- coding: utf-8 -*-
#!/usr/bin/env python
bar = {
 "imie" : "jurek",
 "nazwisko" : "lepper"
  }
# tworzymy listę kluczy i wyświetlamy
lista = bar.keys()
for i in lista:
 print i

#kasujemy jeden z kluczy
del bar["imie"]
print "
"
# tworzymy nową listę, tym razem nie ma klucza "imie"
lista = bar.keys()
for i in lista:
 print i
```

Funkcje

Funkcję definiujemy za pomocą instrukcji **def**. Oto przykład:

```
#!/usr/bin/env python
def nasza_funkcja(argument1, argument2):
  argumenty = argument1+" - "+argument2
  return argumenty

print nasza_funkcja("Poczatek", "Koniec")
def nazwa_funkcji(parametr, parametr, parametr):
  #kod funkcji
  return zmienna
  # lub
  return (tupla, tupla, bla, bla, bla)
```

Co do **return** - zwraca podane wyrażenie (zmienna, obiekt itp.). Jeżeli chcemy zwrócić więcej niż jedną zmienną to stosujemy tuplę (zmienna, zmienna, zmienna). Zmienne w definicji funkcji mogą mieć przypisane wartości domyślne (jeżeli w wywołaniu funkcji nie podamy wartości parametru to użyta zostanie wartość domyślna). Zmienne utworzone wewnątrz funkcji nie są dostępne poza nią, lecz można je zdefiniować jako zmienne globalne za pomocą operatora global:

```
#!/usr/bin/env python
def nasza_funkcja(argument1, argument2 = "Koniec"):
  global a
  argumenty = argument1+" - "+argument2
  a = 1
  return argumenty
print nasza_funkcja("Poczatek")
print a
```

Funkcja skorzysta z wartości domyślnej, zmienna "a" została zdefiniowana jako zmienna globalna i jest ogólnodostępna po wywołaniu funkcji.

Klasy

Instrukcja **class** pozwala definiować klasy - używane w programowaniu obiektowym. Klasy to w uproszczeniu zbiory funkcji powiązanych między sobą, co pozwala na tworzenie w łatwy i niezależny sposób wielu komponentów złożonej aplikacji przez wielu programistów.

```
#!/usr/bin/env python
class koszyk:
  def __init__ (self):
        self.koszyk = []
  def dodaj(self,obiekt):
        self.koszyk.append(obiekt)
  def rozmiar(self):
        return len(self.koszyk)

s = koszyk()
s.dodaj("pierwszy wpis")
s.dodaj("drugi wpis")
print s.rozmiar()
del s
```

Metody klasy definiowane są za pomocą **def**. Pierwszy argument każdej metody odnosi się do obiektu i zazwyczaj stosuje się nazwę "**self**". Wszystkie operacje wykorzystujące atrybuty obiektu muszą odwoływać się do nich poprzez tą zmienną. Metody poprzedzone **__** to metody specjalne, np **__init__** jest wykonywana przy utworzeniu obiektu klasy (wywołaniu klasy - **s = koszyk()**)

Moduły

W przypadku dużych projektów całego skryptu w jednym pliku raczej nie "zmieścimy". Istnieje potrzeba zastosowania szeregu plików z definicjami np. różnych funkcji. Utwórz plik **doda.py**:

```
#!/usr/bin/env python
def dodaj(a,b):
  return a+b
```

W drugim pliku wpisz:

```
#!/usr/bin/env python
import doda
print doda.dodaj(2, 2)
```

Gdzie "**import doda**" to **import NAZWA_PLIKU** (bez rozszerzenia .py). Wszystkie obiekty zawarte w tym module dostępne są poprzez jego nazwę - **NAZWA_PLIKU.dodaj(a,b)**.

Można też załadować określony element - **from doda import dodaj** lub też załadować wszystko do bieżącej przestrzeni nazw **from doda import ***

```
#!/usr/bin/env python
from doda import *
print dodaj(2, 2)
```