Python

Poziom średniozaawansowany

Autor: Michał Nowotka

Plan na dzisiaj

1. Moduły
2. Funkcje – zagadnienia zaawansowane
3. Iteratory i generatory

MODUŁY

Importowanie modułów przez inne moduły.

- Załóżmy, że w bieżącym katalogu mamy dwa pliki:
  - **my_module.py**
  - **intro.py**
- W pliku **my_module.py** umieszczamy:
  - polecenie **print**
  - deklarację zmiennej o nazwie **test**
  - deklarację funkcji **find_index**, która zwraca indeks elementu **target** jeśli występuje on na liście **to_search** lub -1 w przeciwnym przypadku.
- W pliku **intro.py** mamy z kolei deklarację zmiennej będącej listą.
- Na tej liście chcielibyśmy wyszukać pewien element, zatem przydałaby nam się bardzo funkcja **find_index** z moduły **my_module**.
- Zaimportujmy ją i zobaczmy co się stanie.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   import my_module
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = my_module.find_index(courses, 'Math')
5   print(index)
```

```
MacBook-Pro-Micha …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Po wykonaniu pliku **intro.py** poprawnie została wypisana jedynka, jako indeks elementu **‘Math’** na liście **courses**.
- Dodatkowo polecenie import **my_module** sprawiło, że został wypisany napis **‘imported my module...’**
- Tym razem zaimportowaliśmy cały moduł **my_module** a nie pojedynczą klasę lub funkcję, dlatego aby dostać się do funkcji z zaimportowanego modułu musieliśmy użyć operatora kropki (**.**).
- Udało nam się zaimportować moduł **my_module** w module **intro** ponieważ oba pliki leżą w tym samym katalogu.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   import my_module
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = my_module.find_index(courses, 'Math')
5   print(index)
```

```
MacBook-Pro-Micha …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Gdybyśmy uznali, że nazwa modułu **my_module** jest zbyt długa by poprzedzać nią wywołanie każdej funkcji, można skrócić tę nazwę w środku modułu **intro** używając polecenia **import my_module as mm**.
- Oznacza ono: zaimportuj moduł **my_module** jako **mm**. Od tej pory w module **intro** do wszystkich składowych modułu **my_module** będziemy się odnosić poprzez nazwę **mm** ponieważ stała się ona lokalnym aliasem (przezwiskiem) nazwy **my_module**.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   import my_module as mm
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = mm.find_index(courses, 'Math')
5   print(index)
```

```
MacBook-Pro-Micha  …/technology
› python intro.py
imported my module...
1
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Oczywiście nie musimy importować całego modułu tylko jego wybrane składowe.
- Jeśli zamiast całego modułu **my_module** zaimportujemy tylko funkcję **find_index** utracimy dostęp do zmiennej test, również zadeklarowanej w tym module.
- Dzięki temu nie będziemy musieli poprzedzać nazwy funkcji **find_index** nazwą modułu.
- W takim przypadku możemy oczywiście zaimportować również zmienną test wymieniając ją po przecinku.

my_module.py

```
1   print('imported my module...')
2
3   test = 'Test string'
4
5
6   def find_index(to_search, target):
7       for index, value in enumerate(to_search):
8           if value == target:
9               return index
10      return -1
```

intro.py

```
1   from my_module import find_index, test
2
3   courses = ['History', 'Math', 'Physics', 'CompSci']
4   index = find_index(courses, 'Math')
5   print(index)
6   print(test)
```

```
MacBook-Pro-Micha  …/technology
› python intro.py
imported my module...
1
Test string
```

Autor: Michał Nowotka

Importowanie modułów przez inne moduły.

- Jeśli chcemy zaimportować wszystkie atrybuty danego modułu możemy użyć operatora gwiazdki.
- Należy przy tym pamiętać, że nie jest to zalecana praktyka i może się stać przyczyną problemów jeśli w naszym pliku lub innym module, który importujemy w podobny sposób nastąpi kolizja nazw.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
```

```
MacBook-Pro-Micha  …/technology
› python intro.py
imported my module...
1
Test string
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- Powiedzieliśmy już, że moduł **intro** potrafi zaimportować moduł **my_module** ponieważ oba pliki leżą w tym samym katalogu.
- A co by się stało gdyby leżały w innych katalogach? W jaki sposób Python odnajduje moduły?
- Python szuka modułów w lokalizacjach określonych przez **sys.path**.
- Lokalizacje te można bez problemu wyświetlić, wystarczy zaimportować standardowy moduł **sys**.
- W odpowiedzi dostaniemy listę lokalizacji, w których Python szuka modułów - według ich kolejności na liście określonej przez **sys.path**.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

```
MacBook-Pro-Micha  .../technology                                  02:13  23:39:19
↳ python intro.py
imported my module...
1
Test string
['/Users/michalnowotka/PycharmProjects/id_validator/technology', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packages']
```

Skąd Python wie gdzie szukać modułów?

- Pierwszym elementem na liście **sys.path** będzie zawsze katalog bieżący.
- Dalej znajdą się na niej elementy pochodzące ze zmiennej systemowej o nazwie **PYTHONPATH**, o której powiemy na kolejnych slajdach.
- Potem znajdą się lokalizacje standardowych bibliotek Pythona.
- Na końcu znajdą się wszystkie dodatkowe zainstalowane biblioteki.

my_module.py

```
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

```
MacBook-Pro-Micha ► _/technology                    02:13  ✔  23:39:19
↳ python intro.py
imported my module...
1
Test string
['/Users/michalnowotka/PycharmProjects/id_validator/technology', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python37.zip', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7', '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/lib-dynload', '/usr/local/lib/python3.7/site-packages']
```

Skąd Python wie gdzie szukać modułów?

- Przenieśmy moduł **my_module** z bieżącego katalogu na pulpit i wykonajmy plik **intro.py** jeszcze raz.
- Dostaniemy błąd, informujący o tym, że nie znaleziono modułu o nazwie **my_module**…
- Co możemy zrobić w tej sytuacji? Istnieje kilka możliwości.

```
MacBook-Pro-Micha  …/technology
› python intro.py
Traceback (most recent call last):
  File "intro.py", line 2, in <module>
    from my_module import *
ModuleNotFoundError: No module named 'my_module'
```

my_module.py

```python
print('imported my module...')

test = 'Test string'


def find_index(to_search, target):
    for index, value in enumerate(to_search):
        if value == target:
            return index
    return -1
```

intro.py

```python
import sys
from my_module import *

courses = ['History', 'Math', 'Physics', 'CompSci']
index = find_index(courses, 'Math')
print(index)
print(test)
print(sys.path)
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- Po pierwsze, skoro **sys.path** jest listą lokalizacji, które są przeszukiwane to możemy do niej dodać nowy wpis dynamicznie podczas pracy programu.
- To nie jest najlepsze rozwiązanie bo musielibyśmy dopisywać nową lokalizację na samej górze każdego pliku, który korzysta z tego modułu.
- Możemy dodać nową lokalizację do zmiennej systemowej **PYTHONPATH**. **PYTHONPATH** jest zmienną, która mówi Pythonowi gdzie ma szukać dodatkowych modułów. Sposób ustawiania zmiennych systemowych różni się pomiędzy systemami operacyjnymi - inaczej robi się to pod Linuxem, inaczej pod Windowsem.
- Jeśli nie chcemy stać przed wyborem czy ustawić zmienną **PYTHONPATH** na czas trwania pojedynczej sesji, czy też ustawić jej wartość w całym systemie operacyjnym, możemy ją ustawić w PyCharmie lub innym IDE.

```
MacBook-Pro-Micha …/technology
↳ python intro.py
Traceback (most recent call last):
  File "intro.py", line 2, in <module>
    from my_module import *
ModuleNotFoundError: No module named 'my_module'

MacBook-Pro-Micha …/technology
↳ export PYTHONPATH=~/Desktop

MacBook-Pro-Micha …/technology
↳ python intro.py
imported my module...
1
Test string
```

Skąd Python wie gdzie szukać modułów?

- Na koniec warto zauważyć, że Python zawsze zna położenie swoich bibliotek standardowych. Być może któraś z nich implementuje już funkcjonalność, którą chcemy uzyskać.
- Nasz moduł **my_module** dostarcza funkcji, która zwraca indeks szukanego elementu na liście jeśli ten element na niej występuje lub -1 w przeciwnym przypadku.
- Tak się składa, że każda lista posiada metodę **index**, która robi dokładnie to samo, zatem nie potrzebujemy jej już implementować.

```
MacBook-Pro-Micha …/technology
python intro.py
1
```

intro.py

```
1  courses = ['History', 'Math', 'Physics', 'CompSci']
2  index = courses.index('Math')
3  print(index)
4
```

Autor: Michał Nowotka

Skąd Python wie gdzie szukać modułów?

- To, że nie musimy znać położenia plików z biblioteki standardowej nie oznacza wcale, że nie możemy ich poznać. Python w żaden sposób nie ukrywa ich położenia.
- Każdy moduł posiada specjalny atrybut **__file__**, którego wartością jest ścieżka do jego lokalizacji na dysku.
- Możemy więc wypisać w konsoli wartość tego atrybutu i otworzyć w edytorze plik spod wskazanej ścieżki żeby zobaczyć co się w nim kryje.
- Moduły biblioteki standardowej są oczywiście zwykłymi plikami Pythona więc można je bez problemów czytać i można się z nich wiele nauczyć.

```
In [1]: import antigravity

In [2]: antigravity.__file__
Out[2]: '/usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/antigravity.py'

In [3]: !cat /usr/local/Cellar/python/3.7.3/Frameworks/Python.framework/Versions/3.7/lib/python3.7/antigravity.py

import webbrowser
import hashlib

webbrowser.open("https://xkcd.com/353/")

def geohash(latitude, longitude, datedow):
    '''Compute geohash() using the Munroe algorithm.

    >>> geohash(37.421542, -122.085589, b'2005-05-26-10458.68')
    37.857713 -122.544543

    '''
    # https://xkcd.com/426/
    h = hashlib.md5(datedow).hexdigest()
    p, q = [('%f' % float.fromhex('0.' + x)) for x in (h[:16], h[16:32])]
    print('%d%s %d%s' % (latitude, p[1:], longitude, q[1:]))
```

Autor: Michał Nowotka

FUNKCJE – ZAGADNIENIA ZAAWANSOWANE

Funkcje - domyślne argumenty

- W Pythonie funkcja może posiadać domyślne argumenty.
- W takim przypadku w sygnaturze funkcji, możemy wymieniając argumenty dać znak równości i podać domyślny argument.
- Argumenty bez domyślnych wartości muszą poprzedzać argumenty z wartościami domyślnymi, nie można mieszać tych dwóch rodzajów.
- Wywołując funkcję można ograniczyć się do podania parametrów, które nie mają domyślnych wartości albo podać również te, które taką domyślną wartość posiadają aby ją nadpisać.

```
def add(a, b=1):
    return a+b
```

```
In [2]: add(5)
Out[2]: 6

In [3]: add(5, 3)
Out[3]: 8
```

Funkcje - domyślne argumenty

- W Pythonie funkcja może posiadać domyślne argumenty.
- W takim przypadku w sygnaturze funkcji, możemy wymieniając argumenty dać znak równości i podać domyślny argument.
- Argumenty bez domyślnych wartości muszą poprzedzać argumenty z wartościami domyślnymi, nie można mieszać tych dwóch rodzajów.
- Wywołując funkcję można ograniczyć się do podania parametrów, które nie mają domyślnych wartości albo podać również te, które taką domyślną wartość posiadają aby ją nadpisać.

```python
from textwrap import wrap


def format_phone_number(number, area_code='+48', delimiter='-'):
    wrapped_number = delimiter.join(wrap(number, 3))
    return f'{area_code} (0) {wrapped_number}'
```

```
In [2]: format_phone_number()
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-2-91946334e5e4> in <module>
----> 1 format_phone_number()

TypeError: format_phone_number() missing 1 required positional argument: 'number'

In [3]: format_phone_number('697120906')
Out[3]: '+48 (0) 697-120-906'

In [4]: format_phone_number('697120906', '+44')
Out[4]: '+44 (0) 697-120-906'

In [5]: format_phone_number('697120906', '+44', ' ')
Out[5]: '+44 (0) 697 120 906'
```

Autor: Michał Nowotka

Funkcje - argumenty pozycyjne i nazwane

- Kiedy wywołujemy funkcję możemy przekazać jej argumenty w takiej kolejności, w jakiej zostały zadeklarowane. Nie musimy wtedy podawać nazw parametrów.
- Czasami jednak zależy nam by podać argumenty dla parametrów funkcji w specyficznej kolejności. Może tak być ze względu na poprawę czytelności kodu (jeśli obok argumentu podamy parametr, którego ten argument dotyczy to oszczędzimy czytelnikom naszego kodu konieczności zaglądania do deklaracji funkcji) lub, ponieważ chcemy nadpisać domyślną wartość parametru, który w kolejności stoi później niż inny domyślny parametr, dla którego nie chcemy zmieniać domyślnej wartości.

```
In [2]: format_phone_number(area_code='+44', number='697120906')
Out[2]: '+44 (0) 697-120-906'

In [3]: format_phone_number('697120906', delimiter=' ')
Out[3]: '+48 (0) 697 120 906'
```

Autor: Michał Nowotka

Funkcje - argumenty pozycyjne i nazwane

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
    def work_hard(self):
        pass
```

```
def give_a_raise(employee, factor=1.1):
    employee.salary *= factor
```

```
employee = Employee("Joe", 10000)
```

1

```
In [6]: give_a_raise(employee)

In [7]: print(employee.salary)
11000.0
```

```
In [9]: give_a_raise(employee, 1.5)

In [10]: print(employee.salary)
15000.0
```

2

```
In [12]: give_a_raise(employee, factor=2)

In [13]: print(employee.salary)
20000
```

Funkcje - argumenty pozycyjne i nazwane

Zmodyfikujmy poprzednio napisane funkcje, tak by pojawiły się parametry domyślne i sprawdźmy jak to działa i wpływa na działanie kodu.

Funkcje - argumenty pozycyjne i nazwane

Napisz funkcję, która dodaje do siebie dwie liczby podane przez użytkownika

Funkcje - argumenty pozycyjne i nazwane

Napisz funkcję, która dodaje do siebie **trzy** liczby podane przez użytkownika

Funkcje - argumenty pozycyjne i nazwane

Napisz funkcję, która dodaje do siebie **pięć** liczb podanych przez użytkownika

Funkcje - argumenty pozycyjne i nazwane: *args i **kwargs

- Czasami pisząc funkcję chcemy dać możliwość wywołania jej z dowolną ilością nazwanych i pozycyjnych argumentów.
- Wtedy zamiast wymieniać wszystkie pozycyjne parametry możemy napisać ***args**.
- Dla nazwanych parametrów zbiorczą nazwą jest ****kwargs**.
- Należy pamiętać, że **args** i **kwargs** to tylko tradycyjne nazwy tych parametrów (podobnie jak **self**, czy **cls** w klasach) a o tym czy pełnią swoją rolę decydują gwiazdki przed nazwą, a nie sama nazwa.

```
In [1]: paste
def my_sum(*args):
    ret = 0
    for arg in args:
        ret += arg
    return ret

## -- End pasted text --

In [2]: my_sum(1,2,3,4)
Out[2]: 10
```

```
In [1]: paste
def my_sum(**kwargs):
    ret = 0
    for key in kwargs:
        ret += kwargs[key]
    return ret

## -- End pasted text --

In [2]: my_sum(eggs=3, spam=4, cheese=7)
Out[2]: 14
```

```
In [1]: paste
def sum_all(*args, **kwargs):
    ret = 0
    for arg in args:
        ret += arg
    for key in kwargs:
        ret += kwargs[key]
    return ret

## -- End pasted text --

In [2]: sum_all(1, 2, 3, eggs=3, spam=4, cheese=7)
Out[2]: 20
```

Autor: Michał Nowotka

Operatory * oraz ** w wywołaniu funkcji

- Może się zdarzyć, że wszystkie argumenty pozycyjne niezbędne do wywołania funkcji mamy już zgromadzone w postaci listy.
- Podobnie może się okazać, że wszystkie nazwane argumenty trzymamy już w jakimś słowniku.
- W takim przypadku zamiast iterować po liście słowniku, aby przypisać każdy argument do innej zmiennej tylko po to, aby wywołać funkcję, możemy użyć operatorów * i **.
- Przykład wart jest więcej niż tysiąc słów.

```
In [2]: positional_data = ['697120906', '+44', ' ']

In [3]: format_phone_number(*positional_data)
Out[3]: '+44 (0) 697 120 906'

In [4]: keyword_data = {'area_code': '+44', 'number': '697120906', 'delimiter': ' '}

In [5]: format_phone_number(**keyword_data)
Out[5]: '+44 (0) 697 120 906'
```

Autor: Michał Nowotka

Rozpakowywanie kolekcji - operator *

Zdefiniuj poniższe funkcje i sprawdź ich działanie.

```
def f1(a, b): print(a, b)
def f2(a, *b): print(a, b)
def f3(a, **b): print(a, b)
def f4(a, *b, **c): print(a, b, c)
def f5(a, b=2, c=3): print(a, b, c)
def f6(a, b=2, *c): print(a, b, c)
```

Rozpakowywanie kolekcji

- Jeśli mamy krotkę z dużą ilością elementów, zamiast odnosić się do nich po indeksie, możemy je rozpakować do zmiennych, które będą się sensownie nazywać.
- To samo można zrobić z listami.
- Rozpakowywania można również użyć w wyrażeniach.

```
In [1]: numbers_info = ('697120906', '+48', '-')

In [2]: number, area_code, delimiter = numbers_info

In [3]: number
Out[3]: '697120906'

In [4]: area_code
Out[4]: '+48'

In [5]: delimiter
Out[5]: '-'

In [6]: shopping_list = ['apples', 'organges', 'bananas']

In [7]: apples, orages, bananas = shopping_list

In [8]: apples
Out[8]: 'apples'

In [9]: quantities = [('apples', 2), ('oragnges', 3), ('bananas', 4)]

In [10]: shopping_dict = {key: val for key, val in quantities}

In [11]: shopping_dict
Out[11]: {'apples': 2, 'oragnges': 3, 'bananas': 4}
```

Autor: Michał Nowotka

Rozpakowywanie kolekcji - swap trick

- Bardzo częstą operacją w programowaniu jest zmiana wartości dwóch zmiennych.
- W wielu językach aby zamienić wartości dwóch zmiennych należy użyć trzeciej pomocniczej zmiennej.
- Dzięki rozpakowywaniu, w Pythonie można tego uniknąć (oczywiście pod spodem Python i tak stworzy taką zmienną więc nie jest to optymalizacja a wyłącznie czystszy zapis)

```
In [1]: a = 1

In [2]: b = 5

In [3]: a, b = b, a

In [4]: a
Out[4]: 5

In [5]: b
Out[5]: 1
```

Autor: Michał Nowotka

Rozpakowywanie kolekcji - operator *

- Czasami interesuje nas tylko określona ilość elementów na początku lub końcu kolekcji.
- Cała reszta może pozostać nierozpakowana.
- Do oznaczenia takich “nierozpakowanych” elementów możemy użyć operatora *.

```
In [1]: a, b, *rest = range(5)

In [2]: a, b, rest
Out[2]: (0, 1, [2, 3, 4])

In [3]: a, *rest, c, d = range(7)

In [4]: a, rest, c, d
Out[4]: (0, [1, 2, 3, 4], 5, 6)
```

Autor: Michał Nowotka

Rozpakowywanie kolekcji - operator *

Za dużo gwiazdek?

Funkcje - adnotacje typów (type hints)

- Python od wersji 3.5 zaczął umożliwiać sugerowanie jakiego typu powinny być poszczególne parametry funkcji oraz wartość zwracana.
- Oczywiście Python nie przestaje być dynamicznie typowanym językiem, a wskazówki są jedynie sugestiami dla użytkowników.
- Istnieją jednak dodatkowe narzędzia (takie jak **mypy**), które dokonują statycznej analizy typów jako niezależny krok, który podobnie jak testy czy pokrycie może być zintegrowany z systemami Continuous Integration (np. GitLab).
- Również PyCharm rozumie adnotacje typów i wyświetla ostrzeżenia jeśli wykryje problem.

```
1    def is_isogram(word: str) -> bool:
2        return len(word) == len(set(word))
3
4
5    def incorrectly_annotated(foo: int) -> int:
6        foo.lower()
7        return
```

Funkcje - adnotacje typów (type hints)

- Adnotacje umieszczamy na liście argumentów funkcji. Po każdym argumencie piszemy dwukropek, a po nim nazwę typu, jaki powinien posiadać dany argument.
- Oczekiwany typ zwracany piszemy po strzałce.
- Bardziej zaawansowane typy można zaimportować używając modułu **typing**.

```python
from typing import Dict, Collection


def is_isogram(word: str) -> bool:
    return len(word) == len(set(word))


def incorrectly_annotated(foo: int) -> int:
    foo.lower()
    return


def extract_keys(my_dict: Dict[str, int]) -> Collection[int]:
    return my_dict.values()
```

Autor: Michał Nowotka

Funkcje - adnotacje typów (type hints)

Ćwiczenie: powróćmy do naszych wcześniej napisanych funkcji i dodajmy do ich parametrów i zwracanych wartości adnotacje typów.

ITERATORY I GENERATORY

Iteratory

- Iteratory są obiektami, które można używać w pętli.
- Iterator powinien dostarczać dwóch specjalnych metod: __iter__ oraz __next__.
- Metoda next ma zwracać wartość kolejnego obiegu pętli lub wyrzucać specjalny wyjątek StopIteration, aby zakończyć działanie pętli.
- Metoda __iter__ w iteratorze zawsze powinna zwracać jego samego, czyli self.
- Przykładem może być klasa implementująca ciąg liczb Fibonacciego o długości N lub klasa implementująca ciąg potęg dwójki.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max
        self.n = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

Iteratory

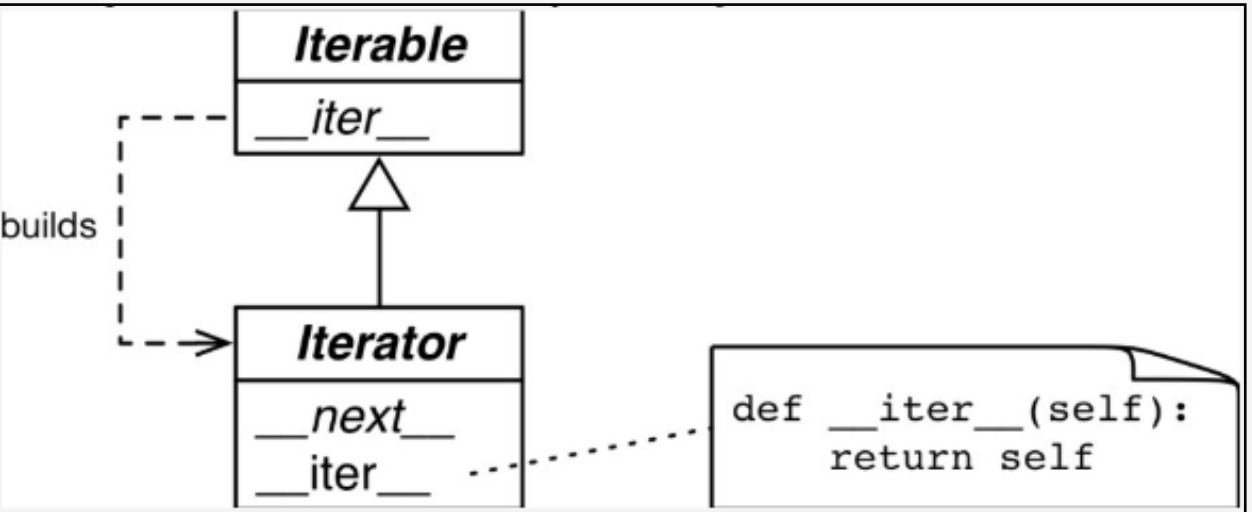
- Iteratory są obiektami, które można używać w pętli.
- Iterator powinien dostarczać dwóch specjalnych metod: __iter__ oraz __next__.
- Metoda next ma zwracać wartość kolejnego obiegu pętli lub wyrzucać specjalny wyjątek StopIteration, aby zakończyć działanie pętli.
- Metoda __iter__ w iteratorze zawsze powinna zwracać jego samego, czyli self.
- Przykładem może być klasa implementująca ciąg liczb Fibonacciego o długości N.

```
In [43]: for i in PowTwo(10):
    ...:     print(i)
    ...:
1
2
4
8
16
32
64
128
256
512
1024
```

Iteratory

- Pojawia się jednak pewien problem - po iteratorze można przejść tylko raz - zużyty iterator wyrzuca ciągle StopIteration i nie da się go ponownie wykorzystać.

- Rozwiązaniem jest podzielenie funkcjonalności pomiędzy Iterable-obiekt, po którym można iterować. Obiekt ten dostarcza metodę __iter__, która za każdym razem zwraca jednorazowy iterator ale dzięki temu można po niej iterować




```
In [50]: power = PowTwo(10)

In [51]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32
64
128
256
512
1024

In [52]: for i in power:
    ...:     print(i)
    ...:
```

Iteratory

```python
class PowTwo:
    def __init__(self, max):
        self.max = max

    def __iter__(self):
        return PowTwoIterator(self.max)


class PowTwoIterator:
    def __init__(self, max):
        self.max = max
        self.n = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration
```

```
In [59]: power = PowTwo(5)

In [60]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32

In [61]: for i in power:
    ...:     print(i)
    ...:
1
2
4
8
16
32
```

Iteratory

Napisz klasę mającą właściwości iteratora z jednym argumentem do konstruktora (liczbą n). W przypadku wrzucenia obiektu tej klasy na iterację do pętli for, powinna ona być w stanie wypisać liczby od 0 do n.

Generatory

- Generator jest rodzajem funkcji.
- Jednak zwyczajna funkcja zwraca wartość raz, za pomocą słowa kluczowego return i kończy swoje działanie.
- Generator z kolei może zwracać wartość wielokrotnie podczas swojego działania. Służy do tego słowo kluczowe yield.
- O generatorze można powiedzieć, że jest funkcją, która zwraca iterator.
- Generatory poprawiają wydajność kodu, ponieważ nie akumulują w pamięci wszystkich wyników tylko zwracają elementy jeden po drugim.
- Jeśli nie chcemy czekać aż funkcja obliczy i zwróci całe zadanie tylko chcemy znać kolejną część rozwiązania tak szybko jak tylko jest dostępna powinniśmy użyć generatora.

```
1  def simple_generator():
2      yield 1
3      yield 2
4      yield 3
```

```
In [2]: gen = simple_generator()

In [3]: for i in gen:
   ...:     print(i)
   ...:
1
2
3

In [4]: gen = simple_generator()

In [5]: list(gen)
Out[5]: [1, 2, 3]

In [6]: list(gen)
Out[6]: []
```

Generatory

- Mając generator możemy po nim iterować w pętli.
- Możemy go też przekształcić w listę ale pamiętajmy że jeśli generator ma wygenerować dużą liczbę elementów to można w ten sposób zawiesić działanie programu.
- Generatory są jednorazowe - wykorzystane raz nie zwrócą nic więcej. Kiedy raz przekształcimy generator na listę dostaniemy wszystkie elementy, które jest on w stanie wygenerować. Kolejna próba zwróci pustą listę.
- Aby ponownie przeiterować po wartościach trzeba wywołać od nowa funkcję.

```
1    def simple_generator():
2        yield 1
3        yield 2
4        yield 3
```

```
In [2]: gen = simple_generator()

In [3]: for i in gen:
   ...:     print(i)
   ...:
1
2
3

In [4]: gen = simple_generator()

In [5]: list(gen)
Out[5]: [1, 2, 3]

In [6]: list(gen)
Out[6]: []
```

Generatory

- Kolejne elementy generatora można otrzymać wywołując wbudowaną funkcję next.
- Kiedy generator się wyczerpie rzuci wyjątek StopIteration.
- Dokładnie tak działa w Pythonie pętla for - na danej sekwencji lub iteratorze jest wywoływana funkcja next tak długo aż zostanie przechwycony wyjątek StopIteration, który kończy działanie pętli.

```
In [7]: gen = simple_generator()

In [8]: next(gen)
Out[8]: 1

In [9]: next(gen)
Out[9]: 2

In [10]: next(gen)
Out[10]: 3

In [11]: next(gen)
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-11-6e72e47198db> in <module>
```

Generatory - przykłady

```
def liczby():
    for i in range(11):
        yield i * 2

for parzysta in liczby():
    print(parzysta)
```

Generatory - przykłady

```
def wznowienia():
    print("wstrzymuje dzialanie")
    yield 1
    print("wznawiam dzialanie")

    print("wstrzymuje dzialanie")
    yield 2
    print("wznawiam dzialanie")

for i in wznowienia():
    print("Zwrocono wartosc: " + str(i))
```

Generatory - przykłady

```
def ret():
    for i in range(5):
        if i == 3:
            return
        else:
            yield i

for x in ret():
    print(x)
```

Generatory

Zaimplementuj rozwiązanie zwracające n-tą liczbę ciągu Fibonacciego. Napisz program w sposób:

a) iteracyjny (z wykorzystaniem pętli),
b) z wykorzystaniem generatora.