Classes (part 1)

Aren't you really tired of all these lists, dictionaries, sets etc and all their whims? Why do I have to put genes in lists, chromosomes in dictionaries, etc.? Wouldn't it be nice to make my own "mechanisms" of managing and storing my data?

This is exactly what classes do!

With the classes you define what operations to do in the data, how they are stored, how they are printed, how I to new entries, how to delete old, and many more..

Let's look at the simplest class that can exist:

In [1]:
```
class Human:
    pass
```

I just made a new class. Lists, dictionaries, sets, exceptions are also classes. Now let's make an *object* from this class:

In [3]:
```
alex = Human()
```

`alex` is a new variable whose type is `Human`

Rember this:

`a = [1,2,3]`

Just as `a` was a variable of type `list`, `alex` is a variable of type `Human`. That is, we have created a new type of variables!

In [3]:
```
type(alex)
```

Out[3]:
```
__main__.Human
```

Here we need to take a break and talk a little bit about terminology. When we write:

`a=3`

then we say that `a` is a variable of type `int` and its value is `3`.

But when we write:

`alex=Human()`

Then what is the value of `alex`?

Precisely because we can not answer this question accurately (and briefly), we say that `alex` is a separate type of variable, which we will call **object**. Or to be more precise, an object of type `Human`.

When we create our own type of variables (that is, when we make an object of a class), then we may add to it any *attribute* we like:

In [4]:
```
alex.name = "Alexandros"
alex.age = 45
```

In [5]:
```
print (alex.name, alex.age)
```

```
Alexandros 45
```

We have seen this `variable.attribute` before, when we did:

```
a = [1,2,4]
a.append(5)
```

Then we used the `append` attribute of the `a` variable which is of type `list`

Let's now make an attribute in `alex` that calculates whether he is an adult or not. One way to do this is:

In [7]:
```
def is_adult(human):
    return human.age >= 18

is_adult(alex)
```

Out[7]: True

Although the above gave us the result we wanted, the `is_adult` function is NOT an alex attribute (such as `name` and `age`).

That is, instead of:

```
is_adult(alex)
```

We want to do:

```
alex.is_adult()
```

Note that `is_adult` now does NOT get an argument!

Then how can I get the `age` to see if he/she is an adult or not?

Python allows us to create functions that can be attributes of an object. To do this, 2 conditions must be met:

- The first condition is that the first argument of the function must be a variable called: `self` . The `self` contains the object for which this function is "called". That is, when we say: `object.method()` , `self` is the `object` . In our case, when we say `alex.is_adult()` , `self` is the `alex` object.
- The function must be defined within the class and not within the object.

These two conditions can be summarized in the following example:

In [6]:
```
class Human:
    # The function is defined inside the class
    def is_adult(self,): # First argument is self
        return self.age >= 18
```

Now I can do:

In [9]:
```
alex = Human()
alex.age = 40
alex.is_adult()
```

Out[9]: True

The above hides a lot of "philosophy" inside. Let's rewrite it:

```
alex = Human()
alex.age = 40
alex.is_adult()
```

Note the following: I have defined a new type of variable (`Human`) which if a add an attribute named `age` then it can calculate whether this `Human` is an adult or not. That is, the new type (Human) creates variables that have a *behavior*: that of calculating whether the variable refers to an adult or not. We also note that one does not need to have an idea of how `is_adult` is implemented, as long as one knows that it is a class *method*. Just like we have no idea how the `append` method has been implemented in lists. A method of a class is an attribute which is a function.

`alex.is_adult()` and `is_adult(alex)` have a huge difference: If we read the first (`alex.is_adult()`) from left to right then we go from general (`alex`) to specific (`is_adult`). While if we read the second then we go from the special to the general.

In "real" life, what is more likely to be asked: "Is Alex an adult?" or "Is adult, Alex?". Classes offer a natural way of defining objects and then referring to the attributes that these objects have.

And a little history

In an effort to make programming more "normal" and closer to human perception, the concept of *object-oriented* programming was introduced in the 1950s. Like almost all programming concepts it has gone through many revisions, implementations and (re)definitions.

What we need to keep in mind is that through OOP (Object Oriented Programming) we can use the python syntax to write things that make sense more easily. An OOP piece of code typically has commands like these:

```
if geneA.is_in_between(geneB)...
if debt.is_paid()....
alex.hire()
if circle_A > circle_B ...
```

The same commands if we assume that we use only lists, dictionaries, sets etc and do not use OOP would be something like this:

```
if geneA['start'] > geneB['start'] and geneA['start'] <
geneB['end']...
if debt['capital']['balance'] == 0.0...
alex['job_status'] = statuses['hired']
if circle_A['radius'] > circle_B['radius']
```

We observe how close to the human perception is the first set of commands.

If you are confused it is normal. Let us briefly summarize the terminology:

- **Class** : A type of data suitable for specific concepts (human, corporate, gene, disease)

- **Object** : A variable whose type is a class. (in correspondence with the above: Alex, public, TPMT, color blindness)
- **Attribute** : An attribute of the class (corresponding to the above: name, number of employees, length, genetic factor)
- **Method** : An attribute that is a function (corresponding to the above: walks, hires, [illegible])

Let's go back to the class we made:

In [10]:
```
class Human:

    def is_adult(self,):
        return self.age >= 18
```

If we make a Human and do not give him/her age, obviously we can not test if he/she is_adult:

In [7]:
```
kostas = Human()
kostas.name="Peter"
kostas.is_adult()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-7ebeb4c15f06> in <module>
      1 kostas = Human()
      2 kostas.name="Peter"
----> 3 kostas.is_adult()

<ipython-input-6-909a17aa7014> in is_adult(self)
      2     # The function is defined inside the class
      3     def is_adult(self,): # First argument is self
----> 4         return self.age >= 18

AttributeError: 'Human' object has no attribute 'age'
```

How can we oblige the programming when declaring an object of type `Human` , to provide his/her `age` ? This can be done with the `__init__()` method:

In [8]:
```
class Human:
    def __init__(self, age):
        self.age = age

    def is_adult(self,):
        return self.age >= 18
```

What is this:

`self.age=age`

Here when we initialize an object, we create an attribute (self.age) and initialize it with the value of the age parameter of `__init__` :

In [14]:
```
alex = Human()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-14-12a27f13d5ce> in <module>
----> 1 alex = Human()

TypeError: __init__() missing 1 required positional argument: 'age'
```

We are obliged to provide an age!

In [15]:
```
alex = Human(age=45)
```

In [16]:
```
alex.is_adult()
```

Out[16]: True

The classes have some "special" methods which are called through python's built-in functions (i.e. `print` , `len` , ...). One of them is `___str___` . This function is called whenever we need a representation of the class in a string (usually via `print`):

In [9]:
```
class Person:
    def __init__(self, name, surname):
        self.name = name
        self.surname = surname
    def __str__(self):
        return '-->{} {}<--'.format(self.name, self.surname)

mitsos = Person("John", "Doe")
print (mitsos)
```

```
-->John Doe<--
```

Another method is `__len__` which is called when we apply `len()` to our object:

In [18]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __len__(self,):
        return self.stop-self.start

tpmt = Gene('TPMT', 150, 200)
len(tpmt) # 200-150
```

Out[18]: 50

Another method is `__getitem__` which allows us to "get" an item from a collection via the operator: `[]` :

In [19]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __getitem__(self, i):
        ret = self.start + i
        if ret > self.stop:
            raise IndexError
        return ret
```

In [20]:
```
tpmt = Gene('TPMT', 150, 200)
tpmt[23]
```

Out[20]: 173

In [21]:
```
tpmt[60]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-39a580f61a36> in <module>
----> 1 tpmt[60]

<ipython-input-19-17b2584805cd> in __getitem__(self, i)
      8         ret = self.start + i
      9         if ret > self.stop:
---> 10             raise IndexError
     11         return ret
     12

IndexError:
```

Two other methods are `__iter__` and `__next__` . By implementing these methods we can do iteration in an object:

In [24]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __iter__(self,):
        self.i = self.start
        return self

    def __next__(self,):
        if self.i == self.stop:
            raise StopIteration

        ret = self.i
        self.i += 1
        return ret
```

In [25]:
```
tpmt = Gene('TPMT', 150, 200)
for x in tpmt:
    print (x)
```

```
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
```

```
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
```

We can even change the behavior of the operators by implementing some special methods. For example we can "support" the `+` operation in the Gene class:

In [35]:
```
class Gene:
    def __init__(self, name, start, stop):
        self.name = name
        self.start = start
        self.stop = stop

    def __add__(self, g):
        return Gene(
            name = f'{self.name}+{g.name}',
            start=self.start,
            stop = self.stop + g.stop-g.start, # We just add the length of
        )

    def __str__(self,):
        return f'Gene: {self.name} Start: {self.start} End:{self.stop}'
```

In [36]:
```
tpmt = Gene('TPMT', 150, 200)
apoe = Gene('APOE', 400, 470)

new_gene = tpmt + apoe
print (new_gene)
```

```
Gene: TPMT+APOE Start: 150 End:270
```

We can see all the "special" methods that a class has:

In [37]:
```
dir(Gene)
```

Out[37]:
```
['__add__',
 '__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__',
```

Classes (part 2)

We have presented how a class can has "length" if it implements the `__len__` method:

In [49]:
```
class Gene:
    def __len__(self,):
        return 50
```

In [50]:
```
b = Gene()
print (len(b))
```

```
50
```

After all, when we apply `len` to a list, `__len__()` is actually executed:

In [51]:
```
a = [1,2,5]
print (a.__len__())
```

```
3
```

Let's create a class that represents a person:

In [10]:
```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

And let's make an object:

In [14]:
```
a = Human('Alex', 50)
```

In [15]:
```
a.name
```

Out[15]:
```
'Alex'
```

As we have seen, `name` and `age` are "attributes" of the class Human. Often a class

needs to have some variables (usually woth constant values) that are the same for ALL objects in that class. Here's how to do it:

In [16]:
```
class Human:
    max_age = 100

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

a = Human('Alex', 50)
b = Human('John', 10)
```

In [17]:
```
print (a.max_age)
```

```
100
```

Notice that we have not set `max_age` in either `mitsos` or in `kwstas`. However, if we `print (a.max_age)` then we get the value that we have set for `Human`. This is because `max_age` exists in ALL objects in the `Human` class. Also if we change the value of this attribute in `Human` then it will change in all objects of this class:

In [18]:
```
Human.max_age = 200
print (b.max_age)
```

```
200
```

Definition: the attributes (fields or methods) of a class that are accessible by the class without the need to create an object, are called static

A method can also be static:

In [58]:
```
class Human:
    max_age = 100

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

    def is_adult(age):
        return age>18

Human.is_adult(30)
```

Out[58]:
```
True
```

The disadvantage of the above implementation is that we can NOT run `is_adult` from an object:

In [19]:
```
alex = Human('Alex', 50)
alex.is_adult(100)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-19-564d0f087293> in <module>
      1 alex = Human('Alex', 50)
----> 2 alex.is_adult(100)

AttributeError: 'Human' object has no attribute 'is_adult'
```

In order for a method to run by referring the class or the object, we must use the

@staticmethod decorator:

In [20]:
```
class Human:
    max_age = 100

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

    @staticmethod
    def is_adult(age):
        return age>18
```

In [21]:
```
print (Human.is_adult(60))
alex = Human('Alex', 50)
print (alex.is_adult(60))
```

```
True
True
```

Note: staticmethod is a decorator , a feature of python that we have not presented ..

A better implementation:

In [22]:
```
class Human:
    max_age = 100
    adult_age = 18

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

    def is_adult(self,):
        return Human.is_adult_2(self.age)

    @staticmethod
    def is_adult_2(age):
        return age >= Human.adult_age



alex = Human('Alex', 50)
print (alex.is_adult()) # True
john = Human('John', 10)
print (john.is_adult()) # False
print (Human.is_adult_2(20)) # True
```

```
True
False
True
```

Classes (Part 3)

One class can "inherit" another class: https://en.wikipedia.org/wiki/Inheritance_(object-oriented_programming)

By doing this, the new class contains all the properties (fields + methods) of the old one. Classic examples are:

- the truck class has inherited the vehicle class
- the DNA class and the RNA class have inherited the sequence class

- the Employee class has inherited the Human class

In python this is done as follows:

In [23]:
```
class Employee(Human):
    pass

alex = Employee('Alex', 40)
```

Employee contains all Human methods:

In [24]:
```
alex.is_adult()
```

Out[24]: True

Let's add a new attribute in Employee:

In [25]:
```
class Employee(Human):
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary

        if not self.is_adult():
            raise Exception('THIS IS ILLEGAL!')

alex = Employee('Alex', 40, 10000)
```

In [26]:
```
john = Employee('John', 15, 10000) # Child labour!
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
<ipython-input-26-61adb21eb4b8> in <module>
----> 1 john = Employee('John', 15, 10000) # Child labour!

<ipython-input-25-2abe1b72610b> in __init__(self, name, age, salary)
      6
      7         if not self.is_adult():
----> 8             raise Exception('THIS IS ILLEGAL!')
      9
     10 alex = Employee('Alex', 40, 10000)

Exception: THIS IS ILLEGAL!
```

Notice that in this piece:

```
self.name = name
        self.age = age
```

Exists in both `Human` and in `Employee` class. Can we avoid this? This is done with the `super` command which calls the parent class.

In [27]:
```
class Employee(Human):
    def __init__(self, name, age, salary):
        super().__init__(name, age)
        self.salary = salary

        if not self.is_adult():
            raise Exception('THIS IS ILLEGAL!')

alex = Employee('Alex', 40, 10000)
```

H `super().__init__()` calls `__init__()` of the parent class.

Multiple inhertance

A class can inherit from more than one classes.

In [30]:
```
class Resident:
    def __init__(self, address):
        self.address = address

    def show_address(self):
        print (self.address)
```

In [31]:
```
class Employee(Human, Resident):
    pass
```

The order in which we declare parent classes is very important. The new class inherits the constructor (`__init__()`) from the first class only:

In [32]:
```
alex = Employee('Alex', 50)
```

Here we notice that the constructor of Resident was not called!

In [33]:
```
alex.show_address()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-33-5f627d50c1b8> in <module>
----> 1 alex.show_address()

<ipython-input-30-3eaf54002b5a> in show_address(self)
      4
      5     def show_address(self):
----> 6         print (self.address)

AttributeError: 'Employee' object has no attribute 'address'
```

We can correct this by calling the constructors in any order we want from the `__init__` of the new class:

In [35]:
```
class Employee(Human, Resident):
    def __init__(self, name, age, address, salary):
        Human.__init__(self, name, age)
        Resident.__init__(self, address)
        self.salary = salary

mitsos = Employee('Alex', 50, "Heraklion", 10000)
print (mitsos.is_adult())
mitsos.show_address()
```

```
True
Heraklion
```

We can have objects in any data structure like dictionaries, lists, ..

In [37]:
```
stuff = {
    0: Employee('Kostas', 40, 'Heraklion', 1000),
    1: Employee('Andreas', 40, 'Patras', 100),
}
```

What happens when we want to save a list / dictionary that has objects in a file? We can not convert them directly to a string:

In [75]:
```
str(stuff)
```

Out[75]:
```
'{0: <__main__.Employee object at 0x7ff76d574880>, 1: <__main__.Employee ob
ject at 0x7ff76d5747c0>}'
```

Nor can we convert them to json:

In [76]:
```
import json
json.dumps(stuff)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-76-54caab5df3b5> in <module>
      1 import json
----> 2 json.dumps(stuff)

~/anaconda3/lib/python3.8/json/__init__.py in dumps(obj, skipkeys, ensure_a
scii, check_circular, allow_nan, cls, indent, separators, default, sort_key
s, **kw)
    229         cls is None and indent is None and separators is None and
    230         default is None and not sort_keys and not kw):
--> 231         return _default_encoder.encode(obj)
    232     if cls is None:
    233         cls = JSONEncoder

~/anaconda3/lib/python3.8/json/encoder.py in encode(self, o)
    197         # exceptions aren't as detailed.  The list call should be r
oughly
    198         # equivalent to the PySequence_Fast that ''.join() would d
o.
--> 199         chunks = self.iterencode(o, _one_shot=True)
    200         if not isinstance(chunks, (list, tuple)):
    201             chunks = list(chunks)

~/anaconda3/lib/python3.8/json/encoder.py in iterencode(self, o, _one_shot)
    255                 self.key_separator, self.item_separator, self.sort_
keys,
    256                 self.skipkeys, _one_shot)
--> 257         return _iterencode(o, 0)
    258
    259 def _make_iterencode(markers, _default, _encoder, _indent, _floatst
r,

~/anaconda3/lib/python3.8/json/encoder.py in default(self, o)
    177
    178         """
--> 179         raise TypeError(f'Object of type {o.__class__.__name__} '
    180                         f'is not JSON serializable')
    181

TypeError: Object of type Employee is not JSON serializable
```

For this purpose we can use the pickle library:

In [38]:
```
import pickle
stuff_serialized = pickle.dumps(stuff)
```

In [39]:
```
stuff_serialized
```

Out[39]:
```
b'\x80\x04\x95\x8c\x00\x00\x00\x00\x00\x00\x00}\x94(K\x00\x8c\x08__main__\x
94\x8c\x08Employee\x94\x93\x94)\x81\x94}\x94(\x8c\x04name\x94\x8c\x06Kosta
s\x94\x8c\x03age\x94K(\x8c\x07address\x94\x8c\tHeraklion\x94\x8c\x06salary\
x94M\xe8\x03ubK\x01h\x03)\x81\x94}\x94(h\x06\x8c\x07Andreas\x94h\x08K(h\t\x
8c\x06Patras\x94h\x0bKdubu.'
```

In [40]:
```
type(stuff_serialized)
```

Out[40]:
```
bytes
```

We can load pickle data:

In [41]:
```
a = pickle.loads(stuff_serialized)
print (a)
```

```
{0: <__main__.Employee object at 0x7fbf4245f970>, 1: <__main__.Employee obj
ect at 0x7fbf4245ff10>}
```

Additional notes

I urge you to read the excellent notes from Spyros Chavlis schavlis@imbb.forth.gr that he had prepared for the 2016 course of the postgraduate program in medicine, on classes and object-oriented programming.