zaj1-blok1

October 22, 2015

1 Wprowadzenie do Pythona

(dla naukowców)

1.1 Uwaga

Używamy pythona **3.X.Y**, gdzie X $>$ 3.

Python 2.7 jest w ciągłym użyciu, ale prawie wszystko jest kompatybilne z 3.X a 2.X powinien umrzeć.

1.2 Ipython notebook

Do stworzenia tej prezentacji używam `ipython notebook`, interaktywnej konsoli pythona w przeglądarce (będzie o niej na trzecich? zajęciach).

Na razie uwierzcie że przykłady te są poprawnym kodem pythona.

2 Wprowadzenie do pythona

Python (dokładniej CPython) to język

- Skryptowy
- Interpretowany (nie do końca)
- Dynamiczny
- Wielo-paradygmatowy
- Bardzo wygodny dla programisty

```
In [15]: # W Pythonie krzyżyk oznacza kometarz do końca linii
         # Odpowiednik // z C/C++/Javy
         # Nie ma wielolinijkowych komentarzy, choć są odpowiedniki
```

```
In [16]: # W Pythonie zmienne nie mają typów
         foo = 1  # Definicja zmiennej, zawierająca liczbę całkowitą
         print(foo) # funkcja print wyświetla linijkę tekstu
```

```
1
```

```
In [1]: foo = 1
        foo = "foo" # Definicja ciągu znaków, jak widzicie jedna zmienna
        # może zmienić swój typ w trakcie życia (jest to nie polecane!)
        print(foo)
```

```
foo
```

```
In [17]: # Wartości zmiennych natomiast mają typ
         foo = 1
         print(type(foo))
```

```
<class 'int'>
```

```
In [18]: bar = "bar"
         print(type(bar))
```

```
<class 'str'>
```

```
In [19]: baz = foo + bar # Dynamiczny język nie oznacza że wszystko się rzutuje samo


         ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-19-96146fb7f6ff> in <module>()
    ----> 1 baz = foo + bar # Dynamiczny język nie oznacza że wszystko się rzutuje samo


        TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

2.1 Garbage collector

Python posiada narzędzia automatycznego zarządzania pamięcią, dokładnieje jest to garbage collector działający na zasadzie liczenia referencji z opcjonalnym rozbijaniem cykli.

Za każdym razem jak jakiś obiekt przypisywany jest do zmiennej wewnętrzny licznik odniesień rośnie o jeden. Jeśli licznik odniesień obiektu spada do zera obiekt jest kasowany.

Opcjonalnie Python potrafi kasować cykle obiektów. Jeśli `A` ma wskaźnik do `B` a `B` do `A` to nigdy liczba odniesień żadnego z tych obiektów nie spadnie do zera, i mówimy że `A` i `B` tworzą cykl. Python potrafi sobie z takimi sytuacjami radzić.

2.2 Garbage collector

```
In [20]: class Foo:
             def __del__(self):      # Metoda wywoływana podczas kasowania obiektu.
                 print("I'm dying") # powinna ona służyć **tylko** do zwolnienia pamięci
                                    # zaalokowanej przez ten obiekt poza metodami
                                    # allokacji w Pythonie (np. za pomocą wywołania malloc)
```

```
In [21]: foo = Foo()
         del foo # operator del kasuje zmienną
```

```
I'm dying
```

```
In [22]: def test():
             foo = Foo() #Zmienna w lokalnym kontekście umrze od razu

         test()
```

```
I'm dying
```

```
In [23]: foo = bar = Foo()
         del foo
```

```
In [24]: del bar
```

```
I'm dying
```

3 Typy danych

3.1 Ciąg znaków

std::string z C++, String z Javy. Domyślnie zawiera znaki w kodowaniu UTF-8.

```
In [25]: my_str = "Ciąg znaków"
         my_str = 'Ciag znaków' # Bez znaczenia czy mamy pojedyńczy czy powdójny cudzysłów!
         print(my_str)
```

```
Ciag znaków
```

```
In [26]: my_str = "Ciąg \nznaków" # Escapes działają jak w C/C++/Javie
         print(my_str)
```

```
Ciąg
znaków
```

```
In [27]: multiline_string = """
         Można stworzyć też wieloznakowe ciągi znaków
         O takie
         """
         print(multiline_string) #W wieloliniowych ciągach znaków
         #znaki nowej linii są zachowane
```

```
Można stworzyć też wieloznakowe ciągi znaków
O takie
```

```
In [3]: raw_string = r'\n\r\u1234' # W ciągach znaków zaczynających się od \r
        print(raw_string)          # escapes nie działają.
```

```
\n\r\u1234
```

4 Operacje na ciągach znaków

```
In [28]: print("foo" + "bar") # Operator + dokonuje konkatenacji znaków
```

```
foobar
```

```
In [29]: print("foobar"[3]) # Można wybierać znaki z ciągu znaków
```

```
b
```

```
In [30]: "bar"[2] = "z" # Ciągów nie można zmieniać


         ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

         <ipython-input-30-3c014a63ceda> in <module>()
    ----> 1 "bar"[2] = "z" # Ciągów nie można zmieniać


         TypeError: 'str' object does not support item assignment
```

```
In [31]: "{} ma {}".format("Ala", "kota") # Formatowanie wiadomości
```

```
Out[31]: 'Ala ma kota'
```

4.1 Lista

Lista to pojemnik przechowujący obiekty dostępne za pomocą indeksu. Można zmieniać zawartość tego pojemnika.

```
In [32]: empty_list = [] # lista (pusta)
         full_lits = [1, 2, 3, 4] # lista (z danymi)
         heterogenous = [1, 2, 3, "kotek"] # Może przechowywać dowolne dane
         print(heterogenous)
```

```
[1, 2, 3, 'kotek']
```

```
In [33]: empty_list.append(1)
         empty_list.extend([2, 3, 4, "kotek"])
         print(len(empty_list))
```

```
5
```

```
In [34]: print(empty_list)
```

```
[1, 2, 3, 4, 'kotek']
```

4.2 Indeksowanie

```
In [35]: my_list = list(range(10)) # list to konstruktor listy, range(N) zwraca **iterable** która zwra
         print(my_list)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [36]: my_list[0]
```

```
Out[36]: 0
```

```
In [37]: my_list[0] = 10
```

```
In [38]: my_list[0]
```

```
Out[38]: 10
```

```
In [39]: my_list[-1] # W pythonie kontenery listopodobne często pozwalają
         # Na indeksy negatywne. Oznacza to element N-ty od końca
```

```
Out[39]: 9
```

```
In [40]: my_list[-1] == my_list[len(my_list)-1] #inwariant
```

```
Out[40]: True
```

4.3 Usuwanie elementów z listy

```
In [41]: my_list = [1, 2, 3]
         del my_list[1] # Usuwamy waartość po indeksie
         print(my_list)
```

```
[1, 3]
```

```
In [42]: my_list = [1, 2, 3, 3, 2, 3]
         my_list.remove(3) # Usuwamy pierwsze pojawienie się wartości
         print(my_list)
```

```
[1, 2, 3, 2, 3]
```

4.4 Indeksowanie II (Slices)

Można łatwo tworzyć wycinki (slices) list. Są to obiekty zawierające podzbiór danych z danej listy.

```
In [43]: my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
         my_list[0:5] #Od pierwszego do piątego elementu

Out[43]: [1, 2, 3, 4, 5]

In [44]: my_list[5:] #Od piątego (wyłącznie) do końca

Out[44]: [6, 7, 8, 9]

In [45]: my_list[5:-3] # Od piątego do trzeciego od końca

Out[45]: [6]

In [46]: my_list[0:-1:2] # Od początku do końca co dwa elementy

Out[46]: [1, 3, 5, 7]

In [47]: my_list[::-1] # Od końca do początku w odwrotnej kolejności

Out[47]: [9, 8, 7, 6, 5, 4, 3, 2, 1]
```

4.5 Slices II

Z wykorzystaniem wycinków można również nadpisywać jakieś fragmenty listy:

```
In [48]: my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]
         print(my_list[1:-1])

[2, 3, 4, 5, 6, 7, 8]

In [49]: my_list[1:-1] = ["a", "b", "c"]
         print(my_list)

[1, 'a', 'b', 'c', 9]
```

4.6 Krotka

Krotka to pojemnik zawierający obiekty dostępne za pomocą indeksu. **Zawrtości krotki nie można zmieniać**

```
In [50]: my_tuple = (1, 3, 4, 5)  #
         my_tuple_2 = 5, 6 ,7 # Nawiasy można ominąć
         print(my_tuple_2)

(5, 6, 7)

In [51]: my_tuple = (1, ) #Jednoelementowe krotki są złośliwe: trzeba podać
         #"wiszący" przecinek na końcu
         print(my_tuple)

(1,)

In [52]: my_tuple_2[0] = 10
```

```
    ---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)

    <ipython-input-52-65dca6cd3416> in <module>()
----> 1 my_tuple_2[0] = 10


    TypeError: 'tuple' object does not support item assignment
```

5 Krotki tworzone w locie

```
In [53]: a = 10
         b = 1
         print((a, b)) #Tutaj powstaje krotka.
```

```
(10, 1)
```

```
In [54]: b, a = a, b # Tutaj powstają dwie krotki, które wymieniają się elementami
         print(a, b)
```

```
1 10
```

```
In [55]: opis = ("Pączuś", "kot", 10, 5)
         imie, gatunek, wiek_lata, masa_kg = opis # Typowe zastosowanie: "rozpakowanie" krotki
         print(imie)
```

```
Pączuś
```

6 Zbiór

Struktura danych zawierająca N elementów. Zawiera jedną kopię każdego elementu. Nie jest uporządkowana.

```
In [56]: my_set = {1, 2, 3, 4}
         5 in my_set # Tak na marginesie in działa też dla list i krotek!
```

```
Out[56]: False
```

```
In [57]: 1 in my_set
```

```
Out[57]: True
```

```
In [58]: my_list = [1 , 2, 3] # operator in działa dla wszystkich kolekcji!
         1 in my_list
```

```
Out[58]: True
```

6.1 Słownik

Zawiera odwzorowanie klucz -> wartość

```
In [59]: zwierze = {
             "imie": "Pączuś",
             "rasa": "kot",
             "masa": (10, "kg"), # Na marginesie: struktury ad-hoc są fajne w pythonie!
             "wiek": (10, "lat")
         }
```

```
In [60]: print(zwierze['imie']) # Słowniki ineksujemy kluczami
```

```
Pączuś
```

```
In [61]: zwierze["kolor oczu"] = "czarne jak noc" # Można dodawać elementy po stworzeniu słownika
         zwierze
```

```
Out[61]: {'kolor oczu': 'czarne jak noc',
          'rasa': 'kot',
          'masa': (10, 'kg'),
          'imie': 'Pączuś',
          'wiek': (10, 'lat')}
```

7 Klucze w słowniku

```
In [62]: klucz = 'foo'
         example = { # Kluczami słownika mogą być dowolne obiekty nie mogące
          1.5: "kotek",  # zmieniać wartości
          3: "pięc",
          klucz: "Tak też działa",
          (1, 2, 3): (4, 5, 6)
         }
         print(example)
```

```
{1.5: 'kotek', (1, 2, 3): (4, 5, 6), 3: 'pięc', 'foo': 'Tak też działa'}
```

```
In [63]: example[[1, 2, 3]] = 8 # Tablice są zmienne i nie mogą być kluczem
```

```
        ---------------------------------------------------------------------------
    TypeError                                 Traceback (most recent call last)

        <ipython-input-63-0bd8918e5c6a> in <module>()
    ----> 1 example[[1, 2, 3]] = 8 # Tablice są zmienne i nie mogą być kluczem


        TypeError: unhashable type: 'list'
```

7.1 Bloki kodu

W Pythonie bloki kody oznaczane są wcięciami

```
In [64]: for ii in range(3): # Drukropek
             print(ii) # Ta instrukcja wykonuje się w pętli
             print(";") # Ta też

         print("END") # A ta nie
```

```
0
;
1
;
2
;
END
```

7.2 Bloki kodu

Długość wcięcia nie ma znaczenia — musi być ono jednak takie samo dla każdej instrukcji

```
In [4]: for foo in range(2):
            for bar in range(2):
             print(foo, bar)
```

```
0 0
0 1
1 0
1 1
```

7.3 Bloki kodu (przykład niepoprawny)

```
In [66]: for foo in range(2):
             print(foo)
            for bar in range(3): # IndentationError
             print(bar)
```

```
  File "<ipython-input-66-633ca24cd76b>", line 3
    for bar in range(3): # IndentationError
                                           ^
IndentationError: unindent does not match any outer indentation level
```

7.4 Istnienie zmiennej

```
In [67]: if 'foo' in globals():
             del foo #Kasuje zmienną
```

```
In [68]: print("Zmienna foo istnieje jako globalna 0 {}".format('foo' in globals()))
         if True:
             print("Zmienna foo istnieje jako globalna 1 {}".format('foo' in globals()))
             foo = 5
             print("Zmienna foo istnieje jako globalna 2 {}".format('foo' in globals()))
         print("Zmienna foo istnieje jako globalna 3 {}".format('foo' in globals()))
```

```
Zmienna foo istnieje jako globalna 0 False
Zmienna foo istnieje jako globalna 1 False
Zmienna foo istnieje jako globalna 2 True
Zmienna foo istnieje jako globalna 3 True
```

DETAL: Z pozostałymi konstrukcjami (`if`, `for`, `while`, `try-catch`) jest podobnie: zmienne w nich zdefiniowane wychodzą trwają nawet po zakończeniu danego bloku. Wyjątkiem jest `catch` – złapany wyjątek umiera

8 Iterowanie

```
In [69]: my_list = [1, 2, 4, "kotek"]

         for element in my_list: #Domyślnie pętla for działa jak for-each
             print(element) # Ciało wykonuje się raz dla każdego elementu listy.
             # Potem powiem po czym poznajemy w pythonie blok kodu!
```

```
1
2
4
kotek
```

```
In [70]: for ii, element in enumerate(my_list): # Możemy też iterować po elementach znając ich indeksy
             print(ii, element) #Służy do tego funkcja enumerate
```

```
0 1
1 2
2 4
3 kotek
```

```
In [71]: second_list = [9, 1, 3] #Można też iterować po dwóch listach na raz
         for a, b in zip(my_list, second_list): # Służy do tego funkcja zip
             print(a, b)
```

```
1 9
2 1
4 3
```

9 Obiekt range

range jest funkcją zwracającą generator, czyli obiekt po którym można przeiterować raz.

```
In [72]: range(5)
```

```
Out[72]: range(0, 5)
```

```
In [73]: range(10000000000000000000000000000000000) # Range nie przechowuje swoich elementów w pamięci
```

```
Out[73]: range(0, 10000000000000000000000000000000000)
```

```
In [74]: for x in range(5):  #Żeby wyświetlić "zawartość" range trzeba po nim przeiterować
             print(x)
```

```
Out[74]: [0, 1, 2, 3, 4]
```

10 Obiekt range

```
In [5]: list(range(5, 10))
```

```
Out[5]: [5, 6, 7, 8, 9]
```

```
In [6]: list(range(10, 5, -1))
```

```
Out[6]: [10, 9, 8, 7, 6]
```

```
In [7]: range(1, 10, 0.1) # Range iteruje po intach
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-7-f14a1061500c> in <module>()
----> 1 range(1, 10, 0.1) # Range iteruje po intach


TypeError: 'float' object cannot be interpreted as an integer
```

10.1 Iterowanie (słowniki)

```
In [78]: zwierze = {
             "imie": "Pączuś",
             "rasa": "kot",
             "masa": (10, "kg"),
             "wiek": (10, "lat")
         }

In [79]: for key in zwierze: # Domyślnie iterujemy po kluczach
             print(key)

rasa
masa
imie
wiek

In [80]: for key, value in zwierze.items(): #Ładny przykład rozpakowania krotki
             print("{}: {}".format(key,  value))

rasa: kot
masa: (10, 'kg')
imie: Pączuś
wiek: (10, 'lat')
```

Z przyczyn bezpieczeństwa (o dokładne przyczyny proszę pytać na konsultacjach) kolejność iterowania po słownikach (i zbiorach) zmienia się pomiędzy różnymi instancjami interpretera,

10.2 Iterowanie (zbiory)

```
In [8]: for x in {'a', 'b', 'c', 'd'}:
            print(x) # Kolejność przypadkowa

d
c
a
b
```

10.3 Instrukcje warunkowe

```
In [82]: if 1 == 1:
             print("Działa")

Działa

In [9]: if 1 == 2:
            print("Oops")
        else:
            print("Działa")

Działa
```

Dwa słowa kluczowe: `True` oraz `False` pozwalają użyć zmiennych logicznych explicite.

```
In [84]: if True:
             pass
```

10.4 Wyrażenia logiczne

```
In [85]: True or False
```

```
Out[85]: True
```

```
In [86]: True and False
```

```
Out[86]: False
```

```
In [87]: True and not False
```

```
Out[87]: True
```

```
In [88]: 'foo' in ['foo', 'bar']
```

```
Out[88]: True
```

```
In [89]: 'foo' not in ['foo', 'bar']
```

```
Out[89]: False
```

10.5 Matematyka

```
In [90]: print(1 + 2 +3 )
```

```
6
```

```
In [91]: import math # Math to moduł pythona, należy go zaimportować.
         print(math.sin(math.pi))
```

```
1.2246467991473532e-16
```

```
In [92]: print(1 / 2 ) # Domyślnie dzielenie intów zwraca floata
```

```
0.5
```

```
In [93]: print(1 // 2) # Teraz dzielenie znane z C++
```

```
0
```

```
In [94]: 2**1024 #Zmienne stałoprzecinkowe mają domyślnie nieograniczony rozmiar!
```

```
Out[94]: 179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322400
```

10.6 Uruchamianie

python nazwa_pliku.py

Na zajęciach numer 1 i 2 proszę korzystać z interpretera w `/opt/python3.4/bin/python`.

```
In [12]: %%bash
         # Z Ipythona można wykonywać też skrypty Basha
         cd /tmp/
         echo 'print("Hello World")' > hello.py
         python hello.py
```

```
Hello World
```

Można też krócej:

```
In [98]: %%bash

         python -c 'print("Hello World")'
```

```
Hello World
```