ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Systems and programming for Automation M

4. Objects in Python

Notice

The course material includes material taken from Prof. Chesani and Prof. Martini courses (with their consent) which have been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

Motivation

- It is well known from AI and Cognitive Studies that we (as humans) tend to construct some knowledge structures in the form of:
  - General concepts, e.g. a car, that have:
    - a number of characteristics (attributes)
    - can be interacted with through certain operations (methods)
  - Individuals, i.e. instances of those concepts, e.g. my car
- We use these mental structures to:
  - project expectations about single instances (my car surely will have a steering wheel)
  - interact with single instances by using proper/meaningful functions/operations.

New Data Type

- When programming we may want to define our own **new data types** to better represent a concept
  - E.g.: a new data type representing a person
- It will be based on existing, predefined types
- A data type is defined through
  - its attributes
    - E.g.: name, surname, date of birth
  - the functions/operators that can be applied to it
- It is up to us to decide how a person is modelled, i.e. which data/attributes, which methods

Object-Oriented Programming (OOP)

- **Class**:
  - **general concept** of our new data type, together with attributes and methods
- **Object/Instance**:
  - a **specific** data type **value**, with the attributes filled properly, and with the possibility of invoking methods on it

Example: Pizza

- Suppose we want to model the idea of a Pizza.
  - The **class** will describe us the concept of pizza:
    - it has a name ('Margherita')
    - it has some toppings ('tomato sauce', 'mozzarella cheese')
    - it has a cost
  - It will have also some **methods**: cutPizza(), bitePizza().

- This what you can find in the restaurant menu…but can you bite it?

- Once we get a description of a pizza, and the methods/operations that are allowed over it, we would like to have a margherita to eat… from the description, to an instance of our pizza, exactly like it happens at the restaurant: from the menu's description, to the real one.

Defining Classes

- Similar to functions, the syntax for defining classes is:

```
class <class_name>(<parent_class>):
    <body>
```

- `<class_name>` is the name of our concept.
  - By convention, class names always start with a capital letter
- The class `<body>` will contain the methods that can be applied to our new data type
- `<parent_class>` later on this, for now is just **`object`**

Attributes and Methods

- Given any object, we can use the `.` symbol to
  - **get** the value of the **attributes**
  - **set** the value of the **attributes**
    - and possibly create new ones
  - **invoke** a **method**

- For example:
  - class `Persona`
    - Attributes: a `att_name` and a `att_family_name`
    - Methods: `introduce()` that prints the attributes
  - Instance `andrea` of the class that represent myself

Attributes and Methods - Example

```
# attributes: a name and a family_name
# methods: introduce() that prints the attributes

print(andrea.att_name) # prints “Andrea”

x = andrea.att_family_name # x = “Galassi”
print(x) # prints “Galassi”

andrea.introduce() # prints “Hello, I’m Andrea Galassi”

andrea.att_name = “Paolo”
andrea.introduce() # prints “Hello, I’m Paolo Galassi”
```

Constructor

- Each class should define a special method, referred as the **constructor** of the object, that will take care of properly instantiating the values for any instance
  - The constructor defines the attributes of the class and assigns the corresponding values for a specific instance
  - It returns the new object (=new instance!)

- The syntax is **`init`** preceded and followed by two _ (underscore)
  - The first parameter is always **`self`**

```
def __init__(self, <list of parameters>):
    <init_body>
```

to initialize the attributes

Constructor - Example

define a class named Person

```
class Person(object):
    def __init__(self, name, familyName):
        self.att_name = name
        self.att_familyName = familyName

p1 = Person('Andrea', 'Galassi')
p2 = Person('Paolo', 'Torroni')
```

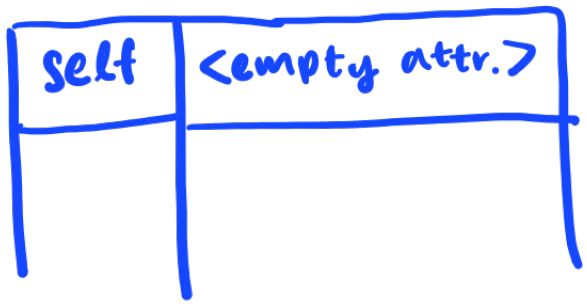
there's an attribute called att_name which links to parameter name)

Constructor – Step 1

```
class Person(object):
    def __init__(self, name, familyName):
```

- As soon as the constructor is called, the Python Machines allocates some memory for the new object, which for now is empty, and it can be accessed through **`self`**

- **`self`** is the new (empty) object



Constructor – Step 2

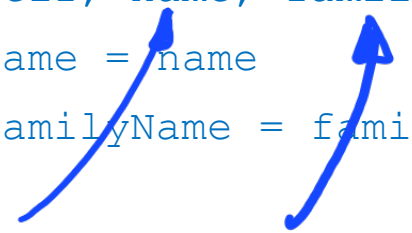
```
class Person(object):
    def __init__(self, name, familyName):
        self.att_name = name
```

- `self` is the new (empty) object
- Inside the constructor, the empty object is populated with attributes
  - `self.<attribute>` refers to the attribute of the object `self`
  - We are **creating** the attribute name and **setting** it to a value

Constructor - Instantiation

```
class Person(object):
    def __init__(self, name, familyName):
        self.att_name = name
        self.att_familyName = familyName

p1 = Person('Andrea', 'Galassi')
p2 = Person('Paolo', 'Torroni')
```



- To create (instantiate) a new object, it is enough to call the class name, passing the required parameters.
  - The arguments must match the formal parameters of the constructor, which is automatically called
  - The `self` parameter is omitted from the arguments

Constructor – Step by Step

```
class Person(object):
    def __init__(self, name, familyName):
        self.att_name = name
        self.att_familyName = familyName

p1 = Person('Andrea', 'Galassi')
p2 = Person('Paolo', 'Torroni')
```

Name	Value
…	…
self	<object>
name	Andrea
familyName	Galassi

Name	Value
…	…
p1	Name / Value: att_name = Andrea; att_familyName = Galassi

Name	Value
…	…
self	Name / Value: att_name = Andrea; att_familyName = Galassi
name	Andrea
familyName	Galassi

Constructor – Attribute Names

- Typically the attributes and the formal parameters of the constructor have the same name

```
class Person(object):
    def __init__(self, name, familyName):
        self.name = name
        self.familyName = familyName

p1 = Person('Andrea', 'Galassi')
p2 = Person('Paolo', 'Torroni')
```

Instance Methods

instanciation?

- We want a set of (meaningful) operations over our new data type.
  - **Methods** that will act on the **specific instance**

- Instance methods are defined as functions within the class body
- They always receive `self` as a first parameter
- If they access attributes (read or write), they must use `self.<name>`

```
class Person(object):
    # ...
    def introduce(self):
        print("Hello, I'm", self.name, self.familyName)
```

```
p1.introduce()
```

name of the instance

name of the method

Inheritance – Motivation

- We mentally organize concepts in **hierarchies** from more general concepts to more specific ones
  - Specific concepts will inherit the features of their parents
  - Specific concepts might have more attributes and methods than the parent ones

a student is a person
↳ specific ↳ generic

- "Each professor is a person"
  - The features of a person usually are present also in a professor
  - The class Professor and the class Student are a specific cases of Person

If we want to model Professor, we do not want to copy everything we have already written for Person

Inheritance – Motivation

- We mentally organize concepts in **hierarchies**, from more general concepts to more specific ones
  - Specific concepts will inherit the features of their parents
  - Specific concepts might have more attributes and methods than the parent ones

- "Each professor is a person"
  - The features of a person usually are present also in a professor
  - The class Professor and the class Student are a specific cases of Person

- If we want to model Professor, we do not want to copy everything we have already written for Person

Inheritance

- Hierarchy between classes:
  - More **general** class: **parent** class, or **super** class
  - More **specific** class: **child** class, or **subclass**

- The inheritance relationship is defined through the definition of the class, as a parameter following the class name

```
class <className>(<parent_classNname>):
```

child → parent

```
class Professor(Person):
```

  - Each instance of Professor will be an instance of Person
  - Each instance of Professor will inherit attributes of Person
  - Each instance of Professor will inherit methods of Person
  - Professor can have new attributes and methods

Inheritance - Constructor

- To instantiate the variables that are inherited from the parent class, in the constructor of the subclass it is possible to invoke the constructor of the super class

```
class Professor(Person):
    def __init__(self, name, familyName, course):
        super().__init__(name, familyName)
        self.course = course
    def printCourse(self):
        print(self.name, self.familyName, self.course)
```

initialize the parent attribute

child

parent

- A Professor has a name and a `familyName`, like all persons
- A Professor has a `course`
- A Professor has method to print the names and the `course`

Specific only to Professor

Inheritance - Constructor

- To instantiate the variables that are inherited from the parent class, in the constructor of the subclass it is possible to invoke the constructor of the super class

```
class Professor(Person):
    def __init__(self, name, familyName, course):
        super().__init__(name, familyName) # invoke parent
        self.course = course
```

- A Professor has a name and a `familyName`, like all persons

Inheritance - Methods

- The subclass inherits all the methods of the parent class
- It can also define new methods

```
class Professor(Person):
    def __init__(self, name, familyName, course):
        super().__init__(name, familyName)
        self.course = course

    def printCourse(self):
            print(self.name, self.familyName, self.course)

p1 = Person("John", "Smith")
p2 = Professor("Andrea", "Galassi", "RTSA")
p2.printCourse() # GOOD!
p1.printCourse() # WRONG!!!
```

Inheritance - Overriding

- Subclasses might redefine (**override**) methods of the parent classes
- Overridden methods can, of course, invoke parent methods

```
class Professor(Person):
    def __init__(self, name, familyName, course):
        super().__init__(name, familyName)
        self.course = course
    def printCourse(self):
        print(self.name, self.familyName, self.course)
    def introduce(self): # overriding
        super().introduce() # invoke parent method
        print( "and I teach", self.course)

p1 = Person("John", "Smith")
p2 = Professor("Andrea", "Galassi", "RTSA")
p2.introduce() # What output?
p1.introduce() # What output?
```

rewrite to create a different method from the parent

use the same method like the parent

addition for the child

Instance and Subclass

- It is possible to test if an object is an instance of a specific class

```
isinstance(<object>,<className>)
```

```
p1 = Professor('Andrea', 'Galassi', 'RTSA')
print(isinstance(p1, Professor)) # prints ??? TRUE
print(isinstance(p1, Person)) # prints ??? TRUE
```

every instance of child is also the instance of parent

- It is possible to test if a class is derived from another

```
issubclass(<childClassName>, <parentClassName>)
```

```
print(issubclass(Professor, Person)) # prints ??? TRUE
```

Identity

every object in Python has identities

- An object is a **value** enveloped in an **identity**
- This identity is unique during a run of the Python machine

- Any value is an object (even int values)
- We may have distinct objects (with different identities) and same values

- Identity of an object is completely controlled by the Python machine
  - We may inspect the identity of an object with `id(<object>)`
  - We cannot change the identity of an object

Equality

- Given two variables, we would like to know if the corresponding objects are **equal**
- What does it mean to be equal?
  - have an EQUAL value
- Does it mean to be the **same thing**?
  - The operator `is` check if two variables refer to the same object, which means if they have the same **identity**
- Or to be two different thing with **equivalent values**?
  - The operator `==` check if the values in the objects are the same
    - BUT it is not defined for generic objects
    - To use it for a new object it is necessary to define it

Equality – Overriding ==

- The behaviour of == can be changed by defining in the class the method `__eq__(self, other)`

```
class Person(object):
    def __init__(self, name, familyName):
        self.name = name
        self.familyName = familyName

    def __eq__(self, other):
        return (self.name.lower()==other.name.lower() and
        self.familyName.lower()==other.familyName.lower())
```

double underscores change the behavior of the operator

EXAM QUESTION

- The same can be done with all the standard operators

REDEFINE EQUAL OPERATOR

Overriding of Built-in Operators

- Similarly, it is possibly to override the standard operators
  - Remember how the semantic of + is different for numbers and strings

Operator	Expression	Internally
Addition	`p1 + p2`	`p1.__add__(p2)`
Subtraction	`p1 - p2`	`p1.__sub__(p2)`
Multiplication	`p1 * p2`	`p1.__mul__(p2)`
Power	`p1 ** p2`	`p1.__pow__(p2)`
Division	`p1 / p2`	`p1.__truediv__(p2)`
Floor Division	`p1 // p2`	`p1.__floordiv__(p2)`
Remainder (modulo)	`p1 % p2`	`p1.__mod__(p2)`
Bitwise Left Shift	`p1 << p2`	`p1.__lshift__(p2)`
Bitwise Right Shift	`p1 >> p2`	`p1.__rshift__(p2)`
Bitwise AND	`p1 & p2`	`p1.__and__(p2)`
Bitwise OR	`p1 | p2`	`p1.__or__(p2)`
Bitwise XOR	`p1 ^ p2`	`p1.__xor__(p2)`
Bitwise NOT	`~p1`	`p1.__invert__()`

Overriding of Built-in Operators

Operator	Expression	Internally
Less than	`p1 < p2`	`p1.__lt__(p2)`
Less than or equal to	`p1 <= p2`	`p1.__le__(p2)`
Equal to	`p1 == p2`	`p1.__eq__(p2)`
Not equal to	`p1 != p2`	`p1.__ne__(p2)`
Greater than	`p1 > p2`	`p1.__gt__(p2)`
Greater than or equal to	`p1 >= p2`	`p1.__ge__(p2)`

Overriding of Special Methods

- These methods can be overridden as well

Function	Description
`__init__()`	initialize the attributes of the object
`__str__()`	returns a string representation of the object
`__len__()`	returns the length of the object
`__add__()`	adds two objects
`__call__()`	call objects of the class like a normal function

- **`__init__`** is used to for the constructor
- **`__str__`** influences `p rint(object)`

Information Hiding

- In OOP, typically is possible to declare attributes as private
  - Objective: limiting the access from portions of code out of the class definition → showing only desired information
  - "**Information Hiding**" principle: separate (detailed) internal representation from external representation
  - Result: different levels of abstraction

- E.g.: A car is made of several components, for example an engine, a gear box, and a breaking system. We are not really interested no how it is made the gear box, but rather we want to change gears using the proper (provided) mechanism.
- In turn, the gear box will be made of other mechanisms (the shift, the pressurized oil pump, the friction,…)

unfortunately, Python makes everything is visible

Attribute Visibility

- In Python, all the attributes of an object are public and can be always accessed using the dot notation
- As a convention, developers can indicate variables that should be treated as private by prefixing the name with an underscore _

```
class Person(object):
    def __init__(self, name, familyName, _age=18):
        self.name = name
        self.familyName = familyName
        self._age = _age
```

- In the example, the developer is saying that the variable _age should be intended as private and should not be accessed
- However, it is legit to write

```
p1 = Person('Federico', 'Chesani', 80)
p1._age = 25
```

POSSIBLE

Attribute Modification

- It is possible to interact with attributes treating them as variable
- Check if an instance has an attribute

```
hasattr(<object>, <attrNameString>)
```

- Create/Modify an attribute

```
setattr(<object>, <attrNameString>, <value>)
```

- Delete an attribute

```
delattr(<object>, <attrNameString>)
```

Class Variables

- So far we have seen instance variables
  - Each person has its own name, family name, and age

- It is possible to set **class variables** (or **class properties**) that are shared across all the instances of that class.
  - For example, we may want to set `species = "homo sapiens"` for all the Persons

- They are defined within a class definition
  - Not within the `__init__` and not prefixed by the `self.` reference

```
class Person(object):
    # …
    species = "homo sapiens"
```

Class Variables Access

- They can be accessed by prefixing them with the name of the class:

```
print(Person.species) # homo sapiens
```

- They can be accessed also through the instances

```
print(p1.species) # homo sapiens
```

- If class variable value is changed, such a change reflects to all classes

```
Person.species = "human"
print(p1.species) # human
print(p2.species) # human
```

every instance changed

- If the change is done on an instance, the change affects only that instance

```
p1.species = "human being"
print(Person.species) # human
print(p1.species) # human being
print(p2.species) # human
```

only single instance changed

Class Methods

- Instance methods
  - are defined within the class
  - are invoked from an instance using dot notation
  - take an instance as a parameter (`self`)
  - affect only attribute of that instance

- Similarly to class variables, we can have also class methods
  - are defined within the class
  - are invoked using the name of the class using dot notation
  - take the class itself (**`cls`**) as a parameter, not an instance of the class
  - can affect class variables

- For example we can define a function to change the class variable species

```
Person.changespecies("human")
```

Class Methods Definition

1) Create a function inside the class using `cls` as first argument
2) Either
   1) invoke `funcname = classmethod(funcname)`
   2) use the decorator `@classmethod` decorator

```
class Person(object):
        # …
        def changeSpecies(cls, newspecies):
                cls.species = newspecies
        changeSpecies = classmethod(changeSpecies)

class Person(object):
        # …
        @classmethod
        def changeSpecies(cls, newspecies):
                cls.species = newspecies
```

Static Methods

- Instance methods are offered by an instance and can impact the instance itself
- Class methods are offered by a class and can impact the class itself

- **Static methods** are offered by a class but only impact the "outside world"
  - For example, a print that can be true for every person
  - They are accessed as Class Methods

```
Person.printhello() # prints "Hello, I am a Person"
```

  - They are defined similarly to class methods, but do not require the `cls` argument
    - First define the method and then either
      - invoke `funcname = staticmethod(funcname)`
      - Decorate it with `@staticmethod`

Static Methods

- Instance methods are offered by an instance and can impact the instance itself
- Class methods are offered by a class and can impact the class itself

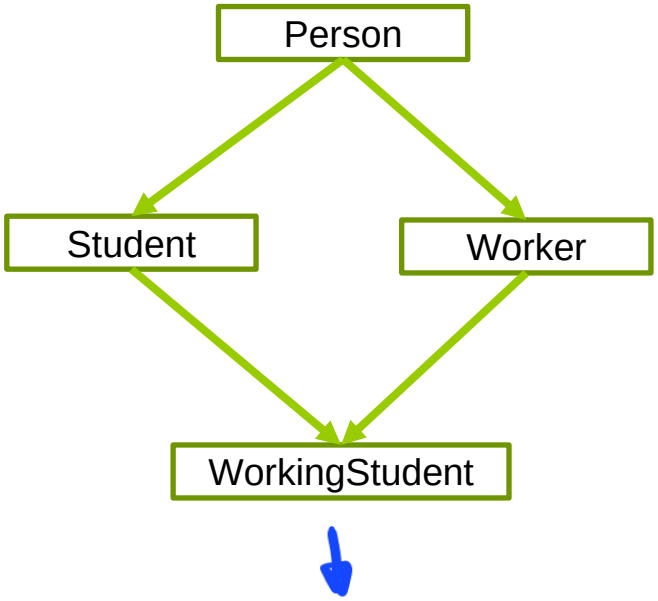
- **Static methods** are offered by a class but only impact the "outside world"
  - For example, a print that can be true for every person
  - They are accessed as Class Methods

```
Person.printhello() # prints "Hello, I am a Person"
```

  - They are defined similarly to class methods, but do not require the `cls` argument
    - First define the method and then either
      - invoke `funcname = staticmethod(funcname)`
      - Decorate it with `@staticmethod`

Multiple Inheritance

- Can a class inherit from more than one class?
  - If we have Students and Workers, can we model a WorkingStudent?
- Python supports **multiple inheritance**



Share the attributes of both inheritance

```
class Person(object):
    def __init__(self, name, familyName):
        self.name = name
        self.familyName = familyName

class Student(Person):
    def __init__(self, name, familyName, course):
        super.__init__(name, familyName)
        self.course = course

class Worker(Person):
    def __init__(self, name, familyName, job):
        super.__init__(name, familyName)
        self.job = job

class WorkingStudent(Worker, Student):
    def __init__(self, name, familyName, job, course):
        super.__init__() # ??? Which parameters? In which order?
```

Multiple Inheritance - MRO

- The order in which the `__init__` methods will be called is decided by Python following the **Method Resolution Order (MRO)**
- We can investigate it with `classname.mro()`

```
print(WorkingStudent.mro())
''' output: [<class '__main__.WorkingStudent'>, <class
'__main__.Worker'>, <class '__main__.Student'>, <class
'__main__.Person'>, <class 'object'>] '''
```

our class

grand parent object

- In the definition of the constructors, we can use `*args` to not worry about it, python will use them in the right order

```
class Student(Person):
    def __init__(self, course, *args):
        super().__init__(*args)
        self.course = course
```

not to worry about the order

```
class WorkingStudent(Worker, Student):
    def __init__(self, *args):
        super().__init__(*args)
```

instead we write this

- In the creation, we must give the arguments in the right order following the MRO

Multiple Inheritance – Example

```
class Person(object):
    def __init__(self, name, familyName):
        self.name = name
        self.familyName = familyName

class Student(Person):
    def __init__(self, course, *args):
        super().__init__(*args)
        self.course = course

class Worker(Person):
    def __init__(self, job, *args):
        super().__init__(*args)
        self.job = job

class WorkingStudent(Worker, Student):
    def __init__(self, *args):
        super().__init__(*args)
```

```
ws = WorkingStudent('researcher',
'RTSA', 'Andrea', 'Galassi')


print('Name:', ws.name)
print('Family:', ws.familyName)
print('Job:', ws.job)
print('Course:', ws.course)


'''
Name: Andrea
Family: Galassi
Job: researcher
Course: RTSA Phylosophy
'''
```