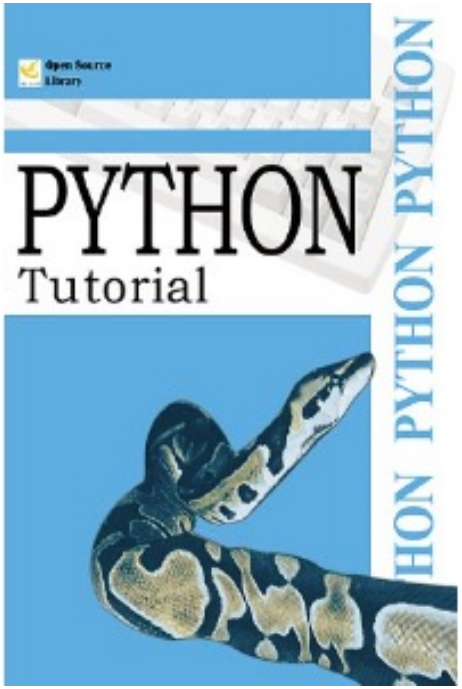
Classes



Dra. Mª Dolores Rodríguez Moreno

Objectives

Specific Objectives

- Understanding class and concepts about class
- Generate classes in Python

Source

- https://python-textbok.readthedocs.io/en/1.0/Classes.html#overriding-magic-methods
- https://docs.python.org/es/3/tutorial/classes.html
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)

Outline

- Classes & Instances
- Initializers
- self
- Access Control
- Instance attributes
- Getters and setters
- Class attributes & methods
- @property

Classes (I)

- Classes group related data and functions
- We've actually been using objects and their methods already!

example

```
s = 'Marien'
print(s.count('M'))
print(s.replace('e', 'a'))
```

- A class is a data type like string or integer
- An instance of a class is an object of that type

Classes (II)

- In Python, everything is an object – instances of some class
- Built-in types and user-defined classes are indistinguishable
- Classes and types are themselves objects, and they are of type “type”

example

```
s = “Marien”
type(s) #<class ‘str’>
type(str) #<class ‘type’>
```

Instances

- Classes make instances of objects
- *string* is a class, '*s*' is an instance of a string
- Make new instances using class name: cad = str(), i = int(2)
- Objects can hold information
- Objects can perform actions: print(s.count('e'))

example

```
s = 'Marien'
print(s.count('M'))
print(s.replace('e', 'a'))
```

Classes as Concepts

- Classes allow us to add new concepts to a language
- Suppose we wanted to add a '*Person*' concept to Python:
  - What information should '*Person*' capture?
  - What actions or operations should '*Person*' provide?

Example

```
class Person:
    pass
```

Outline

- Classes & Instances
- **Initializers**
- self
- Access Control
- Instance attributes
- Getters and setters
- Class attributes & methods
- @property

Initializer

- Special method called *__init__*
- Called automatically when a new instance of a class is created
- The purpose is to initialize the object's attributes
- Commonly referred to as the "constructor" but les accurate since object creation is handled by the *__new__* method (rarely overriden)
- *__init__* can be empty or absent

Example

```
class Person:
    def __init__(...):
        ...
```

Outline

- Classes & Instances
- Initializers
- **self**
- Access Control
- Instance attributes
- Getters and setters
- Class attributes & methods
- @property

self

- A reference to the current instance of the class
- It is used to access variables and methods associated with the object
- It must be the first parameter of any function in the class
- Whenever we call a method on an object, the object itself is automatically passed in as the first parameter (no need to be called *self*)

Example

```
class Person:
    def __init__(self, name, age):
        self.__name = name
        self.__age = age
```

Person Class example

- Data members:
  - name, surname,birthdate, address, telephone, email
- Methods:
  - Age, name, surname…
- Let's code the class

Example: class

```
import datetime #we use this for date objects

class Person:
    def __init__(self, name, surname, birthdate, address, telephone, email):
        self.name = name
        self.surname = surname
        self.birthdate = birthdate
        self.address = address
        self.telephone = telephone
        self.email = email

    def age(self):
        today = datetime.date.today()
        age = today.year - self.birthdate.year
        if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
            age -= 1

        return age
```

Example: instance

```
person = Person(
    "Jane",
    "Doe",
    datetime.date(1992, 3, 12), # year, month, day
    "No. 12 Short Street, Greenville",
    "555 456 0987",
    "jane.doe@example.com"
)

print(person.name)

print(person.email)

print(person.age())
```

Outline

- Classes & Instances
- Initializers
- self
- **Access Control**
- Instance attributes
- Getters and setters
- Class attributes & methods
- @property

Access Control

- Python relies on naming conventions to indicate the intended level of access
  - Public Attributes and Methods
  - Protected Attributes and Methods
  - Private Attributes and Methods

Public members

- Accessible from anywhere, both inside and outside the class
- No special prefix is used

Example

```
class MyClass:

    def __init__(self):

        self.public_attribute = "I am public"

    def public_method(self):

        return "This is a public method"
```

Protected members

- Intended to be accessible within the class and subclasses
- Starting names with a single underscore (_)
- This is just a convention and not enforced by Python

Example

```
class MyClass:

    def __init__(self):

        self._protected_attribute = "I am protected"

    def _protected_method(self):

        return "This is a protected method"
```

Private members

- Intended to be accessible only within the class where they are defined
- Starting names with 2 underscore (__)
- Python performs "name mangling" on these members to make it harder to access them from outside the class

Example

```
class MyClass:

    def __init__(self):
        self.__private_attribute = "I am private"

    def __private_method(self):
        return "This is a private method"
```

Private members: *Name mangling*

- Se aplica a los atributos y métodos con doble guion bajo al inicio de su nombre en una clase
- Esta técnica se utiliza para proporcionar una forma de privacidad en Python, aunque no es una privacidad estricta como en otros lenguajes
- Propósito:
  1. Evitar colisiones de nombres en las subclases
  2. Proporcionar un nivel de privacidad para atributos y métodos
- Funcionamiento:
  1. Cuando se define un atributo o método con doble guion bajo (por ejemplo, __atributo), Python modifica internamente el nombre.
  2. La modificación consiste en anteponer al nombre el prefijo "_NombreClase"
- Fórmula: el nombre se cambia de __atributo a _NombreClase__atributo (Mejor un método)

Ejemplo:

```
# print(objeto.__atributo_privado() #Error
# print(objeto.__metodo_privado()) #Error
```

Private members: *Name mangling (II)*

- Implicaciones:
  - No es una verdadera privacidad, ya que aún se puede acceder conociendo el nombre modificado
  - Dificulta el acceso accidental o no intencionado a estos atributos o métodos
  - Ayuda a evitar conflictos de nombres en la herencia
- Limitaciones:
  - No se aplica a nombres que comienzan y terminan con doble guion bajo (como **init**).
  - No se aplica si el nombre tiene más de un guion bajo al inicio.
- Uso en la práctica:
  5. Se usa cuando realmente se quiere evitar conflictos de nombres en subclases.
  6. No se recomienda usarlo solo para indicar que algo es privado (para eso se usa un solo guion bajo).
- Es importante entender que el name mangling no está diseñado para proporcionar seguridad real, sino para evitar conflictos y accesos accidentales. En Python, la convención es más importante que la restricción forzada, siguiendo el principio de "somos todos adultos aquí"

Outline

- Classes & Instances
- Initializers
- self
- Access Control
- **Instance attributes**
- Getters and setters
- Class attributes & methods
- @property

Instances attributes /Instanciar atributos

- Attributes set in *__init__* do not exhaustively list all possible attributes
- You can add new attributes and methods on the fly
- The initializer is not the only place to set attributes
- *hasattr()* is a built-in function that checks if a specified attribute exists
  Params:
  - object: the object to be inspected for the attribute
  - attribute: the name of the attribute to check for
  - returns: True if attribute exists, False, otherwise
  - commonly used to dynamically check for the presence of attributes

Dynamic Attribute Assignment

Example

```
def age(self):
    if hasattr(self, "age"):  #is there an attribute named “age” then return it
        return self.age

    today = datetime.date.today()

    age = today.year - self.birthdate.year

    if today < datetime.date(today.year, self.birthdate.month, self.birthdate.day):
        age -= 1

    self.age = age

    return age
```

Risks

- WARNING:
  - Attributes can be added outside *__init__* and even add new ones from the outside:

Example

```
print(person.email)
person.pets = ['cat', 'cat1', 'dog']
print(person.pets)
```

  - It's better practice **to initialize attributes** in *__init__*
  - An *__init__* method doesn't have to take any parameters (except self) and it can be completely absent
  - An alternative: __slot__ allows you to restrict the attributes that an object can have, improving performance and memory usage

__slots__

Example: without __slots__

```
class Persona:

   def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

p = Persona("Ana", 25)

p.nombre = "Ana María"

p.edad = 26

p.apellido = "López"
```

__slots__

Example: __slots__

```
class Persona:

    __slots__ = ['nombre', 'edad']  # Only 2 attributes

    def __init__(self, nombre, edad):

        self.nombre = nombre

        self.edad = edad

p = Persona("Ana", 25)

p.nombre = "Ana María"

p.edad = 26

# p.apellido = "López"  # ✗ AttributeError is not in __slots__
```

Exercise

- Improve class Person so we use appropriate access control for attributes and methods
- We add pet as another attribute and should be added to the person through a method
- Output should be:

```
Jane
jane.doe@example.com
32
['cat', `cat1', 'dog']
```

Outline

- Classes & Instances
- Initializers
- self
- Access Control
- Instance attributes
- **Getters and setters**
- Class attributes & methods
- @property

Getters and setters

- Getters: Methods that "get" the values of attributes
- Setters: Methods that "set" the values of attributes
- Some languages encourage using getters and setters for all attributes and make attributes inaccessible except through getters and setters
- Python's Approach:
  - Accessing simple attributes directly is acceptable
  - Using getters and setters for all attributes is considered unnecessarily
  - It provides built-in functions: getattr() and setattr()

getattr()

- Is a built-in function to get the value of an attribute (is NOT a method)
- It allows dynamic access to an attribute's value
- Syntax: getattr(object, 'attribute_name', default_value)
  - object: the object whose attribute's value you want to retrieve
  - attribute_name: the name of the attribute (as a string)
  - default_value: optional; value returned if the attribute does not exist

Example getattr()

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age

person = Person("Alice", 30)

# Using getattr to access attributes

print(getattr(person, 'name'))  # Output: Alice

print(getattr(person, 'age'))   # Output: 30

print(getattr(person, 'address', 'Unknown'))  # Output: Unknown (default value)
```

setattr()

- Is a built-in function to set the value of an attribute (is NOT a method)
- Allows dynamic access to an attribute's value
- Allows adding new attributes to objects on the fly, if the attribute doesn't exist, it will create it
- Syntax: setattr(object, 'attribute_name', value)
  - object: the object whose attribute's value you want to set
  - attribute_name: the name of the attribute (as a string)
  - value: the value to set for the attribute

Example setattr()

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age

person = Person("Bob", 25)

# Using setattr to set attributes

setattr(person, 'name', 'Robert')

setattr(person, 'age', 26)

print(person.name)  # Output: Robert

print(person.age)   # Output: 26
```

Dynamic Attribute Access

- You can use getattr() and setattr() together for dynamic attribute manipulation
- Useful for loops or when attribute names are not known beforehand

- Useful in scenarios where attribute names are stored as strings
- Common in frameworks, libraries, and data manipulation tasks
- You should only use these functions if you **have a good reason** to do so

Example I

```
class Person:

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

person = Person("Charlie", 40)

# Attributes to update

attributes = {
    'name': 'Charles',
    'age': 41,
    'address': '123 Main St'
}
```

```
# Dynamically update attributes

for key, value in attributes.items():

    setattr(person, key, value)



# Dynamically access updated attributes for
key in attributes.keys():

    print(f"{key}: {getattr(person, key, 'Not
Set')}")
```

Example II

```
class Config:

    def __init__(self, **entries):
        for key, value in entries.items():
            setattr(self, key, value)

config = Config(debug=True, db_host="localhost", db_port=3306)

# Accessing configuration values dynamically

print(getattr(config, 'debug'))    # Output: True

print(getattr(config, 'db_host'))  # Output: localhost

print(getattr(config, 'db_port'))  # Output: 3306
```

Outline

- Classes & Instances
- Initializers
- self
- Access Control
- Instance attributes
- Getters and setters
- **Class attributes & methods**
- @property

Class attributes (I)

- Defined in the class body and shared across all instances

Example

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms') # Class Attribute

    def __init__(self, title, name, surname):

      if title not in self.TITLES:

            raise ValueError(f"{title} is not a valid title.")

      self.title = title

      self.name = name

      self.surname = surname
```

Class attributes (II)

- Can be accessed both from instances and the class itself
- They are shared among all instances (but instance attributes are unique to each instance, don't get confused!!)

Example

```
person = Person('Mr', 'John', 'Doe')

print(person.TITLES)    # Access from instance

print(Person.TITLES)    # Access from class
```

Default Values

- Class attributes can provide default values
- Explanation: if an instance attribute with the same name is set, it overrides the class attribute

Example

```
class Person:

    deceased = False

    def mark_as_deceased(self):

        self.deceased = True
```

Mutable Class Attributes

- Be cautious with mutable class attributes (e.g., lists, dictionaries)
- Changes affect ALL instances sharing the attribute

Example: Mutable Class Attributes (WRONG USE)

```
class Person:

    pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()

bob = Person()

jane.add_pet("cat")

print(jane.pets)  # ['cat']

print(bob.pets)   # ['cat']
```

Example: Mutable Instance Attribute (CORRECT USE)

```
class Person:

    def __init__(self):
        self.pets = []

    def add_pet(self, pet):
        self.pets.append(pet)

jane = Person()

bob = Person()

jane.add_pet("cat")

print(jane.pets)  # ['cat']

print(bob.pets)   # []
```

Scope of Class Attributes

- Method definitions are in the same scope as class attribute definitions
- We can use class attribute names as variables in method definitions (without self, which is only defined inside the methods)

Example: Scope

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, title, name, surname, allowed_titles=TITLES):

        if title not in allowed_titles:
            raise ValueError("%s is not a valid title." % title)

        self.title = title

        self.name = name

        self.surname = surname
```

Class Methods

- Special methods within a class that operate on the class
- Useful for operations related to the class, rather than instances
- First parameter is the class itself → we use the special argument *cls*
- Similar to *self*, but it refers to the class rather than the object
- Define it by using *@classmethod*

Example: Class Methods

```
class Person:

    TITLES = ('Dr', 'Mr', 'Mrs', 'Ms')

    def __init__(self, name, surname):

        self.name = name

        self.surname = surname

    @classmethod

    def allowed_titles_starting_with(cls, startswith):

        return [title for title in cls.TITLES if title.startswith(startswith)]

Person. allowed_titles_starting_with('Ms') #Call to the Class Method
```

Static Methods

- Methods within a class that do not receive the calling object/class (*self* or *cls*)
- This means they have no direct access to class or instance attributes
- Can be called from both instances and class objects, but they are more commonly called from classes
- Use Cases:
  - Grouping related functions that do not need access to class or instance data
  - Creating utility functions within a class for convenience
- Define it by using @*staticmethod*

Example I: Static Methods

```
class Person:

    @staticmethod

    def is_adult(age):

        return age >= 18

Person.is_adult(20)   #True
```

Example II: Static Methods

```
class Person:

   %Copy previous code for Person class

    @staticmethod

    def allowed_titles_ending_with(endswith): # static method

        # no parameter for class or instance object
        # we have to use Person directly

        return [t for t in Person.TITLES if t.endswith(endswith)]

jane = Person("Jane", "Smith")

print(Person.allowed_titles_starting_with("M"))#print(jane.allowed_titles_starting_with("M"))

print(Person.allowed_titles_ending_with("s"))#print(jane.allowed_titles_ending_with("s"))
```

Exercise

Create a Person class with the following specifications:

- A public class attribute *species* with the value 'Homo Sapiens'
- Private instance attributes: name, surname, and birthdate
- Set the date of birth as a string: “1990-2-23” (*strptime*)
- Protected method: *_age* to calculate the age from the birthdate
- Class method *from_string* that takes a string in the format "name surname birthdate" and returns an instance of Person
- Static method *is_adult* that takes an age and returns True if the age is 18 or older, and False otherwise

Outline

- Classes & Instances
- Initializers
- self
- Access Control
- Instance attributes
- Getters and setters
- Class attributes & methods
- **@property**

Motivation

- Some languages encourage using getters and setters for all attributes and make attributes inaccessible except through getters and setters
- Python's Approach:
  - Accessing simple attributes directly is acceptable
  - Using getters and setters for all attributes is considered unnecessarily verbose
- Potential Drawback of setters is that they don't allow use of compound assignment operators with immutable types (+=, -=, *=, etc.)
- Python Philosophy:
  - Favours direct attribute access for simplicity
  - Uses properties and methods only when additional logic or control is needed

Example: Setters with Immutable type

```
class Contador:

    def __init__(self):

        self._valor = 0

    @property

    def valor(self):

        return self._valor

    @valor.setter

    def valor(self, v):

        self._valor = v

c = Contador()

# c.valor += 1  # Error because it tries: c.valor = c.valor + 1
```

Example: Setters with mutable type

```
class ListaContainer:

    def __init__(self):

        self._lista = []

    @property

    def lista(self):

        return self._lista

    @lista.setter

    def lista(self, value):

        self._lista = value

contenedor = ListaContainer()

contenedor.lista.append(1)  # It works
```

Example: Setters

```
class Person:

    def __init__(self, height):

        self.height = height

    def get_height(self):

        return self.height

    def set_height(self, height):

        self.height = height

jane = Person(153) # Jane is 153cm tall

jane.height += 1 # Jane grows by a centimetre (use the attribute)

jane.set_height(jane.height + 1) # Jane grows again (use the set method)
```

@property

- Transforms a method into a "getter" attribute
- Allows accessing methods as if they were attributes
- Facilitates encapsulation and control of data access

Getter

```
@property

def height(self):

    ...
```

@property.setter

- Transforms a method into a ”setter"
- Allows accessing methods as if they were attributes
- It can be used to add validation or other logic to the setter method
- Facilitates encapsulation and control of data access

Setter

```
@height.setter

def height(self, new_height):

    ...
```

Example: @property

```
class Person:

    def __init__(self, height):

        self._height = height

    @property

    def height(self):

        return self._height

    @height.setter

    def height(self, new_height):

        self._height = abs(new_height)      #use import math to calculate absolute
```

Example: calling “height”

```
jane = Person(153)  # Jane is 153cm tall

# Using the property directly for reading and modifying with validation

print(jane.height)  # Output: 153

jane.height += 1  # This will work with validation (height becomes 154)

print(jane.height)

jane.height = 160  # Direct assignment also works

print(jane.height)  # Output: 160
```

Inspecting an object

- We can check what properties are defined on an object using the dir function:
- In Python 3 classes inherit from the built-in object *class* by default

dir(jane)

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__','__hash__',
'__init__',  '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__','__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'__weakref__', '_height']
```

- `__init__`: the initialisation method of an object, which is called when the object is created.
- `__str__`: the string representation method of an object, which is called when you use the `str` function to convert that object to a string.
- `__class__`: an attribute which stores the the class (or type) of an object – this is what is returned when you use the `type` function on the object.
- `__eq__`: a method which determines whether this object is equal to another. There are also other methods for determining if it's not equal, less than, etc.. These methods are used in object comparisons, for example when we use the equality operator `==` to check if two objects are equal.
- `__add__` is a method which allows this object to be added to another object. There are equivalent methods for all the other arithmetic operators. Not all objects support all arithemtic operations – numbers have all of these methods defined, but other objects may only have a subset.
- `__iter__`: a method which returns an iterator over the object – we will find it on strings, lists and other iterables. It is executed when we use the `iter` function on the object.
- `__len__`: a method which calculates the length of an object – we will find it on sequences. It is executed when we use the `len` function of an object.
- `__dict__`: a dictionary which contains all the instance attributes of an object, with their names as keys. It can be useful if we want to iterate over all the attributes of an object. `__dict__` does not include any methods, class attributes or special default attributes like `__class__`.

Exercise

Add to the previous Person class:

- A property method fullname that returns the full name of the person
- A setter for the fullname property that allows updating the name and surname using a single string in the format "name surname"

Exercise

Testing the Solution

- Create an instance of Person using the regular constructor and check the values
- Create another instance using the from_string class method
- Check if specific ages correspond to adults using the is_adult static method
- Update the fullname of an instance and verify the changes to name and surname