Programming

6- Object-Oriented Programming

These slides will be available on Arche

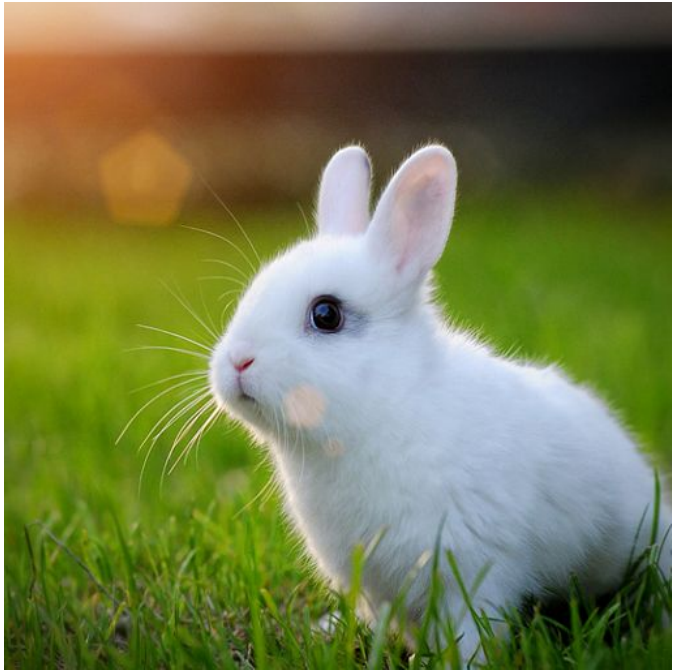
By Gaël Guibon, inspired by Mathieu d'Aquin's course

Object-Oriented Programming Overview

Object-Oriented Programming

Python is fully made of objects!

What is this?



What is this?

A rabbit 🐰

What can it do?

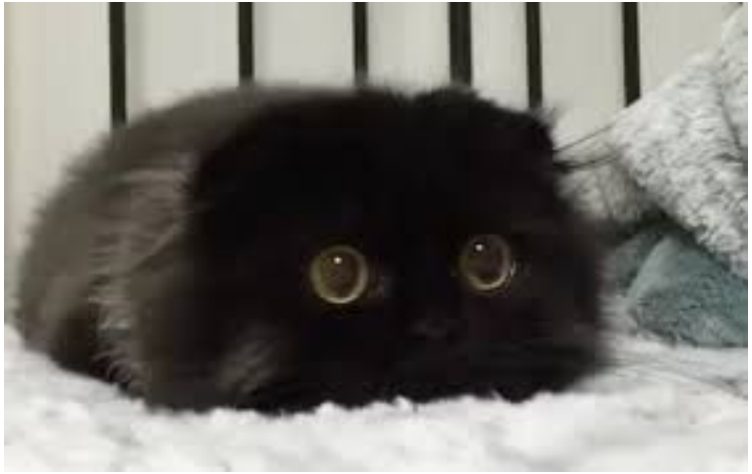
A rabbit 🐰

Possesses a color

It can:

- Flee
- Jump
- Bite
- Eat

What is this?



What is this?

A cat 🐱

What can it do?

A cat 🐱

Possesses a color

It can:

- Flee
- Jump
- Bite
- Eat

What They Really Are

Objects!

What They Really Are

Objects!

```
class Rabbit:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def eat(self, food): # method
        if food == '🥕':
            return f"{self.name} eats the yummy {food} 😋"
        return f"{self.name} does not even look at the {food}"
```

What They Really Are

Objects!

```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def eat(self, food): # method
        if food in ['🐟', '🍖', '🐁']:
            return f"{self.name} eats the yummy {food} 😋"
        return f"{self.name} does not even look at the {food}"
```

What They Really Are

Objects!

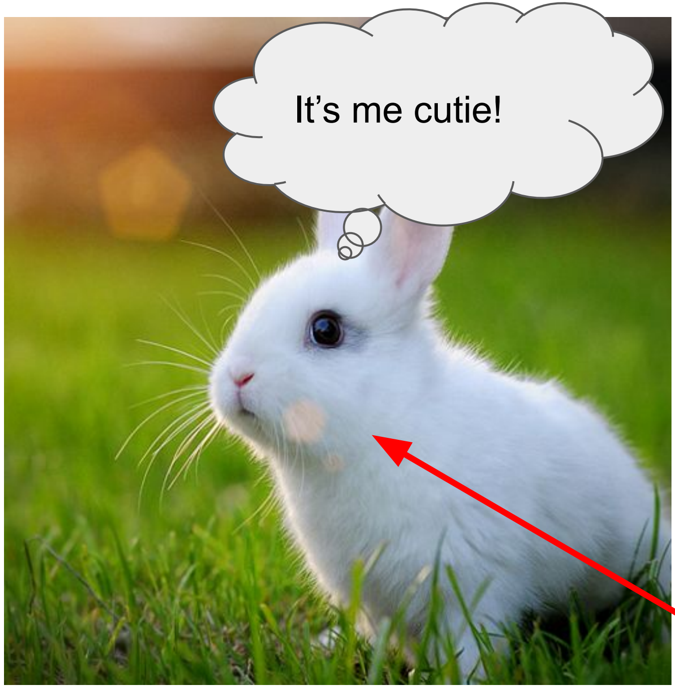
```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def eat(self, food): # method
        if food in ['🐟', '🍖', '🐁']:
            return f"{self.name} eats the yummy {food} 😋"
        return f"{self.name} does not even look at the {food}"
```

What They Really Really Are

Instances of a class!

What They **Really** Really Are

Instance of Rabbit()



```python
class Rabbit:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def eat(self, food): # method
        if food == '🥕':
            return f"{self.name} eats the yummy {food} 😋"
        return f"{self.name} does not even look at the {food}"

white_rabbit = Rabbit("cutie", "white")
```

What They Really Really Are

Instance of Cat()!

```
class Cat:
    def __init__(self, name, color):
        self.name = name
        self.color = color
    def eat(self, food): # method
        if food in ['🐟', '🍖', '🐁']:
            return f"{self.name} eats the yummy {food} 😋"
        return f"{self.name} does not even look at the {food}"

black_cat = Cat("bbkitty", "black")
```

Object-Oriented Programming
In more details

What is Object-Oriented Programming (OOP)

A way to **structure your code**. → one Rabbit class with behavior, etc.

A way to **create new types.** → Rabbit()

A programming paradigm that favors **separation of concerns**.

Fundamentals of Object-Oriented Programming

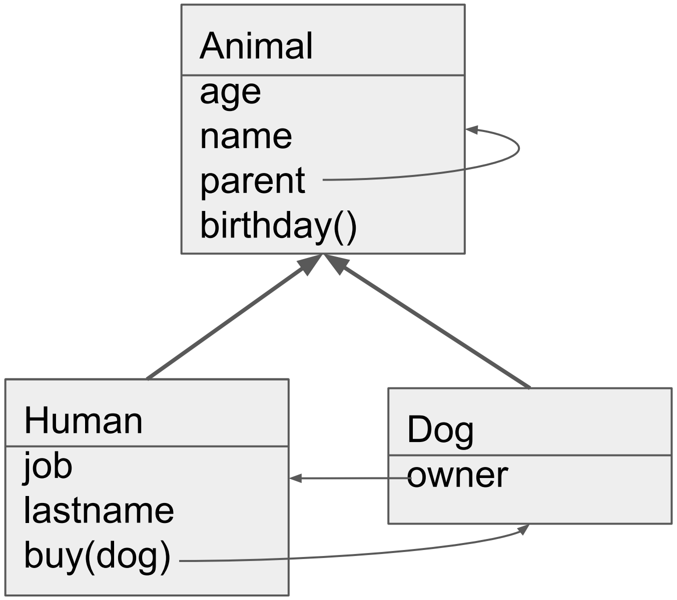
You define ***classes***, which can be the types of ***instances***.

Classes have ***attributes*** that are variables in the scope of the instances of the class.

Functions are associated with classes, and apply to their instances. They are called the ***methods*** of a class.

Classes can have ***sub-classes*** that ***inherit*** their methods and attributes.

Example



```
class Animal:
 def __init__(self,age,name):
   self.age = age
   self.name = name
 def birthday(self):
   self.age += 1

class Human(Animal):
 def __init__(self,age,name,lastname,job):
   super().__init__(age,name)
   self.lastname = lastname
   self.job = job
 def buy(self, dog):
   dog.owner = self

class Dog(Animal):
 def __init__(self, age, name):
   super().__init__(age,name)
```

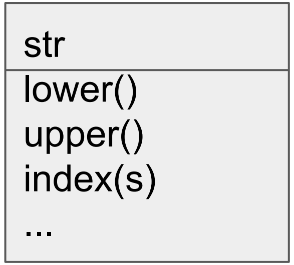
```
class Animal:
 def __init__(self,age,name):
   self.age = age
   self.name = name
 def birthday(self):
   self.age += 1

class Human(Animal):
 def __init__(self,age,name,lastname,job):
   super().__init__(age,name)
   self.lastname = lastname
   self.job = job
 def buy(self, dog):
   dog.owner = self

class Dog(Animal):
 def __init__(self, age, name):
   super().__init__(age,name)
```

```
bob = Human(24,"Bob","Example","Programmer")
print(f"{bob.name} {bob.lastname} {bob.job} {bob.age}")
bob.birthday()
print(f"{bob.name} {bob.lastname} {bob.job} {bob.age}")
bob.parent = Human(55,"Alice", "Example", "Engineer")
print(f"{bob.name}'s parent is {bob.parent.name}")
alice = bob.parent
print(f"{alice.name} {alice.lastname} {alice.job}
{alice.age}")
rex = Dog(4, "Rex")
bob.buy(rex)
print(f"{rex.name}'s owner is {rex.owner.name}")
___________________________________
Bob Example Programmer 24
Bob Example Programmer 25
Bob's parent is Alice
Alice Example Engineer 55
Rex's owner is Bob
```

Classes we have already seen...



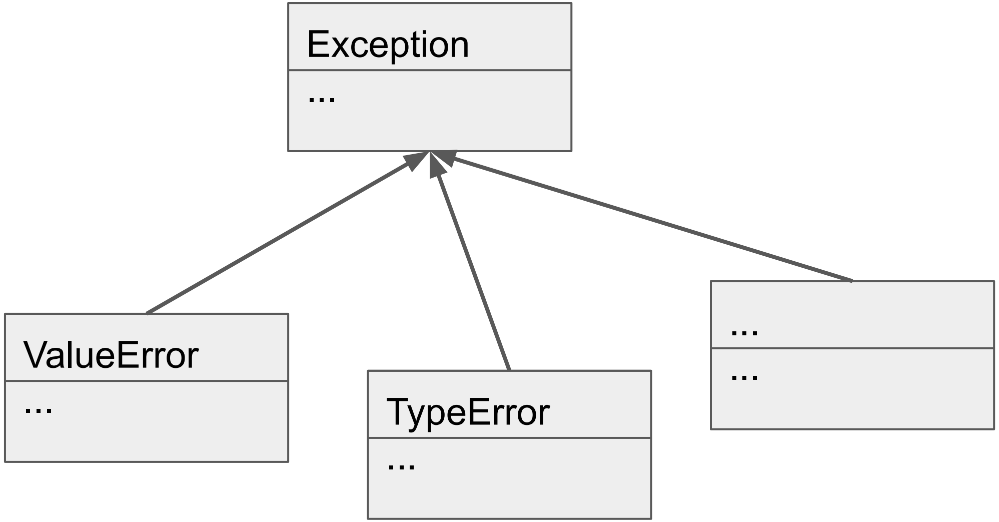
str

lower()

upper()

index(s)

...



Exception

...

ValueError

...

TypeError

...

...

...

Defining a Simple Class

The minimum information a class needs is a name:

```
class Cat:
    pass
```

Then we can create instances of the class (variables with for type this class):

```
moly = Cat()
print(type(moly))
______________________
<class '__main__.Cat'>
```

It’s a type!

Attributes in a class/instance

Attributes are variables which belong **specifically to the instances** of a class. They only make sense in the context of an instance, and can be accessed/assigned using the dot notation:

```
class Cat:
    name = "Cat"

moly = Cat()
print(moly.name)
moly.name = "moly"
print(moly.name)
____________________
Cat
moly
```

Methods in a class/instance

Methods are functions that are associated with classes and are applicable to their instances. They only exist for the instances of the classes in which they are declared.

```
class Cat:
    name = "Cat"      # attribute
    def speak(self): # method
      return f"{self.name} says meow"


moly = Cat()
moly.name = "Moly"
print(moly.speak())

______________________________

Moly says meow
```

Methods in a class/instance

Methods are functions that are associated with classes and are applicable to their instances. They only exist for the instances of the classes in which they are declared.

```
class Cat:
    name = "Cat"      # attribute
    def speak(self): # method
        return f"{self.name} says meow"


moly = Cat()
moly.name = "Moly"
print(moly.speak())
_____________________________
Moly says meow
```

`self` refers to the current instance of the class.
It needs to be the first parameter of methods and to be used to access the values of attributes within those methods

Constructor of a class

The constructor is a **special method**, called `__init__`, that is called when a new instance is created.

```
class Cat:
    name = "Cat"     # attribute
    def __init__(self, name):
      self.name = name
    def speak(self): # method
      return f"{self.name} says meow"

moly = Cat("Moly")
print(moly.speak())

_______________________
Moly says meow
```

Constructor of a class

The constructor is a **special method**, called `__init__`, that is called when a new instance is created.

```
class Cat:
    name = "Cat"     # attribute
    def __init__(self, name):
      self.name = name
    def speak(self): # method
      return f"{self.name} says meow"

moly = Cat("Moly")
print(moly.speak())

_______________________
Moly says meow
```

Subclasses and inheritance

A class can be a subclass of one or several other classes. In this case, it will inherit its (their) attributes and methods.

```
class Animal:
 age = 0
 def __init__(self,age):
   self.age = age
 def setSound(self,s):
   self.sound = s

class Cat(Animal):
 def __init__(self,name):
   self.name = name
   self.setSound("meow")
 def speak(self):
   return f"{self.name} ({self.age}) says {self.sound}"
```

Subclasses and inheritance

A class can be a subclass of one or several other classes. In this case, it will inherit its (their) attributes and methods.

```python
class Animal:
 age = 0
 def __init__(self,age):
   self.age = age
 def setSound(self,s):
   self.sound = s

class Cat(Animal):
 def __init__(self,name):
   self.name = name
   self.setSound("meow")
 def speak(self):
   return f"{self.name} ({self.age}) says {self.sound}"
```

Subclasses and inheritance

A class can be a subclass of one or several other classes. In this case, it will inherit its (their) attributes and methods.

```
class Animal:
 age = 0
 def __init__(self,age):
   self.age = age
 def setSound(self,s):
   self.sound = s

class Cat(Animal):
 def __init__(self,name):
   self.name = name
   self.setSound("meow")
 def speak(self):
   return f"{self.name} ({self.age}) says {self.sound}"
```

```
moly = Cat("moly")
print(moly.speak())
moly.age = 12
moly.setSound("meeeow")
print(moly.speak())
```

```
____________________________
moly (0) says meow
moly (12) says meeeow
```

Polymorphism

Polymorphism is the mechanism through which the **same method can have a different behaviour depending** on the class.

```
class Animal:
 name = "A"
 def speak(self, sound):
   return f"{self.name} says {sound}"

class Dog(Animal):
 def speak(self, sound="woof"):
   return f"{self.name} barks {sound}"

class Cat(Animal):
 def speak(self, sound="meow"):
   return f"{self.name} meows {sound}"
```

Polymorphism

Polymorphism is the mechanism through which the **same method can have a different behaviour** depending on the class.

```
class Animal:
 name = "A"
 def speak(self, sound):
   return f"{self.name} says {sound}"

class Dog(Animal):
 def speak(self, sound="woof"):
   return f"{self.name} barks {sound}"

class Cat(Animal):
 def speak(self, sound="meow"):
   return f"{self.name} meows {sound}"
```

```
rex = Dog()
rex.name = "Rex"
moly = Cat()
moly.name = "Moly"
a = rex
print(a.speak())
a = moly
print(a.speak())
print(a.speak("meeeow"))
___________________
Rex barks woof
Moly meows meow
Moly meows meeeow
```

The `super()` function

The super() function gives you the same instance as self as if it was an instance of the **super class**, so you can access

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

class Cat(Animal):
   def __init__(self, name, sound):
     self.sound = sound
     super().__init__(name)
   def speak(self):
     return f"{self.name} says {self.sound}"

moly = Cat("Moly", "meeeow")
print(moly.speak())
```

```
_______________________________________
Moly says meeeow
```

The `super()` function

The super() function gives you the same instance as self as if it was an instance of the **super class**, so you can access

```
class Animal:
   def __init__(self, name):
      self.name = name

class Cat(Animal):
   def __init__(self, name, sound):
      self.sound = sound
      super().__init__(name)
   def speak(self):
      return f"{self.name} says {self.sound}"

moly = Cat("Moly", "meeeow")
print(moly.speak())
```

```
______________________________________
Moly says meeeow
```

Overriding standard methods: __str__

Some methods are built into classes that can be overridden.

```python
class Animal:
    species = "unspecified"
    def __init__(self, name): self.name = name

class Cat(Animal):
    species = "cat"
    def __init__(self, name): super().__init__(name)

moly = Cat("Moly")
print(moly)
```

```
____________________________
<__main__.Cat object at 0x7fbedd384390>
```

Overriding standard methods: __str__

Some methods are built into classes that can be overridden.

```
class Animal:
   species = "unspecified"
   def __init__(self, name): self.name = name
   def __str__(self): return f"{self.name} the {self.species}"

class Cat(Animal):
   species = "cat"
   def __init__(self, name): super().__init__(name)

moly = Cat("Moly")
print(moly)

____________________________
Moly the cat
```

Overriding standard methods: __str__

Some methods are built into classes that can be overridden.

```
class Animal:
    species = "unspecified"
    def __init__(self, name): self.name = name
    def __str__(self): return f"{self.name} the {self.species}"

class Cat(Animal):
    species = "cat"
    def __init__(self, name): super().__init__(name)

moly = Cat("Moly")
print(moly)

____________________________
Moly the cat
```

Some built-in methods that can be overridden

- __lt__ __le__ __gt__ __ge__ __eq__ __ne__: comparisons (<,<=,>,>=,==,!=)
- __bool__ __int__ ….: typecasting (like __str__)
- __len__ __getitem__ __setitem__ __contains__: related to lists (len, x[i], in)
- __add__ __sub__ __mul__ __truediv__: arithmetic operations (+,-,*,/)
- __and__ __or__ __xor__: Boolean operators
- …

A word on multiple inheritance

It is possible for a class to be a subclass of more than one class.

```
class Animal:
    isAdult=None
    def __init__(self, name): self.name = name
class Adult:
    isAdult=True
class Child:
    isAdult=False
class AdultAnimal(Animal,Adult):
    def __init__(self, name): super().__init__(name)
```

But it can make things complicated with polymorphism.

YOU
PROCEDURAL
OBJECT ORIENTED
imgflip.com

To be seen in labs

Creating and using classes

To represent Pokémons and more