# Kurs z zakresu Python PCAP i PCPP

Dr inż. Marcin Caryk

1

# Rozszerzona składnia argumentów funkcji

## Extended function argument syntax

# Extended function argument syntax

Kiedy mowa o argumentach funkcji, należy pamiętać o następujących faktach:

- niektóre funkcje mogą być wywoływane bez argumentów;
- funkcje mogą wymagać określonej liczby argumentów bez wyjątków;
- należy przekazać wymaganą liczbę argumentów w narzuconej kolejności, aby podążać za definicją funkcji;
- funkcje mogły mieć już zdefiniowane wartości domyślne dla niektórych parametrów, więc nie trzeba przekazywać wszystkich argumentów jako argumentów brakujących, wraz z wartościami domyślnymi;
- parametry z wartościami domyślnymi są prezentowane jako parametry słów kluczowych;
- można przekazywać argumenty w dowolnej kolejności, jeśli przypisane są słowa kluczowe do wszystkich wartości argumentów, w przeciwnym razie argumenty pozycyjne są pierwszymi na liście argumentów.

# Extended function argument syntax

Najbardziej podstawowa funkcja to print():

```
print()
print(3)
print(1, 20, 10)
print('--', '++')
print(f'Lista to {[1, 2, 4]} ')
print('Lista to {}'.format([1, 2, 4],))
print('Lista to %s' % ([1, 2, 4]))
print('{:#>15}'.format("Hello"))
print('{:#<15}'.format("Hello"))
print('{:#^15}'.format("Hello"))
```

Albo funkcje wbudowanych typów danych

```
a_list = list()
b_list = list((10, 20, 43, 54, 23, 23, 34, 23, 2))

print(a_list)
print(b_list)
```

# Extended function argument syntax

Dwa specjalne identyfikatory (nazwane *args i **kwargs) należy umieścić jako dwa ostatnie parametry w definicji funkcji. Ich nazwy można by zmienić, ponieważ jest to tylko konwencja nazywania ich „args” i „kwargs”, ale ważniejsze jest zachowanie kolejności parametrów i wiodących gwiazdek.

Te dwa specjalne parametry odpowiadają za obsługę dowolnej liczby dodatkowych argumentów (umieszczonych obok oczekiwanych argumentów) przekazywanych do wywoływanej funkcji:

***args** – odnosi się do krotki wszystkich dodatkowych, nieoczekiwanych jawnie argumentów pozycyjnych, więc argumenty są przekazywane bez słów kluczowych i przekazywane jako następne po oczekiwanych argumentach. Innymi słowy, *args zbiera wszystkie niedopasowane argumenty pozycyjne;

****kwargs** (keyword arguments) – odnosi się do słownika wszystkich nieoczekiwanych argumentów, które zostały przekazane w postaci par słowo kluczowe=wartość. Podobnie **kwargs zbiera wszystkie niedopasowane argumenty słów kluczowych.

W Pythonie gwiazdki są używane do oznaczenia, że parametry args i kwargs nie są zwykłymi parametrami i należy je rozpakować, ponieważ zawierają wiele elementów.

# Extended function argument syntax

```
def combiner(a, b, *args, **kwargs):
    print(a, type(a))
    print(b, type(b))
    print(args, type(args))
    print(kwargs, type(kwargs))


combiner(10, '20', 40, 60, 30, argument1=50,
argument2='66')
```

Jak widać, definicja funkcji oczekuje dwóch argumentów, **a** i **b**, oraz definicja jest przygotowana na obsługę dowolnej liczby dodatkowych argumentów. Co więcej, wszystkie inne argumenty pozycyjne są dostępne w krotce (jak być może pamiętasz, krotka jest typem sekwencji, ponieważ w tym przypadku kolejność ma znaczenie). Podobnie wszystkie nieoczekiwane parametry słów kluczowych są dostępne w parametrach typu słownikowego.

Teraz, jeśli spojrzeć na wbudowaną definicję funkcji print(), stanie się jasne, w jaki sposób ta funkcja może przyjąć dowolną liczbę argumentów i dlaczego przed jednym z parametrów znajduje się gwiazdka:

```
def print(self, *args, sep=' ', end='\n', file=None):
```

## Rozszerzona składnia argumentów funkcji – przekazywanie argumentów do innych funkcji

```
def combiner(a, b, *args, **kwargs):
    super_combiner(*args, **kwargs)

def super_combiner(*my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_kwargs', my_kwargs)

combiner(10, '20', 40, 60, 30, argument1=50,
argument2='66')
```

Należy zwróć uwagę, jak jest wywoływana funkcja super_combiner() – jest wywoływana z prawdziwymi argumentami, więc te argumenty mogą być obsługiwane w taki sam sposób, jak są obsługiwane przez funkcję łączącą. Jeśli gwiazdki zostaną usunięte z wywołania funkcji, to zarówno krotka, jak i słownik zostaną przechwycone przez my_args, ponieważ ma obsługiwać wszystkie argumenty pozycyjne (żaden z nich nie jest opatrzony słowami kluczowymi).

# Extended function argument syntax

Ostatni przykład w tej sekcji pokazuje, jak połączyć *args, słowo kluczowe, i **kwargs w jednej definicji:

```
def combiner(a, b, *args, c=20, **kwargs):
    super_combiner(c, *args, **kwargs)
def super_combiner(my_c, *my_args, **my_kwargs):
    print('my_args:', my_args)
    print('my_c:', my_c)
    print('my_kwargs', my_kwargs)

combiner(1, '1', 1, 1, c=2, argument1=1, argument2='1')
```

Jak widać, Python oferuje złożoną obsługę parametrów:

- argumenty pozycyjne (a, b) różnią się od wszystkich innych argumentów pozycyjnych (args)
- słowo kluczowe „c” odróżnia się od wszystkich innych parametrów oznaczonych słowami kluczowymi.

2

# Programowanie zorientowane obiektowo

## Object-Oriented Programming



- Podejście obiektowe jest ewolucją dobrych praktyk projektowych, które sięgają samych początków programowania komputerów.
- To bardzo ważne podejście jest obecne w większości aplikacji komputerowych, ponieważ pozwala programistom modelować obiekty reprezentujące rzeczywiste obiekty. Co więcej, OOP umożliwia programistom modelowanie interakcji między obiektami w celu rozwiązywania rzeczywistych problemów w wydajny, wygodny, rozszerzalny i dobrze ustrukturyzowany sposób.
- Terminy w OPP:

**class** — pomysł, schemat lub przepis na instancję;

**instance** — instancja klasy; bardzo często używany zamiennie z terminem „obiekt”;

**object** — reprezentacja danych i metod w Pythonie; obiekty mogą być agregatami instancji;

**attribute** — dowolna cecha obiektu lub klasy; może być zmienną lub metodą;

**method** — funkcja wbudowana w klasę, która jest wykonywana w imieniu klasy lub obiektu; niektórzy twierdzą, że jest to „atrybut wywoływalny”;

**type** — odnosi się do klasy, która została użyta do utworzenia instancji obiektu.



- W Pythonie wszystko jest obiektem (funkcje, moduły, listy, zmienne itp.)
- Obiekt jest niezależną instancją klasy i zawiera oraz agreguje pewne specyficzne i wartościowe dane w atrybutach odpowiednich dla poszczególnych obiektów;
- Obiekt posiada i udostępnia metody wykorzystywane do wykonywania działań.
- Świat zorientowany obiektowo przedstawia koncepcję obiektów, które mają atrybuty (data members) i procedury (member functions), a te funkcje są odpowiedzialne za manipulowanie atrybutami;
- Weźmy na przykład obiekt Samochód. Obiekt Car będzie posiadał atrybuty takie jak poziom paliwa, isSedan, prędkość oraz kierownica i współrzędne, a metody to przyspieszenie(), aby zwiększyć prędkość i takeTurn(), aby samochód skręcił;
- Każda instancja lub zmienna klasy ma swój własny adres pamięci lub tożsamość. Obiekty, które są instancjami klas, oddziałują między sobą, aby służyć celom tworzonej aplikacji.

**Class**

- Klasa wyraża ideę, to plan lub przepis na instancję. Klasa jest czymś wirtualnym, może zawierać wiele różnych szczegółów i zawsze istnieje jedna klasa dowolnego typu.
- Klasy opisują razem atrybuty i funkcjonalności, aby jak najdokładniej przedstawić ideę.
- Możesz zbudować klasę od zera lub zastosować dziedziczenie, aby uzyskać bardziej wyspecjalizowaną klasę opartą na innej klasie.
- Dodatkowo, twoje klasy mogą być używane jako nadklasy dla nowo pochodnych klas (podklas).

```
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')
```

**Instance**

- Instancja to jedna konkretna fizyczna reprezentacja klasy, która zajmuje pamięć i zawiera elementy danych. Do tego właśnie odnosi się słowo „self”, gdy mamy do czynienia z instancjami klas.
- Obiekt to wszystko, na czym można operować w Pythonie, na przykład klasa, instancja, lista lub słownik.
- Termin instancja jest bardzo często używany zamiennie z terminem obiekt, ponieważ obiekt odnosi się do konkretnej instancji klasy. To trochę uproszczenie, ponieważ termin obiekt jest bardziej ogólny niż instancja.
- Relacja między instancjami a klasami jest dość prosta: mamy jedną klasę danego typu i nieograniczoną liczbę instancji danej klasy.
- Każda instancja ma swój własny, indywidualny stan (wyrażony jako zmienne, czyli znowu obiekty) i dzieli swoje zachowanie (wyrażone jako metody, czyli znowu obiekty).

```
duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")
```

**Attribute**

- To obszerny termin, który może odnosić się do dwóch głównych rodzajów cech klasowych:
    - zmienne, zawierające informacje o samej klasie lub instancji klasy;
    - klasy i instancje klas mogą posiadać wiele zmiennych;
    - metody sformułowane jako funkcje Pythona;
    - reprezentują zachowanie, które można zastosować do obiektu.
- Każdy obiekt Pythona ma swój indywidualny zestaw atrybutów. Możemy rozszerzyć ten zestaw, dodając nowe atrybuty do istniejących obiektów, zmieniać je (ponownie przypisywać) lub kontrolować dostęp do tych atrybutów.
- Metody są wywoływane w imieniu obiektu i zwykle są wykonywane na danych obiektu.
- Atrybuty klasy są najczęściej adresowane za pomocą notacji „kropka”.
- Innym sposobem dostępu do atrybutów (zmiennych) jest użycie funkcji getattr() i setattr().

# Object-Oriented Programming – Przykład

```
class Duck:
    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex

    def walk(self):
        pass

    def quack(self):
        return print('Quack')

duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

drake.quack()
print(duckling.height)
```

zmienne

metody

**Type**

- Jest to najważniejszy typ, z którego można odziedziczyć każdą klasę.
- W rezultacie, jeśli szukasz typu klasy, zwracany jest typ.
- We wszystkich innych przypadkach odnosi się do klasy, która została użyta do utworzenia instancji obiektu.
- Jest to nazwa bardzo przydatnej funkcji Pythona, która zwraca informacje o klasach obiektów przekazywanych jako argumenty tej funkcji.
- Zwraca nowy obiekt typu, gdy type() jest wywoływana z trzema argumentami.

Informacje o klasie obiektu zawarte są w __class__.

```
print(Duck.__class__)
print(duckling.__class__)
print(duckling.sex.__class__)
print(duckling.quack.__class__)
```

Python pozwala na używanie zmiennych na poziomie instancji lub klasy. Te używane na poziomie instancji są określane jako zmienne instancji, podczas gdy zmienne używane na poziomie klasy są określane jako zmienne klasowe.

**Zmienna instancji:**

Tego rodzaju zmienna istnieje wtedy i tylko wtedy, gdy jest jawnie tworzona i dodawana do obiektu. Można to zrobić podczas inicjalizacji obiektu metodą __init__ lub później w dowolnym momencie życia obiektu. Ponadto każdą istniejącą własność można usunąć w dowolnym momencie.

Zmienne instancji mogą być tworzone w dowolnym momencie życia obiektu. Ponadto wyświetla zawartość każdego obiektu, używając wbudowanej właściwości __dict__, która jest obecna dla każdego obiektu Pythona.

```
class Demo:
    def __init__(self, value):
        self.instance_var = value

d1 = Demo(100)
d2 = Demo(200)

d1.another_var = 'another variable in the object'

print("d1's instance variable is equal to:", d1.instance_var)
print("d2's instance variable is equal to:", d2.instance_var)

print('contents of d1:', d1.__dict__)
print('contents of d2:', d2.__dict__)
```

**Zmienna klasowe:**

Zmienne klasy są zdefiniowane w konstrukcji klasy, więc zmienne te są dostępne przed utworzeniem jakiejkolwiek instancji klasy. Aby uzyskać dostęp do zmiennej klasy, po prostu uzyskaj do niej dostęp za pomocą nazwy klasy, a następnie podaj nazwę zmiennej.

Ponieważ zmienna klasy jest obecna przed utworzeniem jakiejkolwiek instancji klasy, można jej użyć do przechowywania pewnych metadanych związanych z klasą, a nie z instancjami:

- stałe informacje, takie jak opis, konfiguracja lub wartości identyfikacyjne;
- zmienne informacje, takie jak liczba utworzonych instancji

```
class Demo:
    class_var = 'shared variable'

print(Demo.class_var)
print(Demo.__dict__)

d1 = Demo()
d2 = Demo()

print(Demo.class_var)
print(d1.class_var)
print(d2.class_var)

print('contents of d1:', d1.__dict__)
```

# Object-Oriented Programming – Przykład

```
class Duck:
    counter = 0
    species = 'duck'

    def __init__(self, height, weight, sex):
        self.height = height
        self.weight = weight
        self.sex = sex
        Duck.counter +=1

    def walk(self):
        pass

    def quack(self):
        print('quacks')

class Chicken:
    species = 'chicken'

    def walk(self):
        pass

    def cluck(self):
        print('clucks')
```

```
duckling = Duck(height=10, weight=3.4, sex="male")
drake = Duck(height=25, weight=3.7, sex="male")
hen = Duck(height=20, weight=3.4, sex="female")

chicken = Chicken()

print('So many ducks were born:', Duck.counter)

for poultry in duckling, drake, hen, chicken:
    print(poultry.species, end=' ')
    if poultry.species == 'duck':
        poultry.quack()
    elif poultry.species == 'chicken':
        poultry.cluck()
```

# Object-Oriented Programming – Przykład 2

```
class Phone:
    counter = 0

    def __init__(self, number):
        self.number = number
        Phone.counter += 1

    def call(self, number):
        message = 'Calling {} using own number {}'.format(number, self.number)
        return message

class FixedPhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        FixedPhone.last_SN += 1
        self.SN = 'FP-{}'.format(FixedPhone.last_SN)


class MobilePhone(Phone):
    last_SN = 0

    def __init__(self, number):
        super().__init__(number)
        MobilePhone.last_SN += 1
        self.SN = 'MP-{}'.format(MobilePhone.last_SN)
```

# Object-Oriented Programming – Konstruktor i destruktor

```
class Person:
    counter = 0

    def __init__(self, author="Janusz"):
        self.created_by = str(author)
        type(self).counter += 1

    def __del__(self):
        type(self).counter -= 1

person1 = Person("Artur")
print(person1.counter, person1.created_by)
person2 = Person("Joanna")
print(person1.counter, person2.created_by)
person3 = Person("Karol")
print(person1.counter, person3.created_by)

print(person1.counter, person2.counter, person3.counter)
del person2
print(person1.counter, person1.created_by)
del person3
print(person1.counter, person1.created_by)
```

- __init__ jest tzw. konstruktorem a __del__ jest tak zwanym destruktorem w założeniach obiektowo zorientowanego programowania
- __init__ wywoływany jest przy tworzeniu instancji klasy żeby ustawić wszystkie atrybuty klasy
- __del__ wywoływany jest po usunięciu obiektu. W pythonie destruktory wywoływane są automatycznie kiedy dany obiekt kończy swoje życie.

```
person1 = Person("Robot1")
person2 = Person("Robot2")
person3 = Person("Robot3")

print(person1.counter, person2.counter, person3.counter)
person3.counter = 10
print(person1.counter, person2.counter, person3.counter)
type(person1).counter = 5
print(person1.counter, person2.counter, person3.counter)
```

- W python operator „+” jest faktycznie konwertowany na metodę __add__() a funkcja len() na metodę __len__().
- Metody te muszą być dostarczone przez klasę, aby wykonać odpowiednią akcję.

```
number = 10
print(number + 20)

number = 10
print(number.__add__(20))
```

```
class Person:
    def __init__(self, weight, age, salary):
        self.weight = weight
        self.age = age
        self.salary = salary

    def __add__(self, other):
        return self.weight + other.weight


p1 = Person(30, 40, 50)
p2 = Person(35, 45, 55)

print(p1 + p2)
```

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f'Vector ({self.x},{self.y})'

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)


v1 = Vector(2, 5)
v2 = Vector(5, -2)
print(v1, v2, v1 + v2)
```

# Object-Oriented Programming – specjalne metody

- Każda klasa (typ klasy) ma swoje specjalne metody nazywane też magiczne metody albo dunder methods __<metoda>__.
- W celu sprawdzenia jakie klasa ma metody specjalne dir() i help().
- Można je wywołać używając np. dict(int()), która wyświetla listę wszystkich metod
- Funkcja help(int()) Pythona służy do wyświetlania dokumentacji modułów, funkcji, klas i słów kluczowych.

```
>>> dir(int())
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_length', 'conjugate', 'denominator', 'from_bytes',
'imag', 'numerator', 'real', 'to_bytes']
```

```
Help on int object:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is a number, return x.__int__().  For floating point
 |  numbers, this truncates towards zero.
 |
 |  If x is not a number or if base is given, then x must be a string,
 |  bytes, or bytearray instance representing an integer literal in the
 |  given base.  The literal can be preceded by '+' or '-' and be surrounded
 |  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
 |  Base 0 means to interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Built-in subclasses:
 |      bool
 |
 |  Methods defined here:
 |
 |  __abs__(self, /)
 |      abs(self)
 |
 |  __add__(self, value, /)
 |      Return self+value.
```

# Object-Oriented Programming – specjalne metody

```
class Test: pass
t = Test()

class Test1(Test):
    def __repr__(object):
        return 'Testing.Test1'

class Test2(Test):
    def __str__(object):
        return 'Testing.Test2'
```

```
print(dir(t))



print(t.__repr__())
print(t.__str__())

print(str(Test1()))
print(repr(Test1()))
print(str(Test2()))
print(repr(Test2()))
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__']
<__main__.Test object at 0x0000020B136232E0>
<__main__.Test object at 0x0000020B136232E0>
Testing.Test1
Testing.Test1
Testing.Test2
<__main__.Test2 object at 0x0000020B13623340>
```

object.**__init__** ( self [,args...] ),  obj = className(args) – konstruktor klasy

object.**__del__**( self ),  del obj – destruktor klasy

object.**__repr__**( self ),  repr(obj) – Funkcja Pythona __repr__() zwraca reprezentację obiektu w formacie String. Ta metoda jest wywoływana, gdy na obiekcie wywoływana jest funkcja repr(). Jeśli to możliwe, zwrócony ciąg znaków powinien być prawidłowym wyrażeniem języka Python, którego można użyć do ponownego zrekonstruowania obiektu.

object.**__str__**( self ),  str(obj) – Ta metoda zwraca ciąg reprezentujący obiekt. Ta metoda jest wywoływana, gdy funkcja print() lub str() jest wywoływana na obiekcie. Ta metoda musi zwrócić obiekt String. Jeśli nie zaimplementujemy funkcji __str__() dla klasy, wówczas używana jest wbudowana implementacja obiektowa, która faktycznie wywołuje funkcję __repr__().

```
import datetime
now = datetime.datetime.now()
print(now.__str__())
print(now.__repr__())
```

Funkcja **__str__** ma zwracać format czytelny dla człowieka, który jest dobry do logowania lub wyświetlania informacji o obiekcie. Natomiast funkcja **__repr__** ma zwrócić „oficjalną” reprezentację obiektu w postaci ciągu znaków, która może posłużyć do ponownego skonstruowania obiektu.

# Object-Oriented Programming – specjalne metody

object.__**lt**__(self, other), x < y

object.__**le**__(self, other), x <= y

object.__**eq**__(self, other), x == y

object.__**ne**__(self, other), x != y

object.__**gt**__(self, other), x > y

object.__**ge**__(self, other), x >= y

```
class Ball(object):
    def __init__(self, color, size):
        self.color = color
        self.size = size

ball1 = Ball('blue', 'small')
ball2 = Ball('blue', 'small')

print(ball1 == ball2)
print(id(ball1))
print(id(ball2))
```

Funkcja id() zwraca „tożsamość” obiektu. Tożsamość obiektu jest liczbą całkowitą, która gwarantuje, że będzie unikalna i stała dla tego obiektu przez cały okres jego istnienia. Dwa obiekty o nienakładających się okresach życia mogą mieć tę samą wartość id(). W implementacji CPython jest to adres obiektu w pamięci

```
class Ball(object):

    def __init__(self, color, size):
        self.color = color
        self.size = size

    def __eq__(self, other):
        return self.color == other.color and self.size
== other.size

    def __ne__(self, other):
        return self.color != other.color or self.size !=
other.size


ball1 = Ball('blue', 'small')
ball2 = Ball('blue', 'small')
ball3 = Ball('green', 'small')

print(ball1 == ball2)
print(ball1 == ball3)
```

Python zawiera grupę z podstawowymi wyrażeniami składni i odpowiadającymi im magicznymi metodami

## Metody porównania

| Function or operator | Magic method | Implementation meaning or purpose |
|---|---|---|
| == | __eq__(self, other) | equality operator |
| != | __ne__(self, other) | inequality operator |
| < | __lt__(self, other) | less-than operator |
| > | __gt__(self, other) | greater-than operator |
| <= | __le__(self, other) | less-than-or-equal-to operator |
| >= | __ge__(self, other) | greater-than-or-equal-to operator |

## Jednoargumentowe metody i funkcje

| Function or operator | Magic method | Implementation meaning or purpose |
|---|---|---|
| + | __pos__(self) | **unary** positive, like a = +b |
| - | __neg__(self) | **unary** negative, like a = -b |
| abs() | __abs__(self) | behavior for abs() function |
| round(a, b) | __round__(self, b) | behavior for round() function |

## Typowe, binarne operatory i funkcje

| Function or operator | Magic method | Implementation meaning or purpose |
|---|---|---|
| + | __add__(self, other) | addition operator |
| - | __sub__(self, other) | subtraction operator |
| * | __mul__(self, other) | multiplication operator |
| // | __floordiv__(self, other) | integer division operator |
| / | __div__(self, other) | division operator |
| % | __mod__(self, other) | modulo operator |
| ** | __pow__(self, other) | exponential (power) operator |

## Rozszerzone operatory i funkcje

| Function or operator | Magic method | Implementation meaning or purpose |
|---|---|---|
| += | __iadd__(self, other) | addition and assignment operator |
| -= | __isub__(self, other) | subtraction and assignment operator |
| *= | __imul__(self, other) | multiplication and assignment operator |
| //= | __ifloordiv__(self, other) | integer division and assignment operator |
| /= | __idiv__(self, other) | division and assignment operator |
| %= | __imod__(self, other) | modulo and assignment operator |
| **= | __ipow__(self, other) | exponential (power) and assignment operator |

## Metody konwersji typów

| Function | Magic method | Implementation meaning or purpose |
|---|---|---|
| int() | __int__(self) | conversion to integer type |
| float() | __float__(self) | conversion to float type |
| oct() | __oct__(self) | conversion to string, containing an octal representation |
| hex() | __hex__(self) | conversion to string, containing a hexadecimal representation |

## Dostęp do atrybutu obiektu

| Expression example | Magic method | Implementation meaning or purpose |
|---|---|---|
| object.attribute | __getattr__(self, attribute) | responsible for handling access to a non-existing attribute |
| object.attribute | __getattribute__(self, attribute) | responsible for handling access to an existing attribute |
| object.attribute = value | __setattr__(self, attribute, value) | responsible for setting an attribute value |
| del object.attribute | __delattr__(self, attribute) | responsible for deleting an attribute |

## Introspekcja obiektu

| Function | Magic method | Implementation meaning or purpose |
|---|---|---|
| str() | __str__(self) | responsible for handling str() function calls |
| repr() | __repr__(self) | responsible for handling repr() function calls |
| format() | __format__(self, formatstr) | called when new-style string formatting is applied to an object |
| hash() | __hash__(self) | responsible for handling hash() function calls |
| dir() | __dir__(self) | responsible for handling dir() function calls |
| bool() | __nonzero__(self) | responsible for handling bool() function calls |

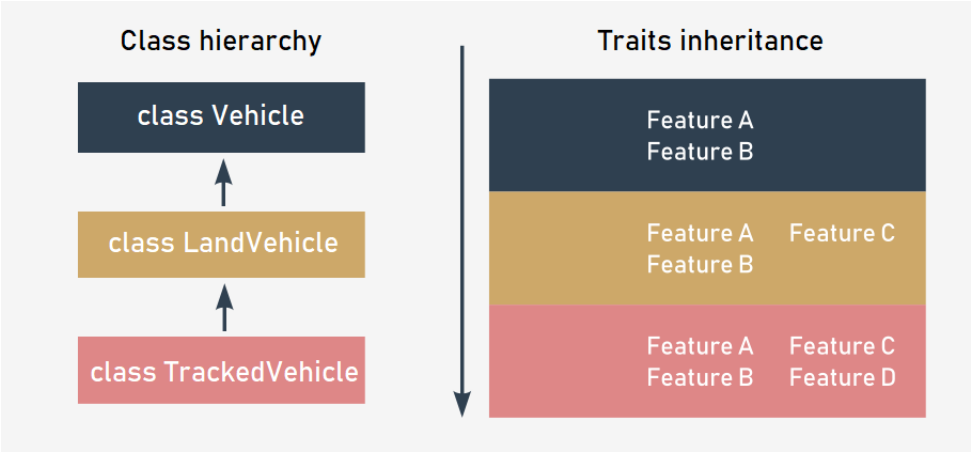
## Metody umożliwiające dostęp do pojemników

| Expression example | Magic method | Implementation meaning or purpose |
|---|---|---|
| len(container) | __len__(self) | returns the length (number of elements) of the container |
| container[key] | __getitem__(self, key) | responsible for accessing (fetching) an element identified by the key argument |
| container[key] = value | __setitem__(self, key, value) | responsible for setting a value to an element identified by the key argument |
| del container[key] | __delitem__(self, key) | responsible for deleting an element identified by the key argument |
| for element in container | __iter__(self) | returns an iterator for the container |
| item in container | __contains__(self, item) | responds to the question: does the container contain the selected item? |

## Retrospekcja obiektu

| Function | Magic method | Implementation meaning or purpose |
|---|---|---|
| isinstance(object, class) | __instancecheck__(self, object) | responsible for handling isinstance() function calls |
| issubclass(subclass, class) | __subclasscheck__(self, subclass) | responsible for handling issubclass() function calls |

- Dziedziczenie jest jednym z podstawowych pojęć programowania obiektowego i wyraża podstawowe relacje między klasami: nadklasami (rodzicami) i ich podklasami (potomkami).
- Dziedziczenie tworzy hierarchię klas. Każdy obiekt związany z określonym poziomem hierarchii klas dziedziczy wszystkie cechy (metody i atrybuty) zdefiniowane w którejkolwiek z nadklas.
- Każda podklasa jest bardziej wyspecjalizowana (lub bardziej szczegółowa) niż jej nadklasa. I odwrotnie, każda nadklasa jest bardziej ogólna (bardziej abstrakcyjna) niż którakolwiek z jej podklas.




```
class Vehicle:
    pass

class LandVehicle(Vehicle):
    pass

class TrackedVehicle(LandVehicle):
    pass
```

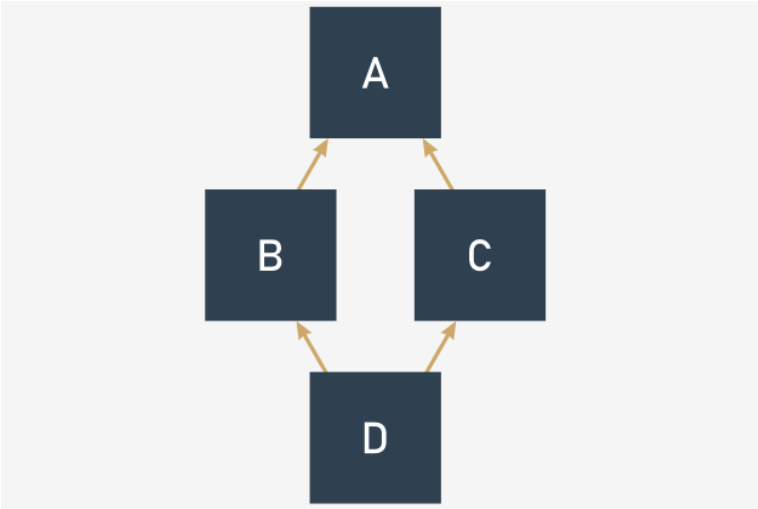


- Nie ma przeszkód, aby używać dziedziczenia wielokrotnego w Pythonie. Dowolną nową klasę można wyprowadzić z więcej niż jednej wcześniej zdefiniowanej klasy.
- Jednak dziedziczenie wielokrotne powinno być stosowane z większą rozwagą niż dziedziczenie pojedyncze, ponieważ pojedyncza klasa dziedziczenia jest zawsze prostsza, bezpieczniejsza i łatwiejsza do zrozumienia i utrzymania
- Wielokrotne dziedziczenie może sprawić, że nadpisywanie metod będzie nieco trudne;
- Użycie funkcji super() może prowadzić do niejasności
- Jest wysoce prawdopodobne, że wdrażając dziedziczenie wielokrotne naruszasz zasadę pojedynczej odpowiedzialności;
- Jeśli Twoje rozwiązanie zwykle wymaga wielokrotnego dziedziczenia, dobrym pomysłem może być rozważenie zaimplementowania kompozycji.

MRO — Method Resolution Order - Kolejność rozwiązania metody

- Spektrum problemów wynikających prawdopodobnie z wielokrotnego dziedziczenia ilustruje klasyczny problem zwany problemem diamentu, a nawet śmiercionośnego diamentu.
- Nazwa odzwierciedla kształt diagramu dziedziczenia

- Istnieje najwyższa nadklasa o nazwie A
- Istnieją dwie podklasy pochodne od A — B i C
- Jest też najniższa podklasa o nazwie D, wywodząca się z B i C (lub C i B, ponieważ te dwa warianty oznaczają różne rzeczy w Pythonie)



# Object-Oriented Programming – MRO

- Powstająca tutaj niejasność wynika z faktu, że klasa B i klasa C są dziedziczone z nadklasy A, a klasa D z obu klas B i C.
- Co naprawdę pokaże metoda info()

- W scenariuszu dziedziczenia wielokrotnego każdy określony atrybut jest wyszukiwany jako pierwszy w bieżącej klasie.
- Jeśli nie zostanie znaleziona, wyszukiwanie jest kontynuowane w bezpośrednich klasach nadrzędnych na pierwszym poziomie w głąb (pierwszy poziom powyżej), od lewej do prawej, zgodnie z definicją klasy.
- Jest to wynik działania algorytmu MRO.

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

D().info()
```



- MRO może zgłaszać niespójności definicji po wprowadzeniu subtelnej zmiany w definicji klasy D, co jest możliwe podczas pracy ze złożonymi hierarchiami klas.
- Zmieniamy definicje klasy D

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

na

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

- Komunikat ten informuje nas, że algorytm MRO miał problemy z określeniem, która metoda (pochodząca z klasy A lub C) powinna zostać wywołana.

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(A, C):
    pass

D().info()
```

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



- Ze względu na MRO należy świadomie wymienić nadklasy w definicji podklasy.
- W przykładzie klasa D jest oparta na klasach B i C, podczas gdy klasa E jest oparta na klasach C i B (kolejność ma znaczenie!).
- W rezultacie klasy te mogą zachowywać się zupełnie inaczej, ponieważ kolejność nadklas jest inna.

```
class A:
    def info(self):
        print('Class A')

class B(A):
    def info(self):
        print('Class B')

class C(A):
    def info(self):
        print('Class C')

class D(B, C):
    pass

class E(C, B):
    pass

D().info()
E().info()
```

Class B
Class C

- W Pythonie polimorfizm to udostępnianie jednego interfejsu obiektom różnych typów. Innymi słowy, jest to umiejętność tworzenia abstrakcyjnych metod z określonych typów w celu ujednolicenia tych typów.
- Na przykład funkcja print, musi obsługiwać inaczej drukowanie ciągu znaków a inaczej dla liczb całkowitych, więc będą dwie implementacje funkcji, które prowadzą do drukowania, ale nazwanie ich wspólną nazwą tworzy wygodny abstrakcyjny interfejs niezależny od typu wartości, która ma zostać wydrukowana.
- Ta sama zasada dotyczy operacji dodawania. Wiemy, że dodawanie jest wyrażane za pomocą operatora „+” i możemy go zastosować, gdy dodajemy dwie liczby całkowite lub łączymy dwa ciągi znaków lub dwie listy.

```
a = 10
print(a.__add__(20))            30
b = 'abc'
print(b.__add__('def'))         abcdef
```

# Object-Oriented Programming – Polimorfizm

- Jednym ze sposobów realizacji polimorfizmu jest dziedziczenie, kiedy podklasy wykorzystują metody klasy bazowej lub je nadpisuje.
- większość kodu można ponownie wykorzystać i zaimplementowane są tylko określone metody, co oszczędza dużo czasu programowania i poprawia jakość kodu;
- kod ma przejrzystą strukturę;
- istnieje jednolity sposób wywoływania metod odpowiedzialnych za te same operacje, zaimplementowany odpowiednio dla typów.

**dziedziczenie**: klasa Radio dziedziczy metodę turn_on() ze swojej nadklasy
**polimorfizm**: wszystkie instancje klas pozwalają na wywołanie metody turn_on() i nadpisują jej funkcjonalność

```
class Device:
    def turn_on(self):
        print('The device was turned on')

class Radio(Device):
    pass

class PortableRadio(Device):
    def turn_on(self):
        print('PortableRadio type object was turned on')

class TvSet(Device):
    def turn_on(self):
        print('TvSet type object was turned on')

device = Device()
radio = Radio()
portableRadio = PortableRadio()
tvset = TvSet()

for element in (device, radio, portableRadio, tvset):
    element.turn_on()
```

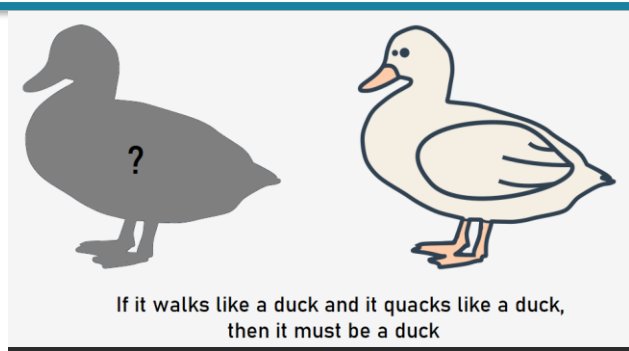
Duck typing to nazwa terminu opisującego zastosowanie testu kaczki: „Jeśli coś chodzi jak kaczka i kwacze jak kaczka, to musi to być kaczka”, która określa, czy obiekt może być użyty do określonego celu. Przydatność obiektu zależy od obecności pewnych atrybutów, a nie od rodzaju samego obiektu.

Duck typing to kolejny sposób na osiągnięcie polimorfizmu i reprezentuje bardziej ogólne podejście niż polimorfizm osiągnięty przez dziedziczenie. Kiedy mówimy o dziedziczeniu, wszystkie podklasy są wyposażone w metody nazwane tak samo, jak metody obecne w nadklasie.

W duck typing wierzymy, że obiekty posiadają wywoływane metody. Jeśli nie są ich właścicielami, powinniśmy być przygotowani na obsługę wyjątków.

Np. ser i wosk maja podobne koncepcyjnie metody, ale reprezentują zupełnie różne rzeczy

- polimorfizm jest używany, gdy różne obiekty klas mają podobne koncepcyjnie metody (ale nie zawsze są dziedziczone)
- polimorfizm wykorzystuje przejrzystość i wyrazistość projektu i rozwoju aplikacji
- gdy zakłada się polimorfizm, mądrze jest zająć się wyjątkami, które mogą się pojawić.




```
class Wax:
    def melt(self):
        print("Wax can be used to form a tool")

class Cheese:
    def melt(self):
        print("Cheese can be eaten")

class Wood:
    def fire(self):
        print("A fire has been started!")

for element in Wax(), Cheese(), Wood():
    try:
        element.melt()
    except AttributeError:
        print('No melt() method')
```



- Hermetyzacja (Encapsulation) jest jednym z podstawowych pojęć w programowaniu obiektowym (między dziedziczeniem, polimorfizmem).
- Opisuje ideę łączenia atrybutów i metod, które działają na tych atrybutach w ramach klasy.
- Hermetyzacja służy do ukrycia atrybutów wewnątrz klasy jak w kapsule, uniemożliwiając dostęp do nich osobom nieupoważnionym.
- W klasie dostępne są publicznie dostępne metody umożliwiające dostęp do wartości, a inne obiekty wywołują te metody w celu pobrania i zmodyfikowania wartości w obiekcie.
- Może to być sposób na wymuszenie pewnej prywatności atrybutów.
- Jeżeli klasa udostępnia te atrybuty to atrybuty te stają się częścią publicznego interfejsu API klasy, będą to atrybuty publiczne.
- Jeżeli do atrybutów będzie dostęp poprzez metody dostępu set i get, a atrybuty będą ukryte dla dostępu z zewnątrz klasy.

# Object-Oriented Programming – Hermetyzacja (kapsułkowanie) atrybutu

```
class TestTemplate(object):
    def __init__(self, tag='', functionality=None, status=False):
        self.__status = status
        self._tag = tag
        self.functionality = functionality

test = TestTemplate(tag="SmokeTest",
functionality="AnyTest", status=True)

print('{:#^15}'.format("Public"))
print(test.functionality)
test.functionality = "ThisTest"
print(test.functionality)

print('{:#^15}'.format("Private"))
print(test._tag)
test._tag = "RegTest"
print(test._tag)

print('{:#^15}'.format("Protected"))
print(test.__status)
print(test._TestTemplate__status)
```

- public bez podkreślenia – jest to atrybut publiczny
- protected _ (pojedyńcze podkreślenie) – ma to charakter informacyjny nie ruszaj chyba że jesteś subclassą
- private __ (podwójne podkreślenie) – dane nie są dostępne poza klassą. Python pozwala dostąć się do tego typu atrybutów poprzez metodę name mangling. _<className>__<memberName>

Metody Getter i Setter:

- Getter: Metoda, która umożliwia dostęp do atrybutu w danej klasie
- Setter: metoda, która pozwala ustawić lub zmienić wartość atrybutu w klasie

W OOP wzorzec pobierający i ustawiający sugeruje, że atrybuty publiczne powinny być używane tylko wtedy, gdy masz pewność, że nikt nigdy nie będzie musiał dołączać do nich zachowania. Jeśli istnieje prawdopodobieństwo, że atrybut zmieni swoją wewnętrzną implementację, należy użyć metod pobierających i ustawiających (getter i setter)

Implementacja wzorca pobierającego (getter) i ustawiającego (seter) wymaga:

- Ustawianie atrybutów jako niepublicznych
- Pisanie metod pobierających i ustawiających dla każdego atrybutu

```
class Label:
    def __init__(self, text, font):
        self.__text = text
        self.__font = font

    def get_text(self):
        return self.__text

    def set_text(self, value):
        self.__text = value

    def get_font(self):
        return self.__font

    def set_font(self, value):
        self.__font = value


arial = Label("Example text", 12)
arial.set_text("New text")
print(arial.get_text())
print(arial.get_font())
```

# Object-Oriented Programming – Hermetyzacja (kapsułkowanie) atrybutu

Python pozwala kontrolować dostęp do atrybutów za pomocą wbudowanej funkcji property() i odpowiedniego dekoratora @property.

- Oznacza metodę, która zostanie wywołana automatycznie, gdy inny obiekt będzie chciał odczytać zawartą w hermetyzacji wartość atrybutu.
- Nazwa wyznaczonej metody zostanie użyta jako nazwa atrybutu instancji odpowiadającego atrybutowi enkapsulacji.
- Należy go zdefiniować przed metodą odpowiedzialną za ustawienie wartości atrybut hermetyzowanego, oraz przed metodą odpowiedzialną za usunięcie atrybutu hermetyzowanego.

```
from datetime import date

class Employee:
    def __init__(self, name, birth_date):
        self.name = name
        self.birth_date = birth_date

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value.upper()

    @property
    def birth_date(self):
        return self._birth_date

    @birth_date.setter
    def birth_date(self, value):
        self._birth_date = date.fromisoformat(value)

emp1 = Employee("Janusz", "1985-02-07")
print(emp1.name)
```

# Object-Oriented Programming – Hermetyzacja (kapsułkowanie) atrybutu

- Obiekt klasy Tank ma atrybut __level, a klasa dostarcza metody odpowiedzialne za obsługę dostępu do tego atrybutu.
- Metoda dekorowana @property to metoda, którą należy wywołać, gdy jakiś inny kod chce odczytać poziom cieczy w naszym zbiorniku. Taką metodę odczytu nazywamy getterem.
- Metoda następująca po dekoratorze nadaje nazwę (level) atrybutowi widocznemu poza klasą. Co więcej, widzimy, że dwie inne metody są nazywane w ten sam sposób, ale ponieważ używamy specjalnie spreparowanych dekoratorów, aby je rozróżnić, nie spowoduje to żadnych problemów:
- @tank.setter() – określa metodę wywoływaną w celu ustawienia wartości atrybutu hermetyzowany;
- @tank.deleter() – oznacza metodę wywoływaną, gdy inny kod chce usunąć enkapsulowany atrybut.

```
class TankError(Exception):
    pass


class Tank:
    def __init__(self, capacity):
        self.capacity = capacity
        self.__level = 0

    @property
    def level(self):
        return self.__level

    @level.setter
    def level(self, amount):
        if amount > 0:
            # fueling
            if amount <= self.capacity:
                self.__level = amount
            else:
                raise TankError('Too much liquid in the tank')
        elif amount < 0:
            raise TankError('Not possible to set negative liquid level')

    @level.deleter
    def level(self):
        if self.__level > 0:
            print('It is good to remember to sanitize the remains from the tank!')
        self.__level = None
```

# Object-Oriented Programming – Hermetyzacja (kapsułkowanie) atrybutu

- Metoda gettera jest ozdobiona „@property”. Określa nazwę atrybutu, który ma być używany przez kod zewnętrzny.
- Metoda ustawiająca jest ozdobiona „@name.setter”. Nazwa metody powinna być nazwą atrybutu.
- Metoda deleter jest ozdobiona „@name.deleter”. Nazwa metody powinna być nazwą atrybutu.

```
# our_tank object has a capacity of 20 units
our_tank = Tank(20)

# our_tank's current liquid level is set to 10 units
our_tank.level = 10
print('Current liquid level:', our_tank.level)

# adding additional 3 units (setting liquid level to 13)
our_tank.level += 3
print('Current liquid level:', our_tank.level)

# let's try to set the current level to 21 units
# this should be rejected as the tank's capacity is 20 units
try:
    our_tank.level = 21
except TankError as e:
    print('Trying to set liquid level to 21 units, result:', e)
```

```
# similar example - let's try to add an additional 15 units
# this should be rejected as the total capacity is 20 units
try:
    our_tank.level += 15
except TankError as e:
    print('Trying to add an additional 15 units, result:', e)

# let's try to set the liquid level to a negative amount
# this should be rejected as it is senseless
try:
    our_tank.level = -3
except TankError as e:
    print('Trying to set liquid level to -3 units, result:', e)

print('Current liquid level:', our_tank.level)

del our_tank.level
```

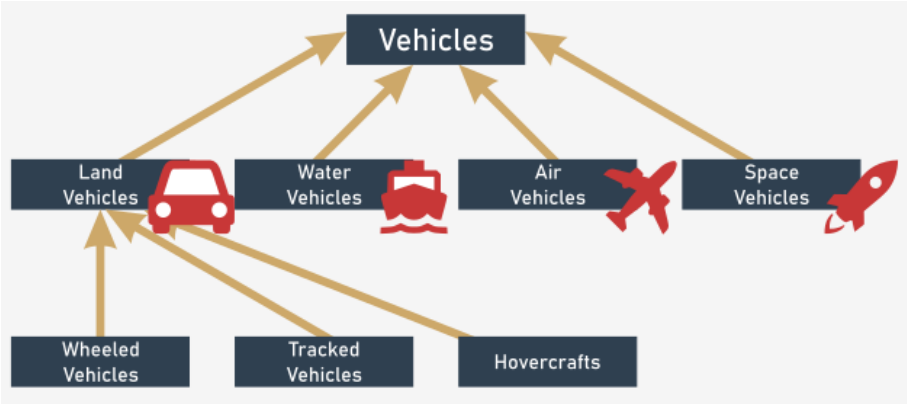
# Object-Oriented Programming – Hermetyzacja (kapsułkowanie) atrybutu

```
class Employee:
    def __init__(self, name, surname):
        self.__name = name
        self.__surname = surname

    @property
    def shortcut(self):
        return (''.join([self.__name[0], self.__name[-1], self.__surname[0],
self.__surname[-1]])).lower()

    @property
    def email(self):
        return self.shortcut + '@company.com'

    @property
    def fullname(self):
        return '{} {}'.format(self.__name, self.__surname)
```

```
    @fullname.setter
    def fullname(self, full_name):
        name, surname = full_name.split(' ')
        self.__name = name
        self.__surname = surname

    @fullname.deleter
    def fullname(self):
        print(f'Delete the name {self.__name} {self.__surname}')
        self.__name = None
        self.__surname = None
e1 = Employee("Marcin", "Caryk")
e1.fullname = "Janusz Bak"

print(e1.fullname)
print(e1.shortcut)
print(e1.email)
```

- Dziedziczenie modeluje ścisłą relację między dwiema klasami: klasą bazową i klasą pochodną, zwaną podklasą.
- Wynikiem tej relacji jest podklasa, która dziedziczy wszystkie metody i wszystkie właściwości klasy bazowej i pozwala podklasie rozszerzyć wszystko, co zostało odziedziczone.
- Rozszerzając klasę podstawową, tworzysz klasę bardziej wyspecjalizowaną. Ponadto mówimy, że klasy te są ściśle powiązane.
- Podstawowym zastosowaniem dziedziczenia jest ponowne wykorzystanie kodu. Jeśli dwie klasy wykonują podobne zadania, możemy stworzyć dla nich wspólną klasę bazową, do której przeniesiemy identyczne metody i właściwości.
- Ułatwi to testowanie i potencjalnie zwiększy niezawodność aplikacji w przypadku zmian. W razie jakichkolwiek problemów łatwiej będzie też znaleźć przyczynę błędu.

- hierarchia rośnie od góry do dołu, jak korzenie drzew, a nie gałęzie. Najbardziej ogólna i najszersza klasa jest zawsze na górze (nadklasa), podczas gdy jej potomkowie znajdują się poniżej (podklasy).
- dziedziczenie wielokrotnemu możesz stworzyć ogromną, złożoną i hierarchiczną strukturę klas
- Zjawisko to znane jest jako problem eksplozji klas i jest jednym z antywzorców programowania



# Object-Oriented Programming – kompozycja a dziedziczenie

- Dziedziczenie nie jest jedynym sposobem konstruowania adaptowalnych obiektów. Podobne cele można osiągnąć za pomocą koncepcji o nazwie kompozycja.
- Ta koncepcja modeluje inny rodzaj relacji między przedmiotami; modeluje to, co ma relację.
- Kompozycja to proces komponowania obiektu przy użyciu innych różnych obiektów.
- Obiekty użyte w kompozycji dostarczają zestawu pożądanych cech (właściwości i/lub metod), więc można powiedzieć, że działają jak klocki do budowy bardziej skomplikowanej konstrukcji.

Ogólnie ujmując:

- dziedziczenie rozszerza możliwości klasy, dodając nowe komponenty i modyfikując istniejące; innymi słowy, kompletny przepis jest zawarty w samej klasie i wszystkich jej przodkach; przedmiot zabiera cały dobytek klasy i czyni z niego użytek;
- kompozycja projektuje klasę jako kontener (nazywany kompozytem) zdolny do przechowywania i używania innych obiektów (pochodzących z innych klas), gdzie każdy z obiektów implementuje część pożądanego zachowania klasy. Warto wspomnieć, że bloki są luźno połączone z kompozytem, a bloki te można wymieniać w dowolnym momencie, nawet w czasie wykonywania programu.

# Object-Oriented Programming – kompozycja a dziedziczenie

- Dziedziczenie nie jest jedynym sposobem konstruowania adaptowalnych obiektów. Podobne cele można osiągnąć za pomocą koncepcji o nazwie kompozycja.
- Ta koncepcja modeluje inny rodzaj relacji między przedmiotami; modeluje to, co ma relację.
- Kompozycja to proces komponowania obiektu przy użyciu innych różnych obiektów.
- Obiekty użyte w kompozycji dostarczają zestawu pożądanych cech (właściwości i/lub metod), więc można powiedzieć, że działają jak klocki do budowy bardziej skomplikowanej konstrukcji.

Ogólnie ujmując:

- dziedziczenie rozszerza możliwości klasy, dodając nowe komponenty i modyfikując istniejące; innymi słowy, kompletny przepis jest zawarty w samej klasie i wszystkich jej przodkach; przedmiot zabiera cały dobytek klasy i czyni z niego użytek;
- kompozycja projektuje klasę jako kontener (nazywany kompozytem) zdolny do przechowywania i używania innych obiektów (pochodzących z innych klas), gdzie każdy z obiektów implementuje część pożądanego zachowania klasy. Warto wspomnieć, że bloki są luźno połączone z kompozytem, a bloki te można wymieniać w dowolnym momencie, nawet w czasie wykonywania programu.

# Object-Oriented Programming – kompozycja a dziedziczenie

```
class Car:
    def __init__(self, engine):
        self.engine = engine


class GasEngine:
    def __init__(self, horse_power):
        self.hp = horse_power

    def start(self):
        print('Starting {}hp gas engine'.format(self.hp))


class DieselEngine:
    def __init__(self, horse_power):
        self.hp = horse_power

    def start(self):
        print('Starting {}hp diesel engine'.format(self.hp))


my_car = Car(GasEngine(4))
my_car.engine.start()
my_car.engine = DieselEngine(2)
my_car.engine.start()
```

# Object-Oriented Programming – kompozycja a dziedziczenie

- Faworyzowanie kompozycji zamiast dziedziczenia jest zasadą projektową, która zapewnia większą elastyczność projektu, ponieważ można wybrać, które obiekty specyficzne dla domeny powinny zostać włączone do ostatecznego obiektu.
- To tak, jakby uzbroić podstawową maszynę w oprzyrządowanie przeznaczone do wykonywania określonego zadania, ale bez budowania szerokiej hierarchicznej struktury klas obejmującej wszystkie możliwe kombinacje sprzętowe.
- W rzeczywistości dzięki podejściu opartemu na kompozycji można łatwiej reagować na zmiany wymagań dotyczące klas, ponieważ nie wymaga ono głębokiego badania zależności, które można zauważyć podczas implementacji kodu z podejściem dziedziczenia.
- Z drugiej strony istnieje wyraźna wada: kompozycja przenosi dodatkowe obowiązki na programistę. Deweloper powinien upewnić się, że wszystkie klasy komponentów, które są używane do budowy kompozytu, implementują metody nazwane w ten sam sposób, aby zapewnić wspólny interfejs.
- W przypadku dziedziczenia, jeśli programista zapomni zaimplementować określoną metodę, zostanie wywołana metoda dziedziczona o tej samej nazwie. Dodatkowo w przypadku dziedziczenia programista musi ponownie zaimplementować tylko określone metody, a nie wszystkie, aby uzyskać wspólny interfejs.

# Object-Oriented Programming – kompozycja a dziedziczenie

- Dziedziczenie i kompozycja nie wykluczają się wzajemnie.;
- Należy traktować zarówno dziedziczenie, jak i kompozycję jako dodatkowe środki rozwiązywania problemów;
- nie ma nic złego w komponowaniu obiektów z… klas, które zostały zbudowane z wykorzystaniem dziedziczenia.
- Trzeba określić każdy przypadek relacji „jest” lub „ma”.
- Jeśli problem można modelować za pomocą relacji „jest a”, wówczas należy zastosować podejście dziedziczenia.
- W przeciwnym razie, jeśli problem można modelować za pomocą relacji „ma”, to wybór jest jasny – rozwiązaniem jest kompozycja.

# Object-Oriented Programming – kompozycja a dziedziczenie

```
class Base_Computer:
    def __init__(self, serial_number):
        self.serial_number = serial_number


class Personal_Computer(Base_Computer):
    def __init__(self, sn, connection):
        super().__init__(sn)
        self.connection = connection
        print('The computer costs $1000')


class Connection:
    def __init__(self, speed):
        self.speed = speed

    def download(self):
        print('Downloading at {}'.format(self.speed))


class DialUp(Connection):
    def __init__(self):
        super().__init__('9600bit/s')

    def download(self):
        print('Dialling the access number ... '.ljust(40), end='')
        super().download()
```

```
class ADSL(Connection):
    def __init__(self):
        super().__init__('2Mbit/s')

    def download(self):
        print('Waking up modem  ... '.ljust(40), end='')
        super().download()


class Ethernet(Connection):
    def __init__(self):
        super().__init__('10Mbit/s')

    def download(self):
        print('Constantly connected... '.ljust(40), end='')
        super().download()

# I started my IT adventure with an old-school dial up connection
my_computer = Personal_Computer('1995', DialUp())
my_computer.connection.download()

# then it came year 1999 with ADSL
my_computer.connection = ADSL()
my_computer.connection.download()

# finally I upgraded to Ethernet
my_computer.connection = Ethernet()
my_computer.connection.download()
```

# 3 Dekoratory

## Deocrators

# Object-Oriented Programming – OPP generalnie

Podstawy oprogramowania zorientowane obiektowo w Pythonie:

- Świat zorientowany obiektowo przedstawia koncepcję obiektów, które mają atrybuty (data members) i procedury (member functions)
- Te funkcje są odpowiedzialne za manipulowanie atrybutami.
- Weźmy na przykład obiekt Samochód. Obiekt Car będzie posiadał atrybuty takie jak poziom paliwa, isSedan, prędkość oraz kierownica i współrzędne, a metody to przyspieszenie(), aby zwiększyć prędkość i takeTurn(), aby samochód skręcił.
- Wszystko w Pythonie jest obiektem.
- Każda instancja lub zmienna klasy ma swój własny adres pamięci lub tożsamość. Obiekty, które są instancjami klas, oddziałują między sobą, aby służyć celom tworzonej aplikacji.

# Object-Oriented Programming – elementy

## Objekt (Object)

- Reprezentują one podmioty w Twojej aplikacji w trakcie opracowywania.
- Jednostki (Entities) współdziałają między sobą w celu rozwiązywania rzeczywistych problemów.
- Na przykład Osoba jest encją (entity) i Samochód jest encją (entity). Osoba prowadzi samochód, aby przemieścić się z jednego miejsca do drugiego.

## Klasa (Class)

- Klasy pomagają programistom w reprezentowaniu rzeczywistych jednostek.
- Klasy definiują obiekty w atrybutach i zrachowaniach. Atrybuty to składowe danych, a zachowania są manifestowane przez funkcje składowe.
- Klasy składają się z konstruktorów, które zapewniają stan początkowy dla tych obiektów.
- Klasy są jak szablony i dlatego można je łatwo ponownie wykorzystać.
- Na przykład klasa Person będzie miała atrybuty name i age oraz funkcję gotoOffice(), która definiuje jego zachowanie podczas podróży do biura do pracy.

## Metoda (Method)

- Reprezentują zachowanie obiektu.
- Metody działają na atrybutach, a także implementują pożądaną funkcjonalność.

```
class Person(object):
    def __init__(self, name, age): #constructor
        self.name = name #data members/ attributes
        self.age = age
    def get_person(self,): # member function
        return "<Person (%s, %s)>" % (self.name, self.age)

p = Person("John", 32) # p is an object of type Person
print("Type of Object:", type(p), "Memory Address:", id(p))
```

```
Type of Object: <class '__main__.Person'> Memory Address: 32311088
```

# Object-Oriented Programming – Główne aspekty programowania obiektowego

## Kapsułkowanie (Encapsulation)

- Zachowanie obiektu jest ukrywane przed światem zewnętrznym lub obiekty zachowują prywatność swoich informacji o stanie.
- Klienci nie mogą zmienić wewnętrznego stanu obiektu, działając bezpośrednio na nich a raczej klienci wpływają na obiekt poprzez żądania, wysyłając komunikaty.
- W zależności od typu żądań, obiekty mogą odpowiadać, zmieniając swój stan wewnętrzny za pomocą specjalnych funkcji składowych, takich jak get i set.
- W Pythonie koncepcja enkapsulacji (ukrywanie danych i metod) nie jest domniemana, ponieważ nie zawiera słów kluczowych, takich jak publiczne, prywatne i które wspierają enkapsulację. Dostępność można ustawić jako prywatną, dodając przedrostek __ w nazwie zmiennej lub funkcji.

## Dziedziczenie (Inheritance)

- Dziedziczenie wskazuje, że jedna klasa wywodzi (w większość) funkcjonalności z klasy nadrzędnej.
- Dziedziczenie jest opisane jako opcja ponownego wykorzystania funkcjonalności zdefiniowanej w klasie bazowej i umożliwienia niezależnych rozszerzeń oryginalnej implementacji oprogramowania.
- Dziedziczenie tworzy hierarchię poprzez relacje między obiektami różnych klas. Python, w przeciwieństwie do Javy, obsługuje dziedziczenie wielokrotne (dziedziczenie z wielu klas bazowych).

## Polimorfizm (Polymorphism)

- Polimorfizm może być dwojakiego rodzaju:
  - Obiekt zapewnia różne implementacje metody na podstawie parametrów wejściowych
  - Ten sam interfejs może być używany przez obiekty różnych typów
- W Pythonie polimorfizm to funkcja wbudowana w język. Na przykład operator + może działać na dwóch liczbach całkowitych, aby je dodać lub może pracować z ciągami, aby je połączyć

```
class A:
    def
a1(self):
    print("a1")
class B(A):
    def b(self):
    print("b")

b = B()
b.a1()
```

## Abstrakcja (Abstraction)

- Zapewnia prosty interfejs dla klientów, w którym klienci mogą wchodzić w interakcje z obiektami klas i wywoływać metody zdefiniowane w interfejsie.
- Abstrahuje złożoność klas wewnętrznych za pomocą interfejsu, dzięki czemu klient nie musi być świadomy wewnętrznych implementacji

```
class Adder:
    def __init__(self):
        self.sum = 0

    def add(self, value):
        self.sum += value

acc = Adder()

for i in range(99):
    acc.add(i)

print(acc.sum)
```

## Kompozycja (Composition)

- Jest to sposób na łączenie obiektów lub klas w bardziej złożone struktury danych lub implementacje oprogramowania
- W kompozycji obiekt jest używany do wywoływania funkcji składowych w innych modułach, udostępniając w ten sposób podstawową funkcjonalność między modułami bez dziedziczenia

```
class A(object):
    def a1(self):
        print("a1")

class B(object):
    def b(self):
        print("b")
        A().a1()

objectB = B()

objectB.b()
```

# Abstract classes – klasy abstrakcyjne

Klasy abstrakcyjne:

- pozwalają ustalić wymagania dla klas w kwestiach interfejsów (metod) udostępnianych przez każdą klasę.
- Klasę abstrakcyjną należy traktować jako wzór dla innych klas, rodzaj kontraktu między projektantem klasy a programistą.
- Projektant klasy określa wymagania dotyczące metod, które muszą zostać zaimplementowane, po prostu je deklarując, ale nie definiując ich szczegółowo. Takie metody nazywane są metodami abstrakcyjnymi.
- Programista musi dostarczyć wszystkie definicje metod, a kompletność zostanie zweryfikowana przez inny, dedykowany moduł. Programista dostarcza definicje metod, zastępując deklaracje metod otrzymane od projektanta klas.

Dlaczego stosujemy klasy abstrakcyjne:

- Kod był polimorficzny, więc wszystkie podklasy muszą dostarczyć zestaw własnych implementacji metod, aby móc je wywoływać przy użyciu wspólnych nazw metod.
- Klasa, która zawiera jedną lub więcej metod abstrakcyjnych, nazywana jest klasą abstrakcyjną.
- Oznacza to, że klasy abstrakcyjne nie ograniczają się do zawierania tylko metod abstrakcyjnych – niektóre metody można już zdefiniować, ale jeśli którakolwiek z metod jest metodą abstrakcyjną, klasa staje się abstrakcyjną.
- Metoda abstrakcyjna to metoda, która ma deklarację, ale nie ma żadnej implementacji. Podamy kilka przykładów takich metod, aby podkreślić ich abstrakcyjny charakter.

# Abstract classes – klasy abstrakcyjne

- Załóżmy, że projektujesz aplikację do odtwarzania muzyki, przeznaczoną do obsługi wielu formatów plików.
- Niektóre formaty są już znane, ale niektóre nie są jeszcze znane.
- Pomysł polega na zaprojektowaniu klasy abstrakcyjnej reprezentującej podstawowy format muzyczny i odpowiednich metod dla „otwierania”, „odtwarzania”, „pobierania szczegółów”, „przewijania” itp., aby zachować polimorfizm.
- Za każdym razem, gdy pojawią się nowe specyfikacje formatu, nie będzie trzeba przerabiać kodu odtwarzacza muzyki, wystarczy, że dostarczysz klasę obsługującą nowy format plików, wypełniając kontrakt narzucony przez klasę abstrakcyjną.
- Pamiętaj, że nie jest możliwe stworzenie instancji klasy abstrakcyjnej i potrzebuje ona podklas, aby zapewnić implementacje metod abstrakcyjnych, które są zadeklarowane w klasach abstrakcyjnych. To zachowanie jest testem wykonywanym przez dedykowany moduł Pythona w celu sprawdzenia, czy programista zaimplementował podklasę, która zastępuje wszystkie metody abstrakcyjne.
- Kiedy projektujemy duże jednostki funkcjonalne, w postaci klas, powinniśmy używać klasy abstrakcyjnej.
- Gdy chcemy zapewnić wspólną zaimplementowaną funkcjonalność dla wszystkich implementacji klasy, możemy również użyć klasy abstrakcyjnej, ponieważ klasy abstrakcyjne częściowo pozwalają nam na implementację klas poprzez dostarczanie konkretnych definicji dla niektórych metod, a nie tylko deklaracji.
- Jest to tzw. wspólny interfejs programowania aplikacji (API) dla zestawu podklas

# Abstract classes – klasy abstrakcyjne

```
import abc

class BluePrint(abc.ABC):
    @abc.abstractmethod
    def hello(self):
        pass

class GreenField(BluePrint):
    def hello(self):
        print('Welcome to Green Field!')


gf = GreenField()
gf.hello()
```

Wielokrotne dziedziczenie - Kiedy planowane jest zaimplementowane dziedziczenie wielokrotne z klas abstrakcyjnych, należy pamiętać, że efektywna podklasa powinna nadpisywać wszystkie metody abstrakcyjne odziedziczone z jej superklas.

- Abstrakcyjna klasa bazowa (ABC) to klasa, której nie można utworzyć.
- Taka klasa jest klasą bazową dla klas konkretnych;
- ABC można odziedziczyć tylko po;
- jesteśmy zmuszeni zastąpić wszystkie metody abstrakcyjne, dostarczając konkretne implementacje metod.

# Abstract classes – klasy abstrakcyjne

```
from abc import ABC, abstractmethod

class Polygon(ABC):
    def __init__(self, sides):
        self.sides = sides

    @abstractmethod
    def noofsides(self):
        '''Returned information how many squares have
        :param sides
        '''
        pass


class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have {} sides".format(self.sides))

class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have {} sides".format(self.sides))
```

```
class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have {} sides".format(self.sides))

class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have {} sides".format(self.sides))

# Driver code
R = Triangle(3)
R.noofsides()

K = Quadrilateral(4)
K.noofsides()

R = Pentagon(5)
R.noofsides()

K = Hexagon(6)
K.noofsides()
```

# Design patterns – SOLID

- W dziedzinie programowania obiektowego i projektowania oprogramowania zasady SOLID to zestaw 5 zasad, które ułatwiają testowanie, utrzymanie i czytelność kodu.
- Korzyści wynikające z przyjęcia przez zespół tych zasad podczas opracowywania kodu obejmują sprawniejsze wdrażanie oprogramowania, zwiększoną skalowalność i możliwość ponownego użycia kodu oraz ulepszone debugowanie.
- Zasady te są podzbiorem zasad określonych przez Roberta C. Martina, znanego jako Wujek Bob, w jego artykule Zasady projektowania i wzorce projektowe.
- SOLID to akronim mnemoniczny, który odnosi się do każdej z zasad za pomocą akronimu w języku angielskim. Te akronimy to:
    - Single Responsibility Principle (SRP) - Zasada pojedynczej odpowiedzialności
    - Open-Closed Principle (OCP) - Zasada otwarte-zamknięte
    - Liskov Substitution Principle (LSP) - Zasada podstawienia Liskowa
    - Interface Segregation Principle (ISP) - Zasada segregacji interfejsów
    - Dependency Inversion Principle (DIP) - Zasada odwrócenia zależności
- Zasady SOLID należą do najbardziej znanych w świecie projektowania oprogramowania i stanowią dobrą podstawę do tworzenia kodu w środowiskach kooperacyjnych, na przykład w dziedzinie inżynierii danych.

# Design patterns – SOLID

## **Zasada pojedynczej odpowiedzialności** (SRP - Single Responsibility Principle)

- Zasada pojedynczej odpowiedzialności - klasa powinna mieć tylko jeden powód do zmiany czyli powinna mieć tylko jedną odpowiedzialność.
- Ta zasada mówi, że kiedy tworzymy klasy, powinny one dobrze odnosić się do danej funkcjonalność.
- Jeśli klasa zajmuje się dwiema funkcjonalnościami, lepiej je rozdzielić. Odnosi się do funkcjonalności jako powodu do zmiany. Na przykład klasa może podlegać zmianom z powodu oczekiwanych od niej różnic w zachowaniu, ale jeśli klasa jest zmieniana z dwóch powodów (w zasadzie zmiany w dwóch funkcjonalnościach), to należy zdecydowanie podzielić klasę.
- Ilekroć następuje zmiana w jednej funkcjonalności, ta konkretna klasa musi się zmienić i nic więcej. Dodatkowo, jeśli klasa ma wiele funkcjonalności, klasy zależne będą musiały przejść zmiany z wielu powodów, czego unika się.

## **Zasada otwarte-zamknięte** (OCP - Open/closed principle)

- Zasada otwórz/zamknij mówi, że klasy lub obiekty i metody powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje.
- W prostym języku oznacza to, że kiedy tworzysz swoją aplikację, upewnij się, że piszesz swoje klasy lub moduły w sposób ogólny, tak aby za każdym razem, gdy czujesz potrzebę rozszerzenia zachowania klasy lub obiektu, nie powinieneś zmienić klasę. Zamiast tego proste rozszerzenie klasy powinno pomóc w zbudowaniu nowego zachowania.
- Istniejące klasy nie ulegają zmianie, a co za tym idzie szanse na regresję są mniejsze
- Pomaga również zachować kompatybilność wsteczną dla poprzedniego kodu

## **Zasada podstawienia Liskov** (LSP - Liskov substitution principle)

- Zasada substytucji mówi, że klasy pochodne muszą być w stanie całkowicie zastąpić klasy bazowe.
- Ta zasada jest dość prosta w tym sensie, że mówi, że kiedy twórcy aplikacji piszą klasy pochodne, powinni rozszerzać klasy bazowe.
- Sugeruje również, że klasa pochodna powinna być jak najbardziej zbliżona do klasy bazowej, tak aby sama klasa pochodna powinna zastąpić klasę bazową bez żadnych zmian w kodzie.

# Design patterns – SOLID

## Zasada segregacji interfejsów (ISP - Interface segregation principle)

- Zgodnie z zasadą segregacji interfejsów, klienci nie powinni być zmuszani do polegania na interfejsach, których nie używają.
- Ta zasada mówi o tym, że programiści dobrze piszą swoje interfejsy. Przypomina programistom/architektom o opracowaniu metod, które odnoszą się do funkcjonalności.
- Jeśli istnieje jakakolwiek metoda, która nie jest powiązana z interfejsem, klasa zależna od interfejsu musi ją niepotrzebnie zaimplementować. Na przykład interfejs Pizza nie powinien mieć metody o nazwie add_chicken(). Klasa Veg Pizza oparta na interfejsie Pizza nie powinna być zmuszana do implementacji tej metody.
- Zmusza programistów do pisania cienkich interfejsów i używania metod, które są specyficzne dla interfejsu.
- Pomaga nie wypełniać interfejsów przez dodawanie niezamierzonych metod.
- https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it

## Zasada odwrócenia zależności (DIP - Dependency inversion principle)

- Zasada odwrócenia kontroli / zależności mówi, że moduły wysokiego poziomu nie powinny być zależne od modułów niskiego poziomu. Obie powinny być zależne od abstrakcji. Szczegóły powinny zależeć od abstrakcji, a nie odwrotnie.
- Ta zasada sugeruje, że dowolne dwa moduły nie powinny być od siebie ściśle zależne. W rzeczywistości moduł podstawowy i moduł zależny powinny być oddzielone warstwą abstrakcji pomiędzy nimi.
- Ta zasada sugeruje również, że szczegóły twojej klasy powinny reprezentować abstrakcje. W niektórych przypadkach filozofia ulega odwróceniu, a szczegóły implementacyjne same decydują o abstrakcji, której należy unikać.
- Ścisłe połączenie modułów nie jest już powszechne, a zatem nie ma złożoności/sztywności w systemie.
- Ponieważ istnieje wyraźna warstwa abstrakcji między zależnymi modułami (dostarczona przez hak lub parametr), łatwo jest lepiej radzić sobie z zależnościami między modułami.

# Design patterns – SOLID - SRP

```
class Journal:
    def __init__(self):
        self.entries = []
        self.count = 0

    def add_entry(self, text):
        self.entries.append(f"{self.count}: {text}")
        self.count += 1

    def remove_entry(self, pos):
        del self.entries[pos]

    def __str__(self):
        return "\n".join(self.entries)

    #SRP
    def save(self, filename):
        file = open(filename, "w")
        file.write(str(self))
        file.close()
```

```
class Journal:
    def __init__(self):
        self.entries = []
        self.count = 0

    def add_entry(self, text):
        self.entries.append(f"{self.count}: {text}")
        self.count += 1

    def remove_entry(self, pos):
        del self.entries[pos]

    def __str__(self):
        return "\n".join(self.entries)


class PersistenceManager:
    @staticmethod
    def save_to_file(journal, filename):
        file = open(filename, "w")
        file.write(str(journal))
        file.close()
```

```
j = Journal()
j.add_entry("Dzien dobry dzis mamy 13.12.2022")
j.add_entry("Teraz omawiamy regule SRP")
print(f"Journal entries:\n{j}\n")

file = r'journal.txt'
PersistenceManager.save_to_file(j, file)


with open(file) as fh:
    print(fh.read())
```

# Design patterns – SOLID - OCP

```
class Animal:
    def __init__(self, name: str):
        self.name = name


    def get_name(self) -> str:
        pass


animals = [
    Animal('lion'),
    Animal('mouse')
]

def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')

        elif animal.name == 'mouse':
            print('squeak')


animal_sound(animals)
```

```
animals = [
    Animal('lion'),
    Animal('mouse'),
    Animal('snake')
]

def animal_sound(animals: list):
    for animal in animals:
        if animal.name == 'lion':
            print('roar')
        elif animal.name == 'mouse':
            print('squeak')
        elif animal.name == 'snake':
            print('hiss')

animal_sound(animals)
```

```
class Animal:
    def __init__(self, name: str):
        self.name = name

    def get_name(self) -> str:
        pass

    def make_sound(self):
        pass


class Lion(Animal):
    def make_sound(self):
        return 'roar'


class Mouse(Animal):
    def make_sound(self):
        return 'squeak'


class Snake(Animal):
    def make_sound(self):
        return 'hiss'
```

```
def animal_sound(animals: list):
    for animal in animals:
        print(animal.make_sound())

animals = [
    Lion('lion'),
    Mouse('mouse'),
    Snake('snake')
]
animal_sound(animals)
```

# Design patterns – SOLID - LSP

```
class Rectangle:
    def __init__(self, width, height):
        self._height = height
        self._width = width

    @property
    def area(self):
        return self._width * self._height

    def __str__(self):
        return f'Width: {self.width}, height: {self.height}'

    @property
    def width(self):
        return self._width

    @width.setter
    def width(self, value):
        self._width = value

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        self._height = value
```

```
class Square(Rectangle):
    def __init__(self, size):
        Rectangle.__init__(self, size, size)

    @Rectangle.width.setter
    def width(self, value):
        _width = _height = value

    @Rectangle.height.setter
    def height(self, value):
        _width = _height = value


rc = Rectangle(2, 3)
print(f'Area: {rc.area}')

sq = Square(5)
print(f'Area: {sq.area}')
```

# Design patterns – SOLID - ISP

```
class Machine:
    def print(self, document):
        raise NotImplementedError()

    def fax(self, document):
        raise NotImplementedError()

    def scan(self, document):
        raise NotImplementedError()


class MultiFunctionPrinter(Machine):
    def print(self, document):
        pass

    def fax(self, document):
        pass

    def scan(self, document):
        pass
```

```
class OldFashionedPrinter(Machine):
    def print(self, document):
        # ok - print stuff
        pass

    def fax(self, document):
        pass  # do-nothing

    def scan(self, document):
        """Not supported!"""
        raise NotImplementedError('Printer cannot scan!')
```

```
printer = OldFashionedPrinter()
printer.fax(123)  # nothing happens
printer.scan(123)  # oops!
```

# Design patterns – SOLID - ISP

```
class Printer:
    @abstractmethod
    def print(self, document): pass


class Scanner:
    @abstractmethod
    def scan(self, document): pass


# same for Fax, etc.

class MyPrinter(Printer):
    def print(self, document):
        print(document)


class Photocopier(Printer, Scanner):
    def print(self, document):
        print(document)

    def scan(self, document):
        pass  # something meaningful
```

```
class MultiFunctionDevice(Printer, Scanner):  # , Fax,
etc
    @abstractmethod
    def print(self, document):
        pass

    @abstractmethod
    def scan(self, document):
        pass


class MultiFunctionMachine(MultiFunctionDevice):
    def __init__(self, printer, scanner):
        self.printer = printer
        self.scanner = scanner

    def print(self, document):
        self.printer.print(document)

    def scan(self, document):
        self.scanner.scan(document)
```

# Design patterns – SOLID - DIP

```python
from abc import abstractmethod
from enum import Enum


class Relationship(Enum):
    PARENT = 0
    CHILD = 1
    SIBLING = 2


class Person:
    def __init__(self, name):
        self.name = name


class RelationshipBrowser:
    @abstractmethod
    def find_all_children_of(self, name): pass


class Relationships(RelationshipBrowser):  # low-level
    relations = []

    def add_parent_and_child(self, parent, child):
        self.relations.append((parent, Relationship.PARENT, child))
        self.relations.append((child, Relationship.PARENT, parent))
```

```python
class Research:
    #dependency on a low-level module directly
    #bad because strongly dependent on e.g. storage type

    def __init__(self, relationships):
        relations = relationships.relations
        for r in relations:
            if r[0].name == 'John' and r[1] == Relationship.PARENT:
                print(f'John has a child called {r[2].name}.')


parent = Person('John')
child1 = Person('Chris')
child2 = Person('Matt')

# low-level module
relationships = Relationships()
relationships.add_parent_and_child(parent, child1)
relationships.add_parent_and_child(parent, child2)

Research(relationships)
```

# Design patterns – SOLID - DIP

```
from abc import abstractmethod
from enum import Enum


class Relationship(Enum):
    PARENT = 0
    CHILD = 1
    SIBLING = 2


class Person:
    def __init__(self, name):
        self.name = name


class RelationshipBrowser:
    @abstractmethod
    def find_all_children_of(self, name): pass
```

```
class Relationships(RelationshipBrowser):  # low-level
    relations = []

    def add_parent_and_child(self, parent, child):
        self.relations.append((parent, Relationship.PARENT, child))
        self.relations.append((child, Relationship.PARENT, parent))

    def find_all_children_of(self, name):
        for r in self.relations:
            if r[0].name == name and r[1] == Relationship.PARENT:
                yield r[2].name


class Research:

    def __init__(self, browser):
        for p in browser.find_all_children_of("John"):
            print(f'John has a child called {p}')
```

```
parent = Person('John')
child1 = Person('Chris')
child2 = Person('Matt')

# low-level module
relationships = Relationships()
relationships.add_parent_and_child(parent, child1)
relationships.add_parent_and_child(parent, child2)

Research(relationships)
```

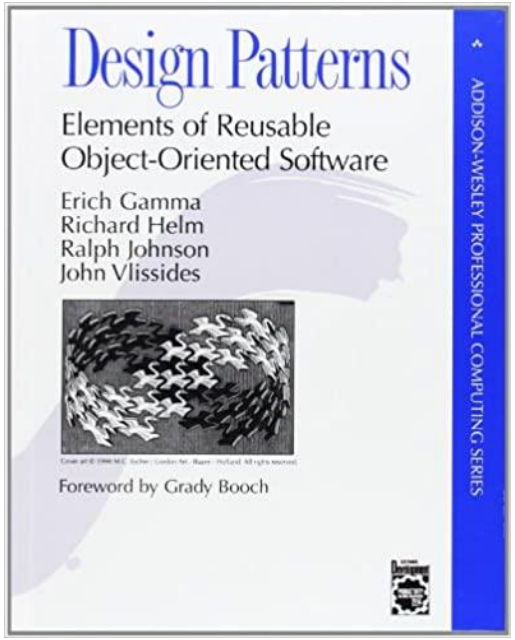
# Zadanie 1

Uwzględniając SOLID i podstawowe informacje o OOP:

1. Stwórz klasę danych Student zawierającą atrybuty takie jak: Imię, Nazwisko, Rok, Grupa, Rozpoczęcie Nauki, Zakończenie Nauki, Instancje klasy Lista przedmiotów oraz atrybuty dostępu do tych wartości
2. Stwórz klasę danych Przedmiot który będzie przechowywała atrybuty takie jak Rok, forma zaliczenia (ocena, zaliczenie), moduły (W, LAB, PROJ) oraz atrybuty dostępu do nich
3. Stwórz klasę Lista Przedmiotów która będzie zawierał listę klas Przedmiot
4. Stwórz klasę Grupa która będzie przechowywała listę studentów oraz będzie dziedziczyła metody po Lista przedmiotów metody takie jak dodaj studenta, usuń studenta, ustaw rok itp.
5. Dodaj metodę wyświetlania listy studentów
6. Dodaj metodę wyświetlania przedmiotów i ocen studenta

# Design patterns – informacje ogólne

- Wzorce projektowe zostały po raz pierwszy wprowadzone przez GoF (Gang of Four), gdzie wspomniał o nich jako o rozwiązaniach określonych problemów.
- Więcej można się dowiedzieć o GoF odsyła do czterech autorów książki Design Patterns: Elements of Reusable Object-Oriented Software. Autorami książki są Erich Gamma, Richard Helm, Ralph Johnson i John Vlissides, ze wstępem Grady'ego Boocha.
- Ta książka zawiera rozwiązania z zakresu inżynierii oprogramowania dla powszechnie występujących problemów w projektowaniu oprogramowania.
- Jako pierwsze zidentyfikowano 23 wzorce projektowe, a pierwsza implementacja została wykonana w odniesieniu do języka programowania Java.
- Wzorce projektowe są odkryciami, a nie wynalazkiem samym w sobie.



# Design patterns – Koncepcja wzorców projektowych

**Kluczowe cechy wzorców projektowych są następujące:**

- Są neutralne językowo i mogą być wdrażane w całym obszarze wiele języków.
- Są dynamiczne, bo co jakiś czas pojawiają się nowe wzorce.
- Są otwarte na dostosowywanie, a zatem przydatne dla programistów.
- Zasadniczo wzorzec projektowy polega na uczeniu się na sukcesach innych, a nie na własnych porażkach.
- Wzorce projektowe to rozwiązania znanych problemów. Dzięki temu mogą być bardzo często wykorzystywane w analizie lub projektowaniu i zgodnie z oczekiwaniami w fazie rozwoju ze względu na bezpośrednią relację w kodzie aplikacji.

**Zalety wzorców projektowych:**

- Można ich używać wielokrotnie w wielu projektach.
- Daje możliwość rozwiązać poziom problemów architektury.
- Są zoptymalizowane pod względem czasu i sprawdzone, co wynika z doświadczenia deweloperów i architektów.
- Mają niezawodność i zależność

# Design patterns – Koncepcja wzorców projektowych

**Taksonomia wzorców projektowych:**

- **Skrawek (Snippet)**: To jest kod w jakimś języku do określonego celu, na przykład łączność, DB w Pythonie może być fragmentem kodu
- **Projekt (Project)**: Rozwiązuje ten konkretny problem.
- **Standard**: jest to sposób na rozwiązanie pewnego rodzaju problemów i może być bardzo ogólny i mieć zastosowanie do danej sytuacji.
- **Wzorzec (Pattern)**: jest to sprawdzone, wydajne i skalowalne rozwiązanie, które rozwiąże całą klasę znanych problemów.

**Kontekst – stosowalność wzorców projektowych:**

- **Uczestnicy (Participants)**: Są to klasy, które są wykorzystywane we wzorcach projektowych. Klasy odgrywają różne role, aby osiągnąć wiele celów we wzorcu.
- **Wymagania niefunkcjonalne (Non-functional requirements)**: wymagania, takie jak optymalizacja pamięci, użyteczność i wydajność, należą do tej kategorii. Czynniki te wpływają na kompletne rozwiązanie programowe i dlatego są krytyczne.
- **Kompromisy**: Nie wszystkie wzorce projektowe pasują do rozwoju aplikacji w takim stanie, w jakim jest, i konieczne są kompromisy. Są to decyzje, które podejmujesz podczas używania wzorca projektowego w aplikacji.
- **Wyniki**: Wzorce projektowe mogą mieć negatywny wpływ na inne części kodu, jeśli kontekst nie jest odpowiedni. Deweloperzy powinni rozumieć konsekwencje i wykorzystanie wzorców projektowych.

# Design patterns – Klasyfikacja wzorców projektowych

## Wzorce twórcze (Creational patterns):

- Działają na podstawie tego, jak można tworzyć obiekty. Wszystkie te wzorce projektowe dotyczą tworzenia instancji klasy.
- Izolują szczegóły tworzenia obiektów.
- Kod jest niezależny od typu tworzonego obiektu.
- Wzorzec ten można dalej podzielić na wzorce tworzenia klas i wzorce tworzenia obiektów.
- Podczas gdy wzorce tworzenia klas skutecznie wykorzystują dziedziczenie w procesie tworzenia instancji, wzorce tworzenia obiektów skutecznie wykorzystują delegację, aby wykonać zadanie.

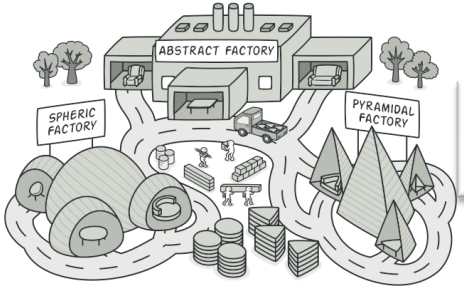
## Wzory strukturalne (Structural patterns):

- Projektują strukturę obiektów i klas tak, aby mogły komponować w celu osiągnięcia większych rezultatów.
- Nacisk kładziony jest na uproszczenie struktury i zidentyfikowanie relacji między klasami a obiektami.
- wykorzystują dziedziczenie do tworzenia interfejsów
- Wyjaśniają, jak łączyć obiekty i klasy w większe struktury (kompozycje), zachowując przy tym elastyczność i wydajność tych struktur.

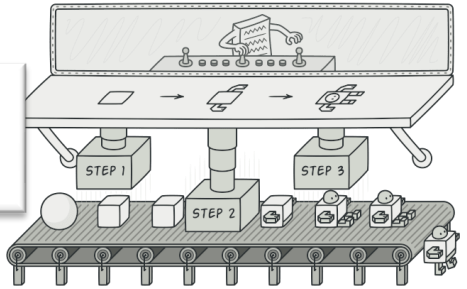
## Wzorce zachowań (Behavioral patterns):

- Zajmują się interakcją między obiektami i odpowiedzialnością obiektów, dotyczą komunikacji obiektów klasy
- Obiekty powinny być w stanie wchodzić w interakcje i nadal być luźno połączone.
- Wzorce behawioralne to te wzorce, które najbardziej dotyczą komunikacji między obiektami.

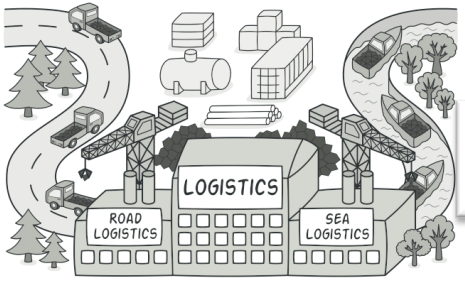
# Design patterns – Klasyfikacja wzorców projektowych - Creational patterns



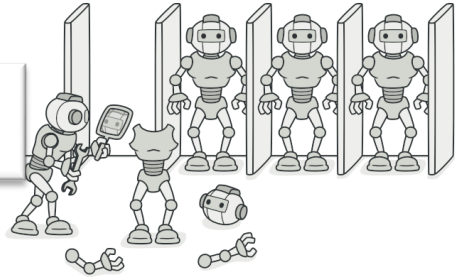

Abstrakcyjna fabryka (Abstract Factory) – umożliwia tworzenie rodzin powiązanych obiektów bez określania ich konkretnych klas, tworzy instancje kilku rodzin klas



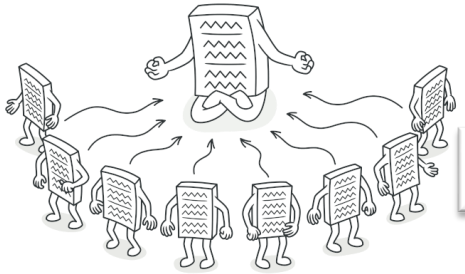

Budowniczy (Builder) - Oddziela konstrukcję obiektu od jego reprezentacji. Pozwala krok po kroku konstruować złożone obiekty. Wzorzec umożliwia tworzenie różnych typów i reprezentacji obiektu przy użyciu tego samego kodu konstrukcyjnego



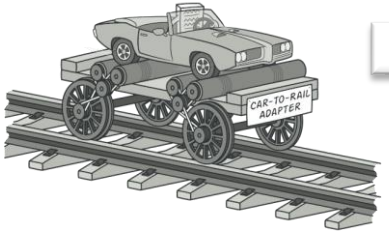

Fabryka (Factory Method) – tworzy instancje kilku klas pochodnych, zapewnia interfejs do tworzenia obiektów w nadklasie, ale pozwala podklasom zmieniać typ tworzonych obiektów



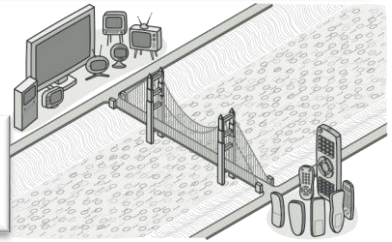
Prototyp (Prototype) - W pełni zainicjowana instancja do skopiowania lub sklonowania. Umożliwia kopiowanie istniejących obiektów bez uzależnienia kodu od ich klas



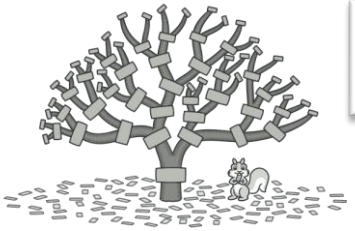
Singleton - Pozwala upewnić się, że klasa ma tylko jedną instancję, zapewniając jednocześnie globalny punkt dostępu do tej instancji.



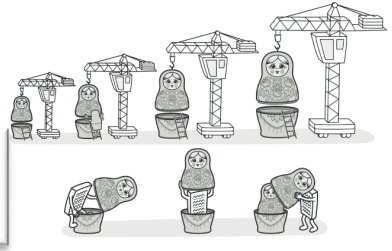

Adapter – umożliwia współpracę obiektów z niekompatybilnymi interfejsami



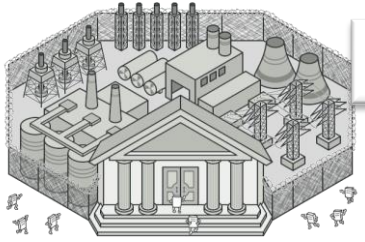
Most (Bridge) – umożliwia podzielenie dużej klasy lub zestawu ściśle powiązanych klas na dwie osobne hierarchie — abstrakcję i implementację — które można rozwijać niezależnie od siebie.



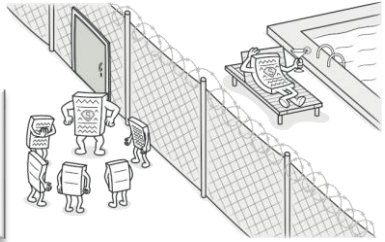
Kompozyt (Composite) – umożliwia komponowanie obiektów w struktury drzewa, a następnie pracę z tymi strukturami tak, jakby były pojedynczymi obiektami



Dekorator (Decorator) – umożliwia dołączanie nowych zachowań do obiektów poprzez umieszczanie tych obiektów w specjalnych obiektach opakowujących zawierających te zachowania

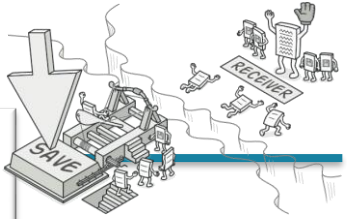


Fasada (Facade) – zapewnia uproszczony interfejs do biblioteki, frameworka lub dowolnego innego złożonego zestawu klas.



Proxy – pozwala zapewnić substytut lub symbol zastępczy dla innego obiektu. Serwer proxy kontroluje dostęp do oryginalnego obiektu, umożliwiając wykonanie czynności przed lub po dotarciu żądania do oryginalnego obiektu.

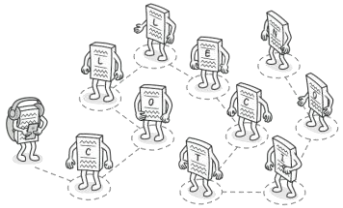
# Design patterns – Klasyfikacja wzorców projektowych - Structural patterns



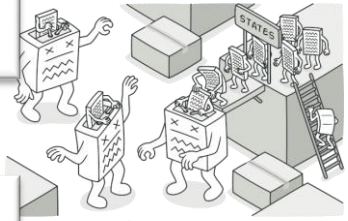


Polecanie (Command) – zamienia żądanie w samodzielny obiekt zawierający wszystkie informacje o żądaniu. Ta transformacja umożliwia przekazywanie żądań jako argumentów metody, opóźnianie lub umieszczanie w kolejce wykonania żądania oraz obsługę operacji, które można cofnąć.

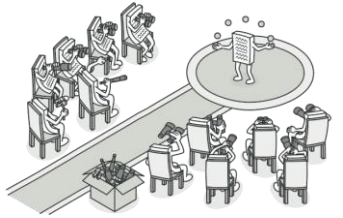
Pamiątka (Memento) – pozwala zapisywać i przywracać poprzedni stan obiektu bez ujawniania szczegółów jego implementacji.



Łańcuch odpowiedzialności (Chain of Responsibility) – umożliwia przekazywanie żądań wzdłuż łańcucha procedur obsługi. Po otrzymaniu żądania każdy program obsługi decyduje, czy przetworzyć żądanie, czy przekazać je następnemu programowi obsługi w łańcuchu.

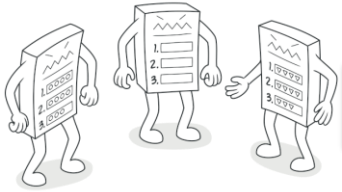



Iterator – możliwia przeglądanie elementów kolekcji bez ujawniania ich podstawowej reprezentacji (listy, stosu, drzewa itp.)



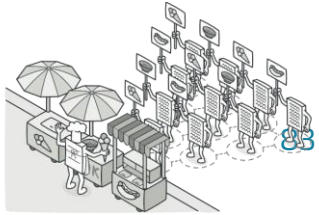
Stan (State) – pozwala obiektowi zmieniać swoje zachowanie, gdy zmienia się jego stan wewnętrzny. Wygląda to tak, jakby obiekt zmienił swoją klasę.

Obserwator (Observer) – umożliwia zdefiniowanie mechanizmu subskrypcji w celu powiadamiania wielu obiektów o wszelkich zdarzeniach, które przytrafiają się obiektowi, który obserwują.



Strategia (Strategy) – pozwala zdefiniować rodzinę algorytmów, umieścić każdy z nich w osobnej klasie i sprawić, by ich obiekty były wymienne.

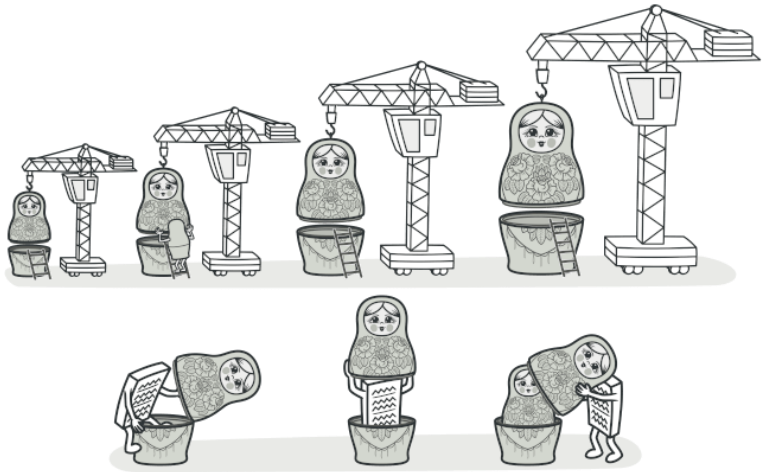
Metoda szablonowa (Template Method) – definiuje szkielet algorytmu w nadklasie, ale pozwala podklasom zastąpić określone kroki algorytmu bez zmiany jego struktury.



Gość (Visitor) – pozwala oddzielić algorytmy od obiektów, na których działają.

# Decorators – podstawy

- Dekorator jest jednym ze wzorców projektowych opisujących strukturę powiązanych obiektów. Python może dekorować funkcje, metody i klasy.
- Działanie dekoratora polega na opakowaniu pierwotnej funkcji nową funkcją (lub klasą) „dekorującą”, stąd nazwa „dekoracja”.
- Odbywa się to poprzez przekazanie oryginalnej funkcji (tj. dekorowanej funkcji) jako parametru funkcji dekorującej, tak aby funkcja dekorująca mogła wywołać przekazaną funkcję. Funkcja dekorująca zwraca funkcję, którą można wywołać później.
- Funkcja dekoracyjna robi więcej, ponieważ może przyjmować parametry funkcji dekorowanej i wykonywać dodatkowe akcje, które czynią z niej prawdziwą funkcję dekoracyjną.
- Dekoratory służą do wykonywania operacji przed i po wywołaniu opakowanego obiektu lub nawet w celu uniemożliwienia jego wykonania, w zależności od okoliczności. Dzięki temu możemy zmienić działanie spakowanego obiektu bez jego bezpośredniej modyfikacji.
- Dekoratory stosowane są w:
    - walidacja argumentów;
    - modyfikacja argumentów;
    - modyfikacja zwracanych przedmiotów;
    - pomiar czasu wykonania;
    - logowanie wiadomości;
    - synchronizacja wątków;
    - refaktoryzacja kodu;
    - buforowanie.



# Decorators – dekorator funkcji

```
def simple_hello():
    print("Hello from simple function!")

def simple_decorator(function):
    print('We are about to call "{}"'.format(function.__name__))
    return function

decorated = simple_decorator(simple_hello)
decorated()
```

```
We are about to call "simple_hello"
Hello from simple function!
```

```
def t_decorator(func):
    def wrapper():
        print("Test: Precondition action")
        func()
        print("Test: Postcondition action")
    return wrapper

def run_test_steps():
    print("Runnin the test steps {}".format([1,2,3]))

run_test_steps = t_decorator(run_test_steps)

run_test_steps()
print(run_test_steps)
```

```
Test: Precondition action
Runnin the test steps [1, 2, 3]
Test: Postcondition action
<function t_decorator.<locals>.wrapper at 0x00000194EADB4700>
```

```
def simple_decorator(function):
    print('We are about to call "{}"'.format(function.__name__))
    return function


@simple_decorator
def simple_hello():
    print("Hello from simple function!")

simple_hello()
```

```
We are about to call "simple_hello"
Hello from simple function!
```

```
def t_decorator(func):
    def wrapper():
        print("Test: Precondition action")
        func()
        print("Test: Postcondition action")
    return wrapper

@t_decorator
def run_test_steps():
    print("Runnin the test steps {}".format([1,2,3,4]))

run_test_steps()
print(run_test_steps)
```

# Decorators – uniwersalność parametrów

- Dekoratory, które powinny być uniwersalne, muszą obsługiwać dowolną funkcję, niezależnie od liczby i rodzaju przekazywanych argumentów.
- W takiej sytuacji możemy posłużyć się pojęciami *args i **kwargs. Możemy również zastosować technikę domknięcia, aby utrwalić argumenty.
- Argumenty przekazane do dekorowanej funkcji są dostępne dla dekoratora, więc dekorator może je wyprintować. To jest prosty przykład, ponieważ argumenty zostały właśnie wyprinotowane, ale nie zostały dalej przetworzone.
- Zagnieżdżona funkcja (internal_wrapper) może odwoływać się do obiektu (own_function) w swoim obejmującym zakresie dzięki domknięciu.

```
def simple_decorator(own_function):

    def internal_wrapper(*args, **kwargs):
        print('"{}" was called with the following
arguments'.format(own_function.__name__))
        print('\t{}\n\t{}\n'.format(args, kwargs))
        own_function(*args, **kwargs)
        print('Decorator is still operating')

    return internal_wrapper


@simple_decorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:", args,
kwargs)

combiner('a', 'b', exec='yes')
```

```
"combiner" was called with the following arguments
                ('a', 'b')
                {'exec': 'yes'}

                Hello from the decorated function; received arguments: ('a', 'b') {'exec': 'yes'}
Decorator is still operating
```

# Decorators – akceptacja własnych atrybutów

- Stwórzmy program, w którym dekorator będzie bardziej ogólny – pozwolimy na przekazanie w argumencie materiału opakowaniowego.
- Utworzona w ten sposób funkcja storage_decorator() stała się dużo bardziej elastyczna i uniwersalna niż 'simple_decorator', ponieważ może obsługiwać różne materiały.
- Zauważ, że nasz dekorator został wzbogacony o jeszcze jedną funkcję, dzięki której jest w stanie obsłużyć argumenty na wszystkich poziomach wywołania.
- Funkcja pack_books zostanie wykonana w następujący sposób:
  - funkcja storage_decorator('kraft') zwróci funkcję opakowującą;
  - zwrócona funkcja opakowująca przyjmie jako argument funkcję, którą ma ozdobić;
  - funkcja wrapper zwróci funkcję internal_wrapper, która dodaje nową funkcjonalność (wyświetlanie materiału) i uruchamia dekorowaną funkcję.

```
def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(our_function.__name__, material))
            our_function(*args)
            print()
        return internal_wrapper
    return wrapper


@warehouse_decorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)


@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)


@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

# Decorators – stack dekoratorów

```
def big_container(collective_material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('<strong>*</strong> The whole order would be packed with',
collective_material)
            print()
        return internal_wrapper
    return wrapper

def warehouse_decorator(material):
    def wrapper(our_function):
        def internal_wrapper(*args):
            our_function(*args)
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(our_function.__name__, material))
        return internal_wrapper
    return wrapper
```

```
@big_container('plain cardboard')
@warehouse_decorator('bubble foil')
def pack_books(*args):
    print("We'll pack books:", args)

@big_container('colourful cardboard')
@warehouse_decorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)

@big_container('strong cardboard')
@warehouse_decorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

```
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')
<strong>*</strong> Wrapping items from pack_books with bubble foil
<strong>*</strong> The whole order would be packed with plain cardboard

We'll pack toys: ('doll', 'car')
<strong>*</strong> Wrapping items from pack_toys with foil
<strong>*</strong> The whole order would be packed with colourful cardboard

We'll pack fruits: ('plum', 'pear')
<strong>*</strong> Wrapping items from pack_fruits with cardboard
<strong>*</strong> The whole order would be packed with strong cardboard
```

# Decorators – stack dekoratorów

- Python pozwala zastosować wiele dekoratorów do wywoływalnego obiektu (funkcji, metody lub klasy).
- Najważniejszą rzeczą do zapamiętania jest kolejność, w jakiej dekoratory są wymienione w twoim kodzie, ponieważ określa kolejność wykonywanych dekoratorów. Kiedy twoja funkcja jest ozdobiona wieloma dekoratorami.

```
@outer_decorator
@inner_decorator
def function():
pass
abcd = subject_matter_function()
```

- sekwencja wywołań będzie wyglądać następująco:
- zewnętrzny_dekorator jest wywoływany w celu wywołania wewnętrznego_dekoratora, a następnie wewnętrzny_dekorator wywołuje twoją funkcję;
- kiedy twoja funkcja kończy swoje działanie, inner_decorator przejmuje kontrolę, a po zakończeniu wykonywania, zewnętrzny_dekorator jest w stanie zakończyć swoje zadanie.
- Ten routing naśladuje klasyczną koncepcję stosu. Przedstawiony lukier syntaktyczny jest odpowiednikiem następujących zagnieżdżonych wywołań:

```
subject_matter_function = outer_decorator(inner_decorator(subject_matter_function()))
abcd = subject_matter_function()
```

# Decorators – dekorowanie funkcji klasami

- W Pythonie może to być klasa pełniąca funkcję dekoratora.
- Możemy zdefiniować dekorator jako klasę iw tym celu musimy użyć specjalnej metody klasowej __call__ .
- Kiedy użytkownik musi utworzyć obiekt, który działa jak funkcja (tj. jest wywoływalny), wówczas dekorator funkcji musi zwrócić obiekt, który można wywoływać, więc specjalna metoda __call__ będzie bardzo przydatna.
- metoda __init__ przypisuje udekorowaną funkcję do self.attribute do późniejszego wykorzystania;
- metoda __call__, która jest odpowiedzialna za obsługę przypadku wywołania obiektu, wywołuje wcześniej przywołaną funkcję.
- klasy dają całą pomocniczość, jaką mogą zaoferować, taką jak dziedziczenie i możliwość tworzenia dedykowanych metod wspomagających.

```
class SimpleDecorator:
    def __init__(self, own_function):
        self.func = own_function

    def __call__(self, *args, **kwargs):
        print('"{}" was called with the following
arguments'.format(self.func.__name__))
        print('\t{}\n\t{}\n'.format(args, kwargs))
        self.func(*args, **kwargs)
        print('Decorator is still operating')


@SimpleDecorator
def combiner(*args, **kwargs):
    print("\tHello from the decorated function; received arguments:",
args, kwargs)


combiner('a', 'b', exec='yes')
```

# Decorators – dekorowanie argumentami

- Gdy przekazujesz argumenty dekoratorowi, mechanizm dekoratora zachowuje się zupełnie inaczej niż w przykładzie dekoratora, który nie przyjmuje argumentów
- odwołanie do dekorowanej funkcji przekazywane jest do metody __call__, która jest wywoływana tylko raz podczas procesu dekorowania, argumenty dekoratora są przekazywane do metody __init__

```
class WarehouseDecorator:
    def __init__(self, material):
        self.material = material

    def __call__(self, own_function):
        def internal_wrapper(*args, **kwargs):
            print('<strong>*</strong> Wrapping items from {} with
{}'.format(own_function.__name__, self.material))
            own_function(*args, **kwargs)
            print()
        return internal_wrapper


@WarehouseDecorator('kraft')
def pack_books(*args):
    print("We'll pack books:", args)


@WarehouseDecorator('foil')
def pack_toys(*args):
    print("We'll pack toys:", args)


@WarehouseDecorator('cardboard')
def pack_fruits(*args):
    print("We'll pack fruits:", args)


pack_books('Alice in Wonderland', 'Winnie the Pooh')
pack_toys('doll', 'car')
pack_fruits('plum', 'pear')
```

* Wrapping items from pack_books with kraft
We'll pack books: ('Alice in Wonderland', 'Winnie the Pooh')

* Wrapping items from pack_toys with foil
We'll pack toys: ('doll', 'car')

* Wrapping items from pack_fruits with cardboard
We'll pack fruits: ('plum', 'pear')

# Decorators – dekoratory klasy

- Dekoratory klas silnie odwołują się do dekoratorów funkcji, ponieważ używają tej samej składni i implementują te same koncepcje.
- Zamiast wrapowania poszczególnych metod dekoratorami funkcji, dekoratory klas umożliwiają zarządzanie klasami lub wrapowanie wywołań metod specjalnych w dodatkową logikę, która zarządza tworzonymi instancjami lub je rozszerza.
- Jeśli weźmiemy pod uwagę składnię, dekoratory klas pojawiają się tuż przed instrukcjami „klas”, które rozpoczynają definicję klasy (podobnie jak dekoratory funkcji, pojawiają się tuż przed definicjami funkcji).
- Najprostsze użycie można przedstawić następująco:

```
@my_decorator
class MyClass:

obj = MyClass()
```

- i jest odpowiedni dla następującego fragmentu:

```
def my_decorator(A):
...
class MyClass:
...
MyClass = my_decorator(MyClass())
obj = MyClass()
```

- Podobnie jak dekoratory funkcji, nowa (udekorowana) klasa jest dostępna pod nazwą „MyClass” i służy do tworzenia instancji.
- Oryginalna klasa o nazwie „MyClass” nie jest już dostępna w Twojej przestrzeni nazw.
- Wywoływalny obiekt zwrócony przez dekorator klasy tworzy i zwraca nową instancję oryginalnej klasy, w pewien sposób rozszerzoną.

# Decorators – dekoratory klasy

Teraz stwórzmy funkcję, która ozdobi klasę metodą wysyłającą alerty za każdym razem, gdy odczytany zostanie atrybut „mileage”.

- linia 1: def object_counter(class_): – ta linia definiuje funkcję dekorującą, która przyjmuje jeden parametr „class_” (zwróć uwagę na podkreślenie)
- linia 2: class_.__getattr__orig = class_.__getattribute__ – dekorator tworzy kopię odwołania do metody specjalnej __getattribute__. Ta metoda odpowiada za zwracanie wartości atrybutów. Odniesienie do tej oryginalnej metody będzie użyte w zmodyfikowanej metodzie
- linia 4: def new_getattr(self, name): – tutaj zaczyna się definicja metody pełniącej rolę nowej metody __getattribute__. Ta metoda akceptuje nazwę atrybutu – jest to string
- linia 5: if name == 'mileage': – w przypadku, gdy jakiś kod poprosi o atrybut 'mileage', zostanie wykonana następna linia
- linia 6: print('Zauważyliśmy, że odczytano atrybut mileage') – generowany jest prosty alert
- linia 7: return class_.__getattr__orig(self, name) – wywoływana jest oryginalna metoda __getattribute__, do której odwołuje się class.__getattr__orig. To kończy definicję funkcji „new_getattr”
- linia 9: class_.__getattribute__ = new_getattr – teraz zdefiniowano 'new_getattr', więc można się do niej odwoływać jako do nowej metody '__getattribute__' przez udekorowaną klasę
- linia 10: zwracana klasa_ – każdy dobrze wychowany i rozwinięty dekorator powinien zwrócić udekorowany obiekt – w naszym przypadku jest to udekorowana klasa

```
def object_counter(class_):
    class_.__getattr__orig = class_.__getattribute__

    def new_getattr(self, name):
        if name == 'mileage':
            print('We noticed that the mileage attribute was read')
        return class_.__getattr__orig(self, name)

    class_.__getattribute__ = new_getattr
    return class_

@object_counter
class Car:
    def __init__(self, VIN):
        self.mileage = 0
        self.VIN = VIN

car = Car('ABC123')
print('The mileage is', car.mileage)
print('The VIN is', car.VIN)
```

```
We noticed that the mileage attribute was read
The mileage is 0
The VIN is ABC123
```

# Decorators – podsumowanie

- Dekorator jest bardzo potężnym i użytecznym narzędziem w Pythonie, ponieważ pozwala programistom modyfikować zachowanie funkcji, metody lub klasy.
- Dekoratory pozwalają nam opakować inny wywoływalny obiekt w celu rozszerzenia jego zachowania.
- Dekoratorzy w dużej mierze polegają na domknięciach oraz *args i **kwargs.
- Ciekawa uwaga:idea dekoratorów została opisana w dwóch dokumentach – PEP 318 i PEP 3129.

# Decorators – przykłady

```python
import functools
import time

def timer(func):
    """Wyświetla czas działania funkcji"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def obliczenia(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])


obliczenia(1000)
```

```python
import functools
import time

def slow_function(func):
    """Sleep 1 second before calling the function"""
    @functools.wraps(func)
    def wrapper_slow_down(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_slow_down

@slow_function
def countdown(from_number):
    if from_number < 1:
        print("Koniec")
    else:
        print(from_number)
        countdown(from_number - 1)


countdown(10)
```

functools to standardowy moduł Pythona dla funkcji wyższego rzędu (funkcji, które działają na inne funkcje lub zwracają je). wraps() to dekorator, który jest stosowany do funkcji opakowania dekoratora. Aktualizuje funkcję opakowującą, aby wyglądała jak opakowana funkcja, kopiując atrybuty, takie jak __name__, __doc__ (dokumentacja) itp.

# Decorators – self

- Do tej pory implementowaliśmy metody, które wykonywały operacje na instancjach (obiektach), a w szczególności na atrybutach instancji, dlatego nazywaliśmy je metodami instancyjnymi.
- Metody instancji jako pierwszy parametr przyjmują parametr self, który jest ich znakiem rozpoznawczym. Warto podkreślić i pamiętać, że self pozwala na odniesienie się do instancji. Oczywiście wynika z tego, że aby skutecznie zastosować metodę instancji, instancja musiała wcześniej istnieć.
- Każdy z obiektów klasy Example posiada własną kopię zmiennej instancji __internal, a metoda get_internal() pozwala na odczytanie zmiennej instancji specyficznej dla wskazanej instancji. Jest to możliwe dzięki wykorzystaniu self.
- Nazwa parametru self została wybrana arbitralnie i możesz użyć innego słowa, ale musisz to zrobić konsekwentnie w swoim kodzie. Z konwencji wynika, że self dosłownie oznacza odniesienie do instancji.

```
class Example:
    def __init__(self, value):
        self.__internal = value

    def get_internal(self):
        return self.__internal

example1 = Example(10)
example2 = Example(99)
print(example1.get_internal())
print(example2.get_internal())
```

# Decorators – metoda klasy

Metoda klasy (Class method)

- Metody klasowe to metody, które podobnie jak zmienne klasowe działają na samej klasie, a nie na tworzonych instancjach obiektów klasy. Można powiedzieć, że są to metody związane z klasą, a nie z obiektem.
- Kontrolujemy dostęp do zmiennych klasy, np. do zmiennej klasy zawierającej informacje o liczbie utworzonych instancji lub numerze seryjnym nadanym ostatnio wytworzonemu obiektowi lub modyfikujemy stan zmiennych klasy
- musimy utworzyć instancję klasy w alternatywny sposób, aby metoda klasy mogła zostać zaimplementowana przez alternatywny konstruktor.
- Aby móc odróżnić metodę klasową od metody instancji, programista sygnalizuje to dekoratorem @classmethod poprzedzającym definicję metody klasowej
- Dodatkowo pierwszym parametrem metody klasowej jest cls, który jest używany do odwoływania się do metod klasowych i atrybutów klas.
- Mają dostęp do stanu klasy, ponieważ przyjmuje parametr klasy, który wskazuje na klasę, a nie instancję obiektu.
- Może modyfikować stan klasy, który miałby zastosowanie do wszystkich wystąpień klasy. Na przykład może zmodyfikować zmienną klasy, która będzie miała zastosowanie do wszystkich instancji.

```
class Example:
    __internal_counter = 0

    def __init__(self, value):
        Example.__internal_counter +=1

    @classmethod
    def get_internal(cls):
        return '# of objects created: {}'.format(cls.__internal_counter)

print(Example.get_internal())

example1 = Example(10)
print(Example.get_internal())

example2 = Example(99)
print(Example.get_internal())
```

```
# of objects created: 0
# of objects created: 1
# of objects created: 2
```

# Decorators – metoda klasy

- Pozwala wykorzystać metodę class jako alternatywny konstruktor, pozwalający obsłużyć dodatkowy argument.
- Metoda include_brand jest metodą klasową i oczekuje wywołania z dwoma parametrami („vin” i „brand”). Pierwszy parametr służy do tworzenia obiektu przy użyciu standardowej metody init__init__.
- Zgodnie z konwencją tworzenie obiektu klasy (czyli między innymi wywołanie metody __init__) odbywa się za pomocą cls(vin).
- Następnie metoda class wykonuje dodatkowe zadanie – w tym przypadku uzupełnia zmienną instancji marki i ostatecznie zwraca nowo utworzony obiekt.
- Dla samochodu z vin=ABCD1234 metoda __init__ została wywołana domyślnie
- Dla samochodu z vin=DEF567 została wywołana metoda class, która wywołuje metodę __init__ a następnie wykonuje dodatkowe akcje i zwraca obiekt.

```
class Car:
    def __init__(self, vin):
        print('Ordinary __init__ was called for', vin)
        self.vin = vin
        self.brand = ''

    @classmethod
    def including_brand(cls, vin, brand):
        print('Class method was called')
        _car = cls(vin)
        _car.brand = brand
        return _car

car1 = Car('ABCD1234')
car2 = Car.including_brand('DEF567', 'NewBrand')

print(car1.vin, car1.brand)
print(car2.vin, car2.brand)
```

```
Ordinary __init__ was called for ABCD1234
Class method was called
Ordinary __init__ was called for DEF567
ABCD1234
DEF567 NewBrand
```

# Decorators – metoda statyczna

Metoda statyczna (Static method)

- Metody statyczne to metody, które nie wymagają (i nie oczekują) parametru wskazującego obiekt klasy lub samą klasę w celu wykonania ich kodu.
- Przydatne sa gdy potrzebujesz metody użytkowej, która jest dostępna w klasie, ponieważ jest powiązana semantycznie, ale nie wymaga obiektu tej klasy do wykonania jej kodu.
- Metoda statyczna nie musi znać stanu obiektów lub klas.
- Aby móc odróżnić metodę statyczną od metody klasowej lub metody instancji, programista sygnalizuje to dekoratorem @staticmethod poprzedzającym definicję metody klasowej
- Metody statyczne nie mają możliwości modyfikowania stanu obiektów czy klas, ponieważ brakuje im parametrów, które by to umożliwiały
- Metoda statyczna nie może uzyskać dostępu ani zmodyfikować stanu klasy.
- Jest obecny w klasie, ponieważ ma sens, aby metoda była obecna w klasie.

```
class Bank_Account:
    def __init__(self, iban):
        print('__init__ called')
        self.iban = iban

    @staticmethod
    def validate(iban):
        if len(iban) == 20:
            return True
        else:
            return False


account_numbers = ['8' * 20, '7' * 4, '2222']

for element in account_numbers:
    if Bank_Account.validate(element):
        print('We can use', element, ' to create a bank account')
    else:
        print('The account number', element, 'is invalid')
```

# Decorators – metoda statyczna a metoda klasy

- metoda klasowa wymaga „cls” jako pierwszego parametru, a metoda statyczna nie.
- metoda klasowa ma możliwość dostępu do stanu lub metod klasy, a metoda statyczna nie.
- metoda klasowa jest ozdobiona przez „@classmethod”, a metoda statyczna przez „@staticmethod”
- metoda klasowa może być używana jako alternatywny sposób tworzenia obiektów, a metoda statyczna jest tylko metodą użytkową.
- Ogólnie rzecz biorąc, metody statyczne nie wiedzą nic o stanie klasy. Są to metody narzędziowe, które pobierają pewne parametry i pracują na tych parametrach. Z drugiej strony metody klasowe muszą mieć klasę jako parametr.
- Ogólnie używamy metody klas do tworzenia metod fabrycznych. Metody fabryki zwracają obiekt klasy (podobny do konstruktora) dla różnych przypadków użycia.
- Ogólnie używamy metod statycznych do tworzenia funkcji narzędziowych.

```
from datetime import date


class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # metoda clasy do tworzenia obiektu Person + data
urodzenia.
    @classmethod
    def fromBirthYear(cls, name, year):
        return cls(name, date.today().year - year)

    # metoda statyczna określająca czy ossoba jest dorosła.
    @staticmethod
    def isAdult(age):
        return age > 18


person1 = Person('ja', 21)
person2 = Person.fromBirthYear('ja', 1996)

print(person1.age)
print(person2.age)
print(Person.isAdult(22))
```

# 4 OOP dodatkowe informacje

## OOP additional information

# Object-Oriented Programming – Dziedziczenie właściwości z klas wbudowanych

- Python daje możliwość stworzenia klasy, która dziedziczy właściwości z dowolnej klasy wbudowanej w Pythonie w celu uzyskania nowej klasy, która może wzbogacić atrybuty lub metody rodzica. W rezultacie twoja nowo utworzona klasa ma przewagę wszystkich dobrze znanych funkcjonalności odziedziczonych po swoim rodzicu, a nawet rodzicach i nadal możesz uzyskać dostęp do tych atrybutów i metod.
- Później możesz zastąpić metody, dostarczając własne modyfikacje dla wybranych metod.

```
class IntegerList(list):

    @staticmethod
    def check_value_type(value):
        if type(value) is not int:
            raise ValueError('Not an integer type')

    def __setitem__(self, index, value):
        IntegerList.check_value_type(value)
        list.__setitem__(self, index, value)

    def append(self, value):
        IntegerList.check_value_type(value)
        list.append(self, value)

    def extend(self, iterable):
        for element in iterable:
            IntegerList.check_value_type(element)

        list.extend(self, iterable)
```

```
int_list = IntegerList()

int_list.append(66)
int_list.append(22)
print('Appending int elements succeed:', int_list)

int_list[0] = 49
print('Inserting int element succeed:', int_list)

int_list.extend([2, 3])
print('Extending with int elements succeed:',
int_list)

try:
    int_list.append('8-10')
except ValueError:
    print('Appending string failed')
```

```
try:
    int_list[0] = '10/11'
except ValueError:
    print('Inserting string failed')

try:
    int_list.extend([997, '10/11'])
except ValueError:
    print('Extending with ineligible element failed')

print('Final result:', int_list)
```

- statyczna, dedykowana metoda sprawdzania typów argumentów. Ponieważ delegowaliśmy tę odpowiedzialność tylko do jednej metody, kod będzie krótszy, czystszy i łatwiejszy w utrzymaniu. Skorzystamy z tej metody kilka razy. W przypadku, gdy typ argumentu nie jest liczbą całkowitą, zgłaszany jest wyjątek ValueError.
- nadpisana metoda __setitem__, która jest magiczną metodą (zwróć uwagę na podkreślenia) odpowiadającą za wstawienie (nadpisanie) elementu w danej pozycji. Ta metoda wywołuje metodę check_value_type(), a później wywołuje oryginalną metodę __setitem__ pochodzącą z klasy nadrzędnej, która wykonuje resztę zadania (ustawia zweryfikowaną wartość na danej pozycji).
- Dalej nadpisana jest metoda append(), która jest odpowiedzialna za dodanie elementu na koniec listy. Ta metoda jest zgodna z poprzednim sposobem postępowania z nowym elementem;
- nadpisana metoda extend() jest w celu weryfikacji i dodania kolekcji elementów do obiektu.
- Wszystkie pozostałe metody pozostały niezmienione, więc nasza nowa klasa listopodobna nadal będzie zachowywać się jak jej rodzic w tych miejscach.
- Aby nasza nowo stworzona klasa była w pełni funkcjonalna konieczne jest dostarczenie implementacji metod: insert(index, object) i __add__()

# Object-Oriented Programming – Dziedziczenie właściwości z klas wbudowanych

- W kolejnym przykładzie stworzymy klasę opartą na wbudowanym słowniku Pythona, która będzie wyposażona w mechanizmy logowania szczegółów operacji zapisu i odczytu wykonanych na elementach naszego słownika.
- Innymi słowy, uzbrajamy słownik Pythona w możliwość rejestrowania szczegółów (czas i typ operacji): tworzenie klas, dostęp do odczytu i tworzenie lub aktualizacja nowego elementu.
- Stworzyliśmy podklasę z dict klasy z nową metodą __init__(), która wywołuje metodę __init__() ze swojej superklasy. Dodatkowo tworzy listę (self.log), która pełni rolę dziennika.
- Na koniec dziennik jest wypełniany komunikatem stwierdzającym, że obiekt został utworzony;
- stworzyliśmy metodę log_timestamp(), która dodaje kluczowe informacje do atrybutu self.log;
- zastąpiliśmy dwie metody właściwe dla klasy słownika (__getitem__() i __setitem__()), aby zapewnić bogatszą implementację, która rejestruje działania.

```
from datetime import datetime


class MonitoredDict(dict):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.log = list()
        self.log_timestamp('MonitoredDict created')

    def __getitem__(self, key):
        val = super().__getitem__(key)
        self.log_timestamp('value for key [{}] retrieved'.format(key))
        return val

    def __setitem__(self, key, val):
        super().__setitem__(key, val)
        self.log_timestamp('value for key [{}] set'.format(key))

    def log_timestamp(self, message):
        timestampStr = datetime.now().strftime("%Y-%m-%d (%H:%M:%S.%f)")
        self.log.append('{} {}'.format(timestampStr, message))


kk = MonitoredDict()
kk[10] = 15
kk[20] = 5

print('Element kk[10]:', kk[10])
print('Whole dictionary:', kk)
print('Our log book:\n')
print('\n'.join(kk.log))
```

- Użyjemy walidatora IBAN, aby mieć pewność, że nasz słownik aplikacji bankowych zawiera tylko zweryfikowane numery IBAN (klucze) i informacje o powiązanym saldzie (wartość).
- IBAN to algorytm używany przez europejskie banki do określania numerów kont. Standardowa nazwa IBAN (International Bank Account Number) zapewnia prostą i dość niezawodną metodę sprawdzania poprawności numerów rachunków pod kątem prostych literówek, które mogą wystąpić podczas przepisywania numeru, np. z dokumentów papierowych, takich jak faktury lub rachunki, do komputerów.
- Więcej informacji można znaleźć tutaj: https://en.wikipedia.org/wiki/International_Bank_Account_Number

Numer konta zgodny z IBAN składa się z:

- dwuliterowy kod kraju zaczerpnięty z normy ISO 3166-1 (np. FR dla Francji, GB dla Wielkiej Brytanii, DE dla Niemiec itd.)
- dwie cyfry kontrolne służące do sprawdzania poprawności – szybkie i proste, ale nie do końca wiarygodne testy, pokazujące, czy numer jest nieprawidłowy (zniekształcony przez literówkę) lub wydaje się być dobry;
- rzeczywisty numer rachunku (do 30 znaków alfanumerycznych – długość tej części zależy od kraju)

Standard mówi, że walidacja wymaga następujących kroków (według Wikipedii):

- (krok 1) Sprawdź, czy całkowita długość numeru IBAN jest poprawna dla kraju (ten program tego nie zrobi, ale jeśli chcesz, możesz zmodyfikować kod, aby spełniał to wymaganie; uwaga: musisz nauczyć kodu wszystkich długości używany w Europie)
- (krok 2) Przenieś cztery początkowe znaki na koniec łańcucha (tj. kod kraju i cyfry kontrolne)
- (krok 3) Zamień każdą literę w ciągu na dwie cyfry, rozszerzając w ten sposób ciąg, gdzie A = 10, B = 11 ... Z = 35;
- (krok 4) Zinterpretuj łańcuch jako dziesiętną liczbę całkowitą i oblicz resztę tej liczby z dzielenia przez 97; Jeśli reszta wynosi 1, test cyfry kontrolnej jest pomyślny i numer IBAN może być ważny.

# Object-Oriented Programming – Dziedziczenie właściwości z klas wbudowanych

- linia 03: poproś użytkownika o podanie numeru IBAN (numer może zawierać spacje, ponieważ znacząco poprawiają czytelność numeru...
- linia 04: ...ale natychmiast je usuń
- linia 05: wprowadzony IBAN musi składać się wyłącznie z cyfr i liter – jeśli nie...
- linia 06: ...wyślij komunikat
- linia 07: numer IBAN nie może być krótszy niż 15 znaków (jest to najkrótszy wariant, używany w Norwegii)
- linia 08: jeśli jest krótsza, użytkownik jest informowany
- linia 09: ponadto IBAN nie może być dłuższy niż 31 znaków (jest to najdłuższy wariant, stosowany na Malcie)
- linia 10: jeśli jest dłuższy, daj ogłoszenie
- linia 11: rozpocznij właściwe przetwarzanie
- linia 12: przenieś cztery początkowe znaki na koniec liczby i zamień wszystkie litery na duże (krok 02 algorytmu)
- linia 13: jest to zmienna służąca do uzupełnienia liczby, utworzona poprzez zamianę liter na cyfry (zgodnie z krokiem 03 algorytmu)
- linia 14: iteruj przez IBAN
- linia 15: jeśli znak jest cyfrą ...wiersz 16: po prostu skopiuj go;
- linia 17: inaczej...
- linia 18: ... przekonwertuj go na dwie cyfry (zwróć uwagę, jak to się robi tutaj)
- linia 19: skonwertowana postać IBAN jest gotowa – zrób z niej liczbę całkowitą
- linia 20: czy reszta z dzielenia iban2 przez 97 jest równa 1?
- linia 21: jeśli tak, to sukces
- linia 22: inaczej...
- linia 23: ...numer jest nieprawidłowy.

```
# IBAN Validator

iban = input("Enter IBAN, please: ")
iban = iban.replace(' ','')
if not iban.isalnum():
    print("You have entered invalid characters.")
elif len(iban) < 15:
    print("IBAN entered is too short.")
elif len(iban) > 31:
    print("IBAN entered is too long.")
else:
    iban = (iban[4:] + iban[0:4]).upper()
    iban2 = ''
    for ch in iban:
        if ch.isdigit():
            iban2 += ch
        else:
            iban2 += str(10 + ord(ch) - ord('A'))
    ibann = int(iban2)
    if ibann % 97 == 1:
        print("IBAN entered is valid.")
    else:
        print("IBAN entered is invalid.")
```

British: `GB72 HBZU 7006 7212 1253 00`
French: `FR76 30003 03620 00020216907 50`
German: `DE02100100100152517108`

# Object-Oriented Programming – Dziedziczenie właściwości z klas wbudowanych

```
class IBANValidationError(Exception):
    pass


def validateIBAN(iban):
    iban = iban.replace(' ', '')

    if not iban.isalnum():
        raise IBANValidationError("You have entered invalid characters.")

    elif len(iban) < 15:
        raise IBANValidationError("IBAN entered is too short.")

    elif len(iban) > 31:
        raise IBANValidationError("IBAN entered is too long.")

    else:
        iban = (iban[4:] + iban[0:4]).upper()
        iban2 = ''
        for ch in iban:
            if ch.isdigit():
                iban2 += ch
            else:
                iban2 += str(10 + ord(ch) - ord('A'))
        ibann = int(iban2)

        if ibann % 97 != 1:
            raise IBANValidationError("IBAN entered is invalid.")

        return True
```

```
test_keys = ['GB72 HBZU 7006 7212 1253 01', 'FR76 30003 03620 00020216907 50',
'DE02100100100152517108' ]

for key in test_keys:
    try:
        print('Status of "{}" validation: '.format(key))
        validateIBAN(key)
    except IBANValidationError as e:
        print("\t{}".format(e))
    else:
        print("\tcorrect")
```

```
Status of "GB72 HBZU 7006 7212 1253 01" validation:
        IBAN entered is invalid.
Status of "FR76 30003 03620 00020216907 50" validation:
        correct
Status of "DE02100100100152517108" validation:
        correct
```

# Object-Oriented Programming – Dziedziczenie właściwości z klas wbudowanych

```
import random


class IBANValidationError(Exception):
    pass


class IBANDict(dict):
    def __setitem__(self, _key, _val):
        if validateIBAN(_key):
            super().__setitem__(_key, _val)

    def update(self, *args, **kwargs):
        for _key, _val in dict(*args, **kwargs).items():
            self.__setitem__(_key, _val)


def validateIBAN(iban):
    iban = iban.replace(' ', '')

    if not iban.isalnum():
        raise IBANValidationError("You have entered invalid characters.")

    elif len(iban) < 15:
        raise IBANValidationError("IBAN entered is too short.")

    elif len(iban) > 31:
        raise IBANValidationError("IBAN entered is too long.")

    else:
        iban = (iban[4:] + iban[0:4]).upper()
        iban2 = ''
        for ch in iban:
            if ch.isdigit():
                iban2 += ch
            else:
                iban2 += str(10 + ord(ch) - ord('A'))
        ibann = int(iban2)

        if ibann % 97 != 1:
            raise IBANValidationError("IBAN entered is invalid.")

        return True


my_dict = IBANDict()
keys = ['GB72 HBZU 7006 7212 1253 00', 'FR76 30003 03620 00020216907 50',
'DE02100100100152517108']

for key in keys:
    my_dict[key] = random.randint(0, 1000)

print('The my_dict dictionary contains:')
for key, value in my_dict.items():
    print("\t{} -> {}".format(key, value))

try:
    my_dict.update({'dummy_account': 100})
except IBANValidationError:
    print('IBANDict has protected your dictionary against incorrect data insertion')
```

```
The my_dict dictionary contains:
                GB72 HBZU 7006 7212 1253 00 -> 420
                FR76 30003 03620 00020216907 50 -> 8
                DE02100100100152517108 -> 583
IBANDict has protected your dictionary against incorrect data insertion
```

# Zadanie 2

- Utwórz dekorator funkcji, który drukuje znacznik czasu (w postaci rok-miesiąc-dzień godzina:minuta:sekundy, np. 2019-11-05 08:33:22)
- Utwórz kilka zwykłych funkcji, które wykonują proste zadania, takie jak dodawanie lub mnożenie dwóch liczb.
- Zastosuj swój dekorator do tych funkcji, aby mieć pewność, że czas wykonywania funkcji może być monitorowany.

5

# Serializacja i deserializacja

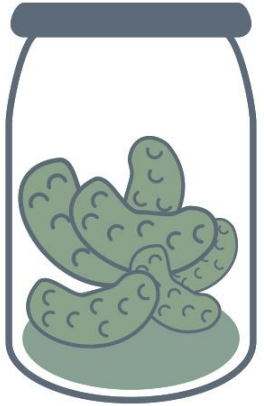
## Serialization and Deserialization

# Serialization and Deserialization – Informacje ogólne

- Serializacja obiektów to proces konwersji stanu obiektu na strumień bajtów czyli postać liniową, nazywany też często marshalingiem
- Strumień bajtów może być dalej przechowywany w dowolnym obiekcie pliko podobnym, takim jak plik dyskowy lub strumień pamięci.
- Może być również przesyłany np. poprzez gniazdo (socket) przez sieć.
- Proces odwrotny, który pobiera strumień bajtów i przekształca go z powrotem w strukturę danych, nazywa się deserializacją lub unmarshallingiem.
- Proces ten używany jest w wielu sytuacjach w których potrzebujemy przechować stan danych w odpowiadającej jej strukturze. Jednym z najczęstszych zastosowań jest zapisywanie stanu sieci neuronowej po fazie uczenia, aby można było z niej później korzystać bez konieczności ponownego uczenia.
- Python oferuje trzy różne moduły w standardowej bibliotece, które umożliwiają serializację i deserializację obiektów:
    - marshal module – nie zaleca się go używać, jest głównie używany przez interpretera a oficjalna dokumentacja ze obiekty mogą być modyfikowane w sposób niekompatybilny wstecz.
    - json module – zalecany jest w wykorzystywaniu komunikacji z innymi językami lub specjalnego zapisu czytelnego dla człowieka jak pliki konfiguracyjne dla aplikacji itp..
    - pickle module – jeżeli nie jest potrzebny zapis czytelny dla człowieka a jedynie jako forma przechowywania danych zaleca się używać tej metody
- Ponadto Python obsługuje język XML, którego można również używać do serializacji obiektów.

# Serialization and Deserialization – modułu pickle

- Moduł „pikle” jest bardzo popularnym i wygodnym modułem do serializacji danych w świecie Pythona.
- Następujące rodzaje mogą być marynowane (pickled):
    - None, warości boole
    - liczby całkowite (intigers), liczby zmiennoprzecinkowe (floating points numbers), liczby zespolone (complex numbers)
    - ciągi znaków (strings) , bajty (bytes) , tablice bajtów (bytes arrays),
    - krotki (tuples), listy (lists), zestawy (sets) i słowniki (dictionaries)
    - obiekty, w tym obiekty z odniesieniami do innych obiektów
    - odniesienia do funkcji i klas, ale nie ich definicje
- Format danych modułu pickle jest bardzo specyficzny dla Pythona.
- Dlatego programy, które nie są napisane w Pythonie, mogą nie być w stanie poprawnie deserializować zakodowanych (marynowanych) danych.



# Serialization and Deserialization – modułu pickle

Wybierzmy nasz pierwszy zestaw danych składający się z

- zagnieżdżony słownik zawierający informacje o walutach
- lista zawierająca łańcuch, liczbę całkowitą i listę.

Kod zaczyna się od instrukcji import odpowiedzialnej za ładowanie modułu pickle

```
import pickle
```

Później możesz zobaczyć, że uchwyt pliku „file_out" jest powiązany z plikiem otwartym do zapisu w trybie binarnym. Ważne jest, aby otworzyć plik w trybie binarnym, ponieważ zrzucamy dane jako strumień bajtów.

```
with open('multidata.pckl', 'wb') as
file_out:
```

Teraz nadszedł czas, aby utrwalić pierwszy obiekt za pomocą funkcji dump(). Ta funkcja oczekuje, że obiekt zostanie utrwalony i uchwyt pliku.

```
pickle.dump(a_dict, file_out)
pickle.dump(a_list, file_out)
```

```
import pickle

a_dict = dict()
a_dict['EUR'] = {'code':'Euro', 'symbol': '€'}
a_dict['GBP'] = {'code':'Pounds sterling', 'symbol': '£'}
a_dict['USD'] = {'code':'US dollar', 'symbol': '$'}
a_dict['JPY'] = {'code':'Japanese yen', 'symbol': '¥'}

a_list = ['a', 123, [10, 100, 1000]]

with open('multidata.pckl', 'wb') as file_out:
    pickle.dump(a_dict, file_out)
    pickle.dump(a_list, file_out)
```

# Serialization and Deserialization – modułu pickle

Teraz nadszedł czas, aby odmarynować (unpickle) zawartość pliku.

Przedstawiony kod jest dość prosty:

- importujemy moduł marynaty;
- plik jest otwierany w trybie binarnym, a uchwyt pliku jest powiązany z plikiem;
- kolejno odczytujemy niektóre porcje danych i deserializujemy je za pomocą funkcji load() ;
- wreszcie badamy rodzaj i zawartość obiektów.

Zwróć uwagę na fakt, że w przypadku modułu „pickle” musisz zapamiętać kolejność utrwalania obiektów i kod deserializacji powinien być zgodny z tą samą kolejnością.

```
import pickle

with open('multidata.pckl', 'rb') as file_in:
    data1 = pickle.load(file_in)
    data2 = pickle.load(file_in)

print(type(data1))
print(data1)
print(type(data2))
print(data2)
```

```
<class 'dict'>
{'EUR': {'code': 'Euro', 'symbol': '€'}, 'GBP': {'code': 'Pounds sterling',
'symbol': '£'}, 'USD': {'code': 'US dollar', 'symbol': '$'}, 'JPY': {'code':
'Japanese yen', 'symbol': '¥'}}
<class 'list'>
['a', 123, [10, 100, 1000]]
```

# Serialization and Deserialization – modułu pickle

- Na początku modułu serializacji wspomnieliśmy, że serializowane obiekty mogą być utrwalane w bazie danych lub przesyłane przez sieć.
- Oznacza to dwie kolejne funkcje odpowiadające funkcjom pickle.dumps() i pickle.loads()
- pickle.dumps(object_to_be_pickled) – oczekuje obiektu początkowego, zwraca obiekt bajtowy. Ten obiekt bajtowy powinien zostać przekazany do bazy danych lub sterownika sieciowego w celu utrwalenia danych;
- pickle.loads(bytes_object) – oczekuje obiektu bytes, zwraca obiekt początkowy.

```
import pickle

a_list = ['a', 123, [10, 100, 1000]]
bytes = pickle.dumps(a_list)
print('Intermediate object type, used to preserve data:', type(bytes))

# now pass 'bytes' to appropriate driver

# therefore when you receive a bytes object from an appropriate driver
you can deserialize it
b_list = pickle.loads(bytes)
print('A type of deserialized object:', type(b_list))
print('Contents:', b_list)
```

```
Intermediate object type, used to preserve data: <class 'bytes'>
A type of deserialized object: <class 'list'>
Contents: ['a', 123, [10, 100, 1000]]
```

# Serialization and Deserialization – modułu pickle

- Pamiętaj, że próby marynowania (pickle) obiektów, których nie można marynować (non-pickling), zgłoszą wyjątek PicklingError.
- Próba piklowania wysoce rekurencyjnej struktury danych (uwaga na cykle) może przekroczyć maksymalną głębokość rekurencji i w takich przypadkach zostanie zgłoszony wyjątek RecursionError.
- Zauważ, że funkcje (zarówno wbudowane, jak i zdefiniowane przez użytkownika) są wybierane na podstawie ich nazwy, a nie wartości.
- Oznacza to, że marynowana jest tylko nazwa funkcji; ani kod funkcji, a nie żaden z jej atrybutów nie są marynowane (pickling).
- Podobnie, klasy są marynowane (pickled) przez nazwane referencje, więc obowiązują te same ograniczenia w środowisku unpiklingowym.
- Zauważ, że żaden kod ani dane klasy nie są marynowane.
- Jest to zrobione celowo, więc możesz naprawić błędy w klasie lub dodać metody do klasy i nadal ładować obiekty, które zostały utworzone za pomocą wcześniejszej wersji klasy.
- Dlatego Twoją rolą jest upewnienie się, że środowisko, w którym klasa lub funkcja jest marynowana (pickled), może zaimportować definicję klasy lub funkcji
- Innymi słowy, funkcja lub klasa musi być dostępna w przestrzeni nazw twojego kodu czytającego plik pickle.

# Serialization and Deserialization – modułu pickle

- Poniższy kod ilustruje sytuację marynowania definicji funkcji
- Nie widzimy żadnych błędów, więc możemy stwierdzić, że f1() zostało pomyślnie zamarynowane i teraz możemy je pobrać z pliku.
- Po wczytaniu otrzymujemy jednak błąd

```
import pickle

import pickle

def f1():
    print('Hello from the jar!')

with open('function.pckl', 'wb') as file_out:
    pickle.dump(f1, file_out)
```

```
import pickle

with open('function.pckl', 'rb') as file_in:
    data = pickle.load(file_in)

print(type(data))
print(data)
data()
```

# Serialization and Deserialization – modułu pickle

- Oto ten sam przykład dotyczący definicji klas i wytrawiania obiektów
- Nie widzimy żadnych błędów, więc możemy stwierdzić, że klasa Cucumber i obiekt zostały pomyślnie zamarynowane i teraz możemy je pobrać z pliku.
- W rzeczywistości utrwalany jest tylko obiekt, ale nie jego definicja, co pozwala nam określić układ atrybutów
- Rozwiązaniem powyższych problemów jest: kod wywołujący funkcje load() lub load() w pickle powinien już znać definicję funkcji/klasy.

```
import pickle

class Cucumber:
    def __init__(self):
        self.size = 'small'

    def get_size(self):
        return self.size

cucu = Cucumber()

with open('cucumber.pckl', 'wb') as file_out:
    pickle.dump(cucu, file_out)
```

```
import pickle

with open('cucumber.pckl', 'rb') as file_in:
    data = pickle.load(file_in)

print(type(data))
print(data)
print(data.size)
print(data.get_size())
```

```
Traceback (most recent call last): File "main.py", line
4, in <module> data = pickle.load(file_in)
AttributeError: Can't get attribute 'Cucumber' on <module
'__main__' from 'main.py'>
```

# Serialization and Deserialization – modułu pickle

- Rozwiązaniem w tym przypadku jest wykluczenie obiektu z procesu serializacji i ponowne zainicjowanie połączenia po deserializacji obiektu.
- Możesz użyć __getstate__() do zdefiniowania, co powinno zostać uwzględnione w procesie marynowania (pickling).
- Ta metoda pozwala określić, co chcesz marynować. Jeśli nie nadpiszesz __getstate__(), wówczas zostanie użyta domyślna instancja __dict__.
- W  przykładzie widać, jak możesz zdefiniować klasę z kilkoma atrybutami i wykluczyć jeden atrybut z serializacji za pomocą __getstate()__:
- W tym przykładzie tworzysz obiekt z trzema atrybutami. Ponieważ jednym atrybutem jest lambda, obiektu nie można wybrać za pomocą standardowego modułu pickle.
- Aby rozwiązać ten problem, określ, co ma zostać zamarynowane, za pomocą funkcji __getstate__(). Najpierw sklonujesz cały __dict__ instancji, aby mieć wszystkie atrybuty zdefiniowane w klasie, a następnie ręcznie usuniesz niemożliwy do pobrania atrybut c.
- Jeśli uruchomisz ten przykład, a następnie dokonasz deserializacji obiektu, zobaczysz, że nowa instancja nie zawiera atrybutu c

```
import pickle

class foobar:
    def __init__(self):
        self.a = 35
        self.b = "test"
        self.c = lambda x: x * x

    def __getstate__(self):
        attributes = self.__dict__.copy()
        del attributes['c']
        return attributes

my_foobar_instance = foobar()
my_pickle_string = pickle.dumps(my_foobar_instance)
my_new_instance = pickle.loads(my_pickle_string)

print(my_new_instance.__dict__)
```

```
{'a': 35, 'b': 'test'}
```

# Serialization and Deserialization – modułu pickle

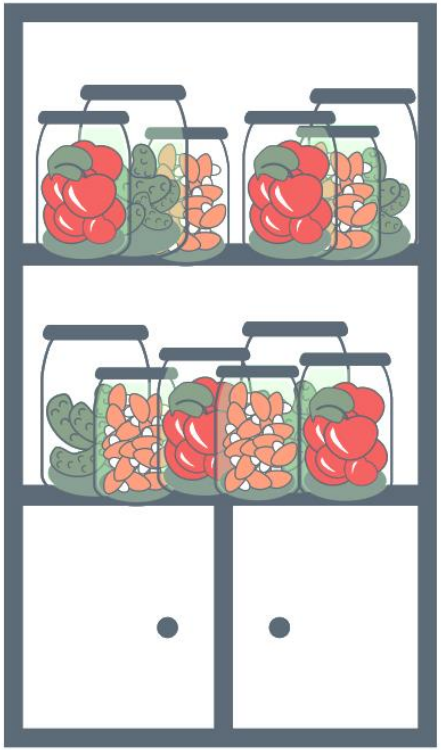
- Ale co by było, gdybyś chciał wykonać kilka dodatkowych inicjalizacji podczas usuwania, na przykład dodając wykluczony obiekt c z powrotem do deserialnej instancji? Możesz to osiągnąć za pomocą __setstate__()
- Przekazując wykluczony obiekt c do __setstate__(), upewniasz się, że pojawi się on w __dict__ niezamarynowanego łańcucha.

```
import pickle

class foobar:
    def __init__(self):
        self.a = 35
        self.b = "test"
        self.c = lambda x: x * x

    def __getstate__(self):
        attributes = self.__dict__.copy()
        del attributes['c']
        return attributes

    def __setstate__(self, state):
        self.__dict__ = state
        self.c = lambda x: x * x

my_foobar_instance = foobar()
my_pickle_string = pickle.dumps(my_foobar_instance)
my_new_instance = pickle.loads(my_pickle_string)
print(my_new_instance.__dict__)
```

# Serialization and Deserialization – modułu shelve

Serializacja obiektów Pythona za pomocą modułu półki (shelve):

- moduł pickle służy do serializacji obiektów jako strumienia jednobajtowego.
- Zarówno strony serializujące, jak i deserializujące muszą przestrzegać kolejności wszystkich elementów umieszczanych w pliku lub bazie danych lub przesyłanych przez sieć.
- Jest jeszcze jeden przydatny moduł, zwany shelve, który jest zbudowany na bazie pickle i implementuje słownik serializacji, w którym obiekty są marynowane i kojarzone z kluczem.
- Klucze muszą być zwykłymi ciągami znaków, ponieważ podstawowa baza danych (dbm) wymaga ciągów znaków
- Dlatego możesz otworzyć plik danych z półki i uzyskać dostęp do marynowanych obiektów za pomocą klawiszy w taki sam sposób, jak w przypadku dostępu do słowników Pythona.
- Może to być wygodniejsze, gdy serializujesz wiele obiektów.



# Serialization and Deserialization – modułu shelve

- Najpierw zaimportujmy odpowiedni moduł i utwórzmy obiekt reprezentujący plikową bazę danych
- Znaczenie dodatkowego parametru:

| Value | Meaning |
|---|---|
| 'r' | Open existing database for reading only |
| 'w' | Open existing database for reading and writing |
| 'c' | Open database for reading and writing, creating it if it doesn't exist (this is a default value) |
| 'n' | Always create a new, empty database, open for reading and writing |

```
import shelve

shelve_name = 'first_shelve.shlv'

my_shelve = shelve.open(shelve_name, flag='c')
my_shelve['EUR'] = {'code':'Euro', 'symbol': '€'}
my_shelve['GBP'] = {'code':'Pounds sterling', 'symbol': '£'}
my_shelve['USD'] = {'code':'US dollar', 'symbol': '$'}
my_shelve['JPY'] = {'code':'Japanese yen', 'symbol': '¥'}
my_shelve.close()

new_shelve = shelve.open(shelve_name)
print(new_shelve['USD'])
new_shelve.close()
```

- Teraz nasz obiekt półki jest gotowy do działania, więc wstawmy kilka elementów i zamknijmy obiekt półki.
- Teraz otwórzmy plik półki

# Serialization and Deserialization – modułu shelve

Powinieneś traktować obiekt półki jak słownik Pythona, z kilkoma dodatkowymi uwagami:

- klucze muszą być łańcuchami
- Python umieszcza zmiany w buforze, który jest okresowo opróżniany na dysk.
- Aby wymusić natychmiastowe opróżnianie, wywołaj metodę sync() na swoim obiekcie półki
- wywołanie metody close() na obiekcie półki powoduje również opróżnienie buforów.

Kiedy traktujesz obiekt półki jak słownik Pythona, możesz skorzystać z narzędzi słownikowych:

- funkcja len()
- operator in;
- metody keys() i items()
- operacja aktualizacji, która działa tak samo, jak w przypadku słownika Pythona
- instrukcja del, używana do usuwania pary klucz-wartość

Po uruchomieniu kodu zauważysz dodatkowo, że niektóre pliki są tworzone w celu obsługi bazy danych. Nie próbuj zmieniać tych plików za pomocą zewnętrznych narzędzi, ponieważ Twoja półka może stać się niespójna, co spowoduje błędy odczytu/zapisu

Korzystanie z półki jest naprawdę łatwe i efektywne. Co więcej, powinieneś wiedzieć, że możesz zasymulować półkę, marynując cały słownik, ale moduł półki bardziej efektywnie wykorzystuje pamięć, więc zawsze, gdy potrzebujesz dostępu do marynowanych obiektów, użyj półki.

# 6 Metaprogramming

## Metaprogramowanie

# Metaprogramming – metaklasy

- Metaprogramowanie to technika programowania, w której programy komputerowe mają możliwość modyfikowania kodów własnych lub innych programów. Może to brzmieć jak pomysł z opowiadania science fiction, ale pomysł narodził się i został zrealizowany na początku lat 60.
- W przypadku Pythona modyfikacje kodu mogą wystąpić podczas wykonywania kodu i być może już tego doświadczyłeś podczas implementacji dekoratorów, przesłaniania operatorów, a nawet implementacji protokołu właściwości.
- Może to wyglądać jak cukier składniowy, ponieważ w wielu przypadkach metaprogramowanie pozwala programistom zminimalizować liczbę linii kodu do wyrażenia rozwiązania, co z kolei skraca czas programowania.
- Ale prawda jest taka, że tę technikę można wykorzystać do przygotowania narzędzi; te narzędzia można zastosować w kodzie, aby dostosować go do określonych wzorców programistycznych lub pomóc w stworzeniu spójnego interfejsu API (Application Programming Interface).
- Innym przykładem metaprogramowania jest koncepcja metaklas
- Tim Peters, guru Pythona, który jest autorem Zen of Python, wyraził swoje odczucia na temat metaklas w grupie dyskusyjnej comp.lang.python 22/12/2002

*[metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).*

# Metaprogramming – metaklasy

- W Pythonie metaklasa to klasa, której instancjami są klasy. Tak jak zwykła klasa definiuje zachowanie pewnych obiektów, metaklasa pozwala na dostosowanie instancji klasy.
- Funkcjonalność metaklasy częściowo pokrywa się z funkcjonalnością dekoratorów klas, ale metaklasy działają w inny sposób niż dekoratory.
- dekoratory wiążą nazwy udekorowanych funkcji lub klas z nowymi wywoływalnymi obiektami. Dekoratory klas są stosowane podczas tworzenia instancji klas.
- metaklasy przekierowują instancje klas do dedykowanej logiki, zawartej w metaklasach.
- Metaklasy są stosowane, gdy definicje klas są odczytywane w celu utworzenia klas, na długo przed utworzeniem instancji klas.
- Metaklasy zwykle wchodzą do gry, gdy programujemy zaawansowane moduły lub frameworki, gdzie trzeba zapewnić dużą precyzyjną automatyzację.
- Typowe przypadki użycia metaklas:
    - Logowanie (logging)
    - Rejestrowanie klas w czasie tworzenia (registering classes at creation time)
    - Sprawdzanie interfejsu (interface checking)
    - Automatyczne dodawanie nowych metod
    - Automatyczne dodawanie nowych zmiennych

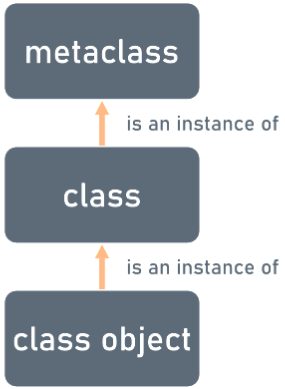
•classes are used to create objects,
•the type of the metaclass `type` is `type`

# Metaprogramming – metaklasy

- W podejściu Pythona wszystko jest obiektem, az każdym obiektem jest powiązany jakiś typ. Aby uzyskać typ dowolnego obiektu, użyj funkcji type()
- Widzimy, że obiekty w Pythonie są definiowane przez ich właściwe klasy
- Przykład pokazuje również, że możemy tworzyć własne klasy, a te klasy będą instancjami typu special class, czyli domyślnej metaklasy odpowiedzialnej za tworzenie klas.

```
class Dog:
    pass


age = 10
codes = [33, 92]
dog = Dog()

print(type(age))
print(type(codes))
print(type(dog))
print(type(Dog))
```

```
for t in (int, list, type):
    print(type(t))
```




Dodatkowo:

- metaklasy służą do tworzenia klas
- klasy służą do tworzenia obiektów
- typem typu metaklasy jest type

# Metaprogramming – metaklasy – typ funkcji

Powinniśmy zapoznać się z kilkoma specjalnymi atrybutami:

- __nazwa__ – nieodłączna dla klas; zawiera nazwę klasy
- __class__ – nieodłączny zarówno dla klas, jak i instancji; zawiera informacje o klasie, do której należy instancja klasy
- __bases__ – właściwe dla klas; jest krotką i zawiera informacje o klasach bazowych klasy
- __dict__ – nieodłączny zarówno dla klas, jak i instancji; zawiera słownik (lub inny obiekt odwzorowania typu) atrybutów obiektu.

```
class Dog:
    pass

dog = Dog()
print('"dog" is an object of class named:', Dog.__name__)
print()
print('class "Dog" is an instance of:', Dog.__class__)
print('instance "dog" is an instance of:', dog.__class__)
print()
print('class "Dog" is  ', Dog.__bases__)
print()
print('class "Dog" attributes:', Dog.__dict__)
print('object "dog" attributes:', dog.__dict__)
```

```
"dog" is an object of class named: Dog

class "Dog" is an instance of: <class 'type'> instance
"dog" is an instance of: <class '__main__.Dog'>

class "Dog" is (<class 'object'>,)

class "Dog" attributes: {'__module__': '__main__',
'__dict__': <attribute '__dict__' of 'Dog' objects>,
'__weakref__': <attribute '__weakref__' of 'Dog'
objects>, '__doc__': None}

object "dog" attributes: {}
```

# Metaprogramming – metaklasy – typ funkcji

- Kiedy funkcja type() jest wywoływana z trzema argumentami, dynamicznie tworzy nową klasę.
- Do wywołania type( , , ):
  - argument określa nazwę klasy, ta wartość staje się atrybutem __name__ klasy
  - argument określa krotkę klas bazowych, z których dziedziczona jest nowo utworzona klasa
  - ten argument staje się atrybutem __bases__ klasy;
  - argument określa słownik zawierający definicje metod i zmienne dla ciała klasy
  - elementy tego argumentu stają się atrybutem __dict__ klasy i określają przestrzeń nazw klasy.

```
Dog = type('Dog', (), {})

print('The class name is:', Dog.__name__)
print('The class is an instance of:', Dog.__class__)
print('The class is based on:', Dog.__bases__)
print('The class attributes are:', Dog.__dict__)
```

```
The class name is: Dog
The class is an instance of: <class 'type'>
The class is based on: (<class 'object'>,)
The class attributes are: {'__module__': '__main__', '__dict__': <attribute
'__dict__' of 'Dog' objects>, '__weakref__': <attribute '__weakref__' of 'Dog'
objects>, '__doc__': None}
```

# Metaprogramming – metaklasy – typ funkcji

- Jak widać, klasa Dog jest teraz wyposażona w dwie metody (feed() i bark()) oraz atrybut instancji age.
- Ten sposób tworzenia klas przy użyciu funkcji type jest istotny dla sposobu tworzenia klas w Pythonie przy użyciu instrukcji class
- po zidentyfikowaniu instrukcji klasy i wykonaniu ciała klasy wykonywany jest kod class = type(, , )
- typ jest odpowiedzialny za wywołanie metody __call__ podczas tworzenia instancji klasy
- ta metoda wywołuje dwie inne metody:__new__(), odpowiedzialny za utworzenie instancji klasy w pamięci komputera; ta metoda jest uruchamiana przed __init__()
- _init__(), odpowiedzialny za inicjalizację obiektu
- Metaklasy zwykle implementują te dwie metody (__init__, __new__), przejmując kontrolę nad procedurą tworzenia i inicjalizacji nowej instancji klasy. Klasy otrzymują nową warstwę logiki.

```
def bark(self):
    print('Woof, woof')

class Animal:
    def feed(self):
        print('It is feeding time!')

Dog = type('Dog', (Animal, ), {'age':0, 'bark':bark})

print('The class name is:', Dog.__name__)
print('The class is an instance of:', Dog.__class__)
print('The class is based on:', Dog.__bases__)
print('The class attributes are:', Dog.__dict__)

doggy = Dog()
doggy.feed()
doggy.bark()
```

# Metaprogramming – metaklasy – typ funkcji

- Metaklasy to klasy tworzone w celu uzyskania klas
- Pierwszym krokiem jest zdefiniowanie metaklasy, która wywodzi się z typu type i uzbraja klasę za pomocą „custom_attribute”
- Należy zwróć uwagę na fakt, że:
    - klasa My_Meta pochodzi od typu. To sprawia, że nasza klasa jest metaklasą
    - nasza własna metoda __new__ została zdefiniowana.
    - Jego rolą jest wywołanie metody __new__ klasy nadrzędnej w celu utworzenia nowej klasy
    - __new__ używa „mcs” w odniesieniu do klasy – to tylko konwencja
    - dodatkowo tworzony jest atrybut class
    - klasa jest zwracana
    - nowa klasa została zdefiniowana w taki sposób, że niestandardowa metaklasa jest wymieniona w definicji klasy jako metaklasa. Jest to sposób, aby powiedzieć Pythonowi, aby używał My_Meta jako metaklasy, a nie zwykłej nadklasy
    - drukujemy zawartość atrybutu class __dict__, aby sprawdzić, czy atrybut niestandardowy jest obecny.

```
class My_Meta(type):
    def __new__(mcs, name, bases, dictionary):
        obj = super().__new__(mcs, name, bases,
dictionary)
        obj.custom_attribute = 'Added by My_Meta'
        return obj

class My_Object(metaclass=My_Meta):
    pass

print(My_Object.__dict__)
```

# Metaprogramming – metaklasy – typ funkcji

- spróbuj zbudować metaklasę odpowiedzialną za uzupełnianie klas metodą (jeśli jej nie ma), aby upewnić się, że wszystkie twoje klasy są wyposażone w metodę o nazwie „greetings”.
- Jak widać, istnieje zdefiniowana funkcja greetings(), która pozdrawia każdego, kto z nią wejdzie w interakcję. W realnym scenariuszu mogłaby to być funkcja, która jest obowiązkowa dla każdej klasy i odpowiada za spójność atrybutów obiektów; może to być funkcja zwracająca sumę kontrolną dla niektórych wartości atrybutu.
- W My_Class1, z założenia, nie ma funkcji powitania, więc kiedy klasa jest konstruowana, jest wyposażana w domyślną funkcję przez metaklasę.
- Natomiast w My_Class2 funkcja greetings jest obecna od samego początku.
- Obie klasy opierają się na tej samej metaklasie.
- Po uruchomieniu kodu zobaczysz, że obie instancje klas są wyposażone w metody greetings(). Dla klasy „biedniejszej” uzupełnia ją metaklasa.
- W ten sposób metaklasy stają się bardzo przydatne – mogą kontrolować proces tworzenia instancji klas i dostosowywać tworzone klasy do wybranych reguł.

```
def greetings(self):
    print('Just a greeting function, but it could be something
more serious like a check sum')

class My_Meta(type):
    def __new__(mcs, name, bases, dictionary):
        if 'greetings' not in dictionary:
            dictionary['greetings'] = greetings
        obj = super().__new__(mcs, name, bases, dictionary)
        return obj

class My_Class1(metaclass=My_Meta):
    pass

class My_Class2(metaclass=My_Meta):
    def greetings(self):
        print('We are ready to greet you!')

myobj1 = My_Class1()
myobj1.greetings()
myobj2 = My_Class2()
myobj2.greetings()
```

```
Just a greeting function, but it could be something more serious like a check sum
We are ready to greet you!
```

# Metaprogramming – metaklasy – podsumowanie

- Klasy to plany obiektów - klasa działa jak plan, gdy tworzymy jej instancję,
- Metaklasy to plany klasy - metaklasa działa jako plan tylko wtedy, gdy klasa jest zdefiniowana.

Najprostsza implementacja metaklasy, która nic nie robi:
- Dziedziczy po klasie *type*
- *__new__* jest metodą klasy nawet bez urzycia *@classmethod*
- *super().__new__* zwraca nową klasę

```
class MyMeta(type):
    def __new__(cls, name, bases, namespace):
        return super().__new__(cls, name, bases, namespace)

class MyClass(metaclass=MyMeta):
    x = 3
```

# Metaprogramming – metaklasy – podsumowanie

Załóżmy że mamy klasę Preson, definiujemy w niej podstawowe elementy:

- Zdefiniujmy listę właściwości obiektu
- Zdefiniujmy metodę __init__ do inicjalizacji atrybutów obiektu
- Zaimplementujmy metody __str__ i __repr__, aby reprezentować obiekty w formatach czytelnych dla ludzi i maszyn
- Zaimplementujmy metodę __eq__, aby porównać obiekty według wartości wszystkich właściwości.
- Zaimplementujmy metodę __hash__, aby używać obiektów klasy jako kluczy słownika lub elementów zestawu.

Załóżmy że chcemy zdefiniować klasę Person magicznie która zawiera automatycznie wszystkie te funkcje.

```
class Person:
    props = ['first_name', 'last_name', 'age']
```

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        self._age = value

    def __eq__(self, other):
        return self.name == other.name and self.age == other.age

    def __hash__(self):
        return hash(f'{self.name, self.age}')

    def __str__(self):
        return f'Person(name={self.name},age={self.age})'

    def __repr__(self):
        return f'Person(name={self.name},age={self.age})'
```

# Metaprogramming – metaklasy – podsumowanie

1. Najpierw definiujemy klasę Data

```
class Data(type):
    pass
```

2. Należy zauważyć, że metoda __new__ jest statyczną metodą metaklasy Data. I nie musisz używać dekoratora @staticmethod, ponieważ Python traktuje go wyjątkowo. Ponadto metoda __new__ tworzy nową klasę, taką jak klasa Person, a nie instancję klasy Person.

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases,
class_dict)
        return class_obj
```

3. Zdefiniujmy klasę Prop, która akceptuje nazwę atrybutu i zawiera trzy metody tworzenia obiektu właściwości (set, get i delete). Metaklasa Data będzie używać tej klasy Prop do dodawania obiektów właściwości do klasy.

```
class Prop:
    def __init__(self, attr):
        self._attr = attr

    def get(self, obj):
        return getattr(obj, self._attr)

    def set(self, obj, value):
        return setattr(obj, self._attr, value)

    def delete(self, obj):
        return delattr(obj, self._attr)
```

# Metaprogramming – metaklasy – podsumowanie

4. Utwórz nową statyczną metodę define_property(), która tworzy obiekt właściwości dla każdego atrybutu z listy props

5. Poniżej zdefiniowano klasę Person korzystającą z metaklasy Data gdzie klasa Person ma dwie właściwości name i age

```
mappingproxy({'__dict__': <attribute '__dict__' of 'Person' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Person' objects>,
              'age': <property object at 0x0000013EE269D770>,
              'name': <property object at 0x0000013EE269D720>,
              'props': ['name', 'age']})
```

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases,
class_dict)
        Data.define_property(class_obj)

        return class_obj

    @staticmethod
    def define_property(class_obj):
        for prop in class_obj.props:
            attr = f'_{prop}'
            prop_obj = property(
                fget=Prop(attr).get,
                fset=Prop(attr).set,
                fdel=Prop(attr).delete
            )
            setattr(class_obj, prop, prop_obj)
```

```
class Person(metaclass=Data):
    props = ['name', 'age']


print(Person.__dict__)
```

# Metaprogramming – metaklasy – podsumowanie

6. Poniżej dodajemy statyczną metodę init i przypisano ją do atrybutu __init__ obiektu klasy
7. Poniższe polecenie tworzy nową instancję klasy Person i inicjalizuje jej atrybuty

{'_age': 25, '_name': 'John Doe'}

p.__dict__ zawiera dwa atrybuty _name i _age oparte na predefiniowanych nazwach na liście props

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        return class_obj

    @staticmethod
    def init(class_obj):
        def _init(self, *obj_args, **obj_kwargs):
            if obj_kwargs:
                for prop in class_obj.props:
                    if prop in obj_kwargs.keys():
                        setattr(self, prop, obj_kwargs[prop])

            if obj_args:
                for kv in zip(class_obj.props, obj_args):
                    setattr(self, kv[0], kv[1])

        return _init
```

```
p = Person('John Doe', age=25)
print(p.__dict__)
```

# Metaprogramming – metaklasy – podsumowanie

8. Poniżej zdefiniowano statyczną metodę repr, która zwraca funkcję i używa jej dla atrybutu __repr__ obiektu klasy

9. Poniższe tworzy nową instancję klasy Person i wyświetla ją

Person(name=John Doe, age=25)

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        return class_obj

    @staticmethod
    def repr(class_obj):
        def _repr(self):
            prop_values = (getattr(self, prop) for prop in class_obj.props)
            prop_key_values = (f'{key}={value}' for key, value in
zip(class_obj.props, prop_values))
            prop_key_values_str = ', '.join(prop_key_values)
            return f'{class_obj.__name__}({prop_key_values_str})'

        return _repr
```

```
p = Person('John Doe', age=25)
print(p)
```

# Metaprogramming – metaklasy – podsumowanie

10. Poniżej zdefiniowano metody eq i hash oraz przypisano je do __eq__ i __hash__ obiektu klasy metaklasy

11. Poniższe tworzy dwie instancje Person i porównuje je. Jeśli wartości wszystkich właściwości są takie same, będą one równe. Inaczej nie będą równe

```
    @staticmethod
    def hash(class_obj):
        def _hash(self):
            values = (getattr(self, prop) for prop in class_obj.props)
            return hash(tuple(values))

        return _hash
```

```
p1 = Person('John Doe', age=25)
p2 = Person('Jane Doe', age=25)

print(p1 == p2)

p2.name = 'John Doe'
print(p1 == p2)
```

False

True

```
class Data(type):
    def __new__(mcs, name, bases, class_dict):
        class_obj = super().__new__(mcs, name, bases, class_dict)

        # create property
        Data.define_property(class_obj)

        # define __init__
        setattr(class_obj, '__init__', Data.init(class_obj))

        # define __repr__
        setattr(class_obj, '__repr__', Data.repr(class_obj))

        # define __eq__ & __hash__
        setattr(class_obj, '__eq__', Data.eq(class_obj))
        setattr(class_obj, '__hash__', Data.hash(class_obj))

        return class_obj

    @staticmethod
    def eq(class_obj):
        def _eq(self, other):
            if not isinstance(other, class_obj):
                return False

            self_values = [getattr(self, prop) for prop in class_obj.props]
            other_values = [getattr(other, prop) for prop in other.props]

            return self_values == other_values

        return _eq
```

# Metaprogramming – metaklasy – podsumowanie

12. Użyjmy dekoratora w celu definicji Employee klasy który używa Data metaklasy.
13. Najpierw zdefiniujmy funkcje dekoratora która zwraca nową klasę, będącą instancją metaklasy Data
14. Następnie użyjmy dekoratora @data dla dowolnej klasy która używa metaklasy Data

```
def data(cls):
    return Data(cls.__name__, cls.__bases__, dict(cls.__dict__))
```

```
@data
class Employee:
    props = ['name', 'job_title']
```

Właśnir stworzone została metaklasa Data na podobieństwo dekoratora @dataclass określonego w PEP 557.



Kiedy możemy zastosować metaklasy:

- **Unikanie powtórzeń dekoratorów lub dekorowanie wszystkich podklas** – gdy używasz wile powtórzeń dekoratorów możesz przenieść pewne mechaniki do mataklasy
- **Walidacja podklas** - mówiąc o klasach, pomyślmy o dziedziczeniu w kontekście wzorca projektowego Metoda szablonowa. Mówiąc najprościej, definiujemy algorytm w klasie bazowej, ale pozostawiamy jeden lub więcej kroków (lub atrybutów) jako metody abstrakcyjne (lub właściwości), które mają być zastąpione w podklasie.
- **Rejestrowanie podklas** – rozszerzalny wzorzec strategii - Korzystając z atrybutów metaklas, możemy również napisać sprytną implementację fabryki typu Open-Closed. Pomysł będzie polegał na prowadzeniu rejestru konkretnych (nieabstrakcyjnych) podklas i budowaniu ich za pomocą nazwy
- **Deklaratywny sposób budowania GUI** - Za tę niesamowitą aplikację metaklas należy się Andersowi Hammarquistowi – autorowi wykładu EuroPython Metaklasy dla zabawy i zysku: Tworzenie deklaratywnej implementacji GUI. Zasadniczo pomysł polega na przekształceniu kodu imperatywnego odpowiedzialnego za budowanie GUI z komponentów.
- **Dodawanie atrybutów** – Model.DoesNotExist Django ORM – zaczerpnięty z Django, które używa metaklas. W przypadku modeli robi wiele rzeczy, od walidacji po dynamiczne dodawanie kilku atrybutów, np. wyjątków DoesNotExist i MultipleObjectsReturned