Școala
informală
de IT

Google

Python Development

Week 5. Object oriented programming

Cuprins

1. Programare orientată pe obiecte
2. Decoratori
3. Clase și obiecte
4. Iteratori

Programare orientată pe obiecte

Week 5. Object oriented programming

Programare orientată pe obiecte - date generale

- Programarea orientată pe obiecte reprezintă o paradigmă de programare ce are ca scop organizarea dezvoltării software în jurul datelor, a obiectelor, în locul funcțiilor.
- Un obiect este definit ca fiind o dată care are atribute și comportament unice.
- Pe scurt, programarea orientată pe obiecte se concentrează pe obiectele pe care programatorul le gestionează, în loc să se concentreze pe logica necesară gestionării lor.
- Acest mod de programare se aplică cu precădere aplicațiilor mari, complexe și/sau care au nevoie permanentă de actualizare sau mentenanță
- Datorită organizării unei aplicații orientată pe obiecte, acestea pot conduce la un mod de dezvoltare colaborativă. O astfel de aplicație poate fi divizată în mai multe grupuri, fiecare dintre ele fiind dezvoltat/actualizat de o echipă diferită.

Programare orientată pe obiecte - date generale

- Primul pas în programarea orientată pe obiecte o reprezintă **data modeling**-ul.
- Acest pas face referire la identificarea tuturor obiectelor implicate în sistem și modul în care acestea interacționează între ele.
- O dată ce obiectele sunt identificate, acestea sunt generalizate sub formă de clasă de obiecte care definește tipul de date respectiv și toată logica necesară gestionării lor.
- În Python, orice tip dată este reprezentată ca un obiect. Tipurile de dată built-in sunt bazate pe clasa **Type**, clasă pe care o moștenesc. Pentru a exemplifica acest lucru puteți folosi metoda **type()**.

```
print(type(int))
print(type(float))
print(type(complex))
print(type(str))
print(type(list))
print(type(tuple))
print(type(set))
print(type(dict))
```

Programare orientată pe obiecte - principii

- Programarea orientată pe obiecte are la bază câteva principii ce trebuie cunoscute înainte de a ne apuca de dezvoltarea unei aplicații bazate pe POO.
- **Încapsularea**
  - implementarea și starea fiecărui obiect sunt gestionate privat, în interiorul unei clase.
  - alte obiecte nu au acces la alte clase și nici dreptul de a schimba starea altor obiecte, dar au acces la o listă de metode publice.
  - acest mod de a ascunde datele oferă o securitatea sporită și reduce riscul de a corupe datele într-un mod neintenționat.
- **Abstractizarea**
  - se referă la expunerea mecanismelor interne către exterior numai în cazul în care acestea sunt relevante, ascunzând orice implementare de cod care nu este necesară.
  - ajută programatorii să facă modificări în timp mult mai ușor.

Programare orientată pe obiecte - principii

- **Moștenirea**
  - o clasă este fie o sublclasă fie o superclasă a unei alte clase cu care împarte aceleași proprietăți.
  - ajută la refolosirea oferind, în același timp, o structură bine definită.
- **Polimorfismul**
  - obiectele au posibilitatea să aibă mai multe forme în funcție de contextul în care sunt folosite.
  - aplicația va decide care formă o are un obiect în fiecare moment al execuției, reducând necesitatea duplicării codului.
  - poate fi obținut prin Method Overriding și/sau Method Overloading. În Python, Method Overloading-ul nu este posibil.

Programare orientată pe obiecte - avantaje

- **Modularitatea**
  - este mult mai ușor de intervenit în cazul unor eventuale defecte.
  - datorită încapsulării, fiecare clasă conține funcționalitate specifică unui anumit tip de obiect. Dacă unul din obiecte are un comportament neașteptat, atunci acel obiect are o problemă.
  - tot datorită încapsulării, mai mulți programatori pot lucra concomitent la obiecte diferite fără riscul de a duplica funcționalități.
- **Reutilizarea codului (code reusability)**
  - datorită moștenirii și abstractizării, funcționalitatea comună mai multor obiecte care împart caracteristici asemănătoare poate fi refolosită.
  - exemplu: o parte a aplicației are nevoie de un obiect care reprezintă o mașină de curse, iar o altă parte are nevoie de un obiect care reprezintă o mașină de teren. Din moment ce ambele împart anumite caracteristici specifice ale unei superclase **Car**, nu este nevoie de duplicarea acestei funcționalități.
- **Flexibilitatea oferită de polimorfism.**
- **Rezolvarea problemelor într-un mod gradual**
  - presupune spargerea unei probleme complexe în bucăți. Acestă practică este benefică pentru că oferă avantajul de a rezolva fiecare problemă în parte, lucrând în același timp la rezolvarea problemei per ansamblu.

Decoratori

Week 5. Object oriented programming

Decoratori

- Un decorator este un concept Python care permite extinderea funcționalității unei funcții, metode sau clase fără a interveni asupra structurii acesteia.
- Funcțiile sunt cetățeni de clasa I (**first class citizens**) - suportă orice operație de bază: poate fi trimisă ca argument, returnată dintr-o funcție, modificată sau asignată unei variabile.

```
def my_initial_function(msg):
    print(msg)


def another_function(function_param):
    function_param("Hello, World!")


another_function(my_initial_function)
```

```
def my_initial_function(msg):
    print(msg)


def another_function():
    return my_initial_function


another_function()("Hello, World!")
```

```
def my_initial_function(msg):
    print(msg)


my_var = my_initial_function


my_var("Hello, World!")
```

- Combinând aceste informații cu cele anterioare legate de nested functions putem declara primul nostru decorator.

Decoratori

- Decoratorii pot fi folosiți atât fără parametri cât și cu parametri.
- În următorul exemplu avem un decorator care primește o funcție și returnează rezultatul inițial ridicat la pătrat. Acesta este un exemplu pentru un decorator care nu primește parametrii.

```
def sum_decorator(custom_function):
    def wrapper(a, b):
        result = custom_function(a, b)
        return result ** 2

    return wrapper


def sum_function(a, b):
    return a + b


decorated_function = sum_decorator(sum_function)
print(decorated_function(2, 3))
```

```
def sum_decorator(custom_function):
    def wrapper(a, b):
        result = custom_function(a, b)
        return result ** 2

    return wrapper


@sum_decorator
def sum_function(a, b):
    return a + b


print(sum_function(2, 3))
```

Decoratori

- În următorul exemplu avem un decorator care primește o funcție și returnează rezultatul inițial la puterea unui alt parametru primit de decorator.

```
def sum_decorator_with_param(param_number):
    def sum_decorator(custom_function):
        def wrapper(a, b):
            result = custom_function(a, b)
            return result ** param_number

        return wrapper

    return sum_decorator


def sum_function(a, b):
    return a + b


decorated_function = sum_decorator_with_param(3)(sum_function)
print(decorated_function(2, 3))
```

```
def sum_decorator_with_param(param_number):
    def sum_decorator(custom_function):
        def wrapper(a, b):
            result = custom_function(a, b)
            return result ** param_number

        return wrapper

    return sum_decorator


@sum_decorator_with_param(param_number=3)
def sum_function(a, b):
    return a + b


print(sum_function(2, 3))
```

Clase și obiecte

Week 5. Object oriented programming

Clase și obiecte

- O clasă reprezintă șablonul unui obiect. Rolul ei este să definim structura și comportamentul unui obiect (folosind atribute și metode).
- Un obiect reprezintă o instanță a unei clase. Fiecare obiect poate propriile date, dar vor avea aceeași structură și comportament cu alte obiecte create pe baza aceeași clase.

```
class MyFirstPythonClass:
    pass


my_first_python_object = MyFirstPythonClass()
```

- În exemplul anterior aveți modul în care se declară o clasă (**MyFirstPythonClass**) și modul în care se creează un obiect (**my_first_python_object**)
- Pentru declararea unei clase vom folosi keyword-ul **class** urmat de numele clasei. Numele unei clase trebuie să fie intuitiv și să fie scris folosind convenția **CapWords**.
- Pentru declararea unui obiect trebuie instanțiată clasa folosind constructorul acesteia.

Clase și obiecte

```
class Dog:
    legs_no = 4

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

    def __str__(self):
        return '%s %s' % (type(self), self.name)

    def change_name(self, name):
        self.name = name

    @staticmethod
    def speak():
        print('Bark! Bark!')


print(Dog.legs_no)  # will print 4

my_dog = Dog('Rex')
print(my_dog)  # will print <class '__main__.Dog'> Rex
print(my_dog.name)  # will print Rex
print(my_dog.legs_no)  # will print 4

my_dog.change_name('Ben')
print(my_dog)  # will print <class '__main__.Dog'> Ben
print(my_dog.name)  # will print Ben

my_dog.speak()  # will print Bark! Bark!
Dog.speak()  # will print Bark! Bark!
```

- În exemplul alăturat puteți vedea definită o clasă reprezentând un câine. Această clasă dispune de următoarele:
  - atribute:
    - **legs_no** - acest atribut este static - este declarat la nivel de clasă. Poate fi accesat prin dot notation.
  - metode:
    - **__init__** - este un constructor. Rolul lui este să ne ajute să creăm obiecte cu date diferite.
    - **__str__** - este o metodă ce are rolul să definească o interpretare a obiectului. Este folosită când se dorește afișarea unui obiect sau convertirea lui la string.
    - **change_name** - este o metodă a clasei ce definește un comportament al obiectului. Ex: unui câine i se poate schimba numele.
    - **speak** - este o metodă statică, deoarece nu se bazează pe o anumită instanță a clasei.

Clase și obiecte

```python
class Dog:
    legs_no = 4

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

    def __str__(self):
        return '%s %s' % (type(self), self.name)

    def change_name(self, name):
        self.name = name

    @staticmethod
    def speak():
        print('Bark! Bark!')


print(Dog.legs_no)  # will print 4

my_dog = Dog('Rex')
print(my_dog)  # will print <class '__main__.Dog'> Rex
print(my_dog.name)  # will print Rex
print(my_dog.legs_no)  # will print 4

my_dog.change_name('Ben')
print(my_dog)  # will print <class '__main__.Dog'> Ben
print(my_dog.name)  # will print Ben

my_dog.speak()  # will print Bark! Bark!
Dog.speak()  # will print Bark! Bark!
```

- În același exemplu, în partea de jos, puteți observa modalitățile de folosire ale celor discutate anterior.
- Atributele statice pot fi folosite fără a avea nevoie de o instanță (obiect), folosind direct numele clasei. Același lucru este valabil și pentru metodele statice. Acest lucru este posibil pentru că acestea nu depind de o anumită instanță a clasei.
- Folosirea constructorului se face prin **NumeClasa(parametrii)**, în acest caz, atributul unui obiect fiind doar **name**.
- Există acces direct la proprietățile unui obiect (my_dog.name)
- Există acces direct la metodele unui obiect (my_dog.change_name)
- Putem accesa atributele și metodele statice și prin intermediul instanței.

Clase și obiecte

- În majoritatea limbajelor de programare există noțiunea de de atribute și metode protected (pot fi folosite doar în interiorul aceluiași pachet/modul) sau private (pot fi folosite doar în interiorul clasei respective).
- Acest lucru, print convenție poate fi obținut și în Python:
  - un atribut/metodă protected poate fi obținută prin prefixarea numelui cu _ (underscore).
  - un atribut/metodă private poate fi obținută prin prefixarea numelui cu __ (dublu underscore).
- În realitate acest lucru nu este posibil în Python pentru că atât un atribut/o metodă protected sau private poate fi accesată din-afară. În cazul protected acestea pot fi accesate fară probleme, în timp ce în cazul celor private, Python le schimbă numele la compilare folosind ***_ClassName__name***.
- Totuși ne putem folosi de acest lucru și de decoratorul **property**, dar trebuie să reținem că Python nu ascunde date. Dacă cineva chiar vrea să acceseze o variabilă sau o metodă...o va face :)

Clase și obiecte

```python
class Dog:
    def __init__(self, name):
        self.__name = name

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

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

    @name.deleter
    def name(self):
        del self.__name


my_dog = Dog(name='Rex')
print(my_dog.name)  # will print Rex

my_dog.name = 'Ben'
print(my_dog.name)  # will print Ben

del my_dog.name
# The following print will raise an
# AttributeError: 'Dog' object has no attribute '_Dog__name'
print(my_dog.name)
```

- Pentru a obține un comportament private pentru atributele noastre ne vom folosi de notația cu __, dar și de avantajele oferite de decoratorul **property**. Acest decorator ne ajută să folosim o metodă cu rol de atribut.
- Cu ajutorul acestuia putem:
  - declara getter, setter, deleter
  - avea control total asupra modului de accesare, setare și ștergere a datelor.
- Cu toate acestea, nu uitați că oricând datele pot fi accesate din exterior.

Clase și obiecte

```
class Cat:
    legs_no = 4

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

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

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

    @name.deleter
    def name(self):
        del self.__name

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

    @staticmethod
    def speak():
        print('Meow! Meow!')
```

- Tot cu ajutorul unei clase putem reprezenta și o pisică.
- Din câte se poate observa în exemplul acesta singurele diferențe dintre clasa **Cat** și clasa **Dog** o reprezintă metoda speak.
- În rest, ambele clase folosesc aceeași proprietate **name**.
- Codul care se repetă în cele două clase este enorm, iar dacă intervine necesitatea unei modificări aceasta va trebui făcută în mai multe locuri.
- **Aici intervin moștenirea și abstractizarea!**

Clase și obiecte

```
class QuadrupedAnimal:
    legs_no = 4

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

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

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

    @name.deleter
    def name(self):
        del self.__name

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

    @staticmethod
    def speak():
        pass
```

- Pentru a evita cod duplicat putem crea o clasă părinte, **QuadrupedAnimal**, care să conțină toate elementele comune ale animalelor patrupede:
  - legs_no
  - proprietatea name
  - metoda de schimbare a numelui
  - definirea unei metode ce necesită implementare individuală. În Python nu există definirea unei interfețe - se aplică conceptul de duck typing (dacă merge ca o rață și măcăne ca o rață atunci este clar o rață) - nu există obligativitatea implementării unei metode dintr-o interfață.

Clase și obiecte

```
class Dog(QuadrupedAnimal):
    @staticmethod
    def speak():
        print('Bark! Bark!')


class Cat(QuadrupedAnimal):
    @staticmethod
    def speak():
        print('Meow! Meow!')


class Cow(QuadrupedAnimal):
    @staticmethod
    def speak():
        print('Muu! Muu!')
```

- În imaginea alăturată avem definirea unor animale patrupede specifice, dar care moștenesc clasa **QuadrupedAnimal**.

```
my_dog = Dog(name='Rex')
my_cat = Cat(name='Kitty')
my_cow = Cow(name="Cow name :))")
```

- În imaginea de mai sus observăm instanțierea unor obiecte din aceste clase.

Clase și obiecte

```python
class QuadrupedAnimal:
    legs_no = 4

    @staticmethod
    def speak():
        pass


class Pet:
    def __init__(self, name):
        self.__name = name

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

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

    @name.deleter
    def name(self):
        del self.__name

    def change_name(self, name):
        self.__name = name
```

- Pentru ca unele animale pot fi domestice, dar nu neaparat animale de companie (ex: vaca), vom rupe o parte din funcționalitatea clasei **QuadrupedAnimal** astfel încât să păstrăm doar ceea ce ține strict de animalele patrupede.
- Funcționalitatea ce ține de numele animalului o vom muta în clasa **Pet**.

```python
class Dog(QuadrupedAnimal, Pet):
    @staticmethod
    def speak():
        print('Bark! Bark!')


class Cat(QuadrupedAnimal, Pet):
    @staticmethod
    def speak():
        print('Meow! Meow!')


class Cow(QuadrupedAnimal):
    @staticmethod
    def speak():
        print('Muu! Muu!')
```

- Din câte se observă în implementarea claselor **Dog** și **Cat** acestea vor moșteni atât clasa **QuadrupedAnimal** cât și clasa **Pet**.
- În cazul în care o clasă moștenește mai multe clase care au atribute/metode ce conțin același nume se va infera atributul/metoda din clasa care se afla prima în lista de clase moștenite.

Iteratori

Week 5. Object oriented programming

Iteratori

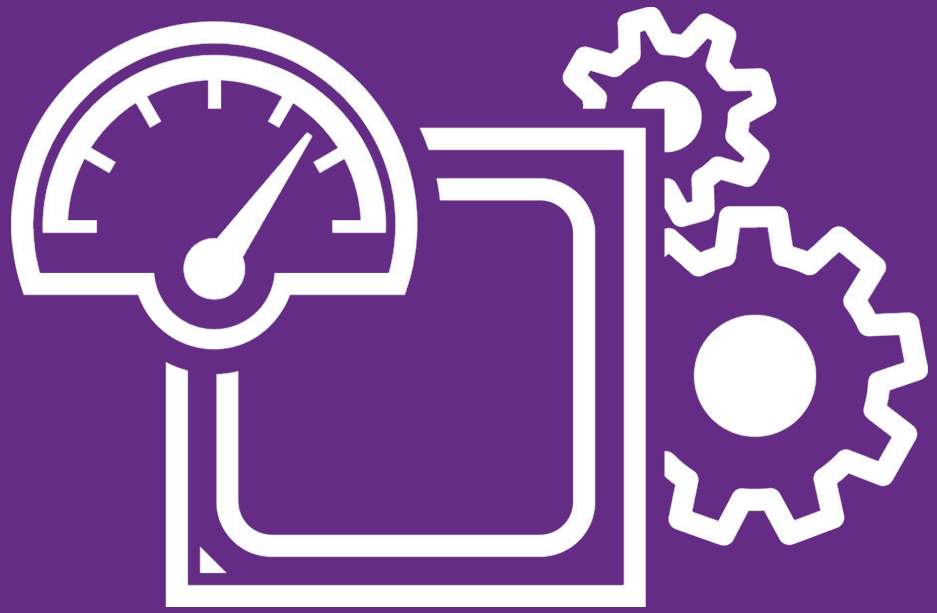
- Un iterator este un obiect care conține o mulțime finită de valori.
- Un iterator poate fi iterat, adică se pot parcurge, în ordine, toate valorile sale.
- În Python, orice clasă care implementează protocolul unui iterator reprezintă un iterator. Protocolul presupune implementarea metodelor **__iter__()** și **__next__()**.
- **Atenție!** Listele, tuplurile, dicționarele și seturi sunt obiecte ce pot fi iterate din care se poate obține un iterator.
- În exemplul următor aveți reprezentat un iterator custom care parcurge numerele din șirul Fibonacci.

```
class FibonacciIterator:
    def __iter__(self):
        self.value = 1
        self.prev = 0
        return self

    def __next__(self):
        value = self.value
        self.value += self.prev
        self.prev = value

        return value
```

Iteratori

- În acest moment, iterarea se poate face la infinit. Pentru a opri interarea infinită trebuie să condiționăm codul din metoda **__next__**. În momentul în care decidem să oprim iterarea vom arunca **StopIteration**.

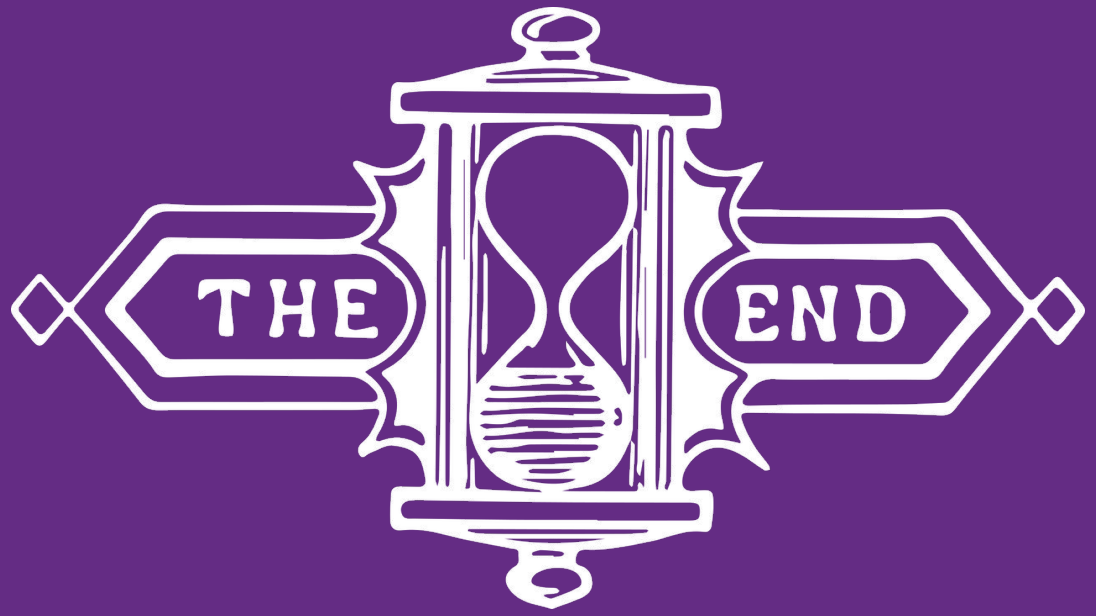
```
class FibonacciIterator:
    def __init__(self, n):
        self.n = n

    def __iter__(self):
        self.count = 0
        self.value = 1
        self.prev = 0
        return self

    def __next__(self):
        if self.count < self.n:
            value = self.value
            self.value += self.prev
            self.prev = value
            self.count += 1

            return value
        else:
            raise StopIteration


fibonacci_instance = FibonacciIterator(10)
fibonacci_iterator = iter(fibonacci_instance)

for number in fibonacci_iterator:
    print(number)
```



Temă

Temă

- Să se scrie o clasă Fractie(numarator, numitor) care sa implementeze următoarele metode:
  - **__init__** : instanțiem numărător și numitor
  - **__str__** : afisam "numărător/numitor"
  - **__add__** : returnam o noua fractie care reprezinta adunarea
  - **__sub__**: returnam o nouă fracție care reprezinta scădearea
  - **inverse**: returnează o nouă fracție (inversa fracției)



Vă mulțumesc!