Classes i Objectes

1. Crear classes i objectes
2. Modificadors d'accés
3. Self
4. Constructor __init__
5. Mètodes de classe
6. Pass
7. Eliminar objectes o atributs d'un objecte
8. @property decorator
9. str i repr

Crear classes i objectes

1. Crear classes i objectes

Python és un llenguatge de programació interpretat **orientat a objectes**. Com a tal, disposem de classes i instàncies de les mateixes (objectes).

```
class MyClass:
    x = 5
    y = 4

obj1 = MyClass()
print(obj1.x)

>>> 5

obj1.x = 6
```

- Aquesta classe permet crear objectes amb dos atributs públics amb un valor inicial.
- Creo objectes de la classe sense passar-li cap valor usant constructor per defecte.
- Un cop creat, podem canviar el valor dels paràmetres.

Modificadors d'accés

2. Modificadors d'accés

Per gestionar l'**encapsulació**, a Python disposem de tres modificadors d'accés: **public, private** i **protected.**

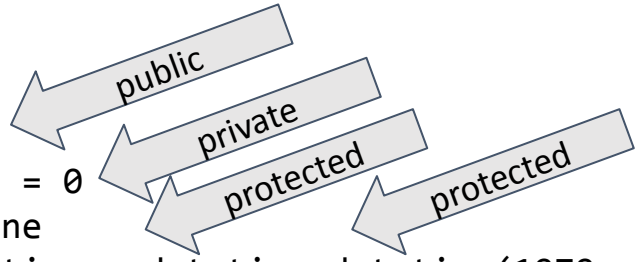
- Els modificadors d'accés **apliquen** als **atributs** i **mètodes** una classe.
- Per determinar el nivell d'encapsulació es fa servir el caràcter *underscore*_
- **Public:** determina que un atribut o mètode és accessible des de qualsevol manera.
- **Private:** determina que un atribut o mètode serà accessible només des de la pròpia classe. S'especifica aquesta condició amb dos *underscores* davant. Exemple :`__name`.
- **Protected:** determina que un atribut o mètode serà accessible des de la classe i des de les classes filles. Farem servir un sol *underscore* per especificar-ho. Ex: `_weight`

2. Modificadors d'accés

- Declarem una classe *Dog* amb alguns atributs usant els **diferents modificadors d'accés** i donant alguns **valors per defecte**.

```
import datetime

class Dog:
    name: str = None
    __chip_code: int = 0
    _breed: str = None
    _birthdate: datetime = datetime.datetime(1979, 1, 1)
```



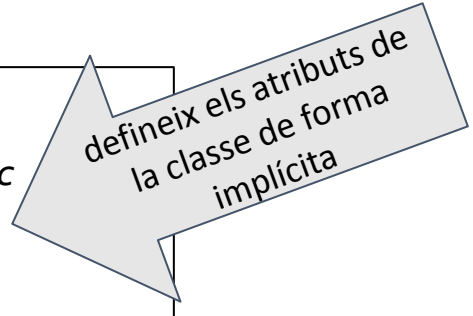
Self

3.Self

El concepte **self** s'usa a Python per fer referència a l'actual instància de la classe i s'usa per accedir a les variables i els mètodes de la classe.

- Altres llenguatges de programació, com ara Java o Kotlin, usen ***this*** per a aquesta funció.
- S'ha d'afegir sempre com a primer paràmetre a tots els mètodes de la classe.

```
class Dog:
 #Defineix el constructor específic
 def __init__(self, name, breed):
     self.__name = name
     self.__breed = breed
```



Constructor __init__

4. Constructor __init__

En programació orientada a objectes, un **constructor de classe** és una funció que ens permet inicialitzar els valors dels paràmetres dels nous objectes que crearem.

- Per defecte i per omissió, tota classe disposa sempre del **constructor per defecte** el qual s'executa amb el nom de la classe i parèntesis buits.
- Normalment, una classe pot tenir més d'un **constructor específic** definint diferents modalitats d'inicialització dels paràmetres. Els valors dels paràmetres esmentats han de ser introduïts dins dels parèntesis en fer la instanciació d'un nou objecte.
- A Python, generalment, només podrem definir **un constructor per defecte** usant la funció **__init__()** sense paràmetres.

4. Constructor __init__()

```
class Dog:
     # Atributs privats per defecte, amb un tipatge de dades
i valor inicial null (None en Python).
     __name: str = None
     __breed: str = None
     # Constructor específic
     def __init__(self, name, breed):
          self.__name = name
          self.__breed = breed
          print("New Dog created with name " + self.__name)
```

- El constructor és un mètode especial que per defecte serà sempre públic.

- Dins la funció __init__() del **constructor específic podem** definir el valor dels paràmetres dels objectes així com executar qualsevol altre codi que calgui.
- Proveu d'executar el codi de la classe i la següent **creació d'objecte**:

```
dog1: Dog = Dog("Pepe", "Rottweiler")
```

Mètodes de classe

5. Mètodes de classe

A banda del constructor específic, a les classes també podrem **declarar funcions** per definir accions a executar. Aquests seran els **mètodes de classe**.

```
class Dog:
      # Declareu atributs privats amb valor None per defecte
      __name = None
      __breed = None
      # Defineix el constructor específic
      def __init__(self, name, breed):
            self.__name = name
            self.__breed = breed
            print("New Dog created with name" + self.__name)
      # Defineix el mètode getter per l'atribut name
      def get_name(self):
            return self.__name
```

- Els atributs seran privats o protected. Poden definir-se directament al constructor.
- Els *getters* i *setters* seran públics.

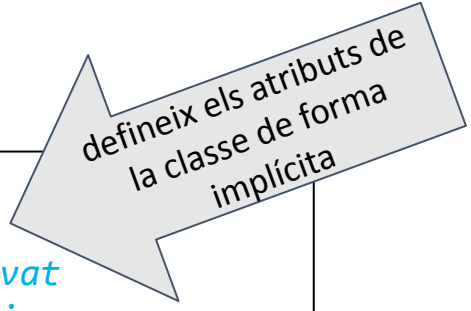
Pass

6. Pass

A Python, el **cos del codi** de **classes** i **funcions** no poden estar buits. Si necessitem declarar classes o funcions sense definir-ne el cos del codi per ser desenvolupades més tard, podem fer servir l'operador **pass**.

```
class Dog:
    def __init__(self, name, breed):
        self.__name = name # Atribut __name definit com a privat
        self.breed = breed # Atribut breed definit com a public
        print("New dog " + self.__name + " has been created.")

    def metode(self):
        pass
```



Eliminar objectes o atributs d’objectes

7. Eliminar objectes o atributs d'objectes

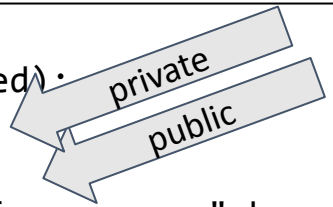
A Python, podem eliminar objectes i també **atributs d'objectes** ja creats.

```
class Dog:
     def __init__(self, name, breed):
          self.__name = name
          self.breed = breed
          print("New dog " + self.__name + " has been created.")


dog1: Dog = Dog("Pepe", "Dalmata")

del dog1.breed # Elimina l'atribut breed de l'objecte dog1 creat

del dog1 # Elimina l'objecte de M RAM
```



@property decorator

8. @property decorator

Per crear un codi *pythonic* farem servir les @property. Això ens permet etiquetar els mètodes per executar accions específiques com ara ***getter***, ***setter*** i ***deleter***.

- Per definir un ***getter*** farem servir @property just abans del mètode i l'anomenarem igual que l'atribut. Això ens permetrà accedir al seu valor com si fos un atribut públic però de forma controlada i crear després el *setter* i *deleter*:

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

- Per definir un ***setter*** farem servir @nom_atribut.setter just abans de la definició del mètode. Això ens permet modificar-ne el valor com si fos públic però de forma controlada. (Hem de declarar la @property primer):

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

8. @property decorator

- Per definir un ***deleter*** farem servir @nom_atribut.deleter just abans de la definició del mètode. Això permetrà eliminar l'atribut de l'objecte creat. *(Hem de declarar la @property del mètode prèviament)*:

```
@name.deleter
def name(self):
    delself.__name
```

Beautiful is better than ugly.

(Primer principi del Zen of Python, "pythonic way")

8. @property decorator

```python
class Dog:
 # Defineix the specific cconstructor
    def __init__(self,name,breed):
        self.__name = name
        self.__breed = breed
        print("New Dog created with name " + self.__name)

    #Defineix the property 'name' which will allow us to access to the __name attribute like if it were public
    @property
    def name(self):
        return self.__name

    #Defineix the property 'breed' which will allow us to access to the __breed attribute like if it were public
    @property
    def breed(self):
        return self.__breed

    # Set a new value for the __name attribute like if it were public. We can control the accepted new values.
    @name.setter
    def name(self, new_name):
        self.__name = new_name

    # Set a new value for the __breed attribute like if it were public. We can control the accepted new values.
    @breed.setter
    def breed(self, new_breed):
        self.__breed = new_breed

    #Define a public method which allows us to delete the atribute breed from an existing Dog object
    @breed.deleter
    def breed(self):
        del self.__breed
```

str i repr

9.str i repr

A Python disposem de dos mètodes específics per imprimir un objecte: **str** i **repr**
És l'equivalent del *toString()* de Kotlin i Java.

- El mètode especial **__str__()** ens permet definir com volem que es mostri un objecte d'una classe definida per l'usuari de manera amigable al ser printat per pantalla.

- El mètode especial **__repr__()** ens permet definir una forma oficial de mostrar un objecte d'un tipus de dades definit per l'usuari de manera que la seva sintaxi permeti crear una còpia de l'objecte de manera àgil.

9.str i repr

```
#Define a public method to print the details of the Dog. __str__
def __str__(self):
   return("The dog is called " + self.__name + " and its breed its " + self.__breed)


#Define a public method to print the details of the Dog. __repr__
def __repr__(self):
   return f'Dog(name = {self.__name}, breed = {self.__breed})'
```

9.str i repr

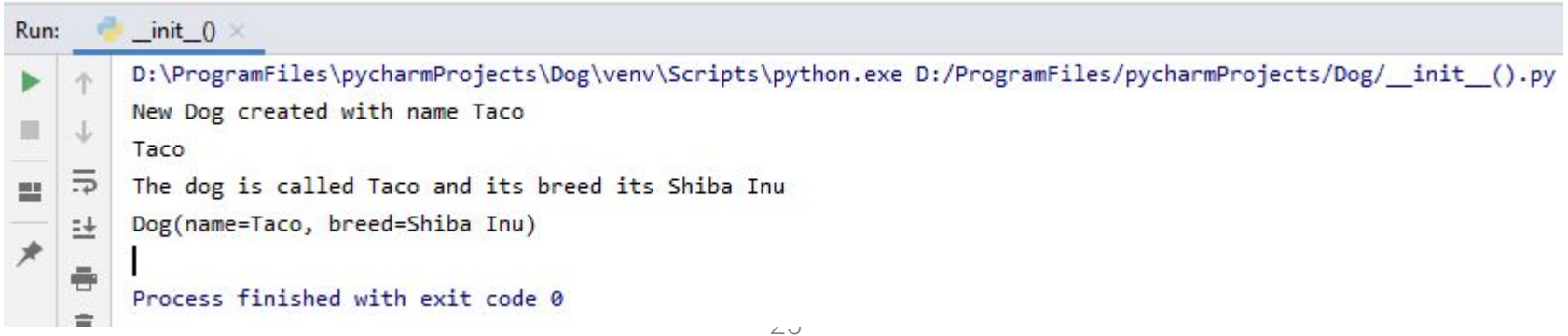
```python
from Dog import*

# Create the dog
d: Dog = Dog("Tac", "Shiba Inu")

# Get the name of the Dog
name = d.name
print(name)

# Using __str__ method
print(d)

# Using __repr__ to print the details of the Dog
print(d.__repr__())
```



"No has de ser perfecte, però has d´estar compromès al 100%"

Alexandria Ocasio Cortez(1989 -)
És la dona més jove triada al Congrés en la història dels Estats Units.

Raimon Izard