OOP_ (INTRO)

Bundling code with data

- “Object oriented programming” is all about bundling data and code together.
- It is important to match your data structure choice to your data.
- Choice of data structure to work with data matters. For example, searching certain value in a `list` can be expensive vs searching a value in a `dict`.
- And that choice makes a huge difference to the complexity of your code.
- Native data types in Python are insufficient to handle complexity of complex data.
- But when Python’s built-in data structures are insufficient to solve complexity, we create classes.

What is `class`?

- Object oriented programming model provides you a way to bundle your code and the data it works on together. This bundling is referred as a `class`.
- In object oriented programming, your code is often referred as `methods` and your data is often referred as `attributes`.
- When you create a particular instance of a `class`, it is referred as `instance` or `object` of a class.
- Each `object` is created from the class and share a similar set of characteristics.

Minimal class

```
class Simple:
    pass
```

Class constructor

```
class Athlete:
    def __init__(self):
        # The code to initialize a "Athlete" object
```

- `class` definition has a “special method” called `__init__()`
- `__init__()` method allows you to control how objects are **initialized**
- `__init__()` is called whenever an object of the class is constructed (this is similar to “constructor” concept in other programming languages).

What is `object`?

- An `object` is an instance of a `class`
- Manipulating `objects` and getting the results is the ultimate goal of “Object Oriented Programming”.
- An `Object` is the basic run-time entity of a `class`.
- Every real-world entity is an `object`. Examples of objects from our daily life are - `mobile`, `camera`, `laptop`, `bike`
- For example: A Chair `object` can have behaviour like Movement, Height Adjustment & Attributes like Colour, Make & Model, and Price.

Syntax of class

```
class <name-of-the-class>(<parent-class>):
    <statement_1>
    <statement_1>
    ...
    ...
    <statement_n>
```

💡 Every class is inherited from some class. If no parent class is provided then implicitly default parent class will be `object` class

```
class MyClass(object):
    a = 100
    b = 200

>>> p = MyClass()
>>> dir(p)
['__class__', '__delattr__',...., 'a', 'b']
```

`class` vs `object`

- An `object` is defined by a `class`
- A `class` is a formal description / layout / definition of how an `object` is designed (i.e. which attribute and methods it has)
- These objects are often called as an `instance` as well.
- A `class` should not be confused with an `object`.

Magic methods `__init__`, `__str__`, `__repr__`, `__del__`

```
class A:
    pass

>>> a = A()
>>> a
<__main__.A object at 0x1034a3130>

>>> repr(a)
'<__main__.A object at 0x1034a3130>'

>>> str(a)
'<__main__.A object at 0x1034a3130>'
```

Understanding `self`

```
>>> a = Athlete()

# `Athlete()` invocation creates a "object" of class "Athlete"
# `a` is a variable / identifier
# `a` holds reference to your "object" of class "Athlete"
```

- `self` is a very important argument and without it, the Python interpreter cannot work out which object instance to apply the method invocation to.
- `self` points to the particular object and method invocation using object name only impacts to the same particular object.
- Note that the class code is designed to be shared among all of the object instances.
- **Every method has first argument as `self`**

```
class Batter:
    """class to define functionality of cricket player"""

    def __init__(self, value=0):
        self.score = value

    def get_score(self):
        return self.score

# Object creation
>>> d = Batter(100)
>>> Batter.__init__(
```

`self` is an “implicit” and “first” argument to every method defined under a class.

`class attributes` VS `instance attributes` VS `method attributes`

- `Attributes` : Anything defined under class and that can be accessed via . (DOT) operator

- `class attribute` :
  - The attributes that are defined at `class` level are called as `class attributes`
  - `class attributes` are owned by `class` and hence, can be accessed directly using class name (along with DOT operator)
  - `class attributes` are shared by objects created by the same class

- `Instance attributes`:
  - `Instance attributes` are owned by objects (i.e. the specific instances of a class)
  - For 2 distinct objects, instance attributes can be different.
  - `instance attributes` are typically defined under `__init__` method (aka constructor method)

Let's take a look at example -

```
class Person:
    citizen = "Indian"     # class attribute
    gender = "male"        # class attribute

    def __init__(self, name_, profession_):
        self.name = name_              # instance attribute
        self.profession = profession_  # instance attribute

    def get_details(self):             # method attribute (aka instance method)
        return self.name + " is " + self.profession
```

- In example above, we have specified what is what.
- Essentially anything is accessible using `. (DOT)` operator is called as an `attribute`.
- `class attributes` can be accessed using `<class-name>.<attribute_name>`.
- It's NOT necessary to create an object of a class to access `class attributes`.

- If it's accessible using `<object-name>.<attribute-name>`, it's called as `instance attribute`.

How to add, access, modify and delete attributes of a class?

```
## how to access attributes ?

```python

>>> x.energy
AttributeError: 'Robot' object has no attribute 'energy'

>>> getattr(x, 'energy', 100)

```
```

Let's define a method and associate that method to a `class`.

```
def hi(obj):
        print("Hi, I am " + obj.name)


class Robot:
    say_hi = hi


x = Robot()
x.name = "Marvin"
Robot.say_hi(x)
```

Instead of defining a function outside of a `class` definition and binding it to a `class attribute`, we define a method directly inside (indented) of a class definition.

```
class Robot:
    name = "prashant"

    def hi(self,):
        print("Hi, I am " + self.name)
```

```
r = Robot()
r.name = "prashant"
Robot.hi(r)
```

- A `method` is "just" a function which is defined inside a `class`.
- The first parameter is used a reference to the calling instance.
- This parameter is usually called `self`.
- `self` corresponds to the Robot object x.

We have seen that a method differs from a function only in two aspects:

It belongs to a class, and it is defined within a class
The first parameter in the definition of a method has to be a reference to the instance, which called the method. This parameter is usually called "self".

As a matter of fact, "self" is not a Python keyword. It's just a naming convention!
