Programowanie w Pythonie wprowadzenia ciąg dalszy

Łukasz Mioduszewski, UKSW 2022

Interpreter

- Sprawdzanie wersji: python --version
- Wychodzenie z interpretera: quit() lub ctrl+d (albo zależnie od systemu ctrl+z)
- Wywoływanie:
  - python –c "pojedyncze polecenie"
  - python –i skrypt.py
    >>>> # tryb interaktywny
- Wykonywalny skrypt pod linuxem ma w 1. linii: #!/usr/bin/env python3

Ostatni wynik w trybie interaktywnym

- To co zostało ostatnio wypisane na ekran jest zachowywane w "zmiennej" _

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

Importowanie pojedynczych komend

- Zamiast importować cały moduł, np. import math, można tylko 1 funkcję, np. from math import sin
- Wtedy można tej funkcji użyć nie poprzedzając jej "math.", np. print(sin(x))
- Ale nie mamy wtedy dostępu do innych funkcji, chyba że zaimportujemy więcej: from math import sin, sqrt

Stringi

- Długość obliczamy funkcją len, np. len('pyton') # zwraca 5
- Wewnątrz cudzysłowów działa \n ale już # to nie komentarz tylko zwykły znak
- Można używać ' ' wewnątrz " " i odwrotnie, czyli " " wewnątrz ' '
- Jeśli chcesz użyć " " wewnątrz " " musisz użyć \, czyli np. "test \"pierwszy\""
- Aby użyć \ jako \ użyj r przed stringiem, np. print(r'C:\some\name’)
- Wiele linii w stringu bez \n osiągniesz poprzez potrójny „ albo ,
  print("""\
  Usage: thingy [OPTIONS]
  -h Display this usage message""") # \ pomija 1 nową linię

Indeksy

- Ujemne indeksy oznaczają liczenie od tyłu, np.
  >>> 'pyton'[-2] # zwraca 'o'
  >>> 'pyton'[-2:] # zwraca 'on'
- Zawsze pierwsza liczba w zakresie jest wliczana a ostatnia nie, np.
  word = 'pyton’
  >>> word[:3] + word[3:] # zwraca 'pyton'
- Cięcie listy lub stringa nie musi używać liczb z zakresu, np.
  >>> 'pyton'[2:42] # zwraca 'ton'

Listy (cdn.)

- Też można je łączyć, np.
  >>>x = [1,2,3] + [4,5,6] # x wynosi teraz [1,2,3,4,5,6]
- Też działa dla nich funkcja len, np. len([1,2,3]) zwraca 3
- Można modyfikować całe fragmenty list, np.
  >>> x[2:4]=[7] # x wynosi teraz [1,2,7,5,6]
- Listy mogą być wewnątrz list, np.
  >>> y=[x,'abc',42,5,x]
  >>> print(y) # wypisuje [[1, 2, 7, 5, 6], 'abc', 42, 5, [1, 2, 7, 5, 6]]
  >>> print(y[1][1]) # wypisuje b
  >>> y[0][1]=18
  >>> print(y) # wypisuje [[1, 18, 7, 5, 6], 'abc', 42, 5, [1, 18, 7, 5, 6]]

Zbiory (set)

- Tak jak słowniki są tworzone w nawiasach klamrowych, ale bez dwukropków: zbior = {'a','b',123}
- Też można liczyć ich elementy funkcją len() # len(zbior) wynosi 3
- Można sprawdzać czy coś jest w zbiorze: if 'a' in zbior: …
- Operatory >, <, <=, >= sprawdzają czy jeden zbiór zawiera się w drugim
- Operatory |, &, -, ^ tworzą odpowiednio sumę, iloczyn, różnicę i różnicę symetryczną

Wejście wyjście

- print nie musi kończyć się nową linią, np.
  >>> for i in range(2):
  … print('a',i, end=',')
  …
  a 0,a 1,>>># tu można wpisać nowe polecenie
- Aby wczytać coś w trakcie wykonywania skryptu, możemy użyć input(), np.
  x = input('Enter your name:')
  print('Hello, ' + x)

Iterowanie po wielu rzeczach naraz

- Iterator może być tuplą, np. gdy iterujemy po słowniku:

```
# tworzymy słownik
users = {'Hans': 'active', 'Éléonore': 'inactive', '景太郎': 'active'}
# iterujemy po kopii słownika
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]
```

A co gdy chcemy iterować po indeksie?

- Można użyć funkcji range (której wynik jest iterowalny, ale nie jest listą)

```
>>> a = ['Ala', 'ma', 'kota']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Ala
1 ma
2 kota
```

A co gdy chcemy iterować po indeksie?

- Można użyć funkcji enumerate, np.

```
>>> seasons = ['Spring', 'Summer', 'Fall', 'Winter']
>>> list(enumerate(seasons))
[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]
```

Instrukcja break wychodzi z pętli

- Inny program do znajdowania liczb pierwszych:

```
import math
for n in range(3, 122):
    for x in range(2, math.ceil(1+math.sqrt(n))):
        if n % x == 0:
            print(n, 'jest równe', x, '*', n//x)
            break
    else: # gdy nie znaleziono dzielnika
        print(n, 'jest liczbą pierwszą')
```

Naprawdę każda zmienna jest referencją

- Przykład funkcji która dziwnie się zachowuje:

```
def f(a, L=[]):
    L.append(a)
    return L
print(f(1)) # wypisze [1]
print(f(2)) # wypisze [1,2]
```

- Jak tego uniknąć:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Instrukcja match (dopiero od Python 3.10)

- Więcej niż switch:

```
match punkt:
    case (0, 0):
        print("Początek układu współrzędnych")
    case (0, x) | (x,0):
        print("Jedna współrzędna wynosi 0 a druga",x)
    case (x, y):
        print("Jedna współrzędna wynosi ",x,"a druga",y)
    case _:
        raise ValueError("To nie jest punkt")
```

Przerwa

- Instrukcja pass nic nie robi

Niektóre wbudowane funkcje

- any oraz all zwracają True, jeśli którykolwiek (any) albo wszystkie (all) elementy iterowalnego argumentu są True, np.

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

- max oraz min są wbudowane (i też mają opcjonalny argument key)
- sorted() to stabilna funkcja sortująca
- sum() sumuje wszystkie elementy argumentu

Rozpakowywanie argumentów

- Czasem mamy argumenty w jednym iterowalnym obiekcie, a chcemy je podać funkcji oddzielnie. Wtedy używamy * przed nazwą obiektu, np.

```
>>> args = [3, 6]
>>> list(range(*args))            # call with arguments unpacked from a list
[3, 4, 5]
```

- Ze słownika możemy wyciągnąć i nazwę, i wartość argumentu – wtedy potrzeba dwóch gwiazdek:

```
def sklep(klient,sprzedawca):
    print("sprzedawca:", sprzedawca, "klient:",klient)
dane = {"sprzedawca": "Jan Kowalski", "klient": "Adam Nowak"}
sklep(**dane) # wypisuje sprzedawca: Jan Kowalski, klient: Adam Nowak
```

Funkcje o zmiennej liczbie argumentów

- Możemy napisać funkcję tak żeby sama rozpakowywała argumenty:

```
def sklep(produkt, *args, **kwargs): # kwargs to nazwane argumenty
    print("Czy jest", produkt, "?")
    for arg in args:
        print("mamy",arg,end=", ")
    print("i więcej.\n","-" * 40)
    for kw in kwargs:
        print(kw, ":", kwargs[kw])
sklep("miód",
        "miód","masło","chleb",
        sprzedawca="Jan Kowalski",
        klient="Adam Nowak")
```

Mniej swobody dla argumentów

- Za tutorialem Pythona:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                   |
        |        Positional or keyword    |
        |                                 - Keyword only
         -- Positional only
```

- To co przed / jest tylko pozycyjne, to po / jest dowolne, to po * jest tylko nazywane

```
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
>>> combined_example(1, 2, kwd_only=3)
1 2 3
```

Zawijanie długich linii

- Używamy \ aby zawijać długie linie, tzn. aby 1 instrukcja zajęła wiele wierszy, np.

```
s_var = 'xxx'
s = 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'\
   + s_var\
   + 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb'
print(s)
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaxxxbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Zawijanie długich linii

- Zamiast \ można użyć dowolnych nawiasów, np.

```
s_var = 'xxx'
s = ('aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa'
   + s_var
   + 'bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb')
print(s)
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaxxxbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Standard PEP8

- Wcięcia z 4 spacji
- Linie krótsze niż 79 znaków (można zawijać)
- Puste linie między funkcjami, klasami i dużymi fragmentami kodu
- Komentarze raczej na oddzielnej linii
- Komentuj funkcje (potrójny cudzysłów tuż po nazwie)
- Spacje przy operatorach i po przecinkach, ale nie po nawiasach, np. a = f(1, 2) + g(3, 4)
- TylkoPierwszaLitera dla klas oraz male_litery_z_podkreslnikiem dla funkcji
- Używaj self jako pierwszego argumentu dla każdej metody (patrz następny wykład)
- Używaj UTF8 albo ograniczaj się do znaków ASCII, ogólnie unikaj niestandardowych znaków

Funkcje lambda

- Lambda oznacza funkcję składającą się tylko z jednej instrukcji. Jej format to:
  - lambda zmienna1,zmienna2,…,zmiennan: jakaś wartość zależna od tych zmiennych
  - lambda zwraca **funkcję** zwracającą tę wartość, np.

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42) # f to funkcja
>>> f(0) # zwraca 42
>>> f(1) # zwraca 43
```

Funkcje lambda i sortowanie

- Funkcje lambda nie mają nazwy ani słowa "return"
- Funkcji lambda można użyć do sortowania listy po czymś co jest funkcją jej elementów:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> print(pairs) # wypisuje [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

- Inny przykład: wartość bezwzględna, wtedy key=lambda x: abs(x)

Iteratory

- Iterator pozwala znaleźć następny element, każdy iterowalny obiekt go ma
- Można uzyskać iterator funkcją iter, np.
  myit=iter('pyton')
- Można dostać wartość następnego elementu, zarazem iterując, funkcją next, np.
  print(next(myit)) # wypisuje p
  print(next(myit)) # wypisuje y
- Uwaga, po 'n' wyrzuci błąd StopIteration, dodaj do next domyślną wartość aby tak nie robił, np.
  print(next(myit,None))

Funkcja zip

- Tworzy iterator tupli z wielu iterowalnych obiektów (tuple mają po jednym elemencie z każdego z nich):
  ```
  >>> list(zip(range(3), ['a', 'b', 'c', 'd']))
  [(0, 'a'), (1, 'b'), (2, 'c')]
  ```
- *zip robi coś odwrotnego:
  ```
  >>> x = [1, 2, 3]
  >>> y = [4, 5, 6]
  >>> z = list(zip(x, y)) # [(1, 4), (2, 5), (3, 6)]
  >>> x2, y2 = zip(*z)   # gdyby zrobić list(zip(*z)), dostalibyśmy [(1,2,3),(4,5,6)]
  >>> x == list(x2) and y == list(y2) # zwraca True
  ```

Funkcja filter

- filter(*function,iterable*) zwraca iterator który iteruje po tych elementach obiektu *iterable*, dla których funkcja *function* zwraca True. Przykład:

```
liczby = list(range(1,11))

def czy_parzysta(liczba):
    """returns True if number is even"""
    return liczba % 2 == 0

iterator_po_parzystych = filter(czy_parzysta, liczby)
parzyste = list(iterator_po_parzystych)
print(parzyste)  # Wypisuje [2, 4, 6, 8, 10]
```

Funkcja filter lubi się z funkcją lambda

- filter(*function,iterable*) zwraca iterator który iteruje po tych elementach obiektu *iterable*, dla których funkcja *function* zwraca True. Przykład:

```
liczby = list(range(1,11))
```

```
iterator_po_parzystych = filter(lambda x: x % 2 == 0, liczby)
parzyste = list(iterator_po_parzystych)
print(parzyste)  # Wypisuje [2, 4, 6, 8, 10]
```

Python to prawie jak angielski

- Przykład na koniec:

```
letters = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

def filter_vowels(letter):
    """a function that returns True if letter is vowel"""
    vowels = ['a', 'e', 'i', 'o', 'u']
    return True if letter in vowels else False

filtered_vowels = filter(filter_vowels, letters)
vowels = tuple(filtered_vowels)
print(vowels) # ('a','e','i','o')
```