Pythonic Objects Part 1

June 30, 2020

1 Agenda

- Object Oriented Programing
- User Defined Classes
  - How we define a class
  - Initialization
  - Holding Data
  - Holding Logic
  - self
- Class Attributes
- Static Methods and Class Methods
- Inheritance
- Dunder or magic methods (str, repr)

2 Resources

- Python Documentation - A First Look at Classes

So far in our lectures we learned about the structure and behaviour of built-in objects, such as `lists` and `dictionaries`

Next we will work with user-defined classes and will learn how to build them in order to behave as real Python objects

3 Object Oriented Programing

Classes are a way of defining a set of rules by which an object is constructed. Similar to a blueprint for construction workers.

Object oriented programing is a programing paradigm which provides principles of structuring programs so that properties and behaviour are grouped together in individual objects. You are

already familiar with python objects from a usability point of view. In order to understand how we build these objects it is relevant to break down this concept:

`Objects are entities that can hold data in the form of attributes and code in the form of methods`

One of the main advantages of OOP are:

- group the logic of our application around the entities we will use (Users, Products, Articles etc.)
- extensibility - write logic common to multiple objects and use it with inheritance

[1]:
```
class User:
    name = 'John'
    age = 41

    def is_young(self):
        return self.age < 65
```

[2]:
```
john = User()

print(john.name)
print(john.is_young())
```

```
John
True
```

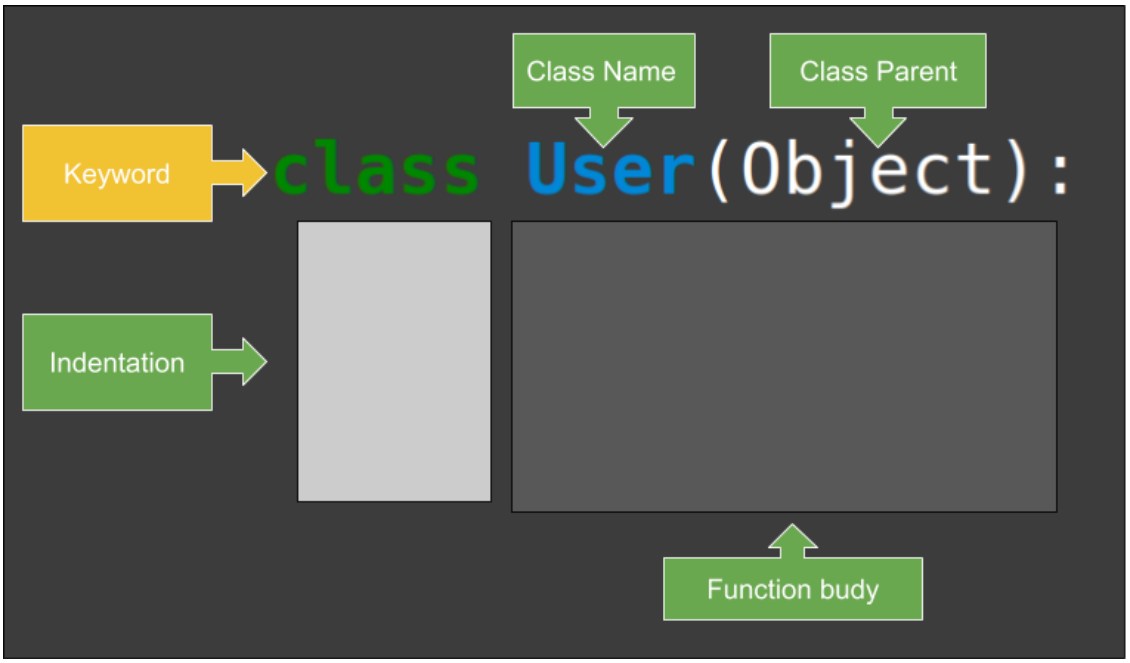
! ! !

It is important to understand that Python is a multi-paradigm language, in which we can program using object oriented programming, functional programming, procedural programming, etc. The choice is based on what kind of problems we are facing and what kind of product we are building.

4 User definde classes -Syntax-

4.1 How we define a class

Similar to how we define a function we use a keyword a name and some arguments. The keyword



[3]:
```
class User(object):

    company = 'Py Future inc.'

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def is_legal_to_drink(self):
        return self.age >= 21
```

Same as functions classes are also objects, that have a name, and id and a type.

[4]:
```
User
print(id(User))
print(type(User))
```

```
94852891807616
<class 'type'>
```

4.2 Initialization

As we mentioned above classes are blueprints for creating objects. The objects that are created using a class are called instances of that class and have all the characteristics of an Python object. In order to create an object with the type of our defined class, we use the `()` call operator. Objects are

created and initialized using the call operator similar to how we call a function with parameters. The arguments we give to the class at the time of the call are the one that will be used for initialization.

[5]:
```
john = User('John', 'Doe', 19)

print(f'Jhon is: {john} of type: {type(john)} with id: {id(john)}')
```

```
Jhon is: <__main__.User object at 0x7f9bcc1dc610> of type: <class
'__main__.User'> with id: 140307121161744
```

In Python, objects have two important phases that they go throw before we can use them in our code. The first phase is when the object is created and the second when the object is initialized.

In order to add data to our custom object, we first need to define what data will it hold. We do this by defining the `__init__` method in our class. The parameters we define in the `__init__` method will be the one we will give to the call operator `()` when we creat each instance of our class. Inside our `__init__` method we need to add the data received from the call as attributes to our object.

[6]:
```
class User(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

[7]:
```
john = User('John', 'Doe', 19)
```

When the above example is evaluated the following steps are performed:

1. An object of type User is created
2. The method `__init__` is called with the object created at step one as the first argument and the rest of the arguments from the call (in our case ('John', 'Doe', 19))
3. Inside `__init__` we add the attributes `first_name`, `last_name` and `age` to the object and assign the values from the call at step 2 to them
4. we have the object ready and the variable `john` pointing to it

4.3 Holding Data

The example below illustrates the capacity of classes to hold information. We are using a dictionary with two keys for creating objects of type `User`. If we check the User class `__init__` method we can see the required arguments in order to create an instance of `User`. We need to adapt the content of the dictionary to accommodate three arguments, in order to do this we will split the name string into first and last name.

[8]:
```
users = [
    {'name':'John Doe', 'age':19},
    {'name':'Jack Fluffy', 'age':22},
    {'name':'Matthew Wu', 'age':43},
    {'name':'Heather Rafferty', 'age':15},
```

```
    {'name':'Randall Blackdall', 'age':76},
    {'name':'Marissa Raynaud', 'age':34},
    {'name':'Marlo Ranbot', 'age':49},
]

users_objects = []
for user  in users:
    users_objects.append(
        User(
            user['name'].split()[0],
            user['name'].split()[1],
            user['age'],

        )
    )
```

If we display the content of our list we will see that we have instances of class `User`. Each instance contains its own information extracted from the initial dictionary. We can access this information by using the following syntax `instance.attribute` where the instance is the user, for us `users_objects[0]...user_objects[n]` and the attribute is one of the fields defined in `__init__`, for us `first_name`, `last_name` and `age`

[9]:
```
users_objects
```

[9]:
```
[<__main__.User at 0x7f9bcc2115d0>,
 <__main__.User at 0x7f9bcc22bad0>,
 <__main__.User at 0x7f9bcc22b690>,
 <__main__.User at 0x7f9bcc22b5d0>,
 <__main__.User at 0x7f9bcc22bc90>,
 <__main__.User at 0x7f9bcc22b790>,
 <__main__.User at 0x7f9bcc22b9d0>]
```

[10]:
```
print(users_objects[0].first_name)
print(users_objects[1].first_name)
print(users_objects[2].first_name)
print(users_objects[4].first_name)
print(users_objects[6].first_name)
```

```
John
Jack
Matthew
Randall
Marlo
```

Q: If we can hold information the same as in a dictionary why we even need classes?

4.4 Holding Logic

As mentioned at the beginning classes are also capable of holding logic about our object. In this way, we can interact with our data, change it, perform calculations on it and have all this logic tight to the object itself. The object is holding this logic inside methods. The methods are similar to functions, both being callable objects. Every time you see a function inside of a class the function will be called method. Methods have slightly different argument structure that functions. If we can define and use the function without parameters, in the case of methods we will always have the object as the first argument of our method.

[11]:
```
class User(object):

    company = 'Py Future inc.'

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def is_legal_to_drink(self, print_result=False):
        if not print_result:
            return self.age >= 21

        message=f'''
        The user {self.first_name} {self.last_name[0]}.
        {"hax" if self.age >= 21 else "dose not have"}
        the legal age for drinking'''
        print(message)
```

We redefined the class from the example above and complicated a bit the `is_legal_to_drink` method. As you can see we added a piece of logic to check if our user has the legal age for drinking. We also added a keyword argument `print_result` that has a default value.

If we want to call our method without a print message we can use the default value.

[12]:
```
user = User('Marissa', 'Raynaud', 20)
user.is_legal_to_drink()
```

[12]:
```
False
```

As you can notice we called the method `is_legal_to_drink` without arguments.

Q1: How was possible to call the method withowt `print_result` ? Q2: How was possible to call the method withowt `self`?

[13]:
```
user.is_legal_to_drink(print_result=True)
```

```
The user Marissa R.
```

```
dose not have
the legal age for drinking
```

4.5 self

Even if we define the method with self as the first argument, we don't need to explicitly call the method with the argument `self`. Python will do the work for us and will always pass the object as the first argument to our methods. When we refer to self we refer to the instances of the class, so when we call a method we use the logic defined for that specific method on the object from which we called the method. In the case of `is_legal_to_drink`, the logic inside the method will check the value of `self.age` to determine the result. The behaviour of each method is coupled with the data we store in our attributes and can't work without them.

`Self` is a convention keyword that can be changed and the behaviour will be the same since Python will always pass the object as the first positional argument. It is not advisable to use other names for the first argument of class methods since is an industry-standard.

5 Your example here:

Let's build a class together that will be included in this material. It needs to have at least 5 attributes and 2 methods. Remember we can model any real-life example using classes

[14]:
```
class _(object):

    def __init__(self,):
        pass
```

6 Class Attributes

As you sow in the examples above we can have attributes defined outside of `__init__`. The one in the User class was `company`. Because the attributes are not defined in `__init__` it means that all the instances of our class will have these attributes regardless of the initialization phase.

If we don't define the `__init__` method our object will be created and have access to the class attributes if there are any defined in our class

[15]:
```
class User(object):

    company = 'Py Future inc.'
```

[16]:
```
john = User()
print(john.company)
```

```
Py Future inc.
```

If we define the `__init__` method we will have both class attributes and instance attribute available on the object. In the class code, both will be available on `self` and when we use the object, both will be attributes on the object

[17]:
```
class User(object):

    company = 'Py Future inc.'

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def user_description(self):
        return f'The user {self.first_name} {self.last_name[0]}. works for␣
↪{self.company}'
```

[18]:
```
john = User('John', 'Doe', 19)
print(john.user_description())
```

```
The user John D. works for Py Future inc.
```

[19]:
```
print(john.first_name, john.company)
```

```
John Py Future inc.
```

7 Inheritance

As we mentioned above one of the advantages of OOP is extensibility. We can achieve this extensibility by using inheritance. Inheritance is the process by which one class takes on the attributes and methods of another. When we create a new class by inheriting from other classes our class will be called child class and the classes we inherit from will be called parent classes. In Python, we can inherit from multiple classes, which is called multiple inheritance.

In the below example, we will take a look at how the inheritance work by creating more specialised types of users.

[20]:
```
class User(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def user_description(self):
        return f'The user {self.first_name} {self.last_name[0]}. age: {self.
↪age}'
```

```
    def is_legal_to_drink(self):
        return self.age >= 21


class Student(User):

        def __init__(self, first_name, last_name, age, courses):
            self.courses = courses
            super().__init__(first_name, last_name, age)

        def is_enrolled(self, course):
            return course in self.courses

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

[21]:
```
john = Student('John', 'Doe', 19, ['Python 1'])
```

[22]:
```
print(john.courses)

print(john.first_name)

print(john.last_name)
```

```
['Python 1']
John
Doe
```

[23]:
```
print(john.user_description())

print(john.is_enrolled('Python 1'))
```

```
The user John D. age: 19
True
```

To break down what happened:

1. We inherit from User all the attributes and all the methods. In our case `User` has two methods (`__init__` and `is_legal_to_drink`)
2. The methods from `User` are now available in class `Student`
3. We overwrite the initialization method `__init__` in order to add `courses` to our `Student` instances.

---

4. We set `courses` on our instances and call the parent class `__init__` by using `super()` to also add the attributes from `User` to our instances.

5. We have our instance with all the methods defined in `User` and all the methods defined in `Student`

Q: What happens to `is_legal_to_drink` since is defined both in `User` and in `Student`

[24]:
```
print(john.is_legal_to_drink())
```

```
True
```

If the same method is available both in child and parent class the method used will be the one from the child class. The method with the same name from the parent class is still accessible by explicitly calling it using `super()`.

8 Static Methods and Class Methods

So far we worked with methods that are dependent on the instance. We can use these methods only by instantiate. We have two more types of methods that we can use: static and class methods.

Static methods are pieces of logic that can work independently of objects. They will not require any data that the class specifies an instance can hold. You can look at static methods exactly as you look at functions, the only difference is that in some use cases you want to link that logic of the static method with the name of the class. By doing this we make the code more clear.

When defining a static method we will us `@staticmethod` decorator on our method. Since the method does not require any interactions with the object the `self` parameter will not be specified.

[25]:
```
class User(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def is_legal_to_drink(self):
        return self.age >= 21

    @staticmethod
    def print_welcome_message():
        print('Hello and welcome')
```

[26]:
```
mark = User('Mark', 'Evens', 29)

User.print_welcome_message()
```

```
Hello and welcome
```

You noticed that we can call the static method directly from class not needing to instantiate an object.

The instance methods we worked on so far were using the instance to retrieve data and perform calculations. The class methods are behaving in similar ways but instead of working with instance objects will work with the class object.

When we define a class method we use `@classmethod` decorator. Our class methods need to work with the class object similar to how the instance methods use the instance object. In this case, Python will do the work for us and will send the class object as the first argument when calling a class method. Same as with `self` we have a convention for naming the first parameter of class methods `cls`.

The most common use cases of class methods are to offer alternatives for constructing the instances, but are not limited to this use case only.

[27]:
```
class User(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    @classmethod
    def from_dict(cls, dictionary):
        first_name = dictionary['first_name']
        last_name = dictionary['last_name']
        age = dictionary['age']

        return cls(first_name, last_name, age) # this is identical with␣
↪User('Mark', 'Evens', 29)

    def is_legal_to_drink(self):
        return self.age >= 21
```

[28]:
```
john = User('John', 'Doe', 19)
mark = User.from_dict(
    {
        'first_name': 'Mark',
        'last_name': 'Evens',
        'age': 22
    }
)
```

[29]:
```
print(john.first_name)
print(mark.first_name)

print(john.is_legal_to_drink())
print(mark.is_legal_to_drink())
```

```
John
Mark
```

```
False
True
```

Same as in the case of static methods the class methods can be called from the class object not needing an instance. This is convenient exactly for this use case of alternative constructors

9 Dunder or magic methods (str, repr)

In Python, we can make our object behave in certain particular ways. We can implement how our object is printed, how two instances of our class will be considered equal or offer support for operators such as + and *. We will give more details about these methods in the next lecture

[30]:
```
class User(object):

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def is_legal_to_drink(self):
        return self.age >= 21
```

[31]:
```
mark = User('Mark', 'Evens', 29)
mark
```

[31]:
```
<__main__.User at 0x7f9bcc1f1610>
```

[32]:
```
print(mark)
```

```
<__main__.User object at 0x7f9bcc1f1610>
```

[33]:
```
class User(object):

    company = 'Py Future inc.'

    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def __repr__(self):
        return f'User: {self.first_name} {self.last_name} with id: {id(self)}'

    def __str__(self):
        return f'User: {self.first_name} {self.last_name[0]}.'

    def is_legal_to_drink(self):
```

```
        return self.age >= 21
```

[34]:
```
mark = User('Mark', 'Evens', 29)
mark
```

[34]:
```
User: Mark Evens with id: 140307120812176
```

[35]:
```
print(mark)
```

```
User: Mark E.
```

10 Let's practice

Build two classes of your choice that can model a real-life example. The class needs to meet the following requirements:

- at least 5 attributes each
- at least 2 methods each
- one class to inherit from another

As a demonstration create at least 5 instances of one class (preferably the child class) and call all the methods it holds

Ex: You can have one class (`Country`) that has general attributes about countries such as `area`, `neighbours`, `cities` etc and methods related to those attributes. The second class can be a specific country (`Romania`) that has more specific attributes such as `attractions`, `universities` etc.