Zaawansowane programowanie w języku Python

Krystian Piękoś

May 11, 2023

CONTENTS

Materiały pomocnicze do szkolenia

0.1 Klasy i obiekty - elementy zaawansowane

0.1.1 Python jako język obiektowy - podstawowe informacje

Wszystko jest obiektem

Python jest językiem w pełni obiektowym - wszystko jest obiektem, także wartości typów prymitywnych i funkcje.

```
(5).__add__(3)
```

```
8
```

Dynamiczne typowanie

Python jest językiem z **dynamicznym typowaniem**. Zmienne przechowują referencję do obiektu, a dopiero obiekty posiadają typ. Dlatego raz zdefiniowana zmienna może "zmienić typ".

```
var = 'text'
var
```

```
'text'
```

```
var = 1
var
```

```
1
```

Dynamiczne typowanie znacznie ułatwia programowanie, ponieważ nie narzuca ograniczeń na typ zmiennych.

Dzięki temu programista może skupić się na bardziej istotnych aspektach, takich jak poprawność kodu, albo po prostu napisać kod szybciej.

Dzięki dynamicznemu typowaniu dostępna jest funkcja `eval`, która pozwala na wykonanie dowolnego, dynamicznie utworzonego wyrażenia:

```
x = 1
eval('x+1')
```

```
2
```

Dynamiczne typowanie umożliwia także łatwe użycie technik metaprogramowania, takich jak metaklasy, które zostaną omówione później. Pozwala to np. na dynamiczne tworzenie nowych typów. Dzięki temu implementacje ORM (*Object-Relational Mapping*, mapowanie obiektowo-relacyjne) są bardziej naturalne.

Tożsamość, typ a wartość

Każdy obiekt posiada:

- **tożsamość** (*identity*) - wskazuje na lokalizację obiektu w pamięci i można ją sprawdzić wywołując wbudowaną funkcję `id`;
- **typ** (*type*) opisuje reprezentację obiektu dla Pythona;
- **wartość** (*value*) to dane przechowywane w obiekcie.

```
numbers = [1, 2, 3]
id(numbers)
```

```
140330206076416
```

```
type(numbers)
```

```
list
```

```
numbers
```

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

0.1.2 Definiowanie klasy

Klasę definiujemy za pomocą słowa kluczowego *class*:

```
class MyClass:
    def method(self):
        pass
```

W Pythonie wszystko jest obiektem, także klasa, dlatego możemy mówić o *obiekcie klasy*. Taki obiekt również ma swój typ:

```
MyClass
```

```
__main__.MyClass
```

```
type(MyClass)
```

```
type
```

0.1.3 Metody specjalne

Metodami specjalnymi nazywane są metody zaczynające i kończące się dwoma podkreślnikami. Implementują one operacje, które wywołujemy przy użyciu specjalnej składni (np. porównanie dwóch obiektów `a < b`, dostęp do atrybutów `obj.attribute` lub składnia `obj[key]`).

Najważniejsze metody specjalne

Kategoria	Nazwy metod
String/bytes representation	`__repr__ __str__ __format__ __bytes__ __fspath__`
Konwersja do liczby	`__bool__ __complex__ __int__ __float__ __hash__ __index__`
Emulacja kolekcji	`__len__ __getitem__ __setitem__ __delitem__ __contains__`
Iteracja	`__iter__ __aiter__ __next__ __anext__ __reversed__`
Funkcje lub korutyny	`__call__ __await__`
Menadżer kontekstu	`__enter__ __exit__ __aexit__ __aenter__`
Tworzenie i niszczenie instancji	`__new__ __init__ __del__`
Zarządzanie atrybutami	`__getattr__ __getattribute__ __setattr__ __delattr__ __dir__`
Deskryptory	`__get__ __set__ __delete__ __set_name__`
Abstract base classes	`__instancecheck__ __subclasscheck__`
Metaprogramowanie	`__prepare__ __init_subclass__ __class_getitem__ __mro_entries__`

Metody operatorowe

Kategoria operatorów	Symbole	Nazwy metod
Jednoargumentowe	`- + abs()`	`__neg__ __pos__ __abs__`
Porównania	`< <= == != > >=`	`__lt__ __le__ __eq__ __ne__ __gt__ __ge__`
Arytmetyczne	`+ - * / // % @ divmod() round() ** pow()`	`__add__ __sub__ __mul__ __truediv__ __floordiv__ __mod__ __matmul__ __divmod__ __round__ __pow__`
Arytmetyczne z przypisaniem	`+= -= *= /= //= %= @= **=`	`__iadd__ __isub__ __imul__ __itruediv__ __ifloordiv__ __imod__ __imatmul__ __ipow__`
Bitowe	`& | ^ << >> ~`	`__and__ __or__ __xor__ __lshift__ __rshift__ __invert__`
Bitowe z przypisaniem	`&= |= ^= <<= >>=`	`__iand__ __ior__ __ixor__ __ilshift__ __irshift__`

Dostęp do atrybutów

Dostęp do atrybutów kontrolują poniższe metody specjalne:

Metoda specjalna	Opis
`__getattr__(self, name)`	Wywoływana, gdy obiekt nie ma atrybutu `name`
`__setattr__(self, name, value)`	Wywoływana podczas przypisywania atrybutów
`__delattr__(self, name)`	Wywoływana przy usuwaniu atrybutu (`del obj.attr`)

Przykład słownika wykorzystującego składnię `dict.key` zamiast `dict[key]`:

```
class Record:
    def __init__(self):
        # Nie możemy użyć poniższego kodu:
        #     self._d = {}
        # ponieważ zakończyłby się on rekurencyjnym wywoływaniem metody __setattr__
        super().__setattr__('_dict', {})

    def __getattr__(self, name):
        print('getting', name)
        return self._dict[name]

    def __setattr__(self, name, value):
        print('setting', name, 'to', value)
        self._dict[name] = value

    def __delattr__(self, name):
        print('deleting', name)
        del self._dict[name]
```

```
person = Record()
person.first_name = "John"
person.first_name
```

```
setting first_name to John
getting first_name
```

```
'John'
```

```
del person.first_name
```

```
deleting first_name
```

Oprócz wspomnianych metod

Metoda specjalna	Opis
`__getattribute__(self, name)`	Wywoływana bezwarunkowo przy dostępie do atrybutów klasy, nawet jeśli dany atrybut istnieje.

```
class Person:
    def __init__(self, first_name):
```

(continues on next page)

(continued from previous page)

```
        self.first_name = first_name

    def __getattribute__(self, name):
        print('getattribute', name)
        return object.__getattribute__(self, name)
```

```
p = Person('John')
p.first_name
```

```
getattribute first_name
```

```
'John'
```

Przykład ilustrujący różnicę między `__getattr__` a `__getattribute__`:

```
class Foo:
    def __init__(self):
        self.a = "a"

    def __getattr__(self,attribute):
        return f"You asked for {attribute}, but I'm giving you default"


class Bar:
    def __init__(self):
        self.a = "a"

    def __getattribute__(self,attribute):
        return f"You asked for {attribute}, but I'm giving you default"
```

```
foo = Foo()
foo.a
```

```
'a'
```

```
foo.b
```

```
"You asked for b, but I'm giving you default"
```

```
getattr(foo, "a")
```

```
'a'
```

```
getattr(foo, "b")
```

```
"You asked for b, but I'm giving you default"
```

```
bar = Bar()
```

```
bar.a
```

```
"You asked for a, but I'm giving you default"
```

```
bar.b
```

```
"You asked for b, but I'm giving you default"
```

```
getattr(bar, "a")
```

```
"You asked for a, but I'm giving you default"
```

```
getattr(bar, "b")
```

```
"You asked for b, but I'm giving you default"
```

0.1.4 Składowe chronione i prywatne

Składowe chronione

Składowe, które powinny być modyfikowane tylko przez klasę, powinny zaczynać się od podkreślnika. Jest to powszechnie przyjęta konwencja oznaczająca, że dana składowa nie powinna być modyfikowana z zewnątrz. Jest to jednak tylko konwencja - Python nie posiada mechanizmu ukrywającego takie składowe. Wciąż można je modyfikować spoza klasy.

```
class BankAccount:
    def __init__(self, initial_balance):
        self._balance = initial_balance

    @property
    def balance(self):
        return self._balance
```

```
account = BankAccount(100.0)
print(account.balance)
print(account._balance) # # składowe chronione są wciąż dostępne z zewnątrz klas
```

```
100.0
100.0
```

Składowe prywatne

Aby ukryć atrybut lub metodę przed dostępem spoza klasy (składowa *private*), należy jej nazwę poprzedzić dwoma podkreślnikami (np. `__atrybut`). Taka składowa jest dostępna tylko wewnątrz tej klasy. Składowe zaczynające się od dwóch podkreślników (nie będące metodami specjalnymi) są traktowane w szczególny sposób - ich nazwa zostaje zmieniona na `_NazwaKlasy__atrybut`. Do tej składowej można się wciąż odwołać z zewnątrz klasy, ale tylko używając zmienionej nazwy.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.__balance = balance

    def withdraw(self, amount):
        self.__balance -= amount

    def deposit(self, amount):
        self.__balance += amount

    def info(self):
        print("owner:", self.owner, "; balance:", self.__balance)
```

```
jk = BankAccount("Jan Kowalski", 1000)
print(jk.__balance) # błąd!
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[29], line 2
      1 jk = BankAccount("Jan Kowalski", 1000)
----> 2 print(jk.__balance) # błąd!

AttributeError: 'BankAccount' object has no attribute '__balance'
```

```
print(jk._BankAccount__balance) # ok
```

```
1000
```

0.1.5 Atrybuty klasy i metody statyczne

Składowe statyczne są wspólne dla wszystkich instancji klasy.

Z kolei metoda statyczna to po prostu funkcja w przestrzeni nazw klasy. Taką funkcję należy udekorować `@staticmethod`. Taka funkcja nie przyjmuje instancji klasy `self`.

```
class CountedObject(object):
    count = 0   # statyczna składowa

    def __init__(self):
        CountedObject.count += 1

    @staticmethod  # statyczna metoda
    def get_count():
        return CountedObject.count
```

```
CountedObject.get_count()
```

```
0
```

```
c1 = CountedObject()
c2 = CountedObject()
cs = [CountedObject(), CountedObject()]

CountedObject.get_count()
```

```
4
```

Czasami atrybutów klasy używa się, aby zainicjalizować domyślną wartość dla pewnych atrybutów instancji. Należy jednak być ostrożnym:

```
class PersonWithDefaultAttributes:
    first_name = 'John'
    last_name = 'Smith'
    phones = []

p1 = PersonWithDefaultAttributes()
p2 = PersonWithDefaultAttributes()

print(p1.first_name)
print(p2.first_name)
```

```
John
John
```

```
p1.first_name = 'Bob'
print(p1.first_name)
print(p2.first_name)
```

```
Bob
John
```

```
PersonWithDefaultAttributes.last_name = 'Williams'
print(p1.last_name)
print(p2.last_name)
```

```
Williams
Williams
```

```
p1.phones.append('+48123456789')
print(p1.phones)
print(p2.phones)
```

```
['+48123456789']
['+48123456789']
```

0.1.6 Metody klasy

Zwykła metoda ma dostęp do instancji klasy (poprzez parametr `self`). Z kolei metoda klasy ma dostęp do klasy, z której została wywołana, lub do **klasy instacji**, z której została wywołana.

Metody klasy są przydatne, jeżeli chcemy pozwolić na więcej niż jeden sposób tworzenia instancji.

```
class Date:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = date_as_string.split('-')
        return cls(int(day), int(month), int(year)) # utworzenie instancji klasy cls
```

```
d1 = Date(20, 1, 2016)
d2 = Date.from_string('20-01-2016')
```

Warto zwrócić uwagę na to, że wewnątrz metody `from_string` tworzona jest nowa instancja klasy `cls`. Nie musi być to klasa `Date`. Tak będzie w przypadku klas dziedziczących po `Date`.

0.1.7 Deskryptory

Deskryptorem jest atrybut (obiekt), który zawiera przynajmniej jedną z trzech metod specjalnych tzw. "protokołu deskryptora".

Metody specjalne deskryptora	Opis
`__get__(self, instance, owner)`	Wywoływana do pobrania atrybutu z obiektu lub klasy "właściciela"
`__set__(self, instance, value)`	Wywoływana do ustawienia wartości atrybutu
`__delete__(self, instance)`	Wywoływana w czasie usuwania atrybutu

Taki obiekt musi pojawić się jako atrybut w obiekcie - "właścicielu". W momencie dostępu do takiego atrybutu wywoływane są odpowiednie metody deskryptora.

Rodzaje deskryptorów

- **non-data descriptor** - posiada tylko metodę `__get__()`. Przy dostępie do atrybutu podjęta może zostać akcja zaimplementowana w metodzie `__get__()`

```
import os

class DirectorySize:
    def __get__(self, instance, owner_class):
        return len(os.listdir(instance.directory_name))

class Directory:
    size = DirectorySize() # descriptor instance

    def __init__(self, directory_name):
        self.directory_name = directory_name # regular instance attribute
```

```
local_dir = Directory('.')
local_dir.size
```

```
28
```

- **data descriptor** - posiada zarówno metodę `__get__()` jak i `__set__()`. Może też zawierać definicję metody `__delete__()`. Umożliwia tworzenie obiektu *mutable*, do którego delegowane są próby dostępu do atrybutu i ustawiania jego wartości.

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAgeAccess:

    def __get__(self, instance, owner_class=None):
        value = instance._age
        logging.info('Accessing %r.%r giving %r', instance, 'age', value)
        return value

    def __set__(self, instance, value):
        logging.info('Updating %r.%r to %r', instance, 'age', value)
        instance._age = value


class Person:
    age = LoggedAgeAccess()             # Descriptor instance

    def __init__(self, name, age):
        self.name = name                # Regular instance attribute
        self.age = age                  # Calls __set__()

    def birthday(self):
        self.age += 1                   # Calls both __get__() and __set__()
```

```
john = Person("John", 31)
mary = Person("Mary", 27)
john.birthday()
mary.age = 30
```

```
INFO:root:Updating <__main__.Person object at 0x000001BE9E42C490>.'age' to 31
INFO:root:Updating <__main__.Person object at 0x000001BE9E9E2450>.'age' to 27
INFO:root:Accessing <__main__.Person object at 0x000001BE9E42C490>.'age' giving 31
INFO:root:Updating <__main__.Person object at 0x000001BE9E42C490>.'age' to 32
INFO:root:Updating <__main__.Person object at 0x000001BE9E9E2450>.'age' to 30
```

Nazwy atrybutów i deskryptory

Jeśli klasa używa wielu deskryptorów, to często powstaje potrzeba związania obiektu deskryptora z nazwą atrybutu, nad którym deskryptor przejmuje kontrolę. Wykorzystuje się do tego metodę `__set_name__()` w klasie deskryptora:

```
import logging

logging.basicConfig(level=logging.INFO)

class LoggedAccess:

    def __set_name__(self, owner, name):
        self.public_name = name
        self.private_name = '_' + name
        logging.info('Setting names: %r and %r', self.public_name, self.private_name)

    def __get__(self, instance, owner_class=None):
        value = getattr(instance, self.private_name)
        logging.info('Accessing %r giving %r', self.public_name, value)
        return value

    def __set__(self, instance, value):
        logging.info('Updating %r to %r', self.public_name, value)
        setattr(instance, self.private_name, value)

class Person:
    name = LoggedAccess()                # First descriptor instance
    age = LoggedAccess()                 # Second descriptor instance

    def __init__(self, name, age):
        self.name = name                 # Calls the first descriptor
        self.age = age                   # Calls the second descriptor

    def birthday(self):
        self.age += 1
```

```
INFO:root:Setting names: 'name' and '_name'
INFO:root:Setting names: 'age' and '_age'
```

```
vars(vars(Person)['name'])
```

```
{'public_name': 'name', 'private_name': '_name'}
```

```
vars(vars(Person)['age'])
```

```
{'public_name': 'age', 'private_name': '_age'}
```

```
james = Person("James", 43)
```

```
INFO:root:Updating 'name' to 'James'
INFO:root:Updating 'age' to 43
```

```
james.birthday()
```

```
INFO:root:Accessing 'age' giving 43
INFO:root:Updating 'age' to 44
```

Kolejność wyszukiwania nazw przy dostępie do atrybutów

Wywołanie z instancji

Przy próbie dostępu do atrybutu `x` dla obiektu `o` (`o.x`):

- **data descriptor**: wartość zwrócona z metody `__get__()` deskryptora pola
- **składowa instancji**: wartość `o.__dict__[x]`
- **non-data descriptor**: wartość zwrócona z metody `__get__()` deskryptora pola
- **składowa klasy**: `type(o).__dict__['x']`
- **składowe klas bazowych**: wyszukiwanie w kolejności MRO,
- **`__getattr__()`**, jeśli ta metoda została zdefiniowana w klasie

Wywołanie z klasy

Przy próbie dostępu do atrybutu `x` dla klasy `C` (`C.x`):

- **data descriptor**: wartość zwrócona z metody `__get__()` deskryptora pola klasy
- **składowa klasy lub klas bazowych**: poszukiwanie w `C.__dict__['x']` lub klasach bazowych
- **non-data descriptor**: wartość zwrócona z metody `__get__()` deskryptora pola

Właściwości (properties)

Przykładem deskryptora jest wbudowany dekorator `@property` pozwalający na enkapsulację obiektu, to znaczy kontrolę dostępu do atrybutów przy użyciu metod dostępowych.

```
class BankAccount:
    def __init__(self, daily_limit):
        self.__daily_limit = daily_limit

    @property
    def daily_limit(self):
        print('getting daily_limit')
        return self.__daily_limit

    @daily_limit.setter
    def daily_limit(self, value):
        if value < 0:
            raise ValueError('Value must be >= 0')
        self.__daily_limit = value
```

```
account = BankAccount(100.0)
account.daily_limit
```

```
getting daily_limit
```

```
100.0
```

```
account.daily_limit = 200.0
account.daily_limit
```

```
getting daily_limit
```

```
200.0
```

```
account.daily_limit = -100.0
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-73-764421fb5f17> in <module>
----> 1 account.daily_limit = -100.0

<ipython-input-70-a7a363ba8f92> in daily_limit(self, value)
     11     def daily_limit(self, value):
     12         if value < 0:
---> 13             raise ValueError('Value must be >= 0')
     14         self.__daily_limit = value

ValueError: Value must be >= 0
```

Gdyby nie został zdefiniowany setter, to znaczy w kodzie nie pojawiłby się `@daily_limit.setter`, wówczas `daily_limit` byłaby właściwością tylko do odczytu. Próby zmiany jej wartości kończyłyby się błędem.

0.1.8 Sloty

Każdy obiekt Pythona posiada `__dict__` - słownik atrybutów typu `dict`. W rezultacie dla każdego obiektu mamy dość spory narzut związany ze zużyciem pamięci i czasem dostępu do składowych słownika.

Jeśli nasza klasa nie zamierza korzystać z dynamicznej natury takiego słownika, to można w klasie zdefiniować atrybut `__slots__` i podać listę wszystkich składowych. Atrybuty wymienione na liście są przechowane w tablicy referencji, która zużywa znacząco mniej pamięci.

```
class Pixel:
    __slots__ = ('x', 'y')
```

```
p = Pixel()
```

Utworzona instancja typu `Pixel` nie posiada słownika `__dict__`:

```
p.__dict__
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-66-33b2432b7e42> in <module>
----> 1 p.__dict__

AttributeError: 'Pixel' object has no attribute '__dict__'
```

Możemy normalnie korzystać z atrybutów `x` i `y`:

```
p.x = 10
p.y = 20
```

Próba ustawienia nowego atrybutu (nie wymienionego na liście slotów) skończy się wyjątkiem:

```
p.z = 30
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-68-cadbcb940683> in <module>
----> 1 p.z = 30

AttributeError: 'Pixel' object has no attribute 'z'
```

0.1.9 Dziedziczenie

Podstawy

Dziedziczenie definiowane jest za pomocą składni:

```
class Base:
    def base_method(self):
        pass

class Derived(Base):
    def derived_method(self):
        pass
```

Method Resolution Order (MRO)

Wszystkie metody zdefiniowane bezpośrednio w klasie `C` są przechowywane w słowniku `C.__dict__`:

```
Base.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'base_method': <function __main__.Base.base_method(self)>,
              '__dict__': <attribute '__dict__' of 'Base' objects>,
              '__weakref__': <attribute '__weakref__' of 'Base' objects>,
              '__doc__': None})
```

```
Derived.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'derived_method': <function __main__.Derived.derived_method(self)>,
              '__doc__': None})
```

```
d = Derived()
d.base_method() # it works
```

W klasie potomnej `Derived` dostępne są metody zdefiniowane w klasie bazowej `Base` (takie jak `base_method`), mimo że nie występują one bezpośrednio w słowniku potomka. W momencie wywoływania metody `d.base_method()`, metoda `base_method` jest poszukiwana w słowniku klasy `Derived`. Ponieważ ten słownik nie ma tej metody, przeszukiwany jest słownik klasy `Base`.

W ogólnym przypadku, przeszukiwane są słowniki wszystkich klas określonych w `C.__mro__`. Ta krotka zawiera klasę `C`, jej klasy nadrzędne, itd.

```
Derived.__mro__
```

```
(__main__.Derived, __main__.Base, object)
```

Dziedziczenie wielobazowe

W Pythonie możliwe jest dziedziczenie po więcej niż jednej klasie.

Przeciążając metody (w szczególności konstruktor `__init__`) należy pamiętać, aby wywołać także wersję rodzica przy użyciu funkcji `super`, która zwraca obiekt proxy służący do wywoływania metod rodzica. Obiekt jest wybierany zgodnie z MRO.

```
class A:
    def __init__(self):
        print("A")

class B(A):
    def __init__(self):
        super().__init__()
        print("B")

class C(A):
    def __init__(self):
        super().__init__()
        print("C")

class D(B, C):
    def __init__(self):
        super().__init__()
        print("D")
```

```
D.__mro__
```

```
(__main__.D, __main__.B, __main__.C, __main__.A, object)
```

```
D()
```

```
A
C
B
D
```

```
<__main__.D at 0x7fc85b329ee0>
```

Czasami nie jest możliwe utworzenie sensownego MRO:

```
class A: pass
class B: pass
class C(A, B): pass
class D(B, A): pass
class E(C, D): pass
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-b8bb9ff5be5c> in <module>
      3 class C(A, B): pass
      4 class D(B, A): pass
----> 5 class E(C, D): pass

TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

Mixins (klasy domieszkowe)

Klasy domieszkowe (*mixins*) to klasy, które dostarczają określoną funkcjonalność innym klasom (poprzez mechanizm wielokrotnego dziedziczenia). Nie są samodzielnymi klasami i, w związku z tym, nie tworzy się instancji klas domieszkowych. Zazwyczaj nazwa takiej klasy kończy się sufixem `Mixin` (np. `ComparableMixin`), ale nie jest to bezwzględnie obowiązująca konwencja.

Definiując klasę i wymieniając jej rodziców należy pamiętać, aby najpierw wymienić wszystkie klasy domieszkowe, a dopiero na końcu podać klasę bazową (chyba że jest nią `object`).

W przypadku porównywania obiektów wywoływana jest jedna z sześciu specjalnych metod (`__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__` lub `__ge__`). Wystarczy jednak zdefiniować dwie z nich (np. `__le__` i `__eq__`), a pozostałe porównania to odpowiednia kombinacja tych dwóch metod.

Poniżej przedstawiono klasę domieszkową `ComparableMixin`. Jej użycie powoduje, że wystarczy zdefiniować metody `__le__` i `__eq__`, aby obiekty klasy dziedziczącej po `ComparableMixin` mogły być porównywane.

```
class ComparableMixin:
    def __ne__(self, other):
        return not (self == other)
    def __le__(self, other):
        return self < other or (self == other)
    def __gt__(self, other):
        return not self <= other
    def __ge__(self, other):
        return self > other or self == other
```

(continues on next page)

(continued from previous page)

```
class MyInteger(ComparableMixin):  # klasą bazową jest "object"
    def __init__(self, i):
        self.i = i
    def __lt__(self, other):
        return self.i < other.i
    def __eq__(self, other):
        return self.i == other.i
```

```
MyInteger(1) > MyInteger(0)
```

```
True
```

Dziedziczenie po typach wbudowanych

Od Pythona 2.2 można dziedziczyć po wszystkich typach wbudowanych.

```
class CountDict(dict):
    def __getitem__(self, key):
        if key in self:
            return super(CountDict, self).__getitem__(key)
        else:
            return 0
```

```
cd = CountDict()
cd['unknown-key']
```

```
0
```

Dziedziczenie po typach niezmiennych

Dla typów niezmiennych *(immutable)* nie działa przeciążanie konstruktora `__init__`. Po utworzeniu obiektu jest już za późno na jakąkolwiek modyfikację.

```
class PositiveInt(int):
    def __new__(cls, value):
        print('__new__')
        obj = int.__new__(cls, value)
        return obj if obj > 0 else -obj
```

```
PositiveInt(-7)
```

```
__new__
```

```
7
```

Przykład ilustrujący gdzie przekazywane są argumenty:

```
class Test:
    def __new__(cls, *args):
        print('__new__', args)
        obj = object.__new__(cls)
        obj.new_attr = "test"
        return obj

    def __init__(self, *args):
        print('__init__', args)
        self.args = args
```

```
t = Test("gadget", 42)
```

```
__new__ ('gadget', 42)
__init__ ('gadget', 42)
```

```
t.args
```

```
('gadget', 42)
```

```
t.new_attr
```

```
'test'
```

Abstract Base Classes

W Pythonie stosowany jest duck-typing, w związku z tym nie ma potrzeby definiowania abstrakcyjnych klas lub interfejsów określających kontrakt. Warto jednak czasami jawnie określić kontrakt, to znaczy stworzyć abstrakcyjną klasę bazową i określić, jakie metody powinny zostać zdefiniowane. Nie tworzy się instancji takiej klasy - służy ona jedynie jako dokumentacja.

```
import abc

class BaseCalculator(abc.ABC):
    @abc.abstractmethod
    def process(self, expr):
        pass

class Calculator(BaseCalculator):
    def process(self, expr):
        return eval(expr)
```

```
c = Calculator()
c.process('2 + 2')
```

```
4
```

Jeżeli w klasie potomnej nie zostanie zdefiniowana wymagana metoda, zostanie rzucony wyjątek.

```
class InvalidCalculator(BaseCalculator):
    pass
```

```
ic = InvalidCalculator()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-101-4973d17f9fb6> in <module>
----> 1 ic = InvalidCalculator()

TypeError: Can't instantiate abstract class InvalidCalculator with abstract␣
 ↪methods process
```

0.1.10 Klasa jako obiekt

W Pythonie wszystko jest obiektem, także klasa. Dlatego możemy mówić o *obiekcie klasy*. *obiekt klasy*, tak samo jak każdy obiekt, może być modyfikowany po jego utworzeniu.

```
class Record:
    pass

Record.name = "John"
```

```
person = Record()
person.name
```

```
'John'
```

```
Record.age = 42
person.age
```

```
42
```

Ma to wpływ na wszystkie instancje.

0.1.11 Class Builders (`@dataclass`)

Moduł `dataclasses` dostarcza dekorator, który umożliwia automatyczne generowanie wybranych metod specjalnych, takich jak `__init__()`, `__repr__()` lub `__eq__()`.

Składowe obiektu, które są wykorzystywane w implementacjach metod, są definiowane na podstawie pól klasy wraz adnotacjami typu.

Na przykład, dla poniższej klasy:

```
from dataclasses import dataclass

@dataclass
class LineItem:
```

(continues on next page)

(continued from previous page)

```
    name: str
    unit_price: float
    quantity: int = 0

    def total(self) -> float:
        return self.unit_price * self.quantity
```

wygenerowana zostanie funkcja `__init__()`, która wygląda następująco:

```
def __init__(self, name: str, unit_price: float, quantity: int = 0):
    self.name = name
    self.unit_price = unit_price
    self.quantity = quantity
```

Stosując dekorator `@dataclass` możemy dużo szybciej tworzyć proste klasy i rozbudowywać je dodając do nich potrzebne metody.

```
line_1 = LineItem("ipad", 7665.0, 1)
repr(line_1)
```

```
"LineItem(name='ipad', unit_price=7665.0, quantity=1)"
```

```
line_1.total()
```

```
7665.0
```

Parametry dekoratora `@dataclass`

Do dekoratora `@dataclass` możemy przekazać następujące paramtetry:

Parametr	Wartość domyślna	Efekt (jeśli `True`)
`init`	`True`	definicja metody `__init__`
`repr`	`True`	definicja metody `__repr__` (jeśli klasa nie definiuje własnej implementacji)
`eq`	`True`	definicja metody `__eq__` (instancje są porównywane jak krotki)
`order`	`False`	definicja metod `__lt__()`, `__le__()`, `__gt__()`, and `__ge__()`
`unsafe_hash`	`False`	definicja metody `__hash__()` (ta opcja implikuje, że instancje powinny być *immutable*)
`frozen`	`False`	próba przypisania wartości dla pola generuje wyjątek (instancje są *immutable*)
`match_args`	`True`	tworzona jest krotka `__match_args__` z listy parametrów metody `__init__()`
`slots`	`True`	generowany jest atrybut `__slots__` i nowa klasa jest zwrócona w miejsce klasy ogryginalnej

```
@dataclass(order=True, unsafe_hash=True, frozen=True)
class Person:
    name: str
    age: int
```

```
people = [Person("John", 33), Person("Eve", 44), Person("Adam", 33)]
sorted(people)
```

```
[Person(name='Adam', age=33),
 Person(name='Eve', age=44),
 Person(name='John', age=33)]
```

```
hash(people[0])
```

```
-5191814165429172402
```

```
people[0].name = "Unknown"
```

```
---------------------------------------------------------------------------
FrozenInstanceError                       Traceback (most recent call last)
<ipython-input-21-2c1af0d06026> in <module>
----> 1 people[0].name = "Unknown"

<string> in __setattr__(self, name, value)

FrozenInstanceError: cannot assign to field 'name'
```

Implementacja składowych

Funkcja `field()` precyzuje sposób, w jaki dane pole jest implementowane.

```
from typing import List
from dataclasses import field


@dataclass(order=True)
class Person:
    name: str
    age: int = field(compare=False)
    friends: List['Person'] = field(default_factory=list)
```

```
p1 = Person("John", 44)
p2 = Person("John", 23)
p1 == p2
```

```
True
```

```
p1.friends.append(p2)
p1
```

```
Person(name='John', age=44, friends=[Person(name='John', age=23, friends=[])])
```

Opcje funkcji `field()`:

- `default` - domyślna wartość dla pola
- `default_factory` - funkcja bezargumentowa, która tworzy domyślną instancję przypisywaną do pola
- `init` - jeśli `True`, pole jest użyte jako parametr metody `__init__()`
- `repr` - jeśli `True`, pole jest użyte w implementacji metody `__repr__()`
- `hash` - jeśli `True`, pole jest użyte w implementacji metody `__hash__()`
- `compare` - jeśli `True`, pole jest użyte w implementacji metod `__eq__()` i innych

Przydatne metody dataclass'y

- `fields()` - zwraca krotkę pól klasy

```
import dataclasses

dataclasses.fields(Person)
```

```
(Field(name='name',type=<class 'str'>,default=<dataclasses._MISSING_TYPE object at␣
↪0x7f8fbd270a60>,default_factory=<dataclasses._MISSING_TYPE object at␣
↪0x7f8fbd270a60>,init=True,repr=True,hash=None,compare=True,metadata=mappingproxy(
↪{}),_field_type=_FIELD),
 Field(name='age',type=<class 'int'>,default=<dataclasses._MISSING_TYPE object at␣
↪0x7f8fbd270a60>,default_factory=<dataclasses._MISSING_TYPE object at␣
↪0x7f8fbd270a60>,init=True,repr=True,hash=None,compare=False,
↪metadata=mappingproxy({}),_field_type=_FIELD),
 Field(name='friends',type=typing.List[ForwardRef('Person')],default=<dataclasses._
↪MISSING_TYPE object at 0x7f8fbd270a60>,default_factory=<class 'list'>,init=True,
↪repr=True,hash=None,compare=True,metadata=mappingproxy({}),_field_type=_FIELD))
```

- `asdict()` - konweruje instancję dataclass'y do słownika

```
dataclasses.asdict(p1)
```

```
{'name': 'John',
 'age': 44,
 'friends': [{'name': 'John', 'age': 23, 'friends': []}]}
```

- `astuple()` - konwertuje instancję do krotki

```
dataclasses.astuple(p2)
```

```
('John', 23, [])
```

- `replace(obj, **changes)` - tworzy nową instancję na podstawie `obj` ze zmienionymi wartościami pól z `**changes`

```
dataclasses.replace(p1, age = 88)
```

```
Person(name='John', age=88, friends=[Person(name='John', age=23, friends=[])])
```

Post-init

Metoda `__post_init__()` umożliwia inicjalizację tzw. pól wyliczanych:

```
@dataclass
class LineItem:
    name: str
    unit_price: float
    quantity: int = 0
    total: float = field(init=False)

    def __post_init__(self):
        self.total = self.unit_price * self.quantity
```

```
line_1 = LineItem("ipad", 7999.99, 2)
line_1.total
```

```
15999.98
```

Pola klasy

```
from typing import ClassVar


@dataclass
class Entity:
    id: int
    count: ClassVar[int] = 0

    def __new__(cls, *args, **kwargs):
        cls.count += 1
        return super().__new__(cls)
```

```
entities = [Entity(id) for id in range(1, 11)]
```

```
Entity.count
```

```
10
```

0.2 Optymalne wykorzystanie wbudowanych typów Pythona

0.2.1 Szybkość działania

Python ma opinię języka, w którym szybko się tworzy, natomiast wykonywany kod jest powolny. Obecnie komputery są na tyle szybkie, że w większości przypadków można zapomnieć o szybkości działania programu. Czas programisty jest cenniejszy od czasu komputera.

Jednak przy przetwarzaniu dużej ilości danych może okazać się, że program wykonuje się za wolno. Pisząc program, którego zadaniem będzie przetwarzanie dużej ilości danych, warto mieć na uwadze, jak szybko wykonują się operacje na kolekcjach, takich jak lista, słownik czy zbiór.

0.2.2 Wydajność operacji na typach wbudowanych

Notacja "Big O"

Notacja *O* pozwala na zwięzłe opisanie, jak szybko wykonuje się dana operacja. Zapis *O(1)* oznacza, że dana operacja będzie wykonywała się zawsze tak samo długo, niezależnie od tego, jak duża jest kolekcja. Zapis *O(n)* oznacza, że czas wykonania danej operacji zależy liniowo od wielkości kolekcji. Na przykład, na pustej liście może ona zająć średnio 50 mikrosekund, na jednoelementowej 60 mikrosekund, na dwuelementowej 70 mikrosekund itd.

Wbudowane algorytmy - lista

Operation	Average Case	Amortized Worst Case
Copy	O(n)	O(n)
Append[1]	O(1)	O(n)
Insert	O(n)	O(n)
Get Item	O(1)	O(1)
Set Item	O(1)	O(1)
Delete Item	O(n)	O(n)
Iteration	O(n)	O(n)
Get Slice	O(k)	O(k)
Del Slice	O(n)	O(n)
Set Slice	O(k+n)	O(k+n)
Extend[1]	O(k)	O(k)
Sort	O(n log n)	O(n log n)
Multiply	O(nk)	O(nk)
x in s	O(n)	O(n)
min(s), max(s)	O(n)	O(n)
Get Length	O(1)	O(1)

k oznacza wielkość wycinka.

Wbudowane algorytmy - słownik

Operation	Average Case	Amortized Worst Case
Copy[2]	O(n)	O(n)
Get Item	O(1)	O(n)
Set Item[1]	O(1)	O(n)
Delete Item	O(1)	O(n)
Iteration[2]	O(n)	O(n)

[1] = Pojedyncza operacja może trwać długo, w zależności od poprzednich wartości słownika

[2] = Dla tych operacji *n* oznacza największą pojemność, jaką kiedykolwiek osiągnął słownik, a nie bieżącą.

Wbudowane algorytmy - zbiór

Operation	Average Case	Amortized Worst Case
x in s	O(1)	O(n)
Union s|t	O(len(s)+len(t))	O(len(s)+len(t))
Intersection s&t	O(min(len(s), len(t))	O(len(s) * len(t))
Difference s-t	O(len(s))	O(len(s))
s.difference_update(t)	O(len(t))	O(len(t))

0.2.3 Moduł collections

Opis modułu

Moduł *collections* zawiera wyspecjalizowane typy danych. Są one alternatywą dla wbudowanych typów danych takich jak `dict`, `list`, `set` i `tuple`.

Nazwa typu	Opis
`namedtuple()`	Funkcja służąca do tworzenia krotek z nazwanymi polami.
`deque`	Kontener podobny do listy, z dostępem zarówno od początku, jak i od końca.
`Counter`	Klasa słownika służąca do liczenia obiektów.
`OrderedDict`	Klasa słownika, która pamięta kolejność, w jakiej dodawano elementy.
`defaultdict`	Klasa słownika, która umożliwia zdefiniowanie brakujących elementów.

`namedtuple`

Nazwane krotki są stosowane wszędzie tam, gdzie potrzebujemy manipulować niewielkimi rekordami, a jednocześnie nie chcemy tworzyć osobnej klasy.

Przykładem jest klasa `Point`:

```
from collections import namedtuple
import math

Point = namedtuple('Point', 'x y z')

def distance(a, b):
    return math.sqrt(
        (a.x - b.x)**2 +
        (a.y - b.y)**2 +
        (a.z - b.z)**2
    )
```

```
a = Point(x=1, y=2, z=3)
b = Point(-1, -2, 42)
print(distance(a, b))
```

```
39.25557285278104
```

Nazwane krotki przydają się także wtedy, gdy chcemy zwrócić więcej niż jeden obiekt. Funkcja lub metoda może zwrócić kilka obiektów w krotce, ale trzeba wówczas pamiętać, w jakiej kolejności są one zwracane. Alternatywą jest zwrócenie nazwanej krotki:

```
from collections import namedtuple

Result = namedtuple('Result', 'quotient remainder')

def div_mod(a, b):
    return Result(quotient=a//b, remainder=a%b)
```

```
r = div_mod(9, 2)
r
```

```
Result(quotient=4, remainder=1)
```

```
r.quotient
```

```
4
```

```
r.remainder
```

```
1
```

0.2.4 Deque

W niektórych algorytmach stosowane są kolejki FIFO (*first in, first out*). Kolejka jest listą, której nowe elementy są dodawane na jej początku. Z kolei inny wątek może pobierać elementy (odczytywać je i usuwać z listy) z jej końca.

Stosowanie listy może spowolnić program, ponieważ dodanie lub usunięcie elementu z początku listy jest operacją kosztowną (złożoność *O(n)*). Dlatego w takich sytuacjach stosuje się wyspecjalizowany kontener `deque`. Jego zaletą jest możliwość szybkiego (w czasie stałym *O(1)*) dodawania i usuwania elementów zarówno z początku, jak i końca.

[deque]{.title-ref} udostępnia ten sam interfejs, co listy. Dostępne są również metody `appendleft`, `extendleft` oraz `popleft`, które działają tak samo jak odpowiednio `append`, `extend` i `pop`, ale działają na początku kolekcji zamiast na jej końcu.

Na przykład, `appendleft(x)` wstawia obiekt *x* na początek kolejki. Jest to równoważne wywołaniu `insert(0, x)`.

```
import collections

dq = collections.deque('abcdefg')
dq
```

```
deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
```

```
len(dq)
```

```
7
```

```
dq[0]
```

```
'a'
```

```
dq[-1]
```

```
'g'
```

```
dq.remove('c')
dq
```

```
deque(['a', 'b', 'd', 'e', 'f', 'g'])
```

```
while True:
    try:
        print(dq.pop())
    except IndexError:
        break
```

```
g
f
e
d
b
a
```

```
dq = collections.deque('abcdefg')

while True:
    try:
        print(dq.popleft())
    except IndexError:
        break
```

```
a
b
c
d
e
f
g
```

```
dq = collections.deque(range(10))
dq
```

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

```
dq = collections.deque(range(10))
dq.rotate(2)
dq
```

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

```
dq = collections.deque(range(10))
dq.rotate(-2)
dq
```

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

0.2.5 Counter

`Counter` jest klasą ułatwiającą zliczanie elementów. Jest to słownik, którego kluczami są elementy, a wartościami częstość ich występowania. Przy tworzeniu instancji należy przekazać kolekcję elementów, które mają zostać zliczone.

Zliczanie znaków występujących w danym napisie można wykonać przy pomocy zwykłego słownika:

```
counter = {}

for char in "Hello there, People!":
    if char not in counter:
        counter[char] = 1
    else:
        counter[char] += 1

print(counter)
```

```
{'H': 1, 'e': 5, 'l': 3, 'o': 2, ' ': 2, 't': 1, 'h': 1, 'r': 1, ',': 1, 'P': 1, 'p
↪': 1, '!': 1}
```

Lub szybciej, przy pomocy klasy `Counter`:

```
from collections import Counter

counter = Counter("Hello there, People!")

print(counter)
```

```
Counter({'e': 5, 'l': 3, 'o': 2, ' ': 2, 'H': 1, 't': 1, 'h': 1, 'r': 1, ',': 1, 'P
↪': 1, 'p': 1, '!': 1})
```

`most_common` jest metodą umożliwiającą sprawdzenie, które elementy występują najczęściej. Jeśli chcemy znaleźć najczęściej występujące słowa w pliku `holmes.txt`, możemy wykonać:

```
import re

words = re.findall('\w+', open('holmes.txt').read().lower())

print(Counter(words).most_common(10))
```

```
[('the', 5810), ('and', 3088), ('i', 3038), ('to', 2823), ('of', 2778), ('a',␣
↪2700), ('in', 1823), ('that', 1767), ('it', 1749), ('you', 1572)]
```

0.2.6 OrderedDict

Słowniki przechowują pary *(klucz, wartość)*, ale nie pamiętają kolejności, w jakiej poszczególne pary zostały dodane. `OrderedDict` jest słownikiem, który zapamiętuje tę kolejność. Podczas iterowania po nim, zwraca on klucze w kolejności, w jakiej zostały dodane:

```
from collections import OrderedDict

d = OrderedDict()
d['c'] = 1
d['b'] = 2
d['a'] = 1

for k in d:
    print(k)
```

```
c
b
a
```

Dodatkowo, ten obiekt udostępnia metodę `popitem`:

```
d.popitem()  # zwraca ostanio dodaną parę (klucz, wartość) (kolejka LIFO)
```

```
('a', 1)
```

```
d.popitem(last=False)  # zwraca pierwszą dodaną parę (kolejka FIFO)
```

```
('c', 1)
```

0.2.7 defaultdict

`defaultdict` jest takim słownikiem, w którym użycie klucza nie będącego w tym słowniku powoduje użycie domyślnej wartości zamiast zgłoszenia wyjątku `KeyError`.

Ten obiekt pozwala uprościć kod wykorzystujący ideę multisłownika, to znaczy słownika, w którym jeden klucz może mieć więcej niż jedną wartość. Taki multisłownik może zostać zaimplementowany jako zwykły słownik, którego wartościami jest lista.

```
s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]
res = {}

for color, item in s:
    if color in res:
        res[color].append(item)
    else:
```

(continues on next page)

(continued from previous page)

```
        res[color] = [item, ]

res.items()
```

```
dict_items([('yellow', [1, 3]), ('blue', [2, 4]), ('red', [1])])
```

Rozwiązanie z użyciem `defaultdict` jest znacznie prostsze. Przy tworzeniu słownika przekazujemy funkcję, która generuje wartość domyślną. W tym przypadku wartością domyślną jest pusta lista, którą generuje wbudowana funkcja `list`.

```
from collections import defaultdict

s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

d = defaultdict(list)
for k, v in s:
    d[k].append(v)

print(d)
```

```
defaultdict(<class 'list'>, {'yellow': [1, 3], 'blue': [2, 4], 'red': [1]})
```

0.3 Funkcje - elementy zaawansowane

0.3.1 Funkcje jako obiekty

Funkcje są w Pythonie "first-class objects". Oznacza to, że funkcje:

- można przekazywać jako argument do innych funkcji, np. do funkcji print,
- mogą być rezultatem działania innej funkcji, np. dekoratory,
- mogą być dynamicznie tworzone, np. funkcje zagnieżdżone.

```
def foo():
    print("foo is happening")

foo()
```

```
foo is happening
```

```
type(foo)
```

```
function
```

```
bar = foo

id(bar) == id(foo)
```

```
True
```

```
print(foo)
```

```
<function foo at 0x7fbe045301f0>
```

```
print(foo)
```

```
<function foo at 0x7fbe045301f0>
```

0.3.2 Atrybuty funkcji

W szczególności, funkcje są instancjami typu function i mają atrybuty:

```
foo.__class__
```

```
function
```

```
foo.__name__
```

```
'foo'
```

```
bar.__name__
```

```
'foo'
```

0.3.3 Klasy jako funkcje

```
class CallableClass:
    def __init__(self):
        self._counter = 0

    def __call__(self):
        self._counter += 1
        print("You have called me {0} times".format(self._counter))
```

```
callable_object = CallableClass()
callable_object()
```

```
You have called me 1 times
```

```
callable_object()
```

```
You have called me 2 times
```

Preferowane jest tworzenie i używanie zwykłych funkcji. Później zawsze istnieje możliwość przekształcenia takiej funkcji w instancję klasy z metodą `__call__`.

0.3.4 Wywoływanie funkcji

`callable` jest wbudowaną funkcją - nie trzeba jej importować, jest zawsze dostępna. Pozwala sprawdzić, czy dany obiekt da się wywołać (czy jest funkcją lub klasą lub instancją klasy z metodą `__call__`):

```
callable(foo)
```

```
True
```

```
callable(callable_object)
```

```
True
```

```
x = 2
callable(x)
```

```
False
```

W Pythonie 3.0 i 3.1 usunięto funkcję `callable`, jednak w Pythonie 3.2+ stała się ponownie wbudowaną funkcją. Jeżeli Twój kod musi działać także pod Pythonem 3.0, 3.1, wówczas należy użyć jednego z dwóch poniższych idiomów:

```
hasattr(foo, '__call__')
```

```
True
```

```
import collections.abc

isinstance(foo, collections.abc.Callable)
```

```
True
```

0.3.5 Funkcje zagnieżdżone

Funkcje można zagnieżdżać, to znaczy zdefiniować jedną funkcję (wewnętrzną) w ciele drugiej funkcji (zewnętrznej).

Ponieważ funkcje są obiektami (*first-class citizen*), funkcja wewnętrzna może zostać zwrócona przez funkcję zewnętrzną.

Jest to szczególnie użyteczne przy tworzeniu *dekoratorów*.

Warto zauważyć, że w poniższym przykładzie funkcja `add` jest tworzona dynamicznie przy każdym wywołaniu funkcji `bind_add`.

Oznacza to, że przy każdym wywołaniu `bind_add` zwracana jest inna funkcja, mającą własną tożsamość (*identity*), co można sprawdzić przy pomocy wbudowanej funkcji `id`.

```
def bind_add(a):
    def add(b):
        return a + b
    return add
```

```
add_1 = bind_add(1)
type(add_1)
id(add_1)
```

```
140454093068400
```

```
add_1(5)
```

```
6
```

```
add_42 = bind_add(42)
id(add_42)
```

```
140454093068688
```

```
add_42(58)
```

```
100
```

0.3.6 Funkcje wyższego rzędu

Funkcja wyższego rzędu wymaga podania jako argumentu innej funkcji lub zwraca funkcję jako rezultat. Przykładową funkcją wyższego rzędu w Pythonie jest funkcja `sorted`. Opcjonalny argument `key` umożliwia przekazanie funkcji, która będzie wywołana dla każdego sortowanego elementu

```
gadgets = ['mp3', 'smartwatch', 'ipod', 'pendrive', 'ipad']
sorted(gadgets, key=len)
```

```
['mp3', 'ipod', 'ipad', 'pendrive', 'smartwatch']
```

Często jako argumenty funkcji wyższego rzędu przekazywane są wyrażenia lambda.

```
lst_numbers = [(0, "zero"), (1, "one"), (2, "two"), (3, "three"), (4, "four"), (5,
↪"five")]
sorted(lst_numbers, key=lambda item : item[1])
```

```
[(5, 'five'), (4, 'four'), (1, 'one'), (3, 'three'), (2, 'two'), (0, 'zero')]
```

0.3.7 Wyrażenia lambda

Wyrażenia lambda pozwalają na zwięzłe stworzenie funkcji bez nazwy, tzw. funkcji anonimowej.

```
add = lambda a, b: a + b

add(2, 3)
```

```
5
```

```
type(add)
```

```
function
```

```
callable(add)
```

```
True
```

```
add.__name__
```

```
'<lambda>'
```

W ciele funkcji nie można umieścić instrukcji, a jedynie pojedyncze wyrażenie (np. `a + b`), które jest rezultatem takiej funkcji.

Dlatego wyrażenia lambda najczęściej wykorzystuje się razem z funkcjami wyższego rzędu `filter` i `map`.

Każdy element z jakiejś kolekcji może zostać przekształcony za pomocą funkcji (`map`) albo przefiltrowany przy pomocy predykatu (`filter`).

```
numbers = [1, -3, 4, -5, 0, 8, 42, 665, 54, -65]
positive_numbers = filter(lambda n: n > 0, numbers)
list(positive_numbers)
```

```
[1, 4, 8, 42, 665, 54]
```

```
squares = map(lambda n: n * n, numbers)
list(squares)
```

```
[1, 9, 16, 25, 0, 64, 1764, 442225, 2916, 4225]
```

Lepiej jest jednak zastąpić `filter` i `map` wyrażeniami listowymi lub generatorowymi, ponieważ wywoływanie funkcji w Pythonie jest związane z dużym narzutem czasowym. Użycie wyrażeń listowych lub generatorowych nie powoduje wielokrotnego wywoływania funkcji.

```
[x for x in numbers if x > 0]
```

```
[1, 4, 8, 42, 665, 54]
```

```
[x * x for x in numbers]
```

```
[1, 9, 16, 25, 0, 64, 1764, 442225, 2916, 4225]
```

0.3.8 Zmienne lokalne, nielokalne i globalne

Python korzysta z przestrzeni nazw (*namespace*), aby śledzić zmienne. Są to słowniki, których kluczami są nazwy zmiennych, a wartościami wartości tych zmiennych. W środku funkcji mamy dostępu do wielu przestrzeni nazw.

Najważniejszą z nich jest lokalna przestrzeń nazw, która zawiera argumenty funkcji i lokalnie zdefiniowane zmienne. Zmienne z tej przestrzeni nie są widoczne na zewnątrz funkcji.

Globalna przestrzeń nazw zawiera wszystkie zmienne zdefiniowane w module. Są to wszystkie zmienne, które nie są "wcięte". Funkcje i klasy to także obiekty, więc w tej przestrzeni nazw znajdują się również one.

W przypadku funkcji zagnieżdżonych, w środku wewnętrznej funkcji możemy mieć do czynienia z przestrzenią nazw zewnętrznej funkcji. Nie jest to ani globalna, ani lokalna przestrzeń.

Gdy odwołujemy się do zmiennej, Python musi zdecydować, z której przestrzeni ma skorzystać. Jeżeli próbujemy odczytać wartość zmiennej, wówczas wykorzystywana jest najbliższa przestrzeń, w której dana zmienna jest zadeklarowana. Najbliższą jest lokalna przestrzeń nazw, potem nielokalne i na końcu globalna.

Jeżeli przypisujemy coś do zmiennej, to Python zakłada, że chcemy ją stworzyć w przestrzeni lokalnej, chyba że użyjemy słów kluczowych `nonlocal` lub `global`.

```
global_var = 2
var = 4

def outer():
    nonlocal_var = 3

    def inner():
        global global_var
        nonlocal nonlocal_var
        global_var = -2  # modyfikujemy zmienną globalną
        var = -4  # tworzymy zmienną lokalną niezależną od zmiennej globalnej var = 3
        nonlocal_var = -3  # modyfikujemy zmienną nielokalną
        print("inner", global_var, nonlocal_var, var)

    inner()

    print("outer", global_var, nonlocal_var, var)
outer()

print("global", global_var, var)
```

```
inner -2 -3 -4
outer -2 -3 4
global -2 4
```

Warto zauważyć, że decyzja o tym, która przestrzeń nazw zostanie wykorzystana, jest podejmowana już w czasie kompilacji funkcji:

```
x = 2

def foo():
    print(x)
    x = 3

foo()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Cell In[33], line 7
      4     print(x)
      5     x = 3
----> 7 foo()

Cell In[33], line 4, in foo()
      3 def foo():
----> 4     print(x)
      5     x = 3

UnboundLocalError: local variable 'x' referenced before assignment
```

W powyższym przykładzie Python założył, że `x` jest zmienną lokalną, ponieważ w środku funkcji znajduje się przypisanie do tej zmiennej. `print(x)` odwołuje się dalej do zmiennej lokalnej, a nie globalnej.

0.3.9 Parametry kontra argumenty

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

x = 3
y = 2

add(x, y)
```

```
5
```

`a` i `b` są parametrami funkcji, natomiast `x` i `y` argumentami.

0.3.10 Parametry funkcji

Wprowadzenie

W Pythonie rozróżniamy cztery różne typy parametrów:

- normalne (*normal parameters*) mają nazwę i pozycję
- nazwane (*keyword parameters*) mają nazwę i domyślną wartość
- zmienne (*variable parameters*) poprzedzone gwiazdką *, mają pozycję
- zmienne nazwane (*variable keyword parameters*) poprzedzone **, mają nazwę

Parametry normalne i nazwane

```
def generate_signature(person, year=2000, place="Paris"):
    print(person, year, place)

generate_signature("Ola", 1995, "Wrocław")
```

```
Ola 1995 Wrocław
```

```
generate_signature("Ala")
```

```
Ala 2000 Paris
```

```
generate_signature("Olek", place="New York")
```

```
Olek 2000 New York
```

```
generate_signature("Alek", year=2010)
```

```
Alek 2010 Paris
```

Parametry zmienne (*args)

Operator * służy do tworzenia funkcji akceptujących dowolną liczbę argumentów:

```
def my_sum(*numbers):
    total = 0
    for number in numbers:
        total += number
    return total

my_sum(1, 2, 3, 4)
```

```
10
```

Parametry zmienne nazwane (**kwargs)

Operator ** służy do tworzenia funkcji akceptujących dowolną liczbę argumentów nazwanych:

```
def dict_without_Nones(**kwargs):
    result = {}
    for k, v in kwargs.items():
        if v is not None:
            result[k] = v
    return result

dict_without_Nones(a="1999", b="2000", c=None)
```

```
{'a': '1999', 'b': '2000'}
```

Parametry zmienne razem (*args, **kwargs)

Funkcja może przyjmować jednocześnie parametry zmienne i zmienne nazwane:

```
def multi(first, *args, **kwargs):
    print(first)
    print(args)
    print(kwargs)

multi(1, 2, 3, 4, 5, ala="1999", ola="2000")
```

```
1
(2, 3, 4, 5)
{'ala': '1999', 'ola': '2000'}
```

0.3.11 Pułapki domyślnego atrybutu

Domyślna wartość dla argumentów nazwanych jest wyliczana w momencie deklarowania funkcji. Wartość ta nie jest ponownie wyliczana przy wywoływaniu funkcji. Zachowanie to nie jest intuicyjne:

```
def its_a_trap(item, seq=[]):
    seq.append(item)
    print(seq)
```

```
its_a_trap(1)
```

```
[1]
```

```
its_a_trap(2)
```

```
[1, 2]
```

Dlatego jako wartości domyślnych należy używać tylko niemodyfikowalnych obiektów, takich jak prymitywne wartości (`0, True, None, 'string'` itp.).

Jeżeli wartość domyślna musi koniecznie być modyfikowalnym obiektem (np. listą), wówczas należy domyślnie użyć `None` i w środku funkcji przypisać pożądaną wartość:

```
def now_its_fine(item, seq=None):
    if seq is None:
        seq = []
    seq.append(item)
    print(seq)
```

```
now_its_fine(1)
```

```
[1]
```

```
now_its_fine(2)
```

```
[2]
```

0.3.12 Adnotacje funkcji

W Pythonie 3 wprowadzono składnię pozwalającą na powiązanie argumentów funkcji i metod oraz zwracaną wartość z dowolnym obiektem. W szczególności, dla każdego:

```
def clip(text:str, max_len:'int > 0'=80) -> str:
    return text[:max_len]
```

Adnotacje funkcji (*function annotations*) są nietypową funkcjonalnością, ponieważ nie określono, do czego konkretnie takie adnotacje mogą zostać użyte.

Adnotacje są dostępne jako specjalny atrybut:

```
clip.__annotations__
```

```
{'text': str, 'max_len': 'int > 0', 'return': str}
```

Przykładowe zastosowanie to dodanie informacji o typach (statyczne typowanie).

Dzięki temu narzędzia takie jak `mypy` mogą zanalizować kod, sprawdzić zgodność typów i w ten sposób wykryć ewentualne błędy jeszcze przed uruchomieniem kodu.

0.3.13 Atrybuty funkcji

W Pythonie wszystko jest obiektem, także funkcje.

Funkcje posiadają specjalne atrybuty ułatwiające ich introspekcję:

```
def foo(arg, kwarg=42, *, kwarg2=43):
    '''docstring'''
    return arg + kwarg + kwarg2
```

```
foo.__name__
```

```
'foo'
```

```
foo.__doc__
```

```
'docstring'
```

```
foo.__defaults__
```

```
(42,)
```

```
foo.__kwdefaults__
```

```
{'kwarg2': 43}
```

0.4 Elementy programowania funkcyjnego

0.4.1 Iteratory

Pętla `for` pozwala w Pythonie na *iterowanie* po elementach jakiejkolwiek sekwencji i wykonanie pewnych operacji dla każdego jej elementu.

Iteracja po liście

Iterować można po liście:

```
for x in [1,4,5,10]:
    print(x, end=' ')
```

```
1 4 5 10
```

Iteracja po słowniku

Iterując po słowniku, uzyskujemy dostęp do jego kluczy:

```
prices = { 'GOOG' : 490.10,
    'AAPL' : 145.23,
    'YHOO' : 21.71
}

for key in prices:
    print(key)
```

```
GOOG
AAPL
YHOO
```

Iteracja po stringu

String (napis) można traktować jako listę znaków. Iterując po napisie, uzyskujemy dostęp do poszczególnych znaków:

```
text = "Yow!"

for character in text:
    print(character)
```

```
Y
o
w
!
```

Iteracja po pliku

Iterować można nie tylko po kolekcjach, ale także obiektach, które w jakiś sposób reprezentują zbiór obiektów. Na przykład, plik można traktować jako zbiór linii. W wyniku iteracji po pliku otrzymujemy linie (razem ze znakiem końca wiersza):

```
for line in open("real.txt"):
    print(line, end='')
```

```
Real Programmers write in FORTRAN
Maybe they do now,
in this decadent era of
Lite beer, hand calculators, and "user-friendly" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes,
Real Programmers wrote in machine code.
Not FORTRAN. Not RATFOR. Not, even, assembly language.
Machine Code.
Raw, unadorned, inscrutable hexadecimal numbers.
Directly.
```

Protokół iteracji

Możliwość iterowania po różnych obiektach wynika z istnienia ścisłego protokołu. Iterować można po każdym obiekcie, który spełnia ten protokół. W szczególności, instancje Twoich własnych klas również mogą być iterowalne.

```
items = [1, 4, 5]

iterator = iter(items)
next(iterator)
```

```
1
```

```
next(iterator)
```

```
4
```

```
next(iterator)
```

```
5
```

```
next(iterator)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[8], line 1
----> 1 next(iterator)

StopIteration:
```

Wbudowana funkcja `iter(x)` wywołuje `x.__iter__()`.

Z kolei `next(x)` deleguje do `x.__next__()` pod Pythonem 3 lub do `x.next()` w przypadku Pythona 2.

```
items = [1, 4, 5]
iterator = items.__iter__()
iterator.__next__() == 1
iterator.__next__()
iterator.__next__() == 5
iterator.__next__()
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-15-600f2bbc793f> in <module>
      4 iterator.__next__()
      5 iterator.__next__() == 5
----> 6 iterator.__next__()

StopIteration:
```

Protokół składa się z dwóch metod:

- Obiekt, który ma być iterowalny, musi mieć metodę `__iter__()`, która powinna zwrócić *iterator*.
- Iterator powinien mieć metodę `__next__()` (lub `next()` w Pythonie 2) zwracającą przy kolejnych wywołaniach kolejne elementy. Jeżeli wszystkie elementy zostały już zwrócone, powinien zostać zgłoszony wyjątek `StopIteration`.

Iterator może być tym samym obiektem, co iterowany obiekt.

W takiej sytuacji implementacja metody `__iter__()` sprowadza się do zwrócenia tego obiektu:

```
class Foo:
    def __iter__(self):
        return self

    def __next__(self):
        """get next element"""
```

Należy jednak pamiętać, że po obiekcie takiej klasy można iterować tylko raz (tak jak w przypadku wyrażeń generatorowych).

Iteracja po własnych typach

Poniżej zostanie przedstawiona implementacja klasy `Countdown` umożliwiającej odliczenia w dół.

Przykład użycia takiej klasy:

```
for i in Countdown(10):
    print(i, end=' ')
```

Implementacja wykorzystuje trik przestawiony wcześniej, to znaczy metoda `__iter__()` zwraca ten sam obiekt. Dzięki temu iterator jest tym samym obiektem, po którym iterujemy. W konsekwencji, nie ma potrzeby pisania dwóch osobnych klas.

```
class Countdown:
    def __init__(self,start):
        self.count = start

    def __iter__(self):
        return self

    def __next__(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

```
for i in Countdown(10):
    print(i, end=' ')
```

```
10 9 8 7 6 5 4 3 2 1
```

Wbudowane funkcje używające obiektów iterowalnych

Python posiada wbudowane funkcje, to znaczy takie, których nie trzeba importować. Niektóre z nich operują na dowolnych obiektach iterowalnych, w szczególności na kolekcjach.

Funkcje `sum`, `min` i `max` agregują przekazaną kolekcję i zwracają jedną wartość (odpowiednio sumę elementów, najmniejszy i największy element). Dwie ostatnie funkcje generują `ValueError`, jeżeli przekazana kolekcja jest pusta.

Funkcje `list`, `tuple`, `set` i `dict` służą do stworzenia nowej kolekcji danego typu. Jeżeli nie zostanie podany żaden element, zwrócona zostanie pusta kolekcja (nie zawierająca żadnego elementu). Jednak najczęściej podaje się jeden argument (dowolny iterowalny obiekt).

Często dysponujemy *generatorami*, to znaczy obiektami przypominającymi kolekcje, ale wyliczającymi elementy na żądanie. Generatory zostaną szczegółowo omówione w następnym rozdziale. Generatory są zwracane na przykład przez funkcje `filter`, `map` i `zip`. Jeżeli chcemy wyświetlić elementy takiego generatora, możemy "przekonwertować" go na listę przy użyciu funkcji `list`:

```
a = [1, 2, 3]
b = ['a', 'b', 'c']
ab_zipped = zip(a, b)
list(ab_zipped)
```

```
[(1, 'a'), (2, 'b'), (3, 'c')]
```

0.4.2 Generatory

Generator jest funkcją, która zwraca sekwencję wyników zamiast pojedynczej wartości. Wewnątrz generatora używana jest instrukcja `yield` zamiast `return`. Służy ona do zwracania kolejnych wartości.

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
```

```
for i in countdown(5):
    print(i, end=' ')
```

```
5 4 3 2 1
```

Wywołanie funkcji generatora tworzy obiekt generatora, ale nie rozpoczyna działania tej funkcji. Przy pierwszym wywołaniu metody `__next__()` następuje wykonanie funkcji generatora aż do napotkania instrukcji `yield`. Wtedy wykonywanie funkcji zostaje wstrzymane, a wartość zwrócona. Przy kolejnych wywołaniach metody `__next__()` następuje wznowienie generatora z miejsca, w którym został on poprzednio wstrzymany.

```
def countdown(n):
    print('start countdown')
    while n > 0:
        print('before yield')
        yield n
        print('after yield')
        n -= 1
```

```
it = countdown(3)
it
```

```
<generator object countdown at 0x7fc079107350>
```

```
next(it)
```

```
start countdown
before yield
```

```
3
```

```
next(it)
```

```
after yield
before yield
```

```
2
```

```
next(it)
```

```
after yield
before yield
```

```
1
```

```
next(it)
```

```
after yield
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-29-bc1ab118995a> in <module>
----> 1 next(it)

StopIteration:
```

yield from

Jeżeli chcemy na raz zwrócić więcej niż jedną wartość, można użyć instrukcji `yield from`.

Jest ona szczególnie przydatna, jeżeli chcemy zwrócić rezultat innego generatora.

```
def flatten_gen():
    yield from ['A', 'B']
    yield from 'CDE'
    yield from range(1, 4)
```

```
list(flatten_gen())
```

```
['A', 'B', 'C', 'D', 'E', 1, 2, 3]
```

Generatory a iteratory

Funkcja generatorowa (lub po prostu generator) różni się od obiektu, który wspiera iterację.

Generator jest operacją jednorazową. Można iterować po generowanych danych tylko raz. Ponowna iteracja wymaga wywołania funkcji generatorowej.

0.4.3 Wyrażenia generatorowe

Wprowadzenie

Wyrażenie generatorowe to generatorowa wersja wyrażenia listowego. Wyrażenie generatorowe zwraca generator, który wylicza kolejne elementy na żądanie.

```
numbers = [1, 2, 3, 4, 5]
squares = [n * n for n in numbers]
squares
```

```
[1, 4, 9, 16, 25]
```

Zamiast tworzyć listę `numbers` i zużywać pamięć można użyć wyrażenia generatorowego:

```
squares_generator = (n*n for n in numbers)
squares_generator
```

```
<generator object <genexpr> at 0x7fc079109190>
```

```
for s in squares_generator:
    print(s, end=' ')
```

```
1 4 9 16 25
```

Wyrażenia generatorowe przydają się przy pracy na dużej ilości danych (np. z dużymi plikami). Jeżeli nie jest możliwe załadowanie wszystkich danych do pamięci, wówczas nie możemy ich przechowywać w liście. Zamiast tego, można użyć wyrażeń generatorowych.

Z drugiej strony, generatory są mniej wygodne, ponieważ można iterować po nich tylko raz.

Składnia

Podobnie jak w przypadku wyrażeń listowych czy słownikowych, możliwe jest kilkukrotne, "zagnieżdżone" iterowanie. Typowym przykładem jest macierz, którą w Pythonie reprezentujemy jako listę list. Wymaga to najpierw iterowania po macierzy, aby uzyskać dostęp do wewnętrznych list reprezentujących poszczególne wiersze lub kolumny, a następnie po poszczególnych wierszach/kolumnach.

Składnia wyrażeń generatorowych jest następująca:

```
(expression for i in s if cond1
            for j in t if cond2
            ...
            if condfinal)
```

Powyższy kod jest równoważny:

```
for i in s:
    if cond1:
        for j in t:
            if cond2:
                ...
                if condfinal:
                    yield expression
```

Co ciekawe, nawiasy można pominąć, jeżeli wyrażenie generatorowe jest jedynym argumentem funkcji:

```
sum((n * n for n in numbers))
```

```
55
```

```
sum(n * n for n in numbers)
```

```
55
```

Funkcje `filter` i `map`

Przy użyciu funkcji `filter` i `map` można wykonać te same operacje, co z użyciem wyrażeń generatorowych. Bardzo często korzysta się wówczas z wyrażenia lambda, pozwalającego na zwięzłe stworzenie anonimowej funkcji:

```
numbers = [1, -3, 4, 5, 42, -665, 5, 3, -7]
positive_numbers = filter(lambda x: x > 0, numbers)
list(positive_numbers)
```

```
[1, 4, 5, 42, 5, 3]
```

W Pythonie 3 obie funkcje zwracają generator. Jest to inne zachowanie niż w Pythonie 2, gdzie zwracana jest lista.

Ze względu na wydajność warto zastąpić `filter` i `map` wyrażeniami generatorowymi. Unikamy narzutu związanego z wielokrotnym wywoływaniem funkcji.

0.4.4 Moduł `itertools`

Python posiada wiele wbudowanych funkcji zwracających iteratory, na przykład `zip`, `map` lub `filter`. Jest wiele innych przydatnych funkcji, które są dostępne w module `itertools` stanowiącym część standardowej biblioteki. Poniżej zostały omówione najważniejsze z nich.

count

`count` jest jak `range`, ale zwraca nieskończony iterator (nie podajemy końcowego indeksu). Podajemy jedynie pierwszy zwracany element (domyślnie zero) oraz krok (domyślnie jeden). Jeżeli nie podamy żadnych argumentów, dostaniemy iterator zwracający kolejne liczby naturalne od zera. Poniżej przedstawiono prosty kalkulator działający w nieskończonej pętli:

```
import itertools

limit = 1000

def find_nth(limit):
    total = 0
    for iter in itertools.count(1):
        total += iter
        if total > limit:
            return iter

n = find_nth(limit)
print(f"Sum of {n} consecutive integers starting from 1 is greater than {limit}")
```

```
Sum of 45 consecutive integers starting from 1 is greater than 1000
```

O ile `range` działa tylko na liczbach całkowitych, to w przypadku `count` krok może być liczbą zmiennoprzecinkową.

islice

Tworzy iterator, który umożliwia zwrócenie określonych `n` elementów (wycinka) z iterowalnego obiektu (np. generatora):

```
from itertools import islice
from typing import Optional

def fibonacci(limit: Optional[int] = None):
    a, b = 0, 1
    while limit is None or b <= limit:
         a, b = b, a+b
         yield a

list(islice(fibonacci(), 15))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610]
```

chain

Przy użyciu operatora + można połączyć (dokonać konkatenacji) dwie listy:

```
a = [1, 2, 3, 4]
b = [5, 6, 7]

a + b
```

```
[1, 2, 3, 4, 5, 6, 7]
```

W przypadku dwóch iteratorów, nie możemy użyć operatora +. Zamiast tego, należy użyć funkcji `chain`:

```
a = range(1, 5)
b = range(5, 8)

ab_chained = itertools.chain(a, b)
list(ab_chained)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

groupby

`groupby` wykonuje tę samą operację, co `GROUP BY` z SQL'a. Przyjmuje listę elementów, a następnie łączy te same elementy w grupy.

Lista lub iterator przekazany jako argument musi być posortowany rosnąco.

```
data = [1, 3, 2, 1, 2, 2, 4, 3, 3, 3, 3, 1]

data = sorted(data)

for element, iter in itertools.groupby(data):
    print(f"{element} - {list(iter)}")
```

```
1 - [1, 1, 1]
2 - [2, 2, 2]
3 - [3, 3, 3, 3, 3]
4 - [4]
```

Podobnie jak w przypadku funkcji `sort`, możemy zdefiniować klucz, według którego elementy będą grupowane. `groupby` zwraca iterator par. Pierwszy element z tej pary to wspólny klucz, natomiast drugi to iterator zwracający wszystkie elementy z danej grupy.

```
animals = ['duck', 'eagle', 'rat', 'giraffe', 'bear',
           'bat', 'dolphin', 'shark', 'lion']

animals.sort(key=len)

for length, group in itertools.groupby(animals, len):
    print(length, '->', list(group))
```

```
3 -> ['rat', 'bat']
4 -> ['duck', 'bear', 'lion']
5 -> ['eagle', 'shark']
7 -> ['giraffe', 'dolphin']
```

takewhile & dropwhile

Funkcje `takewhile` i `dropwhile` działa podobnie do `filter`. Przyjmuje dwa argumenty: funkcję (predykat) zwracającą `True` lub `False` dla każdego elementu kolekcji oraz kolekcję.

`takewhile` przerywa zwracanie po natrafieniu na pierwszy element, dla którego predykat zwrócił `False`.

`dropwhile` pomija wszystkie początkowe elementy spełniające predykat.

```
from itertools import takewhile, dropwhile


list(takewhile(lambda x: x <= 200, fibonacci()))
```

```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144]
```

```
list(takewhile(lambda n: n < 1000, dropwhile(lambda n: n < 100, fibonacci())))
```

```
[144, 233, 377, 610, 987]
```

0.5 Dekoratory funkcji i klas

0.5.1 Domknięcie (*closure*)

Domknięcie, w metodach realizacji języków programowania, jest to obiekt **wiążący funkcję oraz środowisko**, w jakim ta funkcja ma działać. Środowisko przechowuje wszystkie obiekty wykorzystywane przez funkcję, niedostępne w globalnym zakresie widoczności. Realizacja domknięcia jest zdeterminowana przez język, jak również przez kompilator.

Domknięcia występują głównie w językach funkcyjnych, w których funkcje mogą zwracać inne funkcje, wykorzystujące zmienne utworzone lokalnie.

```
def bind_add(x):
    def add(y):
        # x jest "zamknięte" w definicji
        return y + x
    return add
```

```
add_5 = bind_add(5)
add_5(10)
```

```
15
```

```
add_665 = bind_add(665)
add_665(2)
```

```
667
```

0.5.2 Wprowadzenie do dekoratorów

Dekorator to wzorzec projektowy, pozwalający na dynamiczne dodanie nowej funkcjonalności, w trakcie działania programu.

W języku Python jest to metoda modyfikacji obiektu wywoływalnego (funkcji, metod klasy, klas) za pomocą domknięć.

Dekoratory są w Pythonie często spotykaną techniką programistyczną. Ich zalety to redukcja ilości kodu oraz możliwość kontrolowania funkcji (lub innych obiektów wywoływalnych), w szczególności ich danych wejściowych i zwracanych wartości.

0.5.3 Prosty dekorator

Poniżej przedstawiono implementację dekoratora `@shouter`. Funkcje udekorowane nim wyświetlają komunikat na początku i pod koniec ich wywołania.

```
def shouter(func):
    def wrapper():
        print("Before", func.__name__)
        result = func()
        print(result)
        print("After", func.__name__)
        return result
    return wrapper
```

Można tak zdefiniowanej funkcji użyć do "nadpisania" istniejącej już funkcji (tak naprawdę do zmiany tego, na co wskazuje zmienna):

```
def greetings():
    return "Hi"

hello = shouter(greetings)

hello()
```

```
Before greetings
Hi
After greetings
```

```
'Hi'
```

Począwszy od Pythona 2.4, możliwe i rekomendowane jest użycie specjalnej składni:

```
@shouter
def hello():
    return "Hello"
```

```
hello()
```

```
Before hello
Hello
After hello
```

```
'Hello'
```

Użycie `@shouter def hello()` jest równoważne `hello = shouter(hello)`.

0.5.4 Argumenty w dekoratorach

Problem

Przedstawiony dekorator działa tylko z funkcjami, które nie przyjmują żadnych argumentów. Co z funkcjami wymagającymi argumentów?

```
@shouter
def add(x, y):
    '''Docstring for add(x, y)'''
    return x + y
```

```
add(2, 7)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 add(2, 7)

TypeError: shouter.<locals>.wrapper() takes 0 positional arguments but 2 were given
```

Innym problemem jest to, że udekorowana funkcja utraciła swój docstring oraz swoją nazwę:

```
add.__doc__
```

```
add.__name__
```

```
'wrapper'
```

Rozwiązanie

Argumenty przekazywane do *wrapper* muszą zostać przekazane dalej, do właściwej funkcji *func*.

Z kolei problem z docstringiem i nazwą rozwiążemy dekorując funkcję *wrapper* przy pomocy dekoratora `@functools.wraps`, który zadba o skopiowanie docstringa i nazwy:

```
import functools

def shouter(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        print("Before", func.__name__)
        result = func(*args, **kwargs)
        print(result)
        print("After", func.__name__)
        return result
    return wrapper
```

```
@shouter
def add(x, y):
    '''Docstring for add(x, y)'''
    return x + y
```

```
add(5, 6)
```

```
Before add
11
After add
```

```
11
```

```
add.__doc__
```

```
'Docstring for add(x, y)'
```

```
add.__name__
```

```
'add'
```

0.5.5 Dekoratory parametryzowane

Dekoratory, które nie przyjmują żadnych argumentów, są często spotykane. Jednak czasami potrzebujemy przekazać do dekoratora argumenty.

Aby otrzymać parametryzowany dekorator, musimy go "owinąć" w jeszcze jedną funkcję (domknięcie):

```
def tag(tagname):
    def decorator(fun):
        @functools.wraps(fun)
        def wrapper(*args, **kwargs):
            tag_before = f"<{tagname}>"
            tag_after = f"</{tagname}>"
            fresult= fun(*args, **kwargs)
            return tag_before + fresult + tag_after
        return wrapper
    return decorator
```

```
@tag("b")
def output(data):
    return data
```

```
output("TEXT")
```

```
'<b>TEXT</b>'
```

Użycie `@tag("b")` jest odpowiednikiem:

```
output = tag("b")(output)
```

0.5.6 Wiele dekoratorów

Funkcję można owijać w wiele dekoratorów.

```
@shouter
@tag('b')
def my_func(text):
    return text
```

```
my_func("text")
```

```
Before my_func
<b>text</b>
After my_func
```

```
'<b>text</b>'
```

Należy pamiętać, że kolejność ma znaczenie. Składnia `@shouter @tag("b") def my_func()` jest równoważna `my_func shouter(tag("b")(my_function)`

0.5.7 Kiedy uruchamiane są dekoratory

Kluczowe znaczenie dla dekoratorów ma fakt, że są one uruchamiane zaraz po tym jak zdefiniowana została dekorowana funkcja. Najczęściej jest to moment *importu* pakietu.

```
registry = []

def register(func):
    print(f'running register({func})')
    registry.append(func)
    return func

@register
def f1():
    print('running f1()')
```

(continues on next page)

(continued from previous page)

```
@register
def f2():
    print('running f2()')

def f3():
    print('running f3()')

def main():
    print('running main()')
    print('registry ->', registry)
    f1()
    f2()
    f3()

if __name__ == '__main__':
    main()
```

```
running register(<function f1 at 0x7f4677b749d0>)
running register(<function f2 at 0x7f4677b74790>)
running main()
registry -> [<function f1 at 0x7f4677b749d0>, <function f2 at 0x7f4677b74790>]
running f1()
running f2()
running f3()
```

0.5.8 Dekoratory klas

Od Pythona 2.6 można dekorować klasy. W środku dekoratora można zmodyfikować klasę, na przykład zmienić jej metody. Dekoratory klas mają działanie zbliżone do metaklas.

```
id = 0

def add_id(decorated_class):
    original_init = decorated_class.__init__

    def __init__(self, *args, **kwargs):
        print("add_id init")
        global id
        id += 1
        self.id = id
        original_init(self, *args, **kwargs)

    decorated_class.__init__ = __init__ # replacing __init__ in decorated class
    return decorated_class

@add_id
class Foo(object):
    def __init__(self):
        print("Foo class init")
```

```
foo = Foo()
foo.id
```

```
add_id init
Foo class init
```

```
1
```

```
bar = Foo()
bar.id
```

```
add_id init
Foo class init
```

```
2
```

0.5.9 Klasy jako dekoratory

Bardzo ciekawym zastosowaniem jest użycie klasy jako dekoratora. Wystarczy zdefiniować w klasie metodę specjalną `__call__`. Instancja klasy (uzyskana za pomocą operatora `()`) staje się wtedy obiektem, który można wywołać.

Jest to alternatywa dla definiowania nieparametryzowanego dekoratora przy pomocy dwóch zagnieżdżonych funkcji. Kod jest nieco prostszy do zrozumienia:

```
import functools

class Shouter:
    def __init__(self, function):
        print("Inside decorator's __init__()")
        self.function = function
        functools.update_wrapper(self, function)

    def __call__(self, *args, **kwargs):
        print("Before call")
        result = self.function(*args, **kwargs)
        print("After call")
        return result
```

```
@Shouter
def anwser(input):
    print("Inside function()")
    return input * 42
```

```
Inside decorator's __init__()
```

```
anwser('*')
```

```
Before call
Inside function()
After call
```

```
'****************************************'
```

0.6 Menadżery kontekstu

0.6.1 Wprowadzenie

Często spotykany w zarządzaniu zasobami jest następujący idiom:

```
do_setup()
try:
    do_task()
except SomeError:
    handle_the_error()
finally:
    do_cleanup()
```

0.6.2 Wyrażenie `with`

Aby uprościć i uodpornić się na błędy programisty, od Pythona 2.5 wzwyż dostępne jest wyrażenie *with*.

Menedżer kontekstu (*context manager*) jest odpowiedzialny za zarządzanie zasobami wewnątrz bloku kodu.

Najczęściej tworzy te zasoby na początku bloku, a zwalnia na końcu.

Na przykład, menadżer kontekstu dla plików upewnia się, że pliki zostały prawidłowo zamknięte po zakończeniu bloku, nawet jeśli zostanie zgłoszony wyjątek.

```
with open('myfile.txt', 'wt') as f:
    f.write('foo bar')
```

Odpowiednikiem bloku:

```
with VAR = EXPR:
    BLOCK
```

jest zapis:

```
VAR = EXPR
VAR.__enter__()
try:
    BLOCK
finally:
    VAR.__exit__()
```

0.6.3 Protokół menadżera kontekstu

Menedżer kontekstu jest klasą posiadającą dwie metody specjalne:

- `__enter__` - metoda wywoływana na samym początku bloku wewnątrz *with*.
- `__exit__` - metoda jest odpowiednikiem `finally:`, wywoływana po zakończeniu bloku `with`.

Poniżej przedstawiono przykładowy, prosty menadżer kontekstu:

```
class Context:
    def __init__(self):
        print('__init__()')

    def __enter__(self):
        print('__enter__()')
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print('__exit__()')
```

```
with Context():
    print("Doing work inside context")
```

```
__init__()
__enter__()
Doing work inside context
__exit__()
```

Metoda `__enter__`

Wartością zwracaną przez menadżera kontekstu w funkcji `__enter__` może być obiekt, który zostanie przypisany do zmiennej występującej po *as*:

```
import sys

def blackhole(*args, **kwargs):
    pass

class SuppressOutput:
    def __enter__(self):
        print('SuppressOutput.__enter__()')
        self.write, sys.stdout.write = sys.stdout.write, blackhole
        return self.write

    def __exit__(self, exc_type, exc_val, exc_tb):
        sys.stdout.write = self.write
        print('SuppressOutput.__exit__()')
```

```
with SuppressOutput() as stdout_write:
    print('That won\'t be printed')
    stdout_write('But this one will be printed\n')
```

```
SuppressOutput.__enter__()
But this one will be printed
SuppressOutput.__exit__()
```

Metoda `__exit__`

Do metody `__exit__` trafia informacja o wyjątkach, jakie pojawiły się bloku `with`.

- Jeśli metoda `__exit__` zwraca `true`, to wyjątek został obsłużony przez menadżera kontekstu.
- Jeśli zwrócona zostanie wartość `false`, to wyjątek będzie propagowany dalej.

```
class Context:
    def __enter__(self):
        pass

    def __exit__(self, excpt_type, excpt_val, excpt_tb):
        print("Exception type:", excpt_type)
        print("Exception value:", excpt_val)
        print("Traceback object:", excpt_tb)
        return True  # or False
```

```
with Context():
    x = 2
```

```
Exception type: None
Exception value: None
Traceback object: None
```

```
with Context():
    x = 2 / 0
```

```
Exception type: <class 'ZeroDivisionError'>
Exception value: division by zero
Traceback object: <traceback object at 0x7f50a8398900>
```

0.6.4 contextlib.contextmanager

W prostych przypadkach zamiast tworzyć klasę, możemy skorzystać z gotowego dekoratora zawartego w module *contextlib*, który konwertuje składnię funkcji do postaci menadżera kontekstu:

```
from contextlib import contextmanager

@contextmanager
def make_context():
    try:
        prepare_resource()
        yield context_object
    except RuntimeError as err:
        handle_exception_here()
```

(continues on next page)

(continued from previous page)

```
    finally:
        do_clean_up()
```

Przykładowy prosty menadżer kontekstu napisany z użyciem `contextmanager`:

```
from contextlib import contextmanager

@contextmanager
def Shouter():
    print('Going in')
    yield
    print('Coming out')

with Shouter():
    print('Inside')
```

```
Going in
Inside
Coming out
```

Jeżeli chcemy obsłużyć rzucone przez funkcję wyjątki, możemy to zrobić w następujący sposób:

```
@contextmanager
def Shouter():
    print('Going in')
    try:
        yield
    except Exception:
        print('Error!')
    else:
        print('No error')
```

```
with Shouter():
    pass
```

```
Going in
No error
```

```
with Shouter():
    print(1/0)
```

```
Going in
Error!
```

0.7 Structural Pattern Matching

Python 3.10 wprowadza nową instrukcję `match` inspirowaną jezykami funkcyjnymi (Scala, Erlang).

Instrukcja `match` porównuje wartość (**subject**) do kilku różnych wzorców (**patterns**) wymienionych po etykietach `case`, aż znalezione zostanie dopasowanie. Każdy wzorzec (**pattern**) opisuje typ i strukturę akceptowanych wartości. Wzorzec może zawierać też zmienne, do których są wiązane pasujące wartości (**binding**).

Składnia:

```
match <subject_expression>:
    case <pattern_1> [<if guard>]:
        <block to execute if pattern_1 matches>
    case <pattern_n> [<if guard>]:
        <block to execute if pattern_n matches>
```

0.7.1 Wzorzec - Pattern

Wzorzec (**pattern**) jest nowym elementem składni języka, który wygląda jak część wyrażenia służącego do konstrukcji obiektu, np:

- `[first, second, *rest]`
- `Point2D(x, 0)`
- `{id: 665, name: "John}`
- `665`

Podobieństwo ze składnią służącą do konstrukcji jest zamierzone, ale dla wzorca oznacza proces odwrotny, nazywany *dekonstrukcją*. Dekonstrukcja umożliwia ekstrakcję elementów obiektu na podstawie wzorca.

Proces dopasowania wzorca

Instrukcja `match` stara się dopasować obiekt (*subject*) do każdego wzorca podanego po etykiecie `case`. Dla pierwszego pasującego wzorca:

- wiązane są wartości do zmiennych występujące w wzorcu
- wykonywany jest odpowiedający etykiecie blok instrukcji

```
from collections import namedtuple

Point2D = namedtuple('Point2D', 'x y')
Point3D = namedtuple('Point3D', 'x y z')

def make_point_3d(pt):
    match pt:
        case (x, y):
            return Point3D(x, y, 0)
        case (x, y, z):
            return Point3D(x, y, z)
        case Point2D(x, y):
            return Point3D(x, y, 0)
        case Point3D(_, _, _):
            return pt
```

(continues on next page)

(continued from previous page)

```
        case _:
            raise TypeError('a type cannot be converted to Point3D')
```

```
make_point_3d((1, 2, 3))
```

```
Point3D(x=1, y=2, z=3)
```

```
make_point_3d(Point2D(99, 45))
```

```
Point3D(x=99, y=45, z=0)
```

Rodzaje wzorców

Literały

```
number = 42

match number:
    case 0:
        print("Nothing")
    case 1:
        print("Just one")
    case 2:
        print("A couple")
    case -1:
        print("One less than nothing")
    case 1-1j:
        print("Good luck with that...")
```

Wzorce przechwyceń

Wprowadzenie nazwy zmiennej we wzorcu pozwala przypisać tej zmiennej odpowiednią wartość (w przypadku dopasowania):

```
greeting = "John"

match greeting:
    case "":
        print("Hello stranger!")
    case name:
        print(f"Hello {name}")
```

```
Hello John
```

W danym wzorcu określona nazwa może wystąpić tylko raz!

```
data = [1, 4]

match data:
    case [x, x]:
        print(x)
```

```
  Cell In[6], line 4
    case [x, x]:
             ^
SyntaxError: multiple assignments to name 'x' in pattern
```

Symbol zastępczy

Symbol _ jest specjalnym znakiem oznaczającym wzorzec, który zawsze pasuje ale nie powoduje wiązania z wartością:

```
data = [42, 665]

match data:
    case [_]:
        print("A list with just one element")
    case [_, _]:
        print("A list with two elements")
```

```
A list with two elements
```

Stałe i wyliczenia

```
from enum import Enum

class Guitar(str, Enum):
    STRATOCASTER = "Stratocaster"
    TELECASTER = "Telecaster"
    LES_PAUL = "Les-Paul"

my_guitar = Guitar.STRATOCASTER

match my_guitar:
    case Guitar.LES_PAUL: # compares my_guitar == Guitar.LES_PAUL
        print("I have guitar with humbuckers")
    case fender:
        print(f"I have a {fender} guitar")
```

```
I have a Stratocaster guitar
```

Wzorce sekwencji

Wzorce sekwencji mają tą samą semantykę co rozpakowanie przypisania (działają zarówno dla krotek jak i list).

```
collection = [1, 2, [3, 4, 5]]

match collection:
    case 1, x, [y, *others]:
        print(f"Got 1 , {x} , [{y} , {others}]")
```

```
Got 1 , 2 , [3 , [4, 5]]
```

Symbol zastępczy _ może być użyty w połączeniu z `*` w celu określenia zmiennej długości:

- `[*_]` - pasuje do sekwencji o dowolnej długości
- `(_, _, *_)` - pasuje do sekwencji o długości równej dwa lub większej
- `['a', *_, 'z']` - pasuje do sekwencji dowolnej długości zaczynającej się od `'a'` i kończącej się na `'z'`

Wzorce słownikowe

Dopasowywana wartość (**subject**) musi być instancją typu `collections.abc.Mapping`. Dodatkowe klucze są pomijane nawet gdy we wzorcu nie został użyty symbol `**rest`.

```
config = { 'url': "http://localhost", 'port': 8080, 'timeout': 60 }

match config:
    case {'url': url, 'port': port}:
        print(f"Connecting to {url}:{port}")
    case {}:
        print("Connection not configured...")
```

```
Connecting to http://localhost:8080
```

Wzorce klas

Umożliwiają dopasowanie na podstawie typu (odpowiednik `isinstance()`) i destrukturyzację obiektów. Dostępne są dwie opcje dopasowań:

- z wykorzystaniem pozycji np. `Point(1, 2)` - dla danej klasy wymagany jest atrybut `__match_args__`
- z wykorzystaniem nazw np. `Point(x=1, y=2)`

```
from dataclasses import dataclass
from typing import Tuple

@dataclass
class Shape:
    coord: Tuple[int, int]

@dataclass
class Circle(Shape):
```

(continues on next page)

(continued from previous page)

```
    radius: int

class Rectangle(Shape):
    __match_args__ = ('coord', 'width', 'height') # required for positional pattern␣
 ↪matching

    def __init__(self, coord, width, height):
        super().__init__(coord)
        self.height = height
        self.width  = width

shp = Rectangle((0, 0), 90, 665)

match shp:
    case Circle(coord, r):
        print(f"Drawing circle with radius={r} at {coord}")
    case Rectangle(_, w, h):
        print(f"Drawing rectangle with width={w} and height={h}")
    case Shape(_):
        print(f"Drawing a shape!")
```

```
Drawing rectangle with width=90 and height=665
```

Łączenie wielu wzorców (wzorce z OR)

Alternatywne wzorce mogą być połączone w jeden za pomocą `|`. Takie połączenie oznacza, że cały wzorzec zostaje dopasowany, jeśli przynajmniej jedna z alternatyw pasuje.

Alternatywne wzorce są dopasowywane od lewej do prawej i mają właściwość *short-circuit*.

```
something = "something"

match something:
    case 0 | 1 | 2:
        print("small number")
    case [] | [_]:
        print("a short sequence")
    case str() | bytes():
        print("something string-like")
    case _:
        print("something else")
```

```
something string-like
```

Wzorce z warunkiem

Każdy z wzorców umieszczonych na początku instrukcji `match` może zawierać warunek (**guard**) w postaci wyrażenia `if`.

```
MAX_SIZE = 100

coord = (88, 88)

match coord:
    case x, y if x > MAX_SIZE and y > MAX_SIZE:
        print("Both coordinates out of bounds")
    case x, y if x > MAX_SIZE or y > MAX_SIZE:
        print("one coordinate out of bounds")
    case x, y if x == y:
        print("Pixel with x coordinate the same as y")
    case _:
        print(f"Pixel at {coord}")
```

```
Pixel with x coordinate the same as y
```

0.8 Metaklasy

Metaklasą nazywamy obiekt (najczęściej klasę) generujący inne klasy.

"Metaclasses are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't"

– Python Guru Tim Peters

0.8.1 Klasy jako obiekty

Podobnie jak w przypadku funkcji, klasy są obiektami. Służą do tworzenia nowych obiektów (instancji).

```
class MyClass:
    pass
```

Nowy obiekt jest tworzony przy pomocy operatora `()`. Jego typ to nazwa klasy.

```
mc = MyClass()
type(mc)
```

```
__main__.MyClass
```

Jakiego typu jest obiekt klasy?

```
type(MyClass)
```

```
type
```

0.8.2 Dynamiczne tworzenie klas

Skoro są obiekty klas są obiektami typu `type`, to możemy je też dynamicznie tworzyć.

Funkcja `type` działa też jak fabryka klas, która przyjmuje trzy argumenty:

- nazwa klasy
- krotka z klasami bazowymi
- słownik zawierający nazwy atrybutów i ich wartości

W rezultacie klasę utworzoną w klasyczny sposób:

```
class Shape:
    def draw(self):
        pass

class Rectangle(Shape):
    _id = 'RECTANGLE'

    def __init__(self, width, height):
        self.width = width
        self.height = height

    def draw(self):
        print(f'Drawing {Rectangle._id}({self.width}, {self.height})')
```

możemy utworzyć też dynamicznie:

```
def rect_init(self, width, height):
        self.width = width
        self.height = height

RectangleT = type('RectangleT', (Shape, ), {
        '_id': 'RECTANGLE_T',
        '__init__': rect_init,
        'draw' : lambda self: print(f'Drawing {RectangleT._id}({self.width}, {self.
 ↪height})')
})
```

```
RectangleT.__name__
```

```
'RectangleT'
```

```
type(RectangleT)
```

```
type
```

```
rect = RectangleT(10, 20)
```

```
rect.draw()
```

```
Drawing RECTANGLE_T(10, 20)
```

0.8.3 Metaklasy

Typ `type` jest więc wbudowaną w Pythona metaklasą. Jednakże istnieje możliwość stworzenia własnych metaklas.

Pod Pythonem 3, składnia jest następująca:

```
class MyClass(object, metaclass=class_creator):
    ...
```

gdzie `class_creator` to specjalny obiekt, którego należy użyć zamiast `type` do utworzenia obiektu klasy.

Funkcja jako metaklasa

W szczególności, metaklasą może być funkcja. Poniżej przedstawiono metaklasę, która konwertuje nazwy wszystkich atrybutów tak, aby używały wielkich liter.

```
def upper_attr(cls, parents, attrs):
    _attrs = ((name.upper(), value)
              for name, value in attrs.items())
    attrs_upper = dict(_attrs)
    return type(cls, parents, attrs_upper)


class Foo(metaclass=upper_attr):
    bar = 'foo'
```

```
foo = Foo()
foo.BAR
```

```
'foo'
```

Klasa metaklasy

Zazwyczaj jednak metaklasa jest klasą dziedziczącą po `type`.

```
class UpperAttr(type):
    def __new__(cls, name, parents, attrs):
        _attrs = ((name.upper(), value)
                  for name, value in attrs.items())
        attrs_upper = dict(_attrs)
        return type(name, parents, attrs_upper)

class Boo(object, metaclass=UpperAttr):
    bar = 'boo'
```

```
foo = Foo()
foo.BAR
```

```
'foo'
```

Metody specjalne metaklasy

```
from typing import Any, Dict, Mapping, Tuple, Type

class Metaclass(type):
    @classmethod
    def __prepare__(mcs, name: str, bases: Tuple[Type, ...], **kwargs: Any) ->␣
↪Mapping[str, Any]:
        print(f'Metaclass.__prepare__(mcs={mcs},\n'
              f'\tname={name},\n'
              f'\tbases={bases!r},\n'
              f'\tkwargs={kwargs!r})')
        return super().__prepare__(mcs, name, bases, **kwargs)

    def __new__(mcs, name: str, bases: Tuple[Type, ...], namespace: Dict[str, Any],␣
↪**kwargs: Any):
        print(f'Metaclass.__new__(mcs={mcs},\n'
              f'\tname={name},\n'
              f'\tbases={bases!r},\n'
              f'\tnamespace={namespace!r},\n'
              f'\tkwargs={kwargs!r})')
        return super().__new__(mcs, name, bases, namespace)

    def __init__(cls, name: str, bases: Tuple[Type, ...], namespace: Dict[str, Any],␣
↪**kwargs: Any) -> None:
        print(f'{cls}.__init__(name={name},\n\tbases={bases!r},\n\tnamespace=
↪{namespace!r},\n\tkwargs={kwargs!r})')
        super().__init__(name, bases, namespace, **kwargs)

    def __call__(cls, *args: Any, **kwargs: Any) -> Any:
        print(f'{cls}.__call__(args={args!r}, kwargs={kwargs!r})')
        return super().__call__(*args, **kwargs)
```

```
class User:
    pass


class SuperUser(User, metaclass=Metaclass, value = 42):
    id: int = 665

    def __init__(self, id: int, name: str):
        print(f'{self}.__init__(id={id}, name={name})')
        self.id = id
        self.name = name

    def set_password(self, new_password: str):
        pass
```

```
Metaclass.__prepare__(mcs=<class '__main__.Metaclass'>,
        name=SuperUser,
        bases=(<class '__main__.User'>,),
```

(continues on next page)

(continued from previous page)

```
        kwargs={'value': 42})
Metaclass.__new__(mcs=<class '__main__.Metaclass'>,
        name=SuperUser,
        bases=(<class '__main__.User'>,),
        namespace={'__module__': '__main__', '__qualname__': 'SuperUser', '__
 ↪annotations__': {'id': <class 'int'>}, 'id': 665, '__init__': <function␣
 ↪SuperUser.__init__ at 0x7f03eb55dd80>, 'set_password': <function SuperUser.set_
 ↪password at 0x7f03eb55de10>},
        kwargs={'value': 42})
<class '__main__.SuperUser'>.__init__(name=SuperUser,
        bases=(<class '__main__.User'>,),
        namespace={'__module__': '__main__', '__qualname__': 'SuperUser', '__
 ↪annotations__': {'id': <class 'int'>}, 'id': 665, '__init__': <function␣
 ↪SuperUser.__init__ at 0x7f03eb55dd80>, 'set_password': <function SuperUser.set_
 ↪password at 0x7f03eb55de10>},
        kwargs={'value': 42})
```

```
user = SuperUser(667, "admin")
```

```
<class '__main__.SuperUser'>.__call__(args=(667, 'admin'), kwargs={})
<__main__.SuperUser object at 0x7f03e8388520>.__init__(id=667, name=admin)
```

Metoda specjalna `__prepare__`

Zadaniem tej metody jest zwrócenie słownika, który zostanie wykorzystany do zainicjowania obiektu `__dict__` w tworzonym obiekcie typu (klasy). Domyślna implementacja zwraca pusty słownik (typu `dict`), ale można to zmienić (np. zwrócić wstępnie wypełnioną instancję słownika).

Metoda specjalna `__new__`

Jest odpowiedzialna za utworzenie obiektu klasy. Dostaje jako argument wywołania obiekt **metaklasy**, nazwę tworzonego typu (klasy), krotkę klas bazowych i wypełniony słownik z atrybutami. Możliwa jest modyfikacja tych parametrów przed przekazaniem ich (najczęściej) w wywołaniu `__new__()` z klasy bazowej, czyli `type.__new__()`

Metoda specjalna `__init__`

Dostaje jako argument wywołania obiekt utworzonej już klasy, z wypełnionym słownikiem atrybutów.

Metoda specjalna `__call__`

Metoda wywoływana, gdy tworzona jest instancja docelowej klasy (utworzonej za pomocą danej metaklasy). Domyślna implementacja z `type` wywołuje operacjel:

- `__new__(cls, *args, **kwargs)` - utworzenie instancji klasy
- `__init__(self, *args, **kwargs)` - inicjalizacja instancji klasy

Ta implementacja może zostać zmieniona w celu lepszej kontroli sposobu tworzenia instancji klasy.

0.8.4 Zastosowanie metaklas

W praktyce, metaklasy są stosowane tam, gdzie API klasy musi być tworzone dynamicznie (np. ORM w Django) oraz do implementacji niektórych wzorców projektowych.

Singleton i metaklasa

```
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class AsSingleton(metaclass=Singleton):
    pass


class Logger(AsSingleton):
    def __init__(self):
        print(f'Executing Logger.__init__({self})')

    def log(self, msg: str) -> None:
        print(f">>{msg}")
```

```
logger1 = Logger()
logger2 = Logger()

logger1 is logger2
```

```
Executing Logger.__init__(<__main__.Logger object at 0x7f03e838b3a0>)
```

```
True
```

Modyfikacja nazw atrybutów w klasie

```
from typing import Any, Mapping
import inflection

class CaseInterpolationDict(dict):
    def __setitem__(self, key, value):
        super().__setitem__(key, value)
        super().__setitem__(inflection.underscore(key), value)

class CaseInterpolatedMeta(type):
    @classmethod
    def __prepare__(mcs, __name: str, __bases: Tuple[type, ...], **kwds: Any) ->␣
 ↪Mapping[str, object]:
        return CaseInterpolationDict()
```

(continues on next page)

(continued from previous page)

```
class MyUser(metaclass=CaseInterpolatedMeta):
    pass

class User(MyUser):
    def __init__(self, firstName: str, lastName: str):
        self.firstName = firstName
        self.lastName = lastName

    def getDisplayName(self):
        return f"{self.firstName} {self.lastName}"

    def greetUser(self):
        return f"Hello {self.getDisplayName()}!"
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[19], line 2
      1 from typing import Any, Mapping
----> 2 import inflection
      4 class CaseInterpolationDict(dict):
      5     def __setitem__(self, key, value):

ModuleNotFoundError: No module named 'inflection'
```

```
User.__dict__
```

```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.User.__init__(self, firstName: str,␣
 ↪lastName: str)>,
              'getDisplayName': <function __main__.User.getDisplayName(self)>,
              'get_display_name': <function __main__.User.getDisplayName(self)>,
              'greetUser': <function __main__.User.greetUser(self)>,
              'greet_user': <function __main__.User.greetUser(self)>,
              '__doc__': None})
```

```
user = User("John", "Doe")
user.getDisplayName()
user.get_display_name()
```

```
'John Doe'
```

0.9 Moduły i pakiety

0.9.1 Moduły

Wprowadzenie

Modułem jest plik źródłowy z rozszerzeniem *.py* zawierający kod Pythona. Moduł może zawierać definicje funkcji, klas, a także niezależny od nich kod. Moduł może zawierać dokumentację informującą o sposobie jego działania i zastosowaniach.

Wewnątrz modułu jego nazwa (nazwa pliku bez rozszerzenia .py) jest dostępna pod globalną zmienną *__name__*, pod warunkiem że moduł jest importowany. Jeśli jednak moduł nie jest importowany, a uruchomiony z konsoli, wówczas `__name__ == "__main__"`. Dlatego często występującym idiomem jest sprawdzenie, czy moduł został zaimportowany, czy też uruchomiony:

```
if __name__ == "__main__":
    ...  # kod, który ma być wykonany tylko, gdy moduł jest uruchamiany, a nie␣
 ↪importowany
```

Moduł po zaimportowaniu jest obiektem.

W dalszej części rozdziału będziemy posługiwać się modułem umieszczonym w pliku *fibo.py*:

```
# Fibonacci numbers module
def fib(n): # wypisuje na standardowe wyjście ciąg Fibonacciego do n
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b

def fib2(n): # zwraca ciąg Fibonacciego do n, jako listę
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Import modułów

Przed użyciem modułu należy go zaimportować.

Istnieją trzy sposoby importu:

- Najprostszy, służący do zaimportowania całego modułu:

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.__name__
'fibo'
```

- Możliwe jest zaimportowanie jedynie określonych funkcji lub klas z danego modułu. Sam moduł `fibo` nie jest wtedy dostępny. Dostępna jest tylko zaimportowana funkcja `fib`.

```
>>> from fibo import fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> fib2(500)  # funkcja fib2 nie została zaimportowana
NameError                                 Traceback (most recent call last)
...
NameError: name 'fib2' is not defined
>>> fibo  # moduł fibo nie jest dostępny
NameError                                 Traceback (most recent call last)
...
NameError: name 'fibo' is not defined
```

- Możliwe jest zaimportowanie od razu wszystkich obiektów zdefiniowanych w danym module, bez ich wyliczania.

  Nie jest to jednak rekomendowane, ponieważ nie mamy kontroli nad tym, jakie dokładnie obiekty zostaną zaimportowane. W efekcie, możliwa jest sytuacja, w której niechcący przesłaniamy zaimportowaną lub zdefiniowaną wcześniej funkcję lub klasę.

```
>>> def fib(n):
...     print('Ta metoda zostanie przez przypadek przesłonięta.')
...
>>> from fibo import *
>>> fib(500)  # wykonywana jest funkcja fib z modułu fibo, zamiast lokalnej funkcji
1 1 2 3 5 8 13 21 34 55 89 144 233 377
>>> fib2(500)
[1 1 2 3 5 8 13 21 34 55 89 144 233 377]
>>> fibo  # sam moduł nie jest dostępny
NameError                                 Traceback (most recent call last)
...
NameError: name 'fibo' is not defined
```

W dwóch pierwszych przypadkach możliwe jest także zaimportowanie modułu lub zdefiniowanych w nim atrybutów pod inną nazwą:

```
>>> import fibo as fibo2
>>> from fibo import fib, fib2 as fib22
```

Lokalizacja modułów - PYTHONPATH

Kiedy moduł o nazwie *mname* jest importowany, interpreter przeszukuje:

- bieżący katalog,
- listę katalogów określoną w zmiennej systemowej *PYTHONPATH*
- oraz domyślną ścieżkę instalacji (np. `/usr/local/lib/python`).

sys.path jest lista katalogów przeglądanych przez interpreter w trakcie wykonywania instrukcji import.

```
>>> import sys
>>> sys.path
 [''',
 'C:\\Python27\\lib\\site-packages\\rested-1.1.0-py2.7.egg',
```

(continues on next page)

(continued from previous page)

```
 'C:\\Program Files (x86)\\DreamPie\\share\\dreampie',
 'C:\\Python27\\python27.zip',
 'C:\\Python27\\DLLs',
...]
```

W szczególności, lista ta może zostać zmodyfikowana, jeżeli chcemy ładować moduły z określonego katalogu.

Przestrzeń nazw modułu

Pliki modułów podczas operacji importowania stają się obiektami. Obiekty te zawierają zmienne istniejące w module:

```
>>> import fibo
>>> print([x for x in fibo.__dict__ if not x.startswith("__")])
['fib', 'fib2']
```

Podczas importu wykonywane są wszystkie instrukcje zawarte w module. Przypisania na najwyższym poziomie modułu tworzą atrybuty obiektu modułu. Dostęp do przestrzeni nazw modułu odbywa się za pomocą *__dict__* lub *dir(M)*

Modyfikacja modułów

Operacja importowania modułu odbywa się tylko raz, podczas pierwszego użycia instrukcji *import* lub *from*. Ponowne użycie instrukcji *import* lub *from* do zaimportowania raz załadowanego modułu nie spowoduje ponownego wykonania kodu tego modułu, nawet jeżeli instrukcje importu znajdują się w innym pliku. Aby jeszcze raz załadować (wykonać) moduł, należy użyć funkcji `importlib.reload`. W Pythonie 2 jest to wbudowana funkcja, tzn. nie trzeba jej importować.

```
>>> import fibo
>>> fibo.fib(1000)

>>> from importlib import reload
>>> imp.reload(fibo)
```

Funkcja `reload` jest przydatna przy jednoczesnej pracy w pliku i interaktywnej konsoli. Po zmodyfikowaniu pliku, możemy załadować jego nowszą wersję wywołując `reload`.

Ukrywanie danych w modułach

Przy wykonywaniu `from module import *` importowane są tylko atrybuty, które nie zaczynają się od podkreślnika. Tym samym funkcje i klasy zaczynające się od podkreślnika są niejako "chronione" i nie udostępniane innym modułom, chyba że zostaną jawnie zaimportowane (`from module import _funkcja`).

Alternatywnie, w module można zdefiniować specjalny atrybut `__all__`, który określa listę nazw wszystkich obiektów, które są "publiczne", tzn. importowane przy wykonaniu instrukcji `from module import *`:

```
__all__ = ['fib', 'fib2']
```

0.9.2 Pakiety

Wprowadzenie

W celu zorganizowania plików modułów w logiczną całość możemy zorganizować je w strukturę pakietów modułów. Jest ona oparta na hierarchii katalogów (folderów) systemu operacyjnego.

```
>>> import mymodules.fibo
```

Kropka oznacza, że w podkatalogu *mymodules* znajdziemy moduł *fibo*. Katalog *mymodules* musi być umieszczony w jednym z katalogów znajdujących się na ścieżce wyszukiwania modułów (np. w zmiennej środowiskowej *PYTHONPATH*).

Aby katalogi były przeszukiwane podczas importu, muszą zawierać plik o nazwie *__init__.py*.

```
mycode\
    mymodules\
        __init__.py
        fibo.py
        fobo.py
```

W powyższym przykładzie, `mycode` **nie** jest pakietem.

Plik *__init__.py* może zawierać instrukcje, które zostaną automatycznie wykonane podczas importu pakietu.

Importowanie pakietów

Aby wykonać funkcję *foo* z modułu *fibo*, musimy ją zaimportować:

```
>>> import mymodules.fibo
>>> mymodules.fibo.foo(100)
```

lub:

```
>>> import mymodules.fibo as fi
>>> fi.foo(100)
```

lub:

```
>>> from mymodules.fibo import foo
>>> foo(100)
```

Wewnątrz pliku `fobo.py` możemy zaimportować moduł `fibo` używając względnej ścieżki:

```
>>> from . import fibo
>>> fibo.foo(100)
```

lub:

```
>>> from .fibo import foo
>>> foo(100)
```

0.10 Testy jednostkowe

Testowanie programu pozwala sprawdzić, czy program zachowuje się w pożądany sposób. Manualne testowanie programu jest czasochłonne. Dlatego dobrą praktyką jest pisanie automatycznych testów, które mogą zostać szybko wykonane przez komputer. Testuje się nie tylko całe programy, ale też pojedyncze funkcje czy klasy.

Standardowa biblioteka Pythona posiada moduły `unittest` i `doctest` ułatwiające pisanie takich testów.

Alternatywą dla standardowych bibliotek jest `pytest`. Jest to jedna z najczęściej używanych bibliotek to testów jednostkowych.

0.10.1 Unittest

Wprowadzenie

`unittest` jest jedną z najpopularniejszych bibliotek stosowanych do pisania testów jednostkowych. Wynika to jej następujących cech:

- jest częścią standardowej biblioteki Pythona, co oznacza, że jest dostępna wszędzie, gdzie jest zainstalowany Python.
- jest inspirowana biblioteką `JUnit` z Javy. Osoby, które pisały wcześniej testy w `JUnit` bardzo szybko nauczą się pisać testy w Pythonie. Ponadto, wykorzystano sprawdzony interfejs. Z drugiej strony, wzorowanie się na bibliotece Javy powoduje, że testy są dosyć rozwlekłe i "rozgadane".
- potrafi automatycznie znaleźć wszystkie testy i wykonać je.

Przykład

Testy grupuje się w klasach dziedziczących po `unittest.TestCase`. Każda metoda zaczynająca się od `test_` to jeden test. Wywołanie `unittest.main()` powoduje uruchomienie wszystkich testów (nie jest potrzebne ręczne wymienianie nazw wszystkich testów).

Powszechnie przyjętą konwencją jest separacja testów i testowanego kodu, poprzez umieszczenie ich w osobnych plikach. Dodatkowo, testy umieszcza się w osobnym katalogu, a nie w tym samym katalogu co testowany moduł. Dzięki temu możliwa jest instalacja pakietu bez instalowania bibliotek wykorzystywanych tylko przez testy.

```
# converters.py
import re

def url_converter(url):
    if not url:
        raise ValueError("Empty url")

    pattern = r'(http://[\w-]+(\.[\w-]+)*((/[\w-]*)?))'
    regexp = re.compile(pattern)
    return regexp.sub('<a href="\\1">\\1</a>', url)

# test_converters.py
import unittest

from converters import url_converter

class UrlConverterTests(unittest.TestCase):
    def test_converts_url_to_ahref(self):
```

(continues on next page)

(continued from previous page)

```
        url = "http://www.python.org"
        expected_ahref = '<a href="http://www.python.org">http://www.python.org</a>'
        result = url_converter(url)
        self.assertEqual(result, expected_ahref)

if __name__ == "__main__":
    unittest.main()
```

Ostatnie dwie linie powyższego kodu gwarantują uruchomienie testów, ale tylko gdy plik z kodem zostanie bezpośrednio uruchomiony, a nie zaimportowany.

```
python test_converters.py
```

Uruchamianie testów

Możemy wykonać testy wpisując w konsoli.

```
python -m unittest test_my_module.TestAdd
```

Ogromną zaletą biblioteki `unittest` jest możliwość uruchomienia wszystkich testów bez określania, gdzie się one znajdują. W takiej sytuacji biblioteka `unittest` poszukuje testów we wszystkich plikach znajdujących się w aktualnym katalogu, podkatalogach, podkatalogach podkatalogów itd.

```
python -m unittest
```

Przydatnym przełącznikiem jest `--failfast` (lub `-f`), który zatrzymuje wykonywanie testów po pierwszym teście, który nie przeszedł. Ułatwia to skupienie się na naprawieniu testu, ponieważ na wyjściu pojawiają się informacje dotyczące tylko jednego nieprzechodzącego testu.

```
python -m unittest --failfast
```

Niektóre środowiska programistyczne, takie jak PyCharm, posiadają wsparcie dla uruchamiania testów.

`setUp` i `tearDown`

Jeżeli na początku lub na końcu każdego testu wykonujemy operacje, które powtarzają się w innych testach, wtedy można umieścić je w metodach `setUp` i `tearDown`. `setUp` jest metodą wykonywaną na początku każdego testu, natomiast `tearDown` – po wykonaniu testu, niezależnie od tego, czy test przeszedł, czy nie.

Jest to świetne miejsce na:

- uzyskanie zasobów, które są potrzebne w każdym teście (np. połączenie z bazą danych),
- skonfigurowanie środowiska w ten sam sposób dla każdego testu (np. w `setUp` – utworzenie przykładowych tabel i rekordów w bazie danych, a w `tearDown` – "posprzątanie" po teście, tzn. wyczyszczenie testowej bazy danych).

```
class TestAdd(unittest.TestCase):
    def setUp(self):
        print("setUp")

    def tearDown(self):
        print("tearDown")
```

(continues on next page)

(continued from previous page)

```
    def test_one(self):
        print("test one")

    def test_two(self):
        print("test two")
```

```
setUp
test one
tearDown
setUp
test two
tearDown
```

Warto zauważyć, że, generalnie rzecz biorąc, w Pythonie nazwy metod piszemy małymi literami, a poszczególne słowa rozdzielamy podkreślnikami, np. `tear_down`, `set_up`. Jednak konwencja ta nie zawsze jest przestrzegana, nawet w obrębie biblioteki standardowej, czego przykładem jest `unittest`. Jest ona wzorowana na bibliotece `JUnit` napisanej w Javie, gdzie obowiązuje konwencja "camelCase".

self.assert*

W środku każdego testu możemy wykorzystać szereg metod zaczynających się od `assert`, np. `assertEqual`. Test przechodzi, jeżeli wszystkie takie asercje są prawdziwe. Jeżeli chociaż jedna taka asercja nie będzie spełniona, wówczas wykonywanie testu jest natychmiast przerywane i wykonywana jest metoda `tearDown`.

Najczęściej wykorzystywane asercje to:

- `self.assertTrue(condition)` i `self.assertFalse(condition)`,
- `self.assertEqual(got, expected)` i `self.assertNotEqual(got, expected)`,
- `self.assertIn(element, collection)` i `self.assertNotIn(element, collection)`,
- `self.assertIsInstance(obj, class_)` i `self.assertNotIsInstance(obj, class_)`.

Do sprawdzenia, czy dany blok kodu rzuca wyjątek, można użyć `self.assertRaises(ExceptionType)`:

```
class UrlConverterTests(unittest.TestCase):
    def test_raises_exception_for_empty_string(self):
        url = ""

        with self.assertRaises(ValueError):
            url_converter(url)
```

Jeżeli sprawdzenie typu rzuconego wyjątku to za mało, możemy uzyskać do niego dostęp:

```
class UrlConverterTests(unittest.TestCase):
    def test_raises_exception_for_empty_string(self):
        url = ""

        with self.assertRaises(ValueError) as ex:
            url_converter(url)
        self.assertEqual(ex.message, 'Empty url')
```

0.10.2 Doctest

`doctest` jest częścią standardowej biblioteki Pythona, ale oferuje zupełnie inne podejście do testowania. Zamiast umieszczać testy w osobnych plikach, testy można umieścić w docstringu testowanej funkcji, klasy lub metody. Takie podejście ma kilka przewag nad `unittest`:

- Ponieważ testy są częścią docstringa, pełnią wówczas jednocześnie rolę dokumentacji. Jeżeli są to krótkie testy, jest to wówczas bardzo dobra dokumentacja.
- Testy są trzymane blisko obiektu, który jest testowany, co jest generalnie pożądane, ponieważ nie ma potrzeby przeskakiwania między plikami.

Niestety, to podejście ma też pewne istotne wady. Przede wszystkim, takie podejście nie skaluje się wraz z coraz bardziej skomplikowanymi testami, co oznacza, że sprawdza się ono głównie w przypadku krótkich testów dla prostych funkcji.

```
import doctest
import math

def factorial(n):
    """Return the factorial of n, an exact integer >= 0.

    >>> factorial(3)
    6
    >>> factorial(30)
    265252859812191058636308480000000
    >>> factorial(-1)
    Traceback (most recent call last):
        ...
    ValueError: n must be >= 0
    >>> [factorial(n) for n in range(6)]
    [1, 1, 2, 6, 24, 120]
    """

    if not n >= 0:
        raise ValueError("n must be >= 0")
    result = 1
    factor = 2
    while factor <= n:
        result *= factor
        factor += 1
    return result

# Uruchomienie testów
if __name__ == "__main__":
    doctest.testmod()
```

0.10.3 Pytest

Wprowadzenie

Zasadniczą wadą biblioteki `unittest` jest rozwlekłość pisanych w nich testów. Każdy test musi być metodą umieszczoną w klasie. Nazwy metod takie jak `self.assertEqual` są dosyć rozwlekłe.

Czy można pisać krótsze, czytelniejsze testy?

Odpowiedź brzmi tak, wystarczy doinstalować bibliotekę `pytest`.

```
pip install pytest
```

`pytest` pozwala umieścić testy w funkcjach, których nazwa zaczyna się od `test_`. Ponadto, można użyć asercji zamiast metod takich jak `self.assertEqual`.

Proste testy

W bibliotece każda funkcja zaczynająca się od `test_*` jest traktowana jak test.

Asercje są przeprowadzane za pomocą standardowej instrukcji `assert`.

```
# my_module.py
def increment(x):
    if x == 0:
        raise ValueError('Zero is not valid value.')
    return x + 1

# test_my_module.py
import pytest

from my_module import inc

def test_increment_returns_next_value():
    # Poniższa asercja jest znacznie czytelniejsza niż self.assertEqual(inc(3), 4)
    assert increment(3) == 4
```

Asercja wyjątków jest przeprowadzana za pomocą menadżera kontekstu `pytest.raises`:

```
def test_increment_raises_when_invalid_argument():
    with pytest.raises(ValueError):
        increment(0)
```

Grupowanie testów w klasach

Testy mogą być grupowane w klasach. Klasa grupująca powinna zaczynać się od `Test*`, w przeciwnym razie testy są pomijane. Nie jest wymagane dziedziczenie po klasie bazowej, tak jak ma to miejsce w bibliotece `unittest`.

```
class TestMultiple:
    def test_first(self):
        assert 5 == 5

    def test_second(self):
        assert 10 == 10
```

Uruchamianie testów

`pytest`, podobnie jak `unittest` potrafi sam znaleźć wszystkie testy w aktualnym katalogu i podkatalogach.

```
$ pytest -v
```

```
============ test session starts ============
platform linux -- Python 3.7.3, pytest-5.4.3, py-1.8.1, pluggy-0.13.1
cachedir: .pytest_cache
rootdir: /chatapp
collected 3 items

test_simple.py::test_something FAILED [ 33%]
test_simple.py::TestMultiple::test_first PASSED [ 66%]
test_simple.py::TestMultiple::test_second PASSED [100%]
```

Przydatną opcją jest `-k EXPRESSION`, która powoduje uruchamienie tylko testów, których nazwa pasuje do podanego wyrażenia tekstowego (case-insensitive):

```
$ pytest -v -k first

$ pytest -v -k "not something"
```

Fikstury

Dostarczanie obiektów, skonfigurowanych na potrzeby testu jest realizowane w `pytest` za pomocą funkcji z dekoratorem `@pytest.fixture`. Taka funkcja może być później użyta w teście jako parametr. Powoduje to wstrzyknięcie do testu odpowiednio skonfigurowanego obiektu.

```
@pytest.fixture()
def warehouse():
    warehouse = InMemoryWarehouse()
    warehouse.add("ProductA", 50)
    return warehouse

def test_order_is_filled_if_enough_items_in_warehouse(warehouse):
    order_service = OrderService(warehouse)
    order = order_service.process_order("ProductA", 50)
    assert order.is_filled()
```

Fikstury mogą też przyjmować jako parametry inne fikstury:

```
# Arrange
@pytest.fixture
def first_entry():
    return "a"

# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]
```

```
def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]
```

Możemy użyć dowolną liczbę fiktur w teście (lub innej fiksturze):

```
@pytest.fixture
def first_entry():
    return "a"

@pytest.fixture
def second_entry():
    return 2

@pytest.fixture
def order(first_entry, second_entry):
    return [first_entry, second_entry]

@pytest.fixture
def expected_list():
    return ["a", 2, 3.0]

def test_string(order, expected_list):
    order.append(3.0)
    assert order == expected_list
```

Fikstura może być też implementowana jako "fabryka" (może być to potrzebne, gdy wynik fikstury jest potrzebny wiele razy w teście):

```
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {"name": name, "orders": []}

    return _make_customer_record


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

Fikstury - setup & teardown

Implementacja wzorca setup & teardown w `pytest` wykorzystuje generator:

```
@pytest.fixture
def new_user(user_service):
    # Setup
    user = user_service.create_user()

    yield user

    # Teardown
    user_service.delete(user)


def test_sending_email_to_new_user(new_user):
    email = Email(subject="Greetings!", body="Welcome")
    response = new_user.send_email(email)
    assert response.status == 'OK'
```

Wbudowane fikstury

- `tmp_path` - dostarcza tymczasową ścieżkę do plików, która z każdym uruchomieniem testu jest inna

```
def test_tmp(tmp_path):
    f = tmp_path / "file.txt"
    print("FILE: ", f)

    f.write_text("Hello World")

    fread = tmp_path / "file.txt"
    assert fread.read_text() == "Hello World"
```

```
test_tmppath.py::test_tmp
FILE: /tmp/pytest-of-amol/pytest-3/test_tmp0/file.txt
PASSED
```

- `capsys` - pozwala odczytać tekst wypisywany w konsoli (`sys.stdout` i `sys.stderr`)

```
def myapp():
    print("MyApp Started")

def test_capsys(capsys):
    myapp()

    out, err = capsys.readouterr()

    assert out == "MyApp Started\n"
```

Parametryzacja testów

Dekorator `@mark.paramterize` umożliwia łatwą parametryzacje testów:

```
@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Parametryzacja może być również stosowana dla całej klasy testów:

```
@pytest.mark.parametrize("n,expected", [(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```