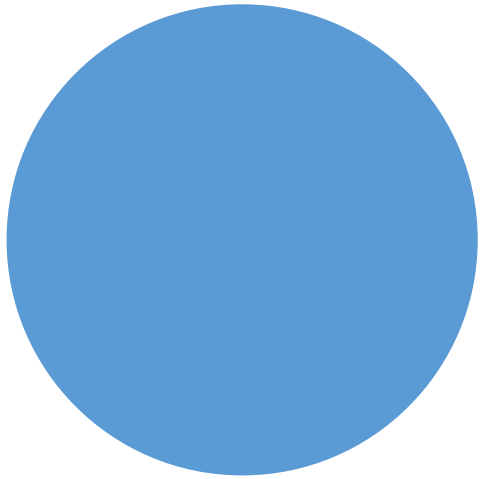
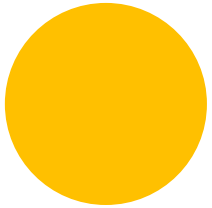



Programare 1

Clase, intro OOP

Cursul 10

Despre ce am discutat în cursul precedent?

Fisiere

- Operatii

Tipuri de fisiere

- Text
- JSON
- CSV
- binare

Despre ce o să discutăm astăzi?

- Programare orientată pe obiecte
- Clase
- Obiecte

Programare orientată pe obiecte

Python este un limbaj de programare orientat pe obiecte. **De ce ?**

Un limbaj de programare sau tehnică este considerată orientată pe obiecte dacă și doar dacă suportă direct:

- **Abstractizare**: oferă o formă de clase și obiecte
- **Moștenire**: oferă posibilitatea de a construii noi abstracții peste cele existente
- **Polimorfism dinamic**: oferă o posibilitatea de legare dinamică a claselor sau obiectelor

Programare orientată pe obiecte

- Terminologie:
  - **Abstractizare:** Posibilitatea de a adăuga noi tipuri de date (abstractizări)
  - **Moștenire:** Posibilitatea de a adăuga noi abstractizări peste cele deja existente
  - **Polimorfism:** Tratarea obiectelor în funcție de tipul de date
  - **Clase:**
    - Descrie unul sau mai multe obiecte
    - Un șablon pentru crearea (sau instanțierea) unor obiecte specifice în cadrul unui program
  - **Obiecte:**
    - O materializare a unei clase

Ce vedeti aici?

Abstractisation

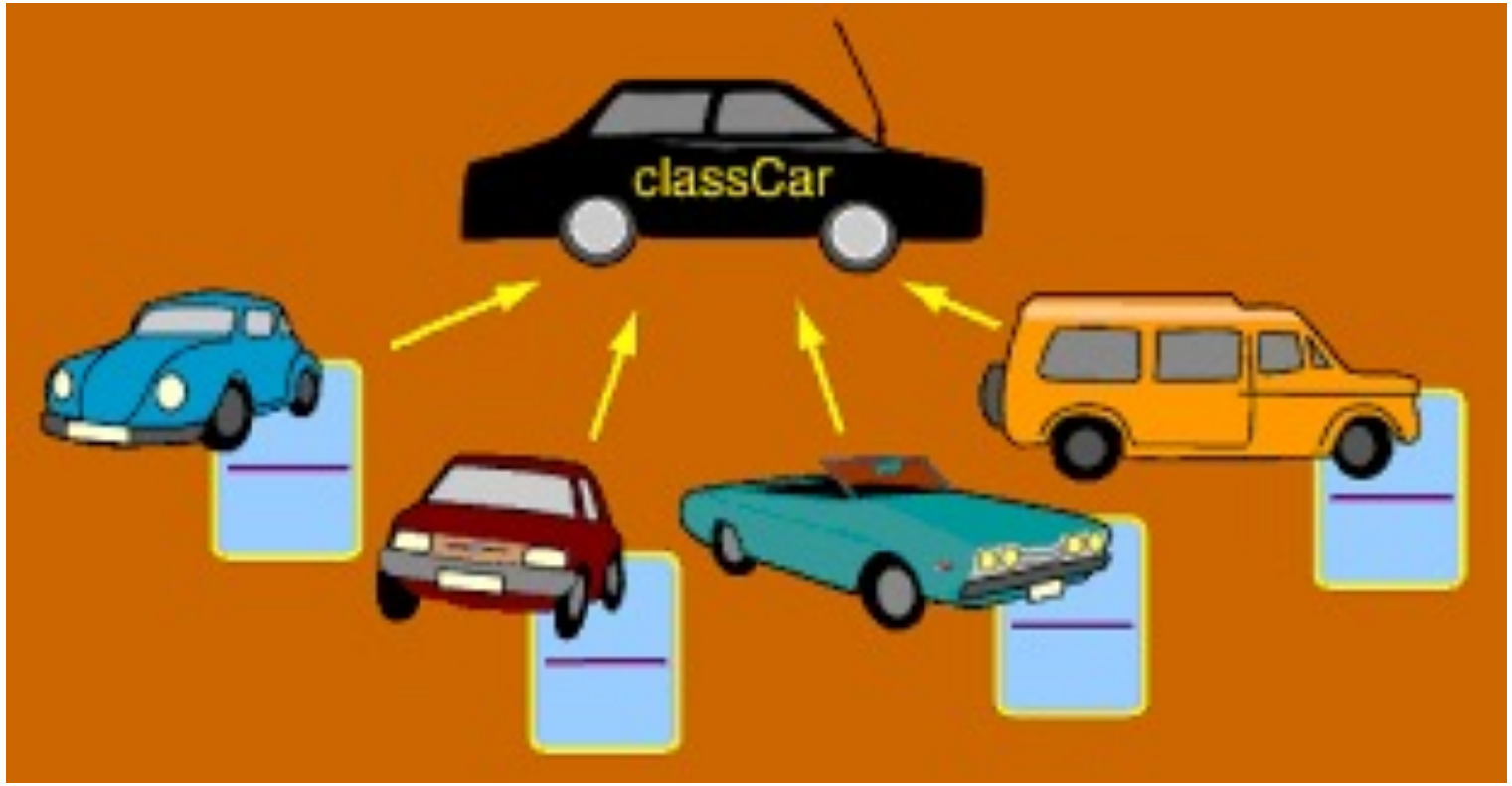
Idee nu este necesar să știi cum o funcție pentru a o folosi

O funcție este o „cutie neagră”, nu știm cum funcționează

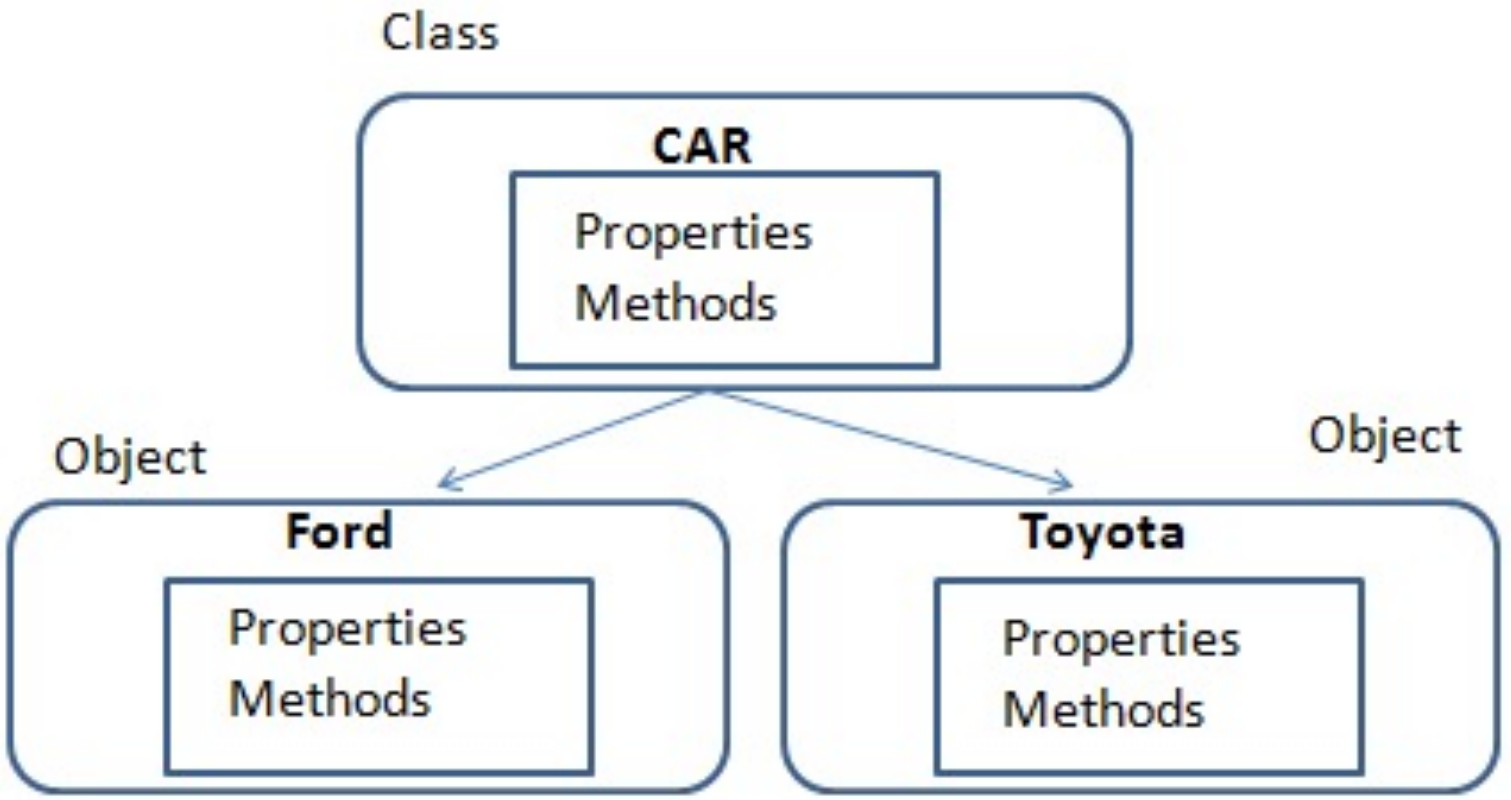
Cunoaștem „interfața” mașinii: cum se pornește/oprește

Cum funcționează această „cutie neagră” când apăsăm cheia?

Abstractizare poate chiar Mostenire



Clase si Obiecte!



Ascundem DETALII folosind ABSTRACTIZARE

- În programare, gândiți-vă la o bucată de cod ca la o cutie neagră
  - nu pot vedea detalii
  - nu trebuie sa vezi detalii
  - nu vreau sa vad detalii
  - ascunde detalii de codare plictisitoare

Cum se defineste o Clasa!

Class Name

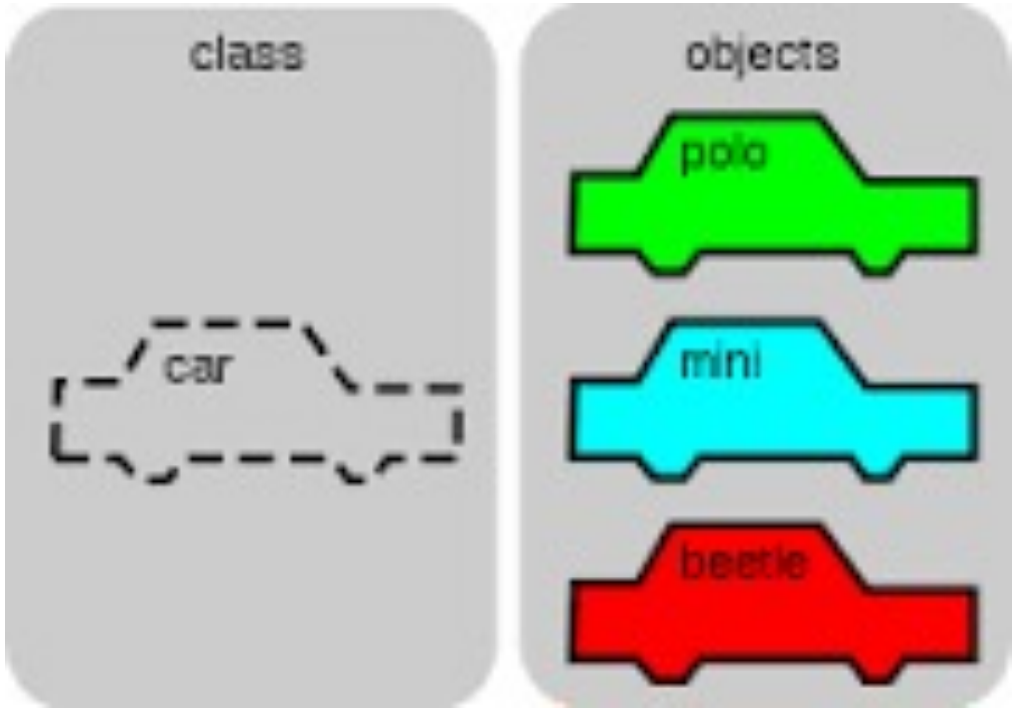
Attributes

Methods

Car
manufacturer color odometerReading …
drive rePaint fillWithGas …

Ce este un Obiect?

- DEFINITION [Object] An object is an **instance** of a class. It can be uniquely identified by its **name**, it defines a state which is represented by the values of its attributes at a particular time, and it exposes a behavior defined by the set of functions (methods) that can be applied to it.
- Un program OOP = colecție de obiecte care interacționează unele cu altele.



Obiecte

Exemple de obiecte Python:

- "Hello" # Un obiect de tipul str
- [1, 2, 3, 4] # Un obiect e tipul list
- {"Programare", "Curs"} # Un obiect de tipul set
- int # un obiect de tipul tip

Fiecare obiect este caracterizat de:

- Un **identificator unic**
- Un **tip**
- O **reprezentare internă**
- Un set de operații care permit **interacțiunea** cu informația stocată în obiect (acest set de operații mai poartă numele de **interfață**)

Ce putem face cu obiectele ?

Putem crea noi obiecte

Putem interacționa cu obiectele. Le putem manipula.

Putem distruge obiectele:

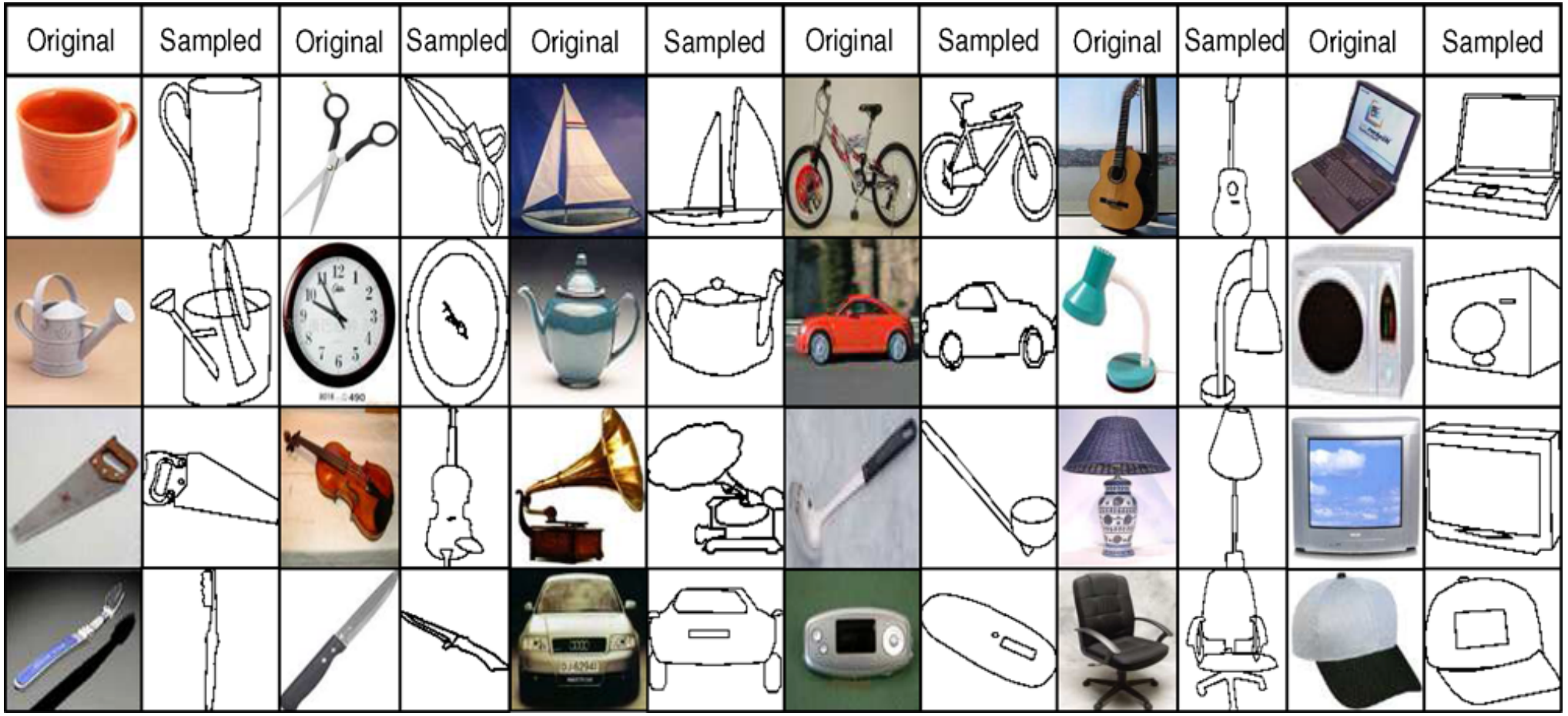
- explicit folosind instrucțiunea `del` sau putem pur și simplu să le ignorăm
- în a doua situație Python va șterge automat obiectele care nu mai sunt referite. Procesul este numit „garbage collection”

Exemple

- Capul unui om
  - Câte obiecte avem ?
  - Putem găsi o descriere care să se potrivească pentru toate ?
  - Există un tip în Python care să descrie corespunzător acest tip de obiect ?

Imagine generată cu http://www.picassohead.com/create.html

Obiecte vs Clase



Clase - Tipuri Proprii

Descriu obiecte similare

Definirea unei clase presupune:

- Definirea numelui clasei
- Definirea membrilor clasei
  - Atribute
  - Metode

Utilizarea claselor presupune:

- Crearea unei noi instanțe (obiect)
- Efectuarea de operații pe obiecte

Să ne definim propriul tip

- Ne folosim de cuvântul cheie `class`

Numele clasei, tip nou de date

Clasele părinte, poate fi omisă situație în care este considerat tipul `object`

Definiția clasei

```
class Coordonate (object):
    # aici definim membrii clasei
```

- Similar cu un cuvântul cheie `def`, indentăm codul pentru a indica care instrucțiuni sunt parte a definiției clasei
- Tipul `object` este tipul primar, iar clasa `Coordonate` moștenește toate atributele și metodele acestuia

Ce sunt membrii unei clase?

Date și metode care „aparțin” clasei

Membrii de date (sau atributele unei clase, sau câmpuri):

- date (variabile) care descriu clasa
- în cazul clasei `Coordonate` ar putea să fie latitudinea și longitudinea punctului pe glob

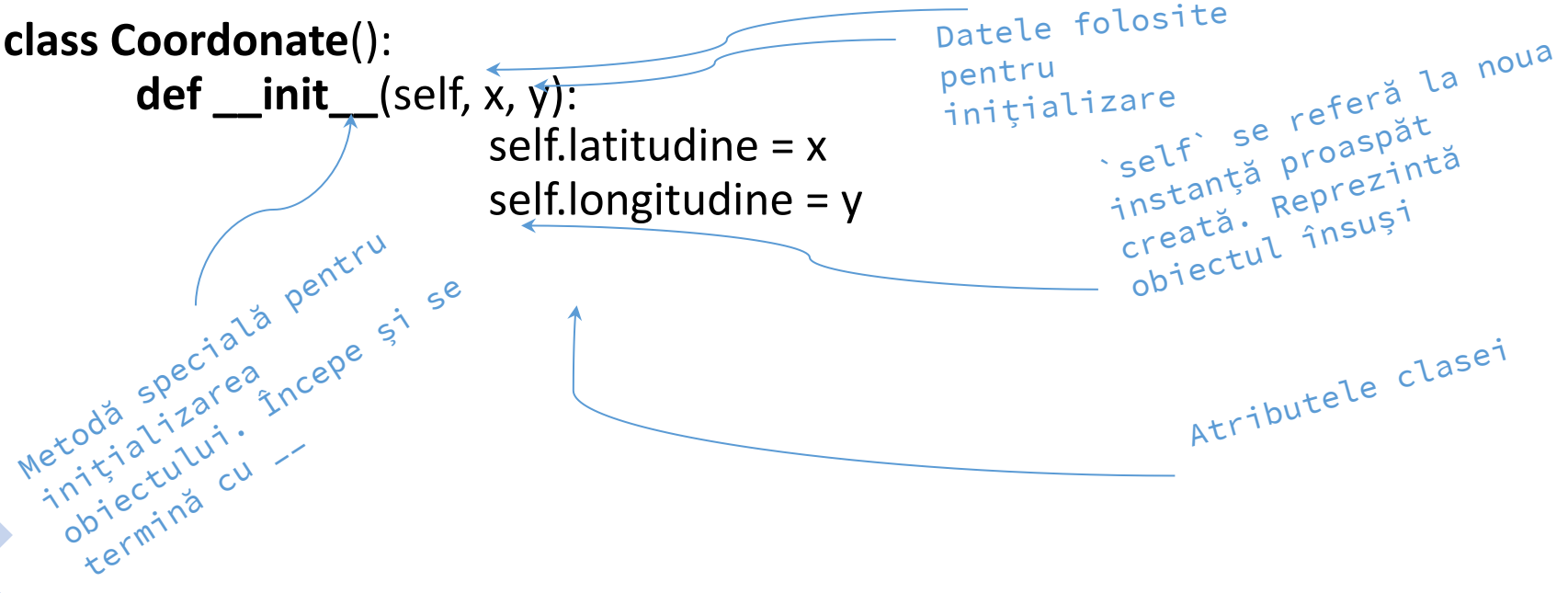
Metode (funcții membru):

- Ne permit să manipulăm datele salvate în clasă
- Permit interacțiunea cu alte obiecte
- Exemple:
  - Reprezentarea coordonatei ca o valoare reală, în grade, minute și secunde
  - Calcularea distanței între două puncte

Definirea modului de creare a unei instanțe

- Pentru a fi utilă trebuie să definim comportamentul unei clase
- O metodă specială `__init__` este folosită pentru inițializarea atributelor
  - Atenție: `__init__` nu este un constructor ci un inițializator

- **class Coordonate**():
  def __init__(self, x, y):
  self.latitudine = x
  self.longitudine = y



Definirea unei metode pentru clasa `Coordonate`

```
class Coordonate():
        def __init__(self, x, y):
                self.latitudine = x
                self.longitudine = y
        def distanta(self, alt_punct):
                x_diff = self.latitudine - alt_punct.latitudine
                y_diff = self.longitudine - alt_punct.longitudine
                return (x_diff**2+y_diff**2)**0.5
```

Folosit pentru a ne referii la instanța curentă

Parametru pentru metodă

Notație cu `.` (punct) pentru accesarea datelor

Cum ne folosim de noul tip ?

Metoda convențională:

```
c = Coordonate(45, 45)
zero = Coordonate(0, 0)
print(c.distanta(zero))
```

Obiectul folosit pentru apelul metodei

Metoda definită de clasă

Parametrul transmis. Nu include `self` acesta fiind adaugat automat de Python

Echivalent cu:

```
c = Coordonate(45, 45)
zero = Coordonate(0, 0)
print(Coordonate.distanta(c, zero))
```

Numele clasei

Metoda clasei

Parametrii metodei apelate, de data aceasta trebuie să specificăm și parametrul care îl reprezintă pe `self`

Afișarea Obiectelor

- >>> c = Coordonate(**3**,**4**)

>>> **print**(c)

- <__main__.Coordonate object at 0x7fa918510488>
  - Funcția print ne afișează o reprezentare neintuitivă implicit
  - Putem definii metoda __str__ pentru o clasă
  - Python apelează automat __str__ atunci când are nevoie de o reprezentare sub formă de șir de caractere
    - Descrie modul sub care ne dorim să vedem detaliile obiectului

- >>> **print**(c)

<3,4>

Afișarea Obiectelor

```
class Coordonate():
        def __init__(self, x, y):
                self.latitudine = x
                self.longitudine = y
        def distanta(self, alt_punct):
                x_diff = self.latitudine – alt_punct.latitudine
                y_diff = self.longitudine – alt_punct.longitudine
                return (x_diff**2+y_diff**2)**0.5
        def __str__(self):
                return "<"+self.latitudine+","+self.longitudine+">"
```

Afișarea Obiectelor

```
>>> c1 = Coordonate(3,4)
>>> c2 = Coordonate(3,4)
>>> l = [c1, c2]
>>> print(l)
[<__main__.Coordonate object at 0x10ebb1fd0>, <__main__.Coordonate object at 0x10ebbc0f0>]
```

- `object.__repr__(self)`: apelată de funcția `repr` pentru determinarea reprezentării „oficiale” a obiectului
- `object.__str__(self)`: apelată de funcția `str` pentru determinarea reprezentării „informale” a obiectului

Obținerea de informații privind obiectele

- Putem determina tipul unui obiect

```
>>> c = Coordonate(3,4)
>>> print(c)
<3, 4>
>>> print(type(c))
<__main__.Coordonate object at 0x7f61980dc2e8>
```

- are sens deoarece:

```
>>> print(Coordinate)
<class __main__.Coordinate>
>>> print(type(Coordinate))
<type 'type'>
```

- folosiți `isinstance()` pentru a verifica dacă un obiect este de un anumit tip

```
>>> print(isinstance(c, Coordinate))
True
```

Operatori Speciali

- +,-,==,<,>,len(), print și mulți alții https://docs.python.org/3/reference/datamodel.html#basic-customization
- putem schimba comportamentul similar cum am făcut pentru __str__
- putem modifica mulți dintre operatori:
  - __add__(self, other)
  - __sub__(self, other)
  - __eq__(self, other)
  - __len__(self)
  - __str__(self)
- … și mulți alții

Operatori Speciali

Supraîncărcarea operatorilor

- Permite claselor să stabilească comportamentul relativ la operatorii limbajului

Abordarea Python pentru supraîncărcarea operatorilor

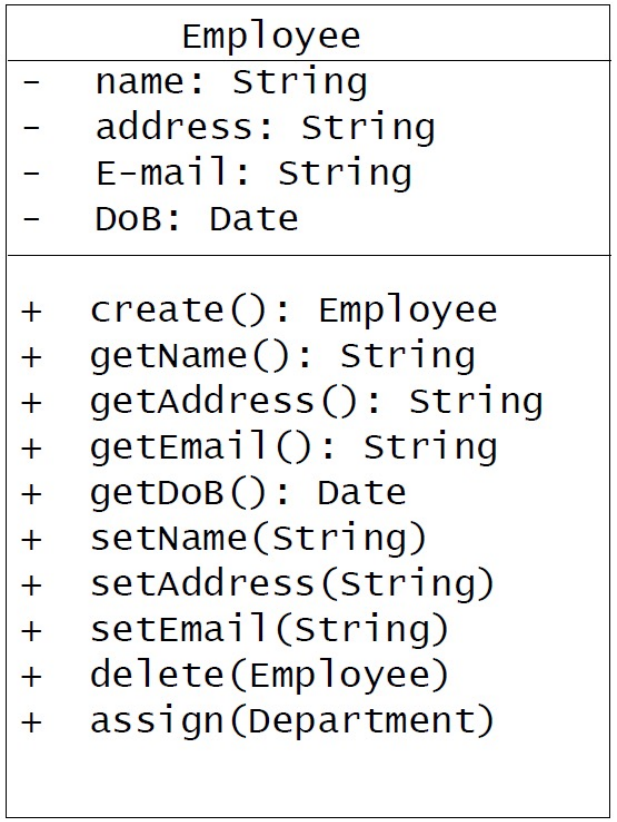
- Folosind metode speciale putem modifica implementarea diferitelor operații care sunt invocate folosind sintaxa limbajului (operații aritmetice, indexare, feliere)

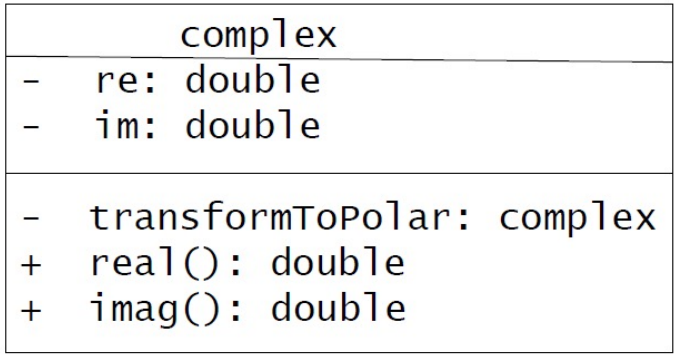
Operatori Speciali - Exemplu

- Creați un tip nou pentru a reprezenta o fracție
- Reprezentarea internă este definită de doi întregi
  - numărătorul
  - numitorul
- „interfața” sau „metoda” sau modul de „interacțiune” cu obiecte de tip „Fracție”
  - adunare, scădere
  - reprezentare pentru print
  - conversia la „float”

Date Publice și Private

- Toate atributele și metodele clasei sunt publice astfel este posibil să fie modificate din exterior
  >>> c = Coordonate(**3**,**4**)
  >>> c.latitude = "a string"
  >>> **print** (c)
  <'a string', 4>
- Pentru a prevenii acest lucru putem proteja datele folosind „accesori”
  - Încapsulare sau ascunderea datelor
  - Accesorii sunt „getters” și „setters”
- Încapsularea este importantă când clasa noastră este folosită de altcineva





- means private
+ means public

Date Publice și Private

- În Python orice care începe cu `__` este considerat privat
  - `__a`, `__variabila_mea`
  - bineînțeles că există și truc prin care le putem acces ;)
- Orice începe cu un singur `_` este considerat „semi-privat” și ar trebui să ne simțim vinovați dacă îl accesăm

Metode GET/SET

- Metode de tipul „GET” returnează valoare unui atribut
- Metode de tipul „SET” setează valoarea unui atribut

```
class Coordonate(object):
        def __init__(self, x, y):
                self.set_latitudine(x)
                self.set_longitudine(y)

        def get_latitudine(self):
                return self.__latitudine

        def set_latitudine(self, x):
                if x<-90 or x>90:
                        raise ValueError("Valoare invalida pentru latitudine")
                self.__latitudine = x
```

Returnăm valoarea privată

Verificăm ca valorile să fie în plaja potrivită

Atribut privat

Metode GET/SET

- Metode de tipul „GET” returnează valoare unui atribut
- Metode de tipul „SET” setează valoarea unui atribut

```
class Coordonate(object):
    def __init__(self, x, y):
        self.set_latitudine(x)

    def get_latitudine(self):
        return self.__latitudine

    def set_latitudine(self, x):
        if x < -90 or x > 90:
            raise ValueError("Valoare invalida pentru latitudine")
        self.__latitudine = x

    latitudine = property(get_latitudine, set_latitudine)
c = Coordonate(3, 4)
c.latitudine
```

Toate operațiile de accesare și modificare vor fi dirijate spre cele două funcții

Încapsulare

Unul dintre principalele beneficii ale claselor este că ascund detaliile de implementare => Încapsulare

O clasă bine gândită are metode care îi permit utilizatorului să acceseze toate datele și funcționalitățile dorite

- Aceasta îi permite să se concentreze pe codul lui și nu pe implementarea noastră

Acest lucru ne permite să schimbăm implementarea unei clase

- Utilizatorii comunică cu interfața noastră și nu cu implementarea
- Face codul mai modular, putem schimba implementarea din spate fără să afectez utilizatorii (dacă au folosit doar funcțiile publice)

Încapsulare

- Datele și funcțiile înrudite sunt grupate într-o capsulă de tip clasă
- Interfața reprezintă suprafața vizibilă a capsulei
  - Interfața definește caracteristicile esențiale ale obiectelor care sunt vizibile pentru lumea exterioară
- Implementarea este ascunsă în capsulă
  - Ascunderea implementării înseamnă că datele pot fi manipulate doar în cadrul clasei/obiectului.

Convenții Clase

- Numele de clase sunt capitalizate. Dacă sunt formate din mai multe cuvinte fiecare cuvânt este capitalizat
  - Exemple: `GeneratorNumere`, `Punct`, etc
- Numele de variabile și funcții încep tot timpul cu o literă mică. Dacă sunt formate din mai multe cuvinte sunt separate cu `_`
  - Exemple: `get_prime_number()`, `random()`
- Definițiile de metode trebuie să se folosească de „docstrings”
- Clasele pot avea și ele „docstrings”

Avantajele OOP

Gruparea datelor și rutinelor în pachete și interacțiunea cu acestea prin interfețe clar definite

Dezvoltare bazată pe metoda bisecției

- implementarea și testarea unei clase separat
- modularitatea ridicată reduce complexitatea

Clasele ne permit să reutilizăm ușor cod

- multe module Python definesc noi clase
- fiecare clasă își are propriul ei mediu/context (nu vom avea conflict de nume, etc)
- moștenirea permite subclaselor să rafineze sau extindă un subset al comportamentului clasei părinte

Obiecte vs Clase