

Python 101

Curs 3 - Programare Orientată Obiect

Programare orientată obiect

- structura fundamentală în jurul căreia se rezolvă o problemă este obiectul
- python propune o structură simplificată a paradigmei, adaptată stilului de scripting

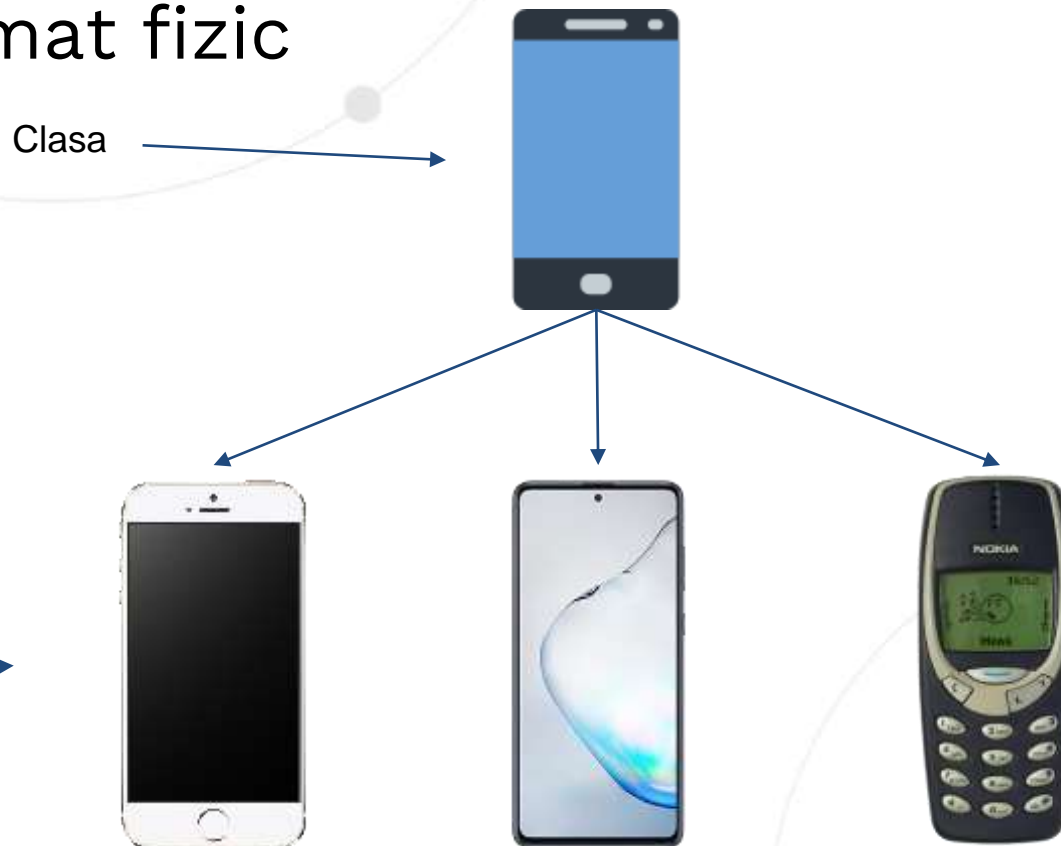
Clasă

- este entitatea abstractă care descrie un obiect și definește caracteristicile lui



Instanță

- este concretizarea unei clase
- **instanța**, numită **obiect**, poate exista în format fizic



PОО in Python

```
class Phone:  
    pass
```



iphone7

= Phone()



#

note10

= Phone()



#

nokia3310

= Phone()



#

Constructor

Constructorul

- este o funcție specială care se apelează la crearea unei instanțe a unei clase și se folosește, de obicei, pentru a inițializa variabilele
- în Python, constructorul se numește **`__init__`**:

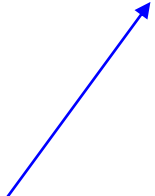
```
class Cat:  
    def __init__(self):  
        pass
```

Self

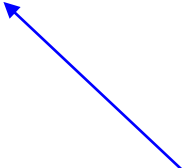
Cuvântul cheie **self** reprezintă o referință către instanța unei clase.

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

Valoarea campului
obiectului



Valoarea parametrului
constructorului



Self (2)

- prin folosirea lui **self** avem acces la attributele și metodele obiectului unei clase.

```
class Cat:

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

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


Constructorul (2)

- un constructor, asemenea unei funcții, poate primi parametrii
- in Python, o clasă poate avea **un singur constructor** cu parametrii care se comportă la fel ca parametrii funcțiilor (cu excepția self, care apare obligatoriu)

```
class Cat:  
    def __init__(self, cat_age, owner="Fred"):  
        self.age = cat_age  
        self.owner = owner
```

Construirea unui obiect

- putem crea o instanță a unui obiect prin apelarea constructorului cu parametrii necesari pentru a putea inițializa attributele clasei

```
class Cat:
    def __init__(self, cat_age, owner= "Fred"):
        self.age = cat_age
        self.owner = owner
```

```
garfield = Cat(1)
rio = Cat(2, "Tom")
```

Atributul

- pentru a diferenția între obiecte, trebuie să descriem niște caracteristici ale lor, numite **atribute**, la care avem acces prin: **instanță.atribut**

```
class Cat:
    def __init__(self, cat_age, owner):
        self.age = cat_age
        self.owner = owner
```

```
rio = Cat(2, "Tom")
print(f"{rio.owner} has a {rio.age}-year old cat")

# Tom has a 2-year old cat.
```

Metode

- în interiorul unei clase se pot defini si funcții, numite metode
- o metodă se apelează, similar funcțiilor, prin:

instanță.metodă(parametrii)

Metode (2)

```
class Cat:
    def __init__(self, age, owner="Fred"):
        self.age = age
        self.owner = owner

    def increase_age(self):
        self.age += 1

    def change_owner(self, new_owner):
        self.owner = new_owner

rio = Cat(2, "Tom")
rio.increase_age()
rio.change_owner("Harry")
print(f"{rio.owner} has a {rio.age}-year old cat")
# Harry has a 3-year old cat.
```

Atribute statice

- există și atribute care pot fi comune tuturor instanțelor
- ele pot fi accesate cu:

numeClasă.numeVariabilă

```
class Car:
    no_cars = 0
    def __init__(self):
        Car.no_cars += 1

toyota = Car()
ford = Car()
print(Car.no_cars) #      2
```

The background of the slide is white with decorative blue elements. On the left, a large blue circle is partially visible. A thin, light blue arc connects a small pink dot on the left circle to a small grey dot on the right. Another thin, light blue arc is visible at the bottom right, with a small grey dot on it. A small blue circle is also visible in the bottom right corner.

Întrebări?

The background features abstract geometric shapes. On the left, a large blue circle is partially visible. A thin, light gray arc curves across the middle of the slide, with a small gray dot on it. In the bottom right corner, another blue circle is partially visible. The word 'Pauzã' is centered on the right side of the slide.

Pauzã

Supraîncărcare

- reprezintă proprietatea unei metode de a se comporta diferit în anumite situații
- în interiorul unei clase, putem defini cum să se comporte diferiți operatori pe instanțele clasei

__str__

- pentru a afișa o instanță a unei clase, trebuie definită metoda **__str__**, care va întoarce un șir cu reprezentarea clasei

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

    def __str__(self):
        return f"{self.x} + {self.y}j"

c1 = ComplexNumber(2, 3)
print(c1)           # 2 + 3j
```

Supraîncărcare +

- supraîncărcarea operatorului + se realizează prin definirea metodei **`__add__`**
- întoarce rezultatul adunării a două obiecte de tipul clasei respective

Supraîncărcare +

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

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

    def __str__(self):
        return f"{self.x} + {self.y}j"

c1 = ComplexNumber(2, 3)
c2 = ComplexNumber(4, 5)
print(c1 + c2)
```

#6 + 8j

Supraîncărcare =

- pentru a verifica egalitatea dintre două instanțe ale unei clase, se supraîncarcă operatorul `=`, prin metoda **`__eq__`**

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

    def __eq__(self, other):
        return (self.x == other.x) and (self.y == other.y)

c1 = ComplexNumber(2, 4)
c2 = ComplexNumber(2, 3)
c3 = ComplexNumber(2, 4)
print(c1 == c2)           # False
print(c1 == c3)           # True
```

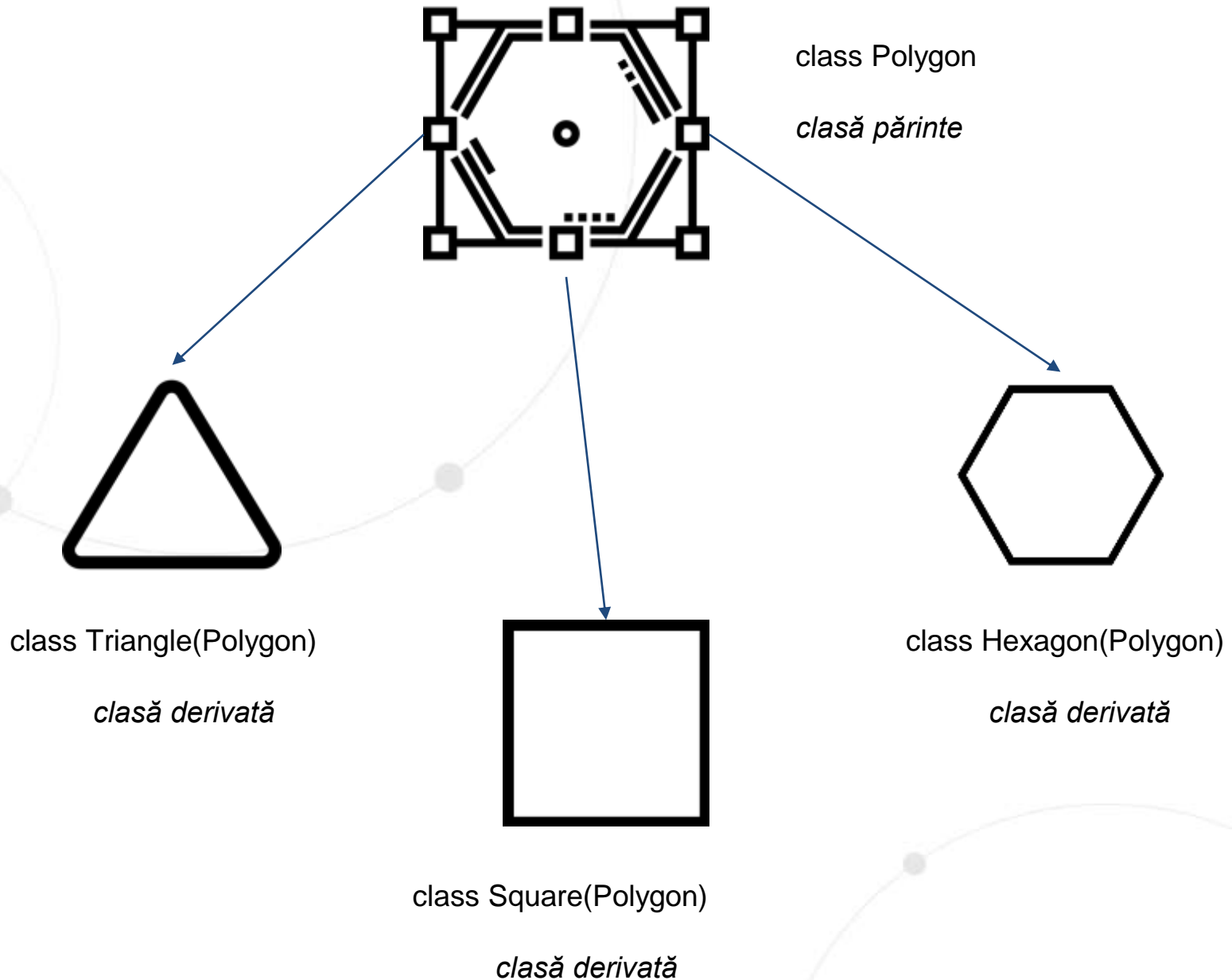
Supraîncărcare operatori

- alte exemple de supraîncărcare:

Operator	Expresie	Metodă
Mai mic	$p1 < p2$	<code>__lt__</code>
Mai mic sau egal	$p1 \leq p2$	<code>__le__</code>
Diferit de	$p1 \neq p2$	<code>__ne__</code>
Mai mare	$p1 > p2$	<code>__gt__</code>
Mai mare sau egal	$p1 \geq p2$	<code>__ge__</code>

Moștenire

- unul dintre conceptele fundamentale ale POO
- presupune definirea unei noi clase, ce poate extinde o clasă deja existentă
- noua clasă se numește **clasă derivată** (sau clasă-copil), iar clasa de la care moștenește se numește **clasă de bază** (sau clasă părinte)



Moștenire (2)

- pentru a moșteni o clasă folosim:
class ClasăDerivată(ClasăPărinte)

```
class Polygon:  
    def geometric_figure(self):  
        print("I am a polygon")
```

```
class Triangle(Polygon):  
    pass
```

```
class Square(Polygon):  
    Pass
```

Moștenire (3)

- o clasă moștenită are **toate** metodele și atributele clasei de bază

```
class Polygon:
    def geometric_figure(self):
        print("I am a polygon")
```

```
class Triangle(Polygon):
    pass
```

```
triangle = Triangle()
triangle.geometric_figure()      # I am a
polygon
```

Moștenire (4)

- o clasă moștenită poate avea metode în plus față de cele din clasa de bază

```
class Polygon:
    def geometric_figure(self):
        print("I am a polygon")

class Triangle(Polygon):
    def num_sides(self):
        print("I have 3 sides")

triangle = Triangle()
triangle.num_sides() # I have 3 sides
```

Suprascrierea

- este o proprietate a claselor derivate de a putea modifica comportamentul unor metode din clasa de bază

```
class Polygon:
    def geometric_figure(self):
        print("I am a polygon")
```

```
class Triangle(Polygon):
    def geometric_figure(self):
        print("I am a triangle")
```

```
triangle = Triangle()
triangle.geometric_figure()                # I am a triangle
```

Moștenire (5)

- putem accesa metodele din clasa de bază în clasa derivată folosind:

`super().nume_metodă(parametrii)`

```
class Polygon:
    def geometric_figure(self):
        print("I am a polygon")
```

```
class Triangle(Polygon):
    def geometric_figure(self):
        print("I am a triangle and also ", end='')
        super().geometric_figure()
```

```
triangle = Triangle()
triangle.geometric_figure() # I am a triangle and also I am a polygon
```

Moștenire (6)

- o bună practică e ca în constructorul clasei derivate să apelăm întotdeauna constructorul clasei de bază, pentru a face corect inițializările

`super().__init__(parametrii)`

```
class Polygon:
    def __init__(self):
        self.area = 0

class Triangle(Polygon):
    def __init__(self):
        super().__init__()
        self.sides = 3
```

Imutabilitate

- este proprietatea unui obiect (tip de date) de a nu putea fi modificat după ce este creat
- tipurile de date imutabile: `int`, `bool`, `str`, `tuple`
- tipuri de date mutabile: `list`, `set`, `dict`

Întrebări?

Nu uitati de feedback: [aici](#)