Wykłady ze Wstępu do Informatyki II

dla kierunku Matematyka

czyli nauka programowania w języku Python

dr Adam Marszałek

Instytut Informatyki
Politechnika Krakowska

Rok akademicki 2017/2018

Dekoratory - czyli opakowywanie funkcji

- Funkcje w Pythonie są obiektami pierwszego rzędu. Co oznacza, że mogą być przekazywane jako parametry wywołania innych funkcji oraz mogą być też wartościami zwracanymi przez te funkcje.
- Na przykład poniższa funkcja pobiera jako argument inną funkcję i wyświetla nazwę podanej funkcji:

```
In [1]: def nazwa_funkcji(f):
            print(f.__name__)
```

- Prosty przykład użycia:

```
In [2]: def foo():
            print('jakis komunikat')

        nazwa_funkcji(foo)
```

```
foo
```

Dekoratory - czyli opakowywanie funkcji

- Poniżej przykład funkcji, która tworzy nową funkcję i zwraca ją jako wynik. W tym wypadku `utworz_dodawanie` tworzy funkcję, która dodaje stałą do jej argumentu:

```
In [1]: def utworz_dodawanie(x):
            def dodaj(y):
                return x + y
            return dodaj

In [2]: dodaj5 = utworz_dodawanie(5)

In [3]: dodaj5(10)
Out[3]: 15
```

Dekoratory - czyli opakowywanie funkcji

- Łącząc obie powyższe możliwości możemy zdefiniować funkcję, która będzie pobierała inną funkcję w parametrze i zwracała jakąś funkcję utworzoną w sposób zależny od podanego parametru.
- Możemy na przykład utworzyć funkcję opakowującą przekazaną funkcję, która będzie pokazywała informacje o każdym wywołaniu tej funkcji:

In [1]:
```
def foo():
    print('jakis komunikat')

def pokaz_wywolanie(f):
    def opakowanie(*args, **kwds):
        print('Wywoluje:', f.__name__)
        return f(*args, **kwds)
    return opakowanie

bar = pokaz_wywolanie(foo)
bar()
```

```
Wywoluje: foo
jakis komunikat
```

Dekoratory - czyli opakowywanie funkcji

- Ponadto jeśli przypiszemy rezultat wywołania funkcji `pokaz_wywolanie` do tej samej nazwy co jej argument, to tym samym zastąpimy oryginalną wersję funkcji naszym opakowaniem:

```
In [2]: foo = pokaz_wywolanie(foo)
        foo()
```

```
Wywoluje: foo
jakis komunikat
```

- Dekoratory w języku Python to tylko tzw. „lukier składniowy” (ang. syntactic sugar). Wygoda ich używania polega na tym, że możemy ich używać w następujący sposób:

```
In [3]: @pokaz_wywolanie
        def func():
            print('Funkcja func została opakowana')
        func()
```

```
Wywoluje: func
Funkcja func została opakowana
```

Dekoratory - czyli opakowywanie funkcji

- Dla przejrzystości kodu, w poprzednich przykładach pominięto, istotny podczas tworzenia własnego dekoratora, dekorator `wraps`. Jego pominięcie powoduje utratę metadanych dekorowanej funkcji (np. docstringa). Zalecane jest, by był on dodawany do tworzonych dekoratorów.
- Wersja bez dekoratora `wraps`:

In [1]:
```
def my_decorator(f):
    def wrapper(*args, **kwds):
        print('Calling decorated function')
        return f(*args, **kwds)
    return wrapper

@my_decorator
def example():
    """Docstring"""
    print('Called example function')

example()
print(example.__name__)
print(example.__doc__)
```

```
Calling decorated function
Called example function
wrapper
None
```

Dekoratory - czyli opakowywanie funkcji

- Wersja z użyciem dekoratora `wraps`:

In [2]:
```
from functools import wraps

def my_decorator(f):
    @wraps(f)
    def wrapper(*args, **kwds):
        print('Calling decorated function')
        return f(*args, **kwds)
    return wrapper

@my_decorator
def example():
    """Docstring"""
    print('Called example function')

example()
print(example.__name__)
print(example.__doc__)
```

```
Calling decorated function
Called example function
example
Docstring
```

Dekoratory - czyli opakowywanie funkcji

- Dekoratory w języku Python zyskały dużą popularność i są szeroko stosowane w nowoczesnym kodzie pythonowym.
- Dekoratory są bardzo użyteczne przy refaktoryzacji kodu. Często zdarza się, iż ta sama funkcjonalność musi zostać wykonana w wielu funkcjach, np. odpisy do logów, synchronizacja wątków, logowanie, cache itp.
- Poza tym składnia dekoratorów pozwala na umieszczanie wyraźnej informacji (jeszcze przed definicją funkcji) w jaki sposób funkcja zostanie udekorowana, a więc w jaki sposób jej funkcjonalność zostanie zmieniona. To zdecydowanie zwiększa czytelność kodu.
- Możliwe jest również przekazywanie argumentów do dekoratorów, tworzenie dekoratorów w formie klas czy dekorowanie klas. Te zagadnienia wykraczają jednak poza kurs podstawowy.

Gettery i settery

- **Gettry i settery** nazywane też **akcesorami/mutatorami**, wykorzystywane są odpowiednio do pobierania i ustawiania wartości atrybutu obiektu. Zapewniają one **enkapsulacje danych**.
- Na początku stwórzmy sobie prostą klasę z getterem i setterem w postaci zwykłych metod:

In [1]:
```
class Osoba(object):
    def __init__(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, name):
        self.__name = name

a = Osoba('Adam')
b = Osoba('')
print(a.get_name(), b.get_name())

b.set_name('Basia')
print(a.get_name(), b.get_name())
```

```
Adam
Adam Basia
```

Gettery i settery - dekorator `@property`

- Możliwe jest zdefiniowanie getterów i setterów dla zmiennych prywatnych w taki sposób aby móc wywoływać je za pomocą składni `zmienna.pole=wartosc`.
- Służy do tego dekorator `@property`, który identyfikuje metodę jako getter. Aby dodać setter należy użyć `@name.setter`, gdzie `name` musi być takie samo jak nazwa pola.

In [1]:
```
class Osoba(object):
    def __init__(self, name):
        self.__name = name

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

    @name.setter
    def name(self, name):
        self.__name = name

a = Osoba('Adam')
b = Osoba('')
print(a.name, b.name)

b.name='Basia'
print(a.name, b.name)
```

```
Adam
Adam Basia
```

Atrybuty statyczne

- Atrybuty zdefiniowane poza metodą `__init__()` traktowane są jako atrybuty statyczne (atrybuty klasy a nie instancji).

In [1]:
```
class Osoba(object):
    counter = 0

    def __init__(self):
        Osoba.counter += 1

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

a = Osoba()
b = Osoba()
print(a.counter, b.counter)
c = Osoba()
d = Osoba()
print(a.counter)
del b
print(a.counter)
```

```
2 2
4
3
```

Metody instancji, klasy i statyczne

- **Metoda instancji** to domyślna metoda utworzona wewnątrz klasy w postaci:

```
def func(self,):
    pass
```

- Metodę tą możemy wykonać na instancji klasy (bez podawania argumenty `self`), lub bezpośrednio z klasy (ale wtedy należy podać w miejsce argumentu `self` jakiś obiekt danej klasy).

In [1]:
```
class Osoba(object):
    def __init__(self, name):
        self.name = name
    def przywitanie(self):
        print('Witaj {}'.format(self.name))

a = Osoba('Adam')
a.przywitanie()
Osoba.przywitanie(a)
```

```
Witaj Adam
Witaj Adam
```

Metody instancji, statyczne i metody klasy

- **Metoda statyczna** to metoda utworzona wewnątrz klasy, która nie operuje na konkretnej instancji klasy. Nie posiadają one argumentu `self`, lecz opatrzone są dekoratorem `@staticmethod`.

```
@staticmethod
def func():
    pass
```

- Statyczną metodę można wywołać zarówno przy pomocy nazwy klasy, jak i jej obiektu, ale w obu przypadkach rezultat będzie ten sam. Technicznie jest to bowiem zwyczajna funkcja umieszczona po prostu w zasięgu klasy zamiast w zasięgu globalnym.

In [1]:
```
class Osoba(object):
    def __init__(self, name):
        self.name = name
    @staticmethod
    def przywitanie():
        print('Witaj !!!')

a = Osoba('Adam')
a.przywitanie()
Osoba.przywitanie()
```

```
Witaj !!!
Witaj !!!
```

Metody instancji, statyczne i metody klasy

- **Metoda klasy** to metoda utworzona wewnątrz klasy, która wywoływana jest na rzecz całej klasy i przyjmuje ową klasę jako swój pierwszy argument (argument ten jest często nazywany `cls`, ale jest to o wiele słabsza konwencja niż ta dotycząca `self`). Metody klasowe opatrzone są dekoratorem `@classmethod`.

```
@classmethod
def func(cls,):
    pass
```

- Podobnie jak metody statyczne, można je wywoływać na dwa sposoby – przy pomocy klasy lub obiektu – ale w obu przypadkach do `cls` trafi wyłącznie klasa. Często wykorzystywane do tworzenia dodatkowych konstruktorów.

```
In [1]: class Osoba(object):
            def __init__(self, name):
                self.name = name
            @classmethod
            def empty(cls):
                return cls('')

        a = Osoba('Adam')
        b = a.empty()
        c = Osoba.empty()
        a.name, b.name, c.name
Out[1]: ('Adam', '', '')
```

Dziedziczenie

- W języku Python składnia mechanizmu dziedziczenia jest bardzo prosta, wystarczy w momencie tworzenia klasy w nawiasach podać nazwy klas po których dana klasa ma dziedziczyć.

```
class KlasaBazowa():
    pass

class KlasaPochodna(KlasaBazowa):
    pass
```

- Nazwa `KlasaBazowa` musi być zdefiniowana w zasięgu zawierającym definicję klasy pochodnej. Zamiast nazwy klasy bazowej dopuszcza się również wyrażenie. Jest to szczególnie przydatne, jeśli nazwa klasy bazowej zdefiniowana jest w innym module, np.:

```
class KlasaPochodna(nazwa_modulu.KlasaBazowa):
    pass
```

- Dziedziczenie umożliwia ponowne wykorzystanie funkcjonalności klas bazowych w klasach pochodnych.

Dziedziczenie - przykład

In [1]:
```
class Student:
    last_index = 0 # atrybut klasy
    def __init__(self, name):
        Student.last_index += 1  # update numer indeksu
        self.name = name # imię
        self.index = Student.last_index # numer indeksu

    def __str__(self):
        return "Student: {} (nr {})".format(self.name, self.index)

class SMatematyk(Student):
    def __init__(self, name):
        Student.last_index += 1  # update numer indeksu
        self.name = name # imię
        self.index = Student.last_index # numer indeksu
        self.kierunek = 'Matematyka'

    def __str__(self):
        return "Student matematyki: {} (nr {})".format(self.name, self.index)

a = Student('Adam')
b = SMatematyk('Basia')
print(a)
print(b)
```

```
Student: Adam (nr 1)
Student matematyki: Basia (nr 2)
```

Dziedziczenie - przykład

- Możliwe jest wywołanie konstruktora klasy nadrzędnej, służy do tego metoda wbudowane `super()`:

In [1]:
```
class Student:
    last_index = 0 # atrybut klasy
    def __init__(self, name):
        Student.last_index += 1  # update numer indeksu
        self.name = name # imię
        self.index = Student.last_index # numer indeksu

    def __str__(self):
        return "Student: {} (nr {})".format(self.name, self.index)

class SMatematyk(Student):
    def __init__(self, name):
        super(SMatematyk, self).__init__(name)
        # można krócej: super().__init__(name)
        self.kierunek = 'Matematyka'

    def __str__(self):
        return "Student matematyki: {} (nr {})".format(self.name, self.index)

a = Student('Adam')
b = SMatematyk('Basia')
print(a.__dict__)
print(b.__dict__)
```

```
{'name': 'Adam', 'index': 1}
{'name': 'Basia', 'index': 2, 'kierunek': 'Matematyka'}
```

Polimorfizm

- Ta sama metoda, różne działanie zależne od typu obiektu.

In [1]:
```
class Kot:
    def glos(self):
        print("Miau")
class Pies:
    def glos(self):
        print("Hau")
class Krowa:
    def glos(self):
        print("Muu")
class Ryba:
    pass # brak metody glos

for zwierze in [Kot(), Pies(), Krowa(), Ryba()]:
    zwierze.glos() # za każdym razem inny typ
```

```
Miau
Hau
Muu

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-1-f1c48142b733> in <module>()
     12
     13 for zwierze in [Kot(), Pies(), Krowa(), Ryba()]:
---> 14     zwierze.glos() # za każdym razem inny typ

AttributeError: 'Ryba' object has no attribute 'glos'
```

Polimorfizm - wymuszanie interfejsu

In [1]:
```
class Zwierze:
    def glos(self):  # pusta metoda glos
        pass

class Kot(Zwierze):
    def glos(self):
        print("Miau")

class Pies(Zwierze):
    def glos(self):
        print("Hau")

class Krowa(Zwierze):
    def glos(self):
        print("Muu")

class Ryba(Zwierze):
    pass # brak metody glos

for zwierze in [Kot(), Pies(), Krowa(), Ryba()]:
    zwierze.glos() # za każdym razem inny typ
```

```
Miau
Hau
Muu
```

Polimorfizm - większe wymuszanie interfejsu

In [1]:
```
class Zwierze:
    def glos(self):  # pusta metoda glos
        raise NotImplementedError("Każde zwiesze musi mieć głos.")
class Kot(Zwierze):
    def glos(self):
        print("Miau")
class Pies(Zwierze):
    def glos(self):
        print("Hau")
class Krowa(Zwierze):
    pass

for zwierze in [Kot(), Pies(), Krowa()]:
    zwierze.glos()
```

```
Miau
Hau

---------------------------------------------------------------------------
NotImplementedError                       Traceback (most recent call last)
<ipython-input-1-01878cbdab9b> in <module>()
     12
     13 for zwierze in [Kot(), Pies(), Krowa()]:
---> 14     zwierze.glos()

<ipython-input-1-01878cbdab9b> in glos(self)
      1 class Zwierze:
      2     def glos(self):  # pusta metoda glos
----> 3         raise NotImplementedError("Każde zwiesze musi mieć głos.")
      4 class Kot(Zwierze):
      5     def glos(self):

NotImplementedError: Każde zwiesze musi mieć głos.
```

Dziedziczenie wielobazowe

- Możliwe jest dziedziczenie po więcej niż jednej klasie.

In [1]:
```
class Samochod:
    def naprzod(self):
        print('Jadę')
    def jedz(self):
        print('Jadę')

class Lodz:
    def naprzod(self):
        print('Płynę')
    def plyn(self):
        print('Płynę')

class Amfibia(Samochod, Lodz):
    pass

pojazd = Amfibia()
pojazd.jedz()
pojazd.plyn()
pojazd.naprzod()

class Amfibia(Lodz, Samochod):
    pass

pojazd = Amfibia()
pojazd.naprzod()
```

```
Jadę
Płynę
Jadę
Płynę
```

Uwagi do dziedziczenia

- Jeśli klasa podrzędna definiuje atrybut o takiej samej nazwie, jaką ma atrybut jej klasy nadrzędnej, to instancje klasy podrzędnej korzystają z jej atrybutów, chyba że atrybut jest jawnie kwalifikowany za pomocą nazwy klasy nadrzędnej (z operatorem kropki).
- Wewnątrz instancji klasy podrzędnej Python szuka nazw atrybutów w następującej kolejności:
  - *przestrzeń nazw instancji* - jest ona dostępna poprzez argument `self` i zawiera zmienne instancji, zmienne prywatne instancji oraz zmienne instancji klasy nadrzędnej,
  - *przestrzeń nazw klasy* - zawiera ona metody, zmienne klasy, metody prywatne i prywatne zmienne klasy,
  - *przestrzeń nazw klasy nadrzędnej* - zawiera ona metody klasy nadrzędnej, zmienne klasy nadrzędnej, metody prywatne klasy nadrzędnej i prywatne zmienne klasy nadrzędnej.

Obiektowość - Przykład - Budujemy klasę `Vector`

Stworzymy implementację klasy Vector, której obiekty będą zachowywać się jak wektory w przestrzeni $\mathbb{R}^n$. W szczególności chcemy aby była możliwość:

- tworzenia wektorów o różnych wymiarach (o zadanych wartościach),
- dodawania, odejmowania wektorów,
- mnożenia wektora przez liczbę (skalowanie),
- obliczenie długości wektora (norma),
- wykonania iloczynu skalarnego wektorów,
- porównywania wektorów,
- obliczenia kąta między wektorami,
- sprawdzenia czy wektory są równoległe, prostopadłe,
- wygenerowania losowego wektora o zadanym wymiarze,
- ...