Introduction to Python

Session 09

Introduction to Object Oriented Programming (OOP)

Special Methods

Introduction to OOP

Object Oriented Programming

Class: Defines the structure: attributes and methods

Student
Name
Surname
Identification Number
Birth date

Instances: Specific realization of any class. Objects that exist in a given program execution

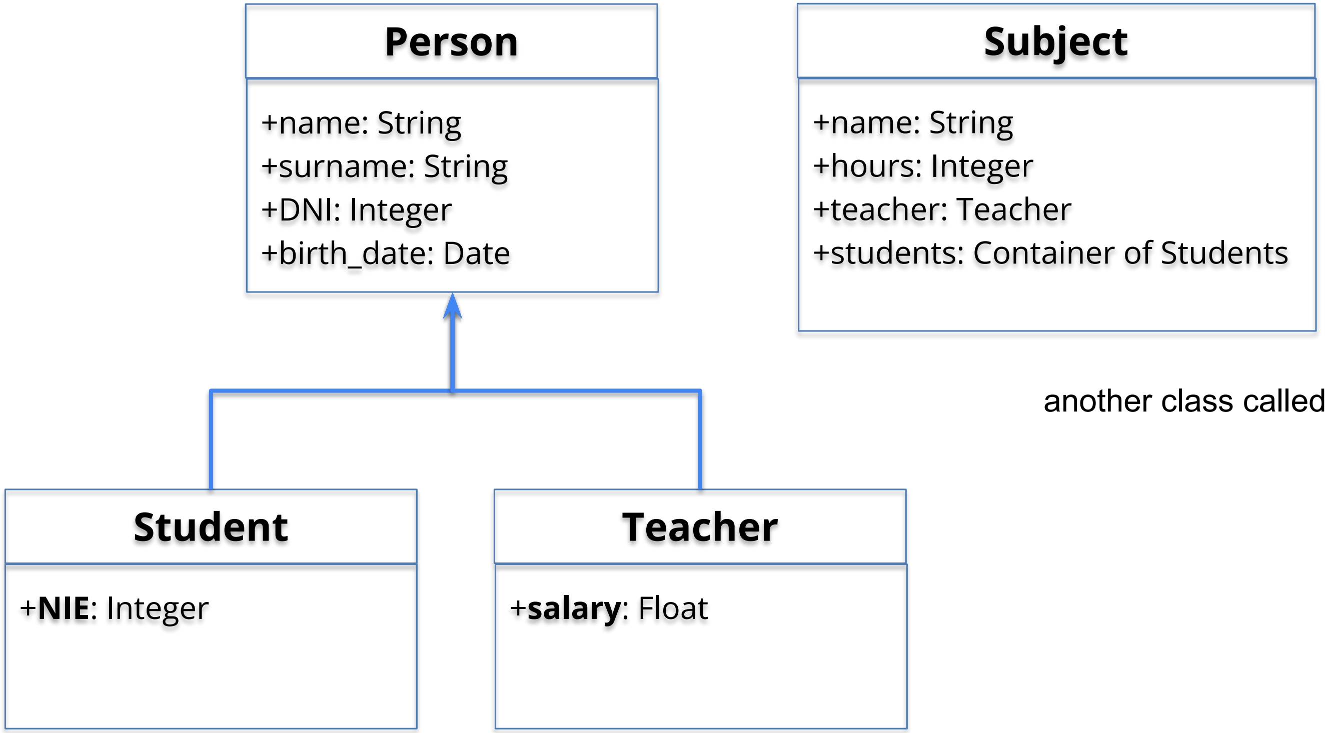
Name: Antonio
Surname: Gómez
Identification Number: 1234
Birth date: 1/1/1990

Name: Alba
Surname: González
Identification Number: 3456
Birth date: 1/1/1992

Name: Agapito
Surname: Garcia
Identification Number: 2827
Birth date: 21/10/1992

Example

superclass called person



another class called subject

this childs will inherit everything form person

Example

```
class Subject:

    def __init__(self, name, hours, teacher):

        self.name = name
        self.hours = hours
        self.teacher = teacher

        self.students = set()

    def add_student(self, student):

        self.students.add(student)
```

Example

Type text here

```
class Person:

    def __init__(self, name, surname, NIE, birth_date):

        self.name = name
        self.surname = surname
        self.NIE = NIE
        self.birth_date = birth_date


class Teacher(Person):

    def __init__(self, name, surname, NIE, birth_date, salary):

        self.salary = salary
        super().__init__(name=name,
                         surname=surname,
                         NIE=NIE,
                         birth_date = birth_date )

class Student(Person):
    pass
```

…

Printing objects

Object conversion to a String representation

```
student_instance1 = Student(   name="Pere",
                               surname="Anton",
                               birth_date="1/1/1990",
                               NIE="u5241")

student_instance2 = Student(   name="Pere",
                               surname="Anton",
                               birth_date="1/1/1990",
                               NIE="u5241”)
```

Printing objects

Object conversion to a String representation

```
>>> print(student_instance1)
<__main__.Student object at 0x102face50>
```

By default, Python prints the class of the object and the memory position of that object instance.

Printing objects

`__str__` is the method called by **`print()`** to get a string representation of the object. It must return a string.

returns a string and what will get when call print

`__repr__` is the method called by the interactive interpreter. Used to recreate the object when passed to **`eval()`**. Used for development and debugging.

when jupyter calls it

```
class Person(object):

    def __str__(self):
        return "I am a Person object instance. My name is %s %s and
my NIE is %s" %(self.name, self.surname, self.NIE)
```

```
>>> print(student_instance1)
‘I am a Person object instance. My name is Pere Anton and my NIE is
u5241’

>>> str(student_instance1)
'I am a Person object instance. My name is Pere Anton and my NIE is
u5241’
```

Comparing objects

```
student_instance1 = Student(   name="Pere",
                               surname="Anton",
                               birth_date="1/1/1990",
                               NIE="u5241")

student_instance2 = Student(   name="Pere",
                               surname="Anton",
                               birth_date="1/1/1990",
                               NIE="u5241")

student_instance1 == student_instance2
False

student_instance1 != student_instance2
True
```

Comparing objects

__eq__ is the method called by the == operator .

```
class Person(object):

    def __init__(self, name, surname, NIE, birth_date):

        self.name = name
        self.surname = surname
        self.NIE = NIE
        self.birth_date = birth_date

    def __eq__(self, other_person):

        return self.NIE == other_person.NIE
```

```
student_instance1 == student_instance2
True

student_instance1 != student_instance2
False
```

Comparing objects: sorting

```
student_instance1 = Student("Pere”, “Anton", "1/1/1990", "u5241")
student_instance2 = Student("Pere", “Anton", "1/1/1990", "u5241")
student_instance3 = Student("Nuria","Gonzalez",1 "20/10/1992","u22312")
student_instance4 = Student("Toni","Bonet","20/10/1992","u22312")


my_student_list = [ student_instance1, student_instance2, student_instance3,
student_instance4 ]


for student in my_student_list:
    print(student.name, student.surname)


('Pere', 'Anton')
('Pere', 'Anton')
('Nuria', 'Gonzalez')
('Toni', 'Bonet')
```

two instances of student,
but diff ids

Same order as
introduced

Comparing objects: sorting

```
student_instance1 = Student("Pere”, “Anton", 123456, "1/1/1990", "u5241")
student_instance2 = Student("Pere", “Anton", 123456, "1/1/1990", "u5241")
student_instance3 = Student("Nuria","Gonzalez",1738172,"20/10/1992","u22312")
student_instance4 = Student("Toni","Bonet",1738172,"20/10/1992","u22312")


my_student_list = [ student_instance1, student_instance2, student_instance3,
student_instance4 ]

my_student_list.sort()

for student in my_student_list:
    print(student.name, student.surname)


('Pere', 'Anton')
('Pere', 'Anton')
('Nuria', 'Gonzalez')
('Toni', 'Bonet')
```

Same order as
introduced again

Comparing objects

define this methods to sort the list!

```
object.__lt__(self, other)        <
object.__le__(self, other)        <=

object.__gt__(self, other)        >
object.__ge__(self, other)        >=

object.__eq__(self, other)        ==
object.__ne__(self, other)        !=
```

Comparing objects: sorting

```
class Person:

    def __init__(self, name, surname, NIE, birth_date):

        self.name = name
        self.surname = surname
        self.NIE = NIE
        self.birth_date = birth_date

    def __eq__(self, other_person):
        return self.NIE == other_person.NIE

    def __lt__(self, other):
        return self.surname < other.surname

    def __le__(self, other):
        return self.surname <= other.surname

    def __gt__(self, other):
        return self.surname > other.surname

    def __ge__(self, other):
        return self.surname >= other.surname
```

Comparing objects: sorting

```
student_instance1 = Student("Pere”, “Anton", 123456, "1/1/1990", "u5241")
student_instance2 = Student("Pere", “Anton", 123456, "1/1/1990", "u5241")
student_instance3 = Student("Nuria","Gonzalez",1738172,"20/10/1992","u22312")
student_instance4 = Student("Toni","Bonet",1738172,"20/10/1992","u22312")


my_student_list = [ student_instance1, student_instance2, student_instance3,
student_instance4 ]

my_student_list.sort()

for student in my_student_list:
    print(student.name, student.surname)


('Pere', 'Anton')
('Pere', 'Anton')
('Toni', 'Bonet')
('Nuria', 'Gonzalez')
```

Ordered by surname !

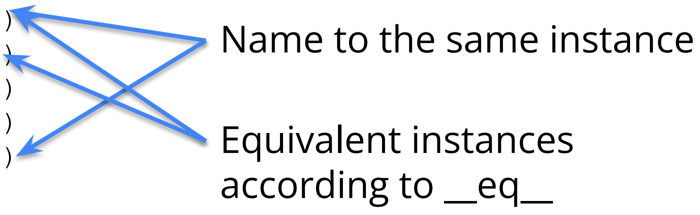
Comparing objects: hashing

```
teacher_instance = Teacher(name="Xavi",surname="Jalencas", NIE=”u1234”,
                           birth_date="1/1/1980", salary=100)

PYT = Subject(name="PYT", hours=125, teacher=teacher_instance)

PYT.add_student(student_instance 1)
PYT.add_student(student_instance 2)
PYT.add_student(student_instance 3)
PYT.add_student(student_instance 4)
PYT.add_student(student_instance 1)


TypeError: unhashable type: 'Student'
```



Name to the same instance

Equivalent instances according to __eq__

As in the class Subject students are added to a Set, duplicated objects are removed.

→ `student_instance1` will appear only once in the set, although we have added it twice.

`student_instance1` and `student_instance2` have equivalent attributes and are considered “equal” according to `__eq__`

→ What python should do ?

Comparing objects: hashing

`__hash__`

- Method that must return an integer
- The hash value returned is used for operations on members of hashed collections: dictionaries, sets…
- Used normally for “immutable” objects.

Comparing objects: hashing

__hash__

Without __hash__: Python treats them as two different objects because they are in different memory locations.

With __hash__: Python sees they have the same fingerprint and same data. It treats them as a duplicate. The set length would be 1.

```
class Person:

    ...

    def __hash__(self):

        return self.NIE.__hash__()
```

In this example, we set the hash of a person as the hash defined by its NIE.

Person objects having the same NIE will be considered duplicates in sets, dictionaries, etc.

The __hash__ method is a special function in Python that allows an object to be used in places that require "uniqueness," such as being a key in a dictionary or an element in a set.

If two objects have the same "fingerprint" (hash), Python then checks if they are actually equal using __eq__.

In Python, if you define __hash__, you must also define __eq__.

Comparing objects: hashing

```
student_instance1 = Student("Pere”,"Anton”,123456,"1/1/1990", NIE="u5241")
student_instance2 = Student(“Pere”,"Anton”,123456,"1/1/1990", NIE="u5241")
student_instance3 = Student("Nuria","Gonzalez",1738172,"20/10/1992","u22312")
student_instance4 = Student("Toni","Bonet",1738172,"20/10/1992","u22312")

teacher_instance = Teacher(name="Xavi",surname="Jalencas", DNI=18276165,
                          birth_date="1/1/1980", salary=100)

PYT = Subject(name="PYT", hours=125, teacher=teacher_instance)

PYT.add_student(student_instance 1)
PYT.add_student(student_instance 2)
PYT.add_student(student_instance 3)
PYT.add_student(student_instance 4)
PYT.add_student(student_instance 1)


print(“Number of students: %d” %(len(PYT.students))

Number of students:  2
```

As students 1 and 2 share same DNI, and students 3 and 4 share same DNI, only 2 students are added to the set

Other special methods

`__len__`

- Returns an integer. Method called when the `len()` function is used.

```
class Subject(object):

        …

        def __len__(self):
            return len(self.students)

print("Number of students: %s" %len(PYT))
```

Other special methods

__contains__ (self, item)

- Should return `True` if item is in self, `False` otherwise.

```
class Subject(object):

        …

        def __contains__(self, student):
                return student in self.students


student_instance1 in PYT
```

Other special methods

`__getitem__` (self, key)

— Method called when accessing using `[ ]`.

Examples:

- Get a specific character or substring in a string object

- Get a specific item or sublist in a List object

`PYT[0]`

Other special methods

`__add__` (self, other)

- Method called when accessing using the operator +

  . When objects are integers, it means “sum”

  . When objects are strings, it means “concatenate”

  . When objects are lists, it means “extend”

```
student_instance_1 + student_instance_2
```

Other special methods

`__iter__`(self)

- Returns an iterator object. Called when the object is iterated.

```
class Subject(object):

        …

        def __iter__(self):
            for student in self.students:
                yield student

for student in PYT:
    print(student)
```

Other special methods

__sub__(self, other):	-
__mul__(self,other):	*
__pow__(self,other):	**
__and__(self, other):	and
__or__(self, other):	or

__abs__

__pos__

__neg__

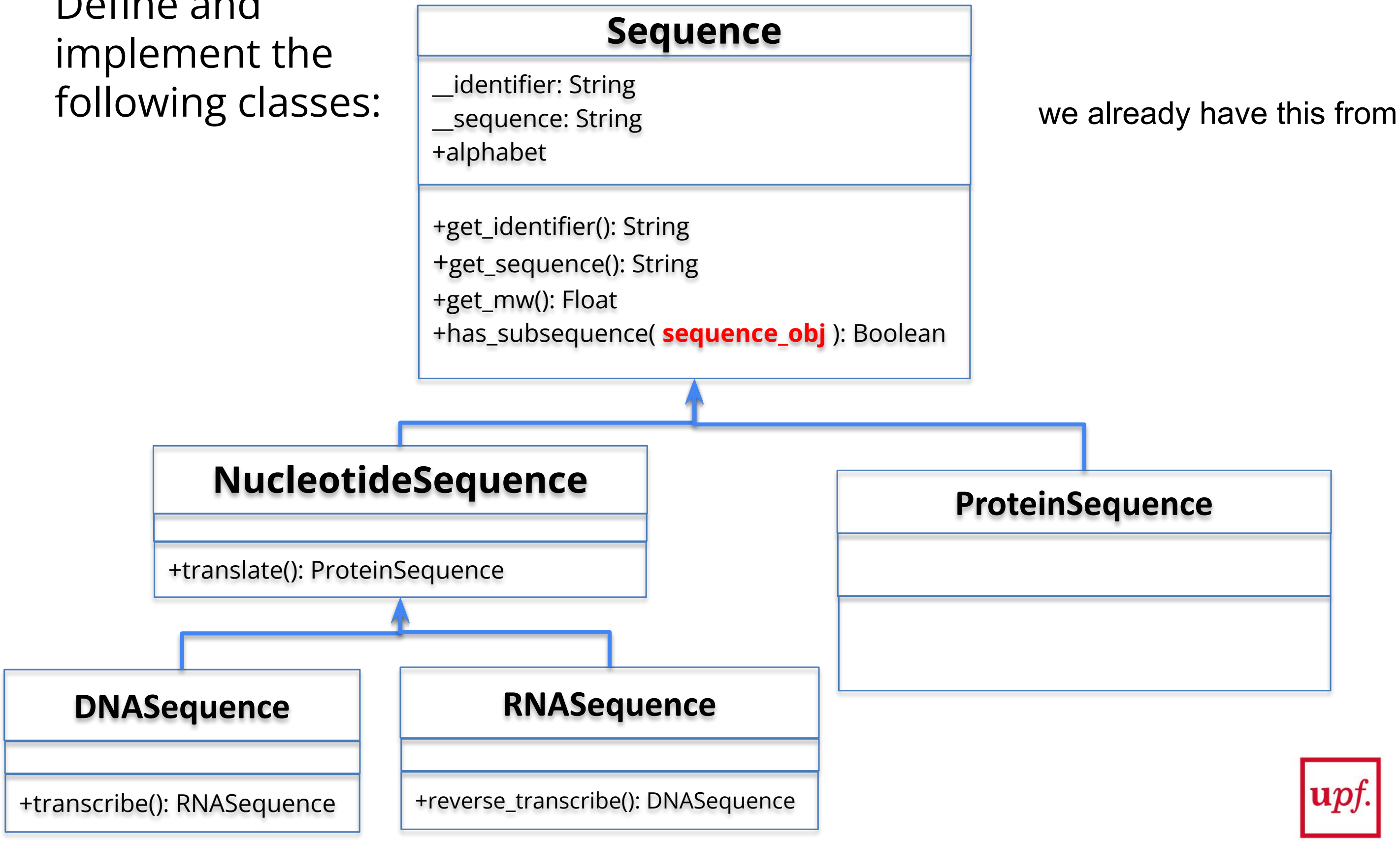
__float__

__int__

Exercises. Session 09

Create a python script called **<uID>IE_S09.py** with:

1) Define and implement the following classes:


Sequence
__identifier: String
__sequence: String
+alphabet
+get_identifier(): String
+get_sequence(): String
+get_mw(): Float
+has_subsequence(**sequence_obj**): Boolean
NucleotideSequence
+translate(): ProteinSequence
ProteinSequence
DNASequence
+transcribe(): RNASequence
RNASequence
+reverse_transcribe(): DNASequence

we already have this from the previous exercises define the special methods we have been seen

Exercises. Session 09

Define the following behaviour for the classes defined in exercise block 2 part 2:

1) **`len(Sequence)`**: should return the length of the sequence.

2) **`sequence1 == sequence2`**: return True if sequence strings are exactly the same (without taking into account the identifiers). don't care about the identifiers, only the seq

3) **`sequence1 != sequence2`**: return True if sequences are different, without taking into account the identifiers.

add seq of the same class (dna +dna, rna+rna)

4) **`Sequence + Sequence`**: Create a new sequence object instance with their sequences concatenated. Sequence object has to be of the same class as the operands. It should not be applicable to different classes (i.e. ProteinSequence, RNASequence). The identifiers should also be concatenated with a "+" as a glue between both identifiers.

```
def __add__(self, other):
    return Sequence(self.__sequence + other.sequence)
```

Exercises. Session 09

Define the following behaviour for the classes defined in exercise block 2 part 2:

5) **`Sequence[i]:`** should return the sequence element at position i. Position 0 corresponds to the first position. list

6) **`in`** **operator**: should return a boolean if the string is a substring of the attribute sequence. def __contains__

7) **Comparing sequences**. Implement the necessary method(s) to define how sequences should be ordered. The objective is that when sorting a list of sequences, they are sorted according to their molecular weight. def __lt__(self, other):

8) Adapt the sequence class so that it can be used as key in a dictionary or it can be added to a set. Two sequences should be considered the same object in terms of set or key if they share both the identifier **and** the sequence.

uses hash function