Python

Poziom średniozaawansowany

Plan na weekend

1. *Funkcje – zagadnienia dodatkowe*
2. *Wyjątki – jak radzić sobie z błędami*
3. *Iteratory i generatory*
4. *Lambdy – funkcje anonimowe*
5. *Dekoratory*

FUNKCJE – ZAGADNIENIA DODATKOWE

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'
```

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'
```

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: *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
```

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'
```

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}
```

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
```

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)
```

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()
```

Funkcje - adnotacje typów (type hints)

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

WYJĄTKI – OBSŁUGA BŁĘDÓW W PROGRAMIE

Ćwiczenie na rozgrzewkę: napisz obsługę błędu błędnego użycia funkcji podaj_imie. Funkcja powinna przyjmowac jeden parametr imie typu str. Jeżeli użytkownik poda inny typ, funkcja zamiast wyprintować podane imię powinna wypisać komunikat o błędzie i zwrócić -1.

Obsługa wyjątków

- Wyjątek jest następstwem pewnej specjalnej sytuacji podczas wykonania programu.
- Sytuacja, która doprowadziła do podniesienia wyjątku nie należy do głównego scenariusza wykonania programu.
- Wyobraźmy sobie, że piszemy funkcję która jako argument przyjmuje ścieżkę do pliku a zwraca liczbę słów w danym pliku.
- Spodziewamy się, że w optymistycznym oraz najczęstszym scenariuszu plik, do którego podana jest ścieżka, będzie istniał.
- Może jednak zajść sytuacja w której ktoś popełni błąd i poda ścieżkę, która nie istnieje. W takim razie nasza funkcja może podnieść wyjątek, informując wywołującego o wystąpieniu specjalnej sytuacji.

Obsługa wyjątków

- Każdą specjalną sytuację należy jakoś obsłużyć.
- Każdy wyjątek, który nie zostanie obsłużony spowoduje zakończenie działania programu.

```
In [2]: print(10/0)
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-2-fe01563e1bc6> in <module>
----> 1 print(10/0)

ZeroDivisionError: division by zero
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [1]: try:
   ...:     print(10/0)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [3]: try:
   ...:     print(10/0)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...: finally:
   ...:     print("Blok finally. Zawsze wykonywany.")
   ...:
Nie dziel przez zero!!!
Blok finally. Zawsze wykonywany.
```

Obsługa wyjątków

- Jeśli wywołujemy funkcję, o której wiemy, że może podnieść wyjątek i chcemy obsłużyć tę sytuację, to należy ją wywołać w bloku **try**.
- Po bloku try musi wystąpić blok **except** i/lub blok **finally**.

```
In [4]: try:
   ...:     print(10/10)
   ...: except:
   ...:     print("Nie dziel przez zero!!!")
   ...: finally:
   ...:     print("Blok finally. Zawsze wykonywany.")
   ...:
1.0
Blok finally. Zawsze wykonywany.
```

Obsługa wyjątków

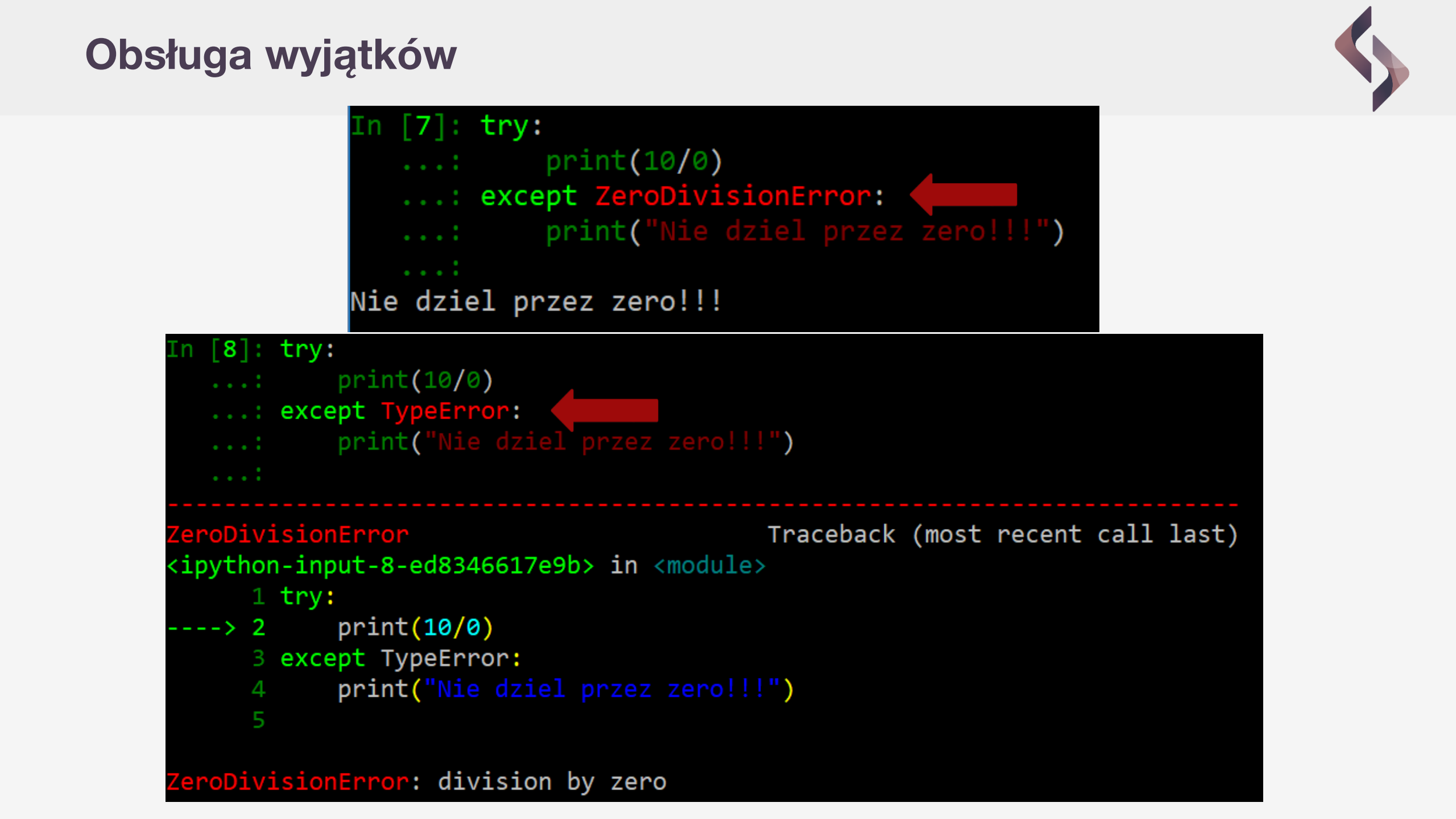
- Instrukcje w bloku finally wykonają się zawsze, niezależnie od tego czy wyjątek zostanie wyrzucony czy nie.
- Blok finally jest bardzo ważną konstrukcją kiedy używamy cennych zasobów, które musimy zwolnić nawet jeśli wystąpi wyjątkowa sytuacja.

Obsługa wyjątków

- W bloku **except** znajduje się kod obsługi przechwyconego wyjątku. Po słowie kluczowym **except** może zostać podana nazwa klasy wyjątku, którego obsługi dotyczy ten blok.
- Wszystkie podane niżej wyjątki (jak również pozostałe) dziedziczą po ogólnej klasie **`BaseException`**. Przy tworzeniu własnych wyjątków zaleca się dziedziczyć po typie **`Exception`**.

`TypeError, AssertionError, KeyError, ModuleNotFoundError, IndexError, NameError, ZeroDivisionError, SyntaxError`

Obsługa wyjątków



```
In [7]: try:
   ...:     print(10/0)
   ...: except ZeroDivisionError:
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

```
In [8]: try:
   ...:     print(10/0)
   ...: except TypeError:
   ...:     print("Nie dziel przez zero!!!")
   ...:
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-8-ed8346617e9b> in <module>
      1 try:
----> 2     print(10/0)
      3 except TypeError:
      4     print("Nie dziel przez zero!!!")
      5

ZeroDivisionError: division by zero
```

Obsługa wyjątków

- Można poinformować program, że spodziewamy się wystąpienia jednego z dwóch lubi więcej wyjątków (lepiej tego nie nadużywać, jeśli nie jesteśmy pewni)

```
In [9]: try:
   ...:     print(10/0)
   ...: except (ZeroDivisionError, TypeError):
   ...:     print("Nie dziel przez zero!!!")
   ...:
Nie dziel przez zero!!!
```

Obsługa wyjątków - przykłady

- TypeError – operacja/funkcja wykonywana ze złym typem obiektu (np. ‘2’ + 2)
- KeyError – klucz (np. w słowniku) nie może być znaleziony
- IndexError – próba dostania się do nieistniejącego elementu (np. w liście)
- ModuleNotFoundError – importowany (cały) moduł nie może być znaleziony
- AssertionError – asercja (wykorzystywana np. w testach) kończy się niepowodzeniem
- NameError – obiekt nie może być znaleziony (np. zmienna przed deklaracją)
- SyntaxError – niepoprawna składnia (np. wyrażenie: `a(=):1+2)`)
- ZeroDivisionError – dzielenie przez zero

Obsługa wyjątków

- Aby własnoręcznie podnieść wyjątek (wymusić go) w naszym kodzie, używamy słowa kluczowego **raise**. Podajemy po nim nazwę klasy wyjątku, który chcemy podnieść albo konkretną instancję tej klasy.

```
In [15]: try:
    ...:     raise SyntaxError
    ...: except SyntaxError:
    ...:     print("Nastąpił błąd syntaktyczny!")
    ...:
Nastąpił błąd syntaktyczny!
```

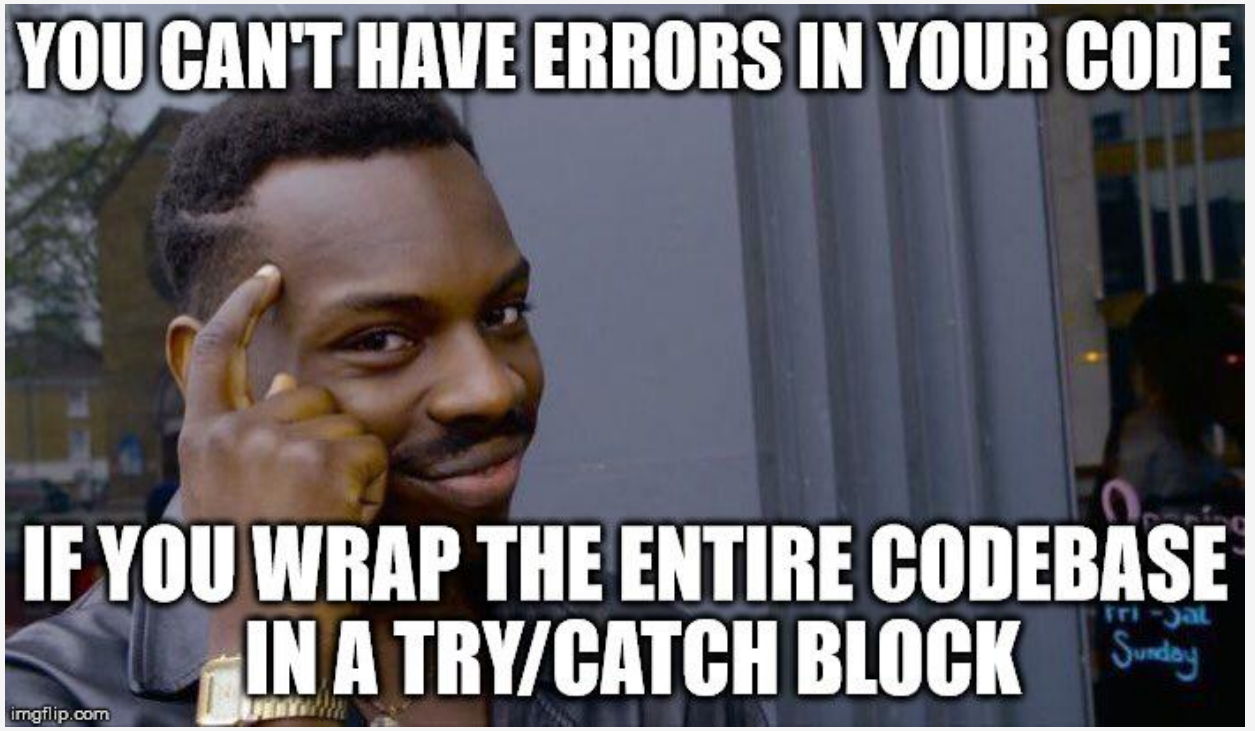
Obsługa wyjątków

- Bloków **except** może być wiele, np. po jednym bloku na każdy spodziewany typ wyjątku.
- Jeśli kilka różnych typów wyjątków będzie obsłużonych w tym samym bloku, ich typy można podać w postaci krotki.
- Po typie wyjątku można użyć słowa kluczowego **as** oraz nazwy zmiennej, do której zostanie przypisana konkretna instancja wyrzuconego wyjątku. Możemy jej użyć w kodzie obsługi np. aby zbadać treść komunikatu o błędzie.

```
In [21]: try:
    ...:     print(10/0)
    ...: except ZeroDivisionError as e:
    ...:     print(e, e.args, type(e))
    ...:
    ...:
division by zero ('division by zero',) <class 'ZeroDivisionError'>
```

Obsługa wyjątków

- Samo słowo **except** bez podania typu albo konstrukcja **except Exception** obsłużą każdy wyjątek, niezależnie od jego typu. Takie przechwytywanie wszystkich wyjątków, nie zważając na ich typ jest nazywane Pokemon exception handling i powszechnie uważane za antywzorzec programowania!



Obsługa wyjątków

- Tworzenie własnych wyjątków polega na dziedziczeniu po klasie **`Exception`** .
- Takie wyjątki możemy podnosić jak każde inne w naszym kodzie.

```
In [3]: class MyNewException(Exception):
   ...:     pass
   ...:

In [4]: raise MyNewException("Mayday!")
---------------------------------------------------------------------------
MyNewException                            Traceback (most recent call last)
<ipython-input-4-a19b402f2158> in <module>
----> 1 raise MyNewException("Mayday!")

MyNewException: Mayday!
```

Spróbujmy wymusić zwrócenie przez program kilku wyjątków…

Napisz funkcję przyjmującą jeden argument. Sprawdź jego typ we wnętrzu funkcji, w przypadku kiedy będzie to:

- typ str – podnieś własny wyjątek StrArgTypeError,
- typ int/float – podnieś własny wyjątek NumberArgTypeError,
- typ bool – podnieś własny wyjątek BoolArgTypeError.

Jeżeli argument będzie miał długość większą niż 1 (np. jako lista, tuple'a, set itd) podnieś wyjątek LenGreaterThanOneError.

Funkcja powinna wyprintować 'None jest okej', jeżeli argument będzie miał wartość None. Wszystkie wyjątki należy obsłużyć wewnątrz bloku if __name__ == '__main__' przy wywoływaniu funkcji.

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

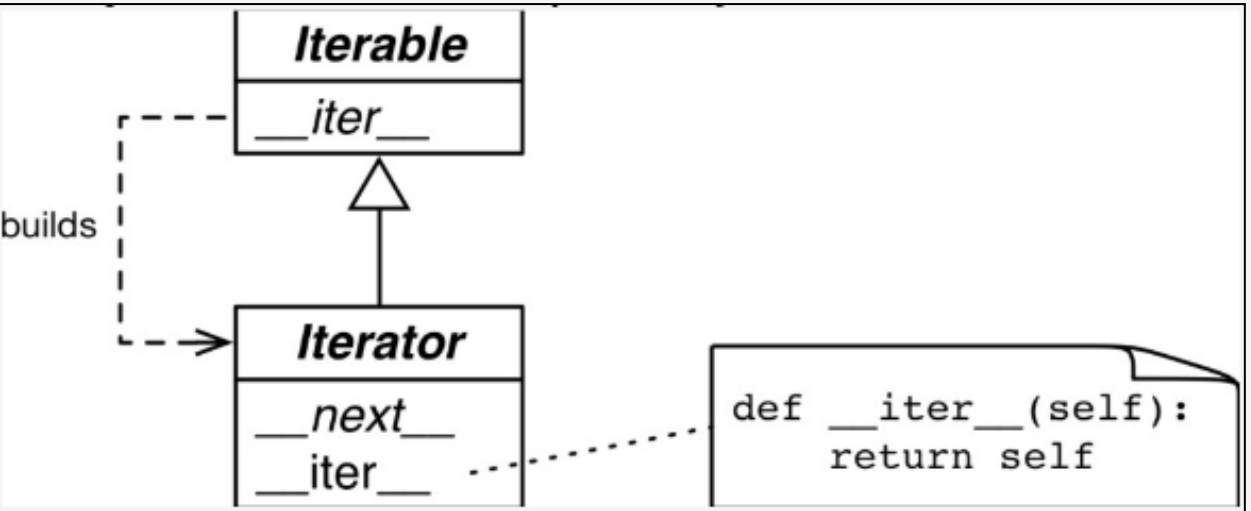
- Iteratory są obiektami, które można używać w pętli.
- Iterator powinien dostarczać dwóch specjalnych metod: __iter__ oraz __next__.
- Metoda next ma zwracać wartość kolejnego obiegu pętli lub wyrzucać specjalny wyjątek StopIteration, aby zakończyć działanie pętli.
- Metoda __iter__ w iteratorze zawsze 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.