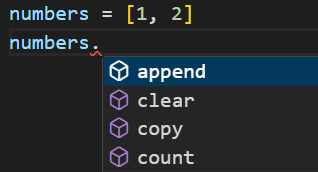


**Classes**:

We learned about list previously,



When we use dot notation, we can automatically access various methods supported by lists to do some operations.

It would be nice if we could create an object like *shopping*\_*cart* and have method like,

shopping\_cart.add()

shopping\_cart.delete()

shopping\_cart.get\_total()

or a point object with methods like,

point.draw()

point.move()

point.get\_distance(1, 2)

This is where classes comes in picture.

***Classes are the blueprint for creating new objects***.

Throughout the course we saw classes in action, for example.

x = 1

print(type(x)) //<class 'int'>

This variable is class of type ‘int’. So we can say that in python we have a class for creating integers.

Similarly, we have classes for creating Booleans, lists, dictionaries and so on.

*So every object in python is created using a class which is a blueprint for creating objects of that type*.

In this section we will learn how to create classes like customer, shopping\_cart, point and so on…

***Objects are the instance of a class***.

For example human is a class which represent all human beings on earth while John, Mary and Jack are objects which are instances of that class.

**Creating Classes**:

Let us create a *Point* class, by using *class* keyword and end this statement with a colon **:** .

class Point:

Note: We use Pascal naming convention to name our classes.

After the colon : we have a code block where we will define all the functions related to *Point* class.

class Point:

    def draw(self):

here we defined a *draw* function, note that all functions defined inside a class has a parameter called ***self***.

class Point:

    def draw(self):

        print("Draw")

Now we have a class or a blueprint for creating Point objects.

Every point object we create will have this *draw* method.

To create a *point* object we have to call this class like a function which in turn returns an object or instance of this *Point* class.

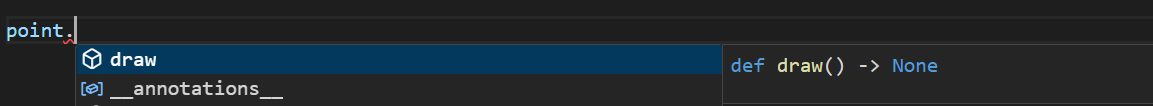
class Point:

    def draw(self):

        print("Draw")

point = Point() //point object created

Now when we use dot operator, we see draw method associated with it.



Note: There are a bunch of other methods we can also see with this point object(*that we did not define*) but our *point* object got these methods from another object in python through a mechanism called ***inheritance***.

class Point:

    def draw(self):

        print("Draw")

point = Point()

print(type(point)) //<class '\_\_main\_\_.Point'>

when we check to see type of this object we see *\_\_main\_\_*. This main is the name of our module.

We have another useful method called *isInstance*. Sometimes we want to check if an object is an instance of a given class.

Let us see if this *point* object is an instance of this *Point* class.

print(isinstance(point, Point)) //True

print(isinstance(point, int)) //False

**Constructors**:

In this example, when we create a point object we also need to provide initial values for x and y coordinates like this,

class Point:

    def draw(self):

        print("Draw")

point = Point(1, 2)

For this purpose we need a ***constructor***, which is a *special method that is called when we create a new object*. Here is how we create a constructor,

class Point:

    def \_\_init\_\_():

In the *Point* class, we define a new function *\_\_init\_\_*,(*which is a magic method and is executed when we create a new object*).

Note: All the methods that we define in a class should have at least one parameter called *self*. It is a reference to the current *point* object.

class Point:

    def \_\_init\_\_(self):

For example, when we call the *Point* class,

point = Point(1,2)

python will internally create a *point* object in memory itself and set *self* to reference that point object.

Optionally we add any number of parameters in *\_\_init\_\_* that we require to create a *point* object. These parameters will be attributes of this *point*, which we see when we use *dot* operator.

class Point:

    def \_\_init\_\_(self, x, y):

Now we can set these x, y parameters to some default value like 0 or set it to *x* argument that we receive from user.

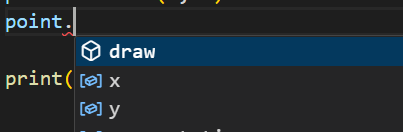
class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Now, when we use dot operator on *point* object we see draw method as well as two new attributes x and y.



We can see value of x on the terminal now,

point = Point(1, 2)

print(point.x) //1

Let us shift our focus on *draw* method,

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def draw(self):

        print("Draw")

This *draw* method also has a ‘*self’* parameter which we can use for referencing the current *point* object and read x, y values and print them on terminal.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def draw(self):

        print(f"Point ({self.x}, {self.y})")

point = Point(1, 2)

point.draw() //Point (1, 2)

In other words,

***“****Class or an object bundles data and functions related to that data into one unit****”***.

**Class vs Instance Attributes**:

In last lecture we defined two attributes x & y for our point objects in the constructor of the Point class.

Now whenever we create a new *point* object, this object will have these attributes by default.

We can also define an attribute *after we create a point object*,

point = Point(1, 2)

point.z = 10 🡪like this

because objects in python are dynamic, similar to JavaScript. So *we do not have to define all the attributes in the constructor*, we can define them later when we need them.

Note: All the attributes we have defined so far (x, y and z) are ***instance attributes***. In other words these are attributes that belong to *Point* instances or *point* objects. So *every point object can have different values for these attributes*.

point = Point(1, 2)

point.draw() //Point (1, 2)

another = Point(3, 4)

another.draw() //Point (3, 4)

We have two *point* objects, completely independent of each other. So these are *instance attributes*.(*just like john and Mary can have different eye colors*)

But we can also define ***Class attributes***. *These attributes are defined at the class level and they are same across all instances of the class*(*each human has two eyes by default*).

class Point:

    default\_color = "red" 🡪A class level attribute

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

This class level attribute *default\_color* can be read via class reference or object reference.

point = Point(1, 2)

print(Point.default\_color) //red (via class reference)

print(point.default\_color) //red (via object reference)

point.draw()

One more thing,

Class level attributes are shared across all instances of a class, if we change their value, the change will be reflected across all instances of that class.

Point.default\_color = "Yellow"

point = Point(1, 2)

print(point.default\_color) //Yellow

another = Point(3, 4)

print(another.default\_color) //Yellow 🡪default\_color changed here as well.

**Instance vs Class Methods**:

In the last lecture we learned about instance vs class attributes. We have the same concept around the methods that we define in the class.

So we have instance methods as well as class methods. In this example,

class Point:

    def \_\_init\_\_(self, x, y): 🡪 Instance method 1

        self.x = x

        self.y = y

    def draw(self): 🡪 Instance method 2

        print(f"Point ({self.x}, {self.y})")

Both our \_\_*init*\_\_ constructor and *draw* methods are ***instance methods****, so we can call them using an instance of the Point class using an object*.

Generally we use these instance methods when we need an object reference. For example when drawing a point, you really need to work with a specific point object. That is why this draw method is defined as an instance method.

But *there are times when you do not need really need an existing object and that is when we use a* ***class method***.

For example in our program there are many cases where we want to create a point with these (0, 0) as (x, y) value.

point = Point(0,0)

point.draw()

This is one way to create a point object…

But we can come up with a different way to create a point object with these values like this,

Point.zero()

Where *zero*() is a method that we define at class level and when we call it, it returns a *point* object initialized with (0, 0) values.

In this example we refer to this *zero* class method as ***Factory method***, because like a factory it creates a new object. Sometimes creating a complex object might require us to pass many values to the class like *Point(0, 0, 1, “a”, “command”)*. Which will be tiresome if we need to create a large amount of such objects.

So instead ***“****we can define a factory method which will create an object with already defined initial values. This way we can move the complexity of creating this object into a factory method****”***.

*How to define method at class level*:

We start by defining a new function called *zero* inside our class.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def zero(cls):

Note: By Python convention when we define a class method, we call its first parameter as *cls*, *this parameter is the reference to the class itself*. So we are not working with *point* object or *Point* instance.

Now, To make this method a class method, we need to decorate it with **@***classmethod*. Using *Decorators is a way to extend the behavior of a method or function*.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    @classmethod 🡪Decorator

    def zero(cls): 🡪class method

Since we have a reference to our class (*cls*) in our class method, with this we can create a point object with initial values.

    @classmethod

    def zero(cls):

       return cls(0, 0) 🡪 exactly like calling Point(0, 0)

We are just returning Point(0, 0)!!

point = Point.zero()

point.draw() // Point (0, 0)

At runtime when we call the *zero* method *python interpreter will automatically pass a reference to the Point class to the zero method and return it*.

**Magic Methods**:

You can identify these magic methods by their unique syntax, two underscores at the beginning and end of their name and they are called automatically by Python interpreter depending on how we use our objects and classes.

For example, Here we have \_\_*init*\_\_ method which python calls automatically when we create a new point object.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def draw(self):

        print(f"Point ({self.x}, {self.y})")

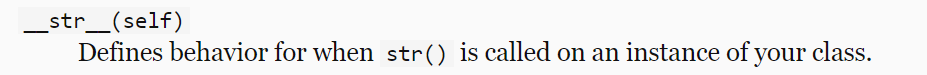
point = Point(1, 2)

point.draw()

We have quite a few magic methods in python.

<https://rszalski.github.io/magicmethods/>

**🡪** \_\_***str***\_\_ method:



This method is used when we try to convert an object to a string.

For example,

point = Point(1, 2)

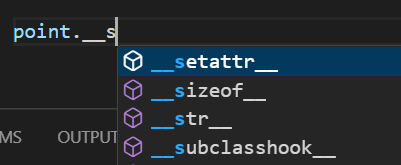
print(point) // <\_\_main\_\_.Point object at 0x000001ADFD71B2D0>

when we simply print this *point* object on the terminal, we get

<\_\_main\_\_.Point object at 0x000001ADFD71B2D0>

We get the name of our module, followed by the class name *Point* and the address of this *point* object. This is the default implementation of the \_\_str\_\_ magic method in our *point* object.

If we use dot operator on *point* object, we can see magic methods that it has inherited,



We can change the default implementation of this method, to get a better result.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_str\_\_(self):

        return f"({self.x}, {self.y})" 🡪 returning x and y as name

point = Point(1, 2)

print(point) // (1, 2)

**Comparing Objects**:

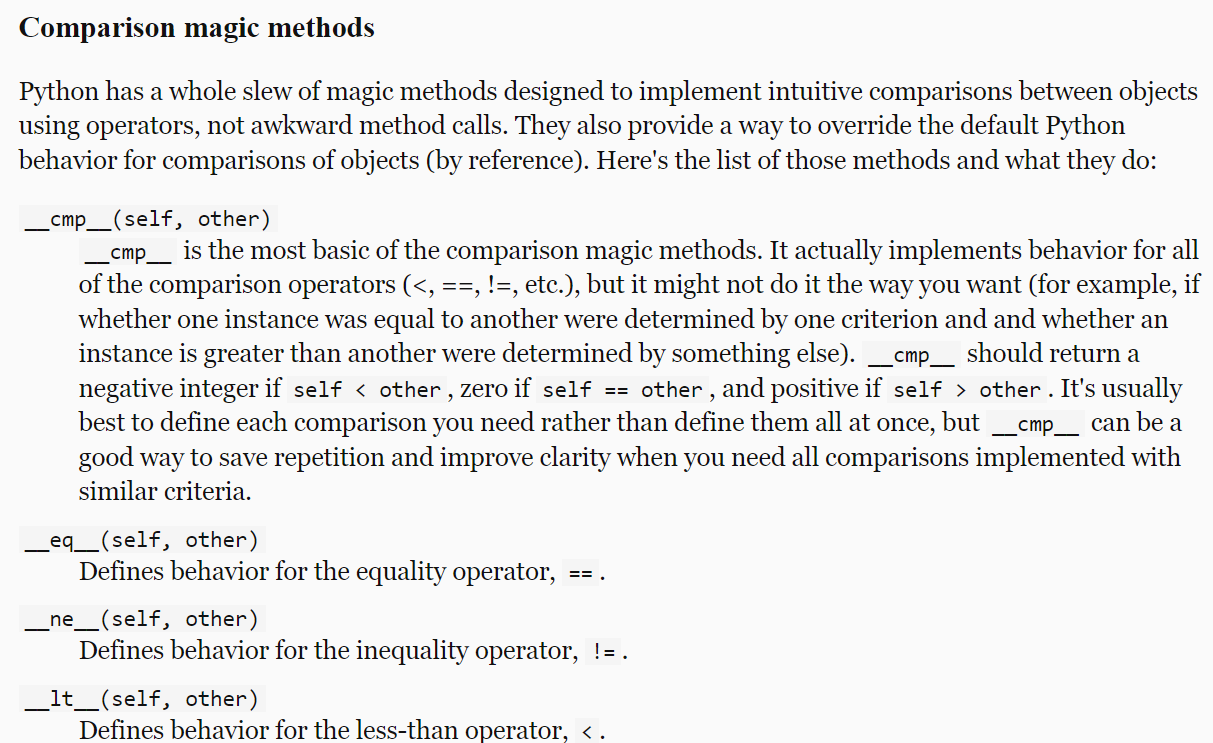
If we compare two objects with exact same attributes,

point = Point(1, 2)

other = Point(1, 2)

print(point == other) //False

We get false because by default this equality operator == compares the reference or addresses of these two objects in memory. To solve this problem, we need a magic method and that *magic method will be called automatically when we compare two objects*.



Refer to comparison magic methods, for example we have \_\_*eq*\_\_ for the equality operator ==, \_\_*gt*\_\_ for greater than > operator and so on…

Let us define \_\_***eq***\_\_ magic method in our class, which takes two parameters *self* and *other* and return a Boolean value.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_eq\_\_(self, other):

        return self.x == other.x and self.y == other.y

point = Point(1, 2)

other = Point(1, 2)

print(point == other) //Now, we get True

Similarly if we check greater than > operator,

point = Point(10, 20)

other = Point(1, 2)

print(point > other)

We get TypeError: '>' not supported between instances of 'Point' and 'Point'.

So we introduce \_\_*gt*\_\_ magic method in our class,

 def \_\_gt\_\_(self, other):

     return self.x > other.x and self.y > other.y

point = Point(10, 20)

other = Point(1, 2)

print(point > other) //True

Note: This code will work automatically for less than operator < as well we do not have to define \_\_*lt*\_\_ separately, Python will figure it out itself.

point = Point(10, 20)

other = Point(1, 2)

print(other < point) //True

**Supporting Arithmetic Operations**:

We also have magic methods to perform arithmetic operations between two objects.

point = Point(10, 20)

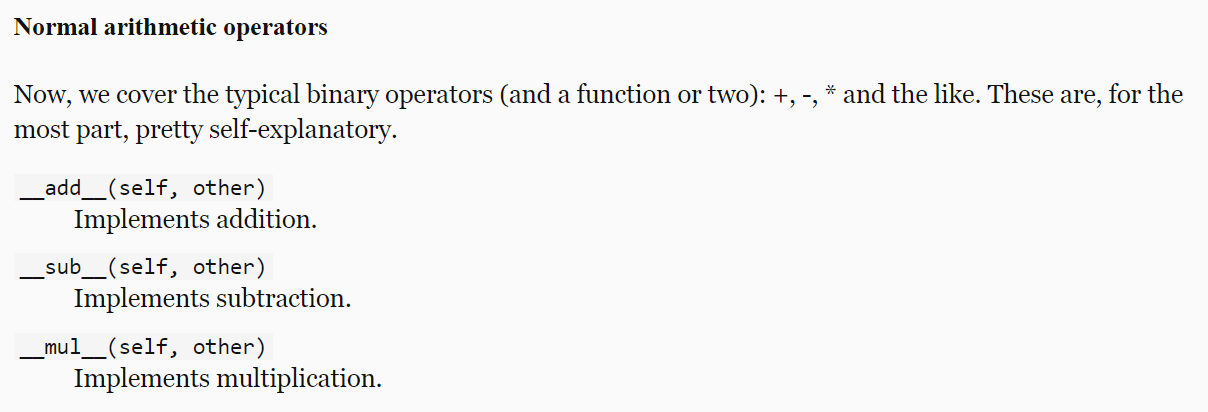
other = Point(1, 2)

print(other + point)

We get the same,

TypeError: unsupported operand type(s) for +: 'Point' and 'Point'.

To solve this refer back to magic method document,



Let us now implement this,

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_add\_\_(self, other):

        return Point(self.x + other.x, self.y + other.y)

point = Point(10, 20)

other = Point(1, 2)

combined = point + other

print(combined.x) //11

**Making custom Containers**:

We learned about common data structures in python like lists, sets, dictionaries and so on. These data structures or *containers* are pretty useful and sufficient for most cases, but there are times when we need to create our own custom container types.

For example, we have this class *TagCloud*,

class TagCloud:

with this class we can keep track of number of various tags on a blog. For example, *how many articles do we have that are tagged with “Python”.* Since this class represents a container, it supports various operations around containers.

Say we have an object *cloud* that we created with this container.

cloud = TagCloud()

We should be able to perform some operations like this,

🡪 find number of items in this container

len(cloud)

🡪 target specific items by their key, like get the number of articles tagged with “python”

cloud["python"]

🡪We should also be able to set it,

cloud["python"] = 10

🡪 iterate over this object,

for tag in cloud:

    print(tag)

--------------------------------------------------------------------------------

Let us implement this, Internally we will use one or more built in data structure like lists, dictionaries and so on.

--------------------------------------------------------------------------------

In this case we will use *dictionary* because it is the easiest way to quickly get the number of any given tag.

class TagCloud:

    def \_\_init\_\_(self):

        self.tags = {}

Here in the \_\_*init*\_\_ constructor we initialize tags attribute to an empty dictionary.

Next we implement an *add* method that takes a tag. Here we should *check to see If we have this tag in dictionary* , if we do not have this, we will set its value to one, otherwise we will increment it by one.

class TagCloud:

    def \_\_init\_\_(self):

        self.tags = {}

    def add(self, tag):

        self.tags[tag] = self.tags.get(tag, 0) + 1

Here we use the *get* method to get an item by its key and supply a default value 0, if we do not have them.

Learn more about **get** method.

Let us test our program up to this point by creating a *cloud* object.

cloud = TagCloud()

cloud.add("Python") //Here we add “Python” 3 times.

cloud.add("Python")

cloud.add("Python")

print(cloud.tags) // {'Python': 3}

But if I use lowercase “python”…

cloud.add("Python")

cloud.add("Python")

cloud.add("python")

print(cloud.tags) //{'Python': 2, python: 1}

To solve this issue whatever tag we receive convert it to lowercase using *lower*() when setting it as well as reading it.

class TagCloud:

    def \_\_init\_\_(self):

        self.tags = {}

    def add(self, tag):

        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1 🡪to lowercase

cloud = TagCloud()

cloud.add("Python")

cloud.add("Python")

cloud.add("python")

print(cloud.tags) //{'python': 3}

Now the problem is solved.

Let us add more complexity in this class, we should be *able to read the count of a tag using square brackets*, **cloud[“**python**”]**.

For this we will use another magic method called \_\_**getitem**\_\_.

This function takes *self* as well as ***key***(*which is tag in our case*).

    def \_\_getitem\_\_(self, tag):

        return self.tags.get(tag.lower(), 0) //zero in case no such tag present

print(cloud["python"]) //3

Now let us take this to the next level, with the current implementation we can only read number of given tag. *To also set it*, we need another magic method called \_\_**setitem**\_\_.

    def \_\_setitem\_\_(self, tag, count):

        self.tags[tag.lower()] = count

This \_\_**setitem**\_\_ takes three parameters *self*, *key* and *value*.

In our case key is *tag* and value is *count*. With this implementation we can set the count for a given tag like this,

cloud["python"] = 10

Now we will implement a magic method for getting the number of items or *len*(cloud) called \_\_**len**\_\_.

    def \_\_len\_\_(self):

        return len(self.tags)

To be *able to iterate over this object*, we will implement another magic method \_\_**iter**\_\_ then all we have to do is to use one of the built in functions *iter* to get an iterator object (*iter function returns an iterator object gives one object at a time in for loop*).

    def \_\_iter\_\_(self):

        return iter(self.tags)

🡪 complete *TagCloud* class looks like this:

class TagCloud:

    def \_\_init\_\_(self):

        self.tags = {}

    def add(self, tag):

        self.tags[tag.lower()] = self.tags.get(tag.lower(), 0) + 1

    def \_\_getitem\_\_(self, tag):

        return self.tags.get(tag.lower(), 0)

    def \_\_setitem\_\_(self, tag, count):

        self.tags[tag.lower()] = count

    def \_\_len\_\_(self):

        return len(self.tags)

    def \_\_iter\_\_(self):

        return iter(self.tags)

**Private members**:

The *TagCloud* class that we built, has a tiny problem.

cloud = TagCloud()

cloud.add("Python")

cloud.add("Python")

cloud.add("Python")

print(cloud["PYTHON"]) //3

so far so good, means our program is working as expected.

However if I try to access the underlying dictionary in this class like this,

cloud = TagCloud()

cloud.add("Python")

cloud.add("Python")

cloud.add("Python")

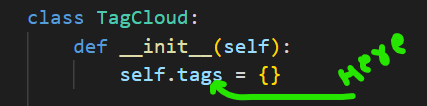
print(cloud.tags["PYTHON"]) // KeyError: 'PYTHON'

We get KeyError: 'PYTHON', because we do not have this key in our dictionary.

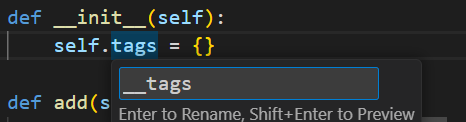
The problem with this class is that it gives us access to the underlying dictionary that is used to keep track of *tag* count.

To fix this we need to hide this attribute from the outside.

Put the cursor here, on the *tags* attribute and press **F2**.(*this is used to renaming objects in VS code*)



Prefix it with double underscores **\_\_** and press **ENTER**.



You will see all occurrences of *self.tags* is converted to *self.\_\_tags*

Now if you use the dot operator on cloud object, we do not see tags or \_\_tags. So if we print this…

print(cloud.\_\_tags)

we get an exception,

AttributeError: 'TagCloud' object has no attribute '\_\_tags'

So *this is how we can make certain attributes or certain methods in a class private by prefixing them with* ***\_\_ double underscores***.

Note: Technically these members are still accessible from the outside only a little harder to access. So the point of this practice is not security, it is more of an alert or warning to someone who is using this class.

We can access these private members by using \_\_dict\_\_ property inherited by every python object. *This is a dictionary that holds all the attributes of the class*.

print(cloud.\_\_dict\_\_) //{'\_TagCloud\_\_tags': {'python': 3}}

Notice we have an attribute in this class,

\_TagCloud\_\_tags, python interpreter just prefixed class name.

print(cloud.\_TagCloud\_\_tags) //{'python': 3}

We can still access it from outside. So in python unlike languages like Java or C#, we do not really have the concept of private members. Using \_\_ double underscores is more of a convention to prevent accidental access of these private members.

**Properties**:

There are times when you want to have control over an attribute in a class.

class Product:

    def \_\_init\_\_(self, price):

        self.price = price

For example here we have a *product* class and in the constructor we set the *price* attribute.

With this implementation, you can create a *product* object and give it a negative price.

product = Product(-50)

and python interpreter will execute this without any problem.

So how can we ensure that our products do not have negative price or more precisely *how can we have more control over our attributes*?

***First way***:

We can make this attribute/ field private using \_\_ (*double underscores*)and then define two methods for getting and setting the value of this attribute.

class Product:

    def \_\_init\_\_(self, price):

        self.set\_price(price) //instead of directly setting price attribute

// we call set\_price to set it.

    def get\_price(self):

        return self.\_\_price

    def set\_price(self, value):

        if value < 0:

            raise ValueError("Price cannot be negative")

        self.\_\_price = value

product = Product(-50) //ValueError: Price cannot be negative

Note: This kind of implementation is what we call “*unpythonic*”. *Pythonic* is one of those expressions, we hear a lot in python community. So when we are not using python’s best practices, it is *unpythonic*.

So if we want to use python’s power to its fullest potential here, we should use a *property*. *A property is an object that sits in front of an attribute and allows us to get or set the value of an attribute*.

Let us see how to define a property here,

After the *get*\_*price* and *set\_price* methods, we will define a class attribute with an ideal name, which is *price* in our case.

price = property()

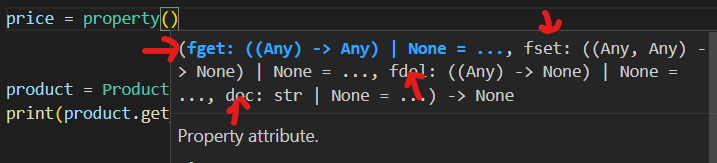
Here we call built in *property* function, which takes four optional parameters.

First parameter **fget** is a function for getting value of an attribute.

Second parameter **fset** is a function for setting the value of an attribute.

Third function **fdel**, is a function for deleting that attribute.

And last one **doc** is for documentation.

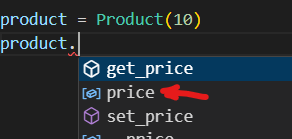


In our case, we need to supply two arguments, *get\_price* and *set\_price*.

 price = property(get\_price, set\_price)

Note: We are not calling these function (), we are simply passing a reference to them.

So when we call built in *property* function, with these arguments, this function will return a property object which will use *get\_price* for getting the value of price attribute and set\_price for setting it.



We can see we have a *price* property attached to our object which we can use a regular field.

product = Product(10)

print(product.price) //10

We can set it as well,

product = Product(10)

product.price = 15

print(product.price) //15

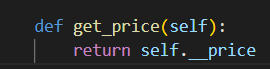
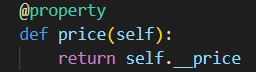
So *a property looks like a regular attribute from the outside, but internally it can have two methods called a* ***getter*** *and a* ***setter***.

*How to hide get\_price and set\_price methods to make the interface of our object simpler*?

These two methods are still accessible from the outside. One way to solve this issue is by making them private using \_\_ double underscores.

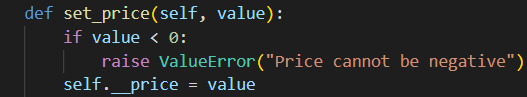
But there is better, shorter and cleaner way to achieve the same,

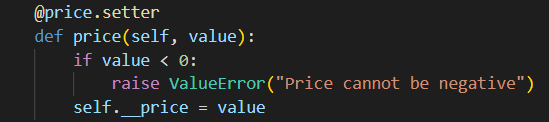
Instead of explicitly calling *property* method to create a property object, we can apply a *property* decorator to these methods and rename them to ideal names.

 🡪 🡪 

When python interpreter will see this code, it will automatically create a property object called *price*.

Similarly we need to apply another decorator to our setter,

🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪🡪



The name of the decorator starts with name of our property then *.setter*.

With these two decorators we can easily create a property in a cleaner way.

Note: We do not always have to define setter for our properties, if we comment out setter method in this example we will get read – only property.

class Product:

    def \_\_init\_\_(self, price):

        self.price = price

    @property

    def price(self):

        return self.\_\_price

    # @price.setter

    # def price(self, value):

    #     if value < 0:

    #         raise ValueError("Price cannot be negative")

    #     self.\_\_price = value

product = Product(10)

product.price = 15

print(product.price)

AttributeError: property 'price' of 'Product' object has no setter

If we still try to set its value from outside,

We get AttributeError: property 'price' of 'Product' object has no setter.

**Inheritance**:

As you build various classes, you may notice that some of these classes might have one or more features or functions in common.

Take a *Mammal* class as an example,

class Mammal:

    def eat(self):

        print("Eat")

    def walk(self):

        print("walk")

All mammals should be able to eat and walk.

We create another class called *Fish*, which is a little different.

class Fish:

    def eat(self):

        print("Eat")

    def swim(self):

        print("swim")

So you see, we have repeated same method *eat* in both the classes.

In real world applications the method can be 10-15 lines of code and it is not a good practice to repeat this in all the classes.

In programming we have a concept called **DRY** which means

***DO NOT REPEAT YOURSELF***

To solve this problem, we have two solutions, we can either use *inheritance* or *composition*.

***“****Inheritance is a mechanism that allows us to define the common behavior or common function in one class and then inherit them in other classes****”***.

class Animal: //Parent or Base class

    def eat(self):

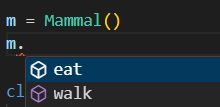
        print("Eat")

class Mammal(Animal): //Child class

    def walk(self):

        print("walk")

Here we defined another class called *Animal* and use the eat method there. Mammal class which is a child class in this case, inherited this method from Animal class or its parent class.

 🡨 Now, *eat* method available for Mammal object.

Inheritance is not limited to methods, we can also inherit attributes of a base class.

class Animal:

    def \_\_init\_\_(self):

        self.age = 1 //age attribute initialized to 1

    def eat(self):

        print("Eat")

class Mammal(Animal):

    def walk(self):

        print("walk")

m = Mammal()

print(m.age) //1

**The Object class**:

In the last lecture, we defined a mammal object. Now we will see a couple of useful built in functions.

🡪 isinstance, this method tells *whether an object is an instance of a given class*.

m = Mammal()

print(isinstance(m, Mammal)) //True

If we pass Animal here,

m = Mammal()

print(isinstance(m, Animal)) //True

Still true, because Mammal class inherits from Animal class.

Let me show you something interesting,

This Animal class inherits from *object* class, even though we have not explicitly defined it like *class Animal (object)*.

So *every python class we have directly or indirectly derives from object class*.

m = Mammal()

print(isinstance(m, object)) //True

🡪 Another useful method in certain situations is issubclass.

*With this we can see if a class derives from another class*.

print(issubclass(Mammal, Animal)) //True

print(issubclass(Mammal, object)) //True

**Method Overriding**:

class Animal:

    def \_\_init\_\_(self):

        self.age = 1

    def eat(self):

        print("Eat")

class Mammal(Animal):

    def walk(self):

        print("walk")

In this example, inside Animal class, we have this constructor where we initialize age attribute to one.

What if we want to initialize a constructor in Mammal class and initialize its weight?

class Animal:

    def \_\_init\_\_(self):

        self.age = 1

    def eat(self):

        print("Eat")

class Mammal(Animal):

    def \_\_init\_\_(self):

        self.weight = 2 //here we initialized Mammal’s constructor

    def walk(self):

        print("walk")

Now if we print age and weight of Mammal object.

m = Mammal()

print(m.age) //AttributeError

print(m.weight)

We get AttributeError: 'Mammal' object has no attribute 'age'

The reason this happened was because the constructor in Animal class was not executed.

In other words ***“****constructor defined in child class replaced constructor defined in its base or parent class****”***. This is what we call ***method overriding***.

So by default we are overriding or replacing a method in the base class, but there is a way to still execute base constructor.

class Mammal(Animal):

    def \_\_init\_\_(self):

        super().\_\_init\_\_() //call base constructor with super()

        self.weight = 2

Here Inside the constructor of Mammal class, we can use the built in ***super*** function, *to get access to super or base class*.

To verify if we are calling base constructor inside constructor of child, let us put print statements in both.

class Animal:

    def \_\_init\_\_(self):

        print("Animal Constructor")

        self.age = 1

    def eat(self):

        print("Eat")

class Mammal(Animal):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        print("Mammal Constructor")

        self.weight = 2

    def walk(self):

        print("walk")

m = Mammal() //Animal Constructor Mammal Constructor

We can also change the order of these method calls, by simply calling *super().\_\_init\_\_()* after initializing Mammal constructor.

class Mammal(Animal):

    def \_\_init\_\_(self):

        print("Mammal Constructor")

        self.weight = 2

super().\_\_init\_\_() //Mammal Constructor Animal Constructor

In summary, *Method overriding means replacing or extending a method defined in the base class*. In this example we extended the \_\_init\_\_ method defined in the Animal class.

**Multi – level Inheritance**:

Too much inheritance between classes can increase complexity and create various issues.

For example, some people try to create inheritance in Employee class like this,

*Employee 🡪 Person 🡪 Living Creature 🡪 A thing…*

This is example of *multi-level inheritance*, just because it is true does not mean we have to create classes like this (*or create an entire model of universe*).

The methods we add in classes are there to solve a business problem, which should be the focus of our software.

***“****If you want to use inheritance limit it to one or at most two level, going beyond that is not recommended***”**

**Multiple Inheritance**:

In python, a class can have multiple base classes.

class Employee:

    def greet(self):

        print("Employee Greet")

class Person:

    def greet(self):

        print("Person Greet")

class Manager(Employee, Person):

    pass

Here we have *Manager* Class which has two base classes, *Employee* and *Person*. This is what we call *multiple inheritance* (*it is a source of issues*). If we do not use it properly, it can introduce all sorts of bugs in our program.

*If we create a manager object and call the greet method, which one will be called*?

class Manager(Employee, Person):

    pass

manager = Manager()

manager.greet() //Employee Greet

The reason greet method from Employee class gets called is because Employee class is added first.

So when python interpreter tries to execute *greet* method, *first it looks inside* ***Manager*** *Class then* ***first base*** *class and finally* ***second base*** *class*. From *wherever it gets this method first, lookup will terminate then and there*.

It is an issue because,

If tomorrow someone change the order of base class we will have completely different output behavior.

It is good to use when,

Base classes have nothing in common and we want to inherit their features in a separate class. For example as below,

class Flyer:

    def fly(self):

        print("fly")

class Swimmer:

    def swim(self):

        print("swim")

class FlyingFish(Flyer, Swimmer): //It can fly and swim, both different activity

    pass

**A good example of inheritance**:

Imagine you want to model the concept of a stream of data. We can read a stream of data from a file, from a network or from the memory.

All these streams have a few things in common. We can open, close and read data from them.

But how we read data from a stream is dependent upon the type of the stream, because reading data from a file is different from reading it from a network.

class Stream:

    def open(self):

We start by defining a base class called *Stream*. In this class we are going to define methods like *open* and *closed*.

In *open* method we might need a flag to see if this stream is open or not. So we define a constructor and in this constructor set the opened flag initially to False and when we call the *open* method it will set its value to True.

class Stream:

    def \_\_init\_\_(self):

        self.opened = False //initially False

    def open(self):

        self.opened = True // open method change it to True

If we try to open a stream that is already open (*which is an invalid operation*), an exception must be raised.

Since this is an invalid operation, we must create a custom python exception called *InvalidOperationError*.

Note: Every custom exception class we create in python must derive from a base class called *Exception*.

class InvalidOperationError(Exception): 🡪 Base class Exception

    pass

class Stream:

    def \_\_init\_\_(self):

        self.opened = False

    def open(self):

        if self.opened:

            raise InvalidOperationError("Stream is already opened") <--

        self.opened = True

Now we will define the close method in a similar way:

class InvalidOperationError(Exception):

    pass

class Stream:

    def \_\_init\_\_(self):

        self.opened = False

    def open(self):

        if self.opened:

            raise InvalidOperationError("Stream is already opened")

        self.opened = True

    def close(self):

        if not self.opened:

            raise InvalidOperationError("Stream is already closed")

        self.opened = False

Now let us go ahead and create a file stream which inherit from *Stream* class. We know by now that, how we read data from a file is different from how we read data from network.

class FileStream(Stream):

    def read(self):

        print("Reading data from a file")

So here we define a method called *read*.

We create another class for reading data from network,

class NetworkStream(Stream):

    def read(self):

        print("Reading data from a network")

This is a good example of inheritance.

**Abstract Base classes**:

We will continue with example from last lecture. There are a couple of issues with this implementation.

1. First issue, is that we can create a stream object and call then open method,

stream = Stream()

stream.open()

This is an issue because this Stream class is an abstract concept.

Because when we apply open method on stream, *are we working with file stream or a network stream*?

So *we should not be able to directly create an instance of the Stream class*. *We should always sub class it and then create an instance of sub class*.

We only used Stream class as a base class to provide some code that we are going to reuse across different kind of streams.

1. Second issue, if you look at the implementation of *FileStream* and *NetworkStream*. You can see both these classes have a *read* method.

class FileStream(Stream):

    def read(self):

        print("Reading data from a file")

class NetworkStream(Stream):

    def read(self):

        print("Reading data from a network")

If tomorrow we decided to create a different kind of stream, we should remember to implement this same *read* method and call it exactly that.

If we call it readLine or readSTR or whatever, it is not going to be consistent with other kinds of streams we have here.

In other words, *currently there is no way to enforce a common interface across different types of sub classes*.

It will nice to have a common contract or common interface across these different types of streams.

This is where we use an *abstract base class*.

***“****An abstract base class is like a half-baked cookie which is not ready to be eaten yet. Its purpose is just to provide some common code to its derivatives****”***.

In order to make this *Stream* class as an abstract base class, we need to import *ABC* class and *abstractmethod* function (*to be used as a decorator*) from *abc* module.

from abc import ABC, abstractmethod

To make our Stream class as abstract base class derive it from *ABC* class.

class Stream(ABC):

Next step is to define a common interface for all streams, Let us say a *read* method.

    @abstractmethod 🡪decorator

    def read(self):

        pass

We decorate this read method with the *abstractmethod* decorator.

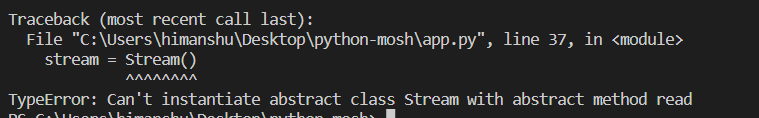
With this implementation both issues are solved.

Solution#1:

stream = Stream()

stream.open()

If I run the program, I get a TypeError: *Can't instantiate abstract class Stream with abstract method read*



It means *when a class has an abstract method*, *it is considered an abstract class* and we cannot instantiate them which means *we cannot create an instance of them*.

Solution#2:

Tomorrow if we are going to create a new kind of stream say *MemoryStream* which we derive from Stream class.

class MemoryStream(Stream):

    pass

stream = MemoryStream() // TypeError

We get the same *TypeError*: *Can't instantiate abstract class MemoryStream with abstract method read*.

In our stream class, we defined an abstract method called *read*.

    @abstractmethod

    def read(self):

        pass

If a class (*MemoryStream*) then derives from *Stream* (*abstract base*) class, *it has to implement this method otherwise that class will also be considered an abstract*.

So in this example, if we want to convert MemoryStream from abstract to concrete class (*in order to instantiate it*), we need to implement *read* method inside it.

class MemoryStream(Stream):

    def read(self):

        print("Reading data from a Memory")

Now this MemoryStream is a concrete class which can be instantiated and it also follows the contract or the interface of the stream class. So all our streams will have the *read* method.

*Complete Code*:

from abc import ABC, abstractmethod

class InvalidOperationError(Exception):

    pass

class Stream(ABC):

    def \_\_init\_\_(self):

        self.opened = False

    def open(self):

        if self.opened:

            raise InvalidOperationError("Stream is already opened")

        self.opened = True

    def close(self):

        if not self.opened:

            raise InvalidOperationError("Stream is already closed")

        self.opened = False

    @abstractmethod

    def read(self):

        pass

class FileStream(Stream):

    def read(self):

        print("Reading data from a file")

class NetworkStream(Stream):

    def read(self):

        print("Reading data from a network")

class MemoryStream(Stream):

    def read(self):

        print("Reading data from a Memory")

stream = MemoryStream()

stream.read()

**Polymorphism**:

As another example here we have define an abstract base class called *UIControl*. This class has an abstract method called *draw* (*which has no implementation*).

So *this class only defines the contract or interface that all these derivatives should follow*.

from abc import ABC, abstractmethod

class UIControl(ABC):

    @abstractmethod

    def draw(self):

        pass

After that we have two classes that derive from *UIControl*, one is *TextBox* and other is *DropDownList*.

Both these classes implement the *draw* method.

class TextBox(UIControl):

    def draw(self):

        print("TextBox")

class DropDownList(UIControl):

    def draw(self):

        print("DropDownList")

Nothing new so far…

But next we have something different, it is a function called *draw* that may take one UI control object (*like DropDownList or TextBox object*) and calls the *draw* method available to this object

def draw(control):

    control.draw()

Now I will create a DropDownList object and pass it to this draw function (*not method*).

ddl = DropDownList()

draw(ddl) //DropDownList

Same goes with TextBox object as well,

tb = TextBox()

draw(tb) //TextBox

*But what is the use for it*?

Suppose instead of just 2, we have 1000 of objects on which we want to call draw method.

Then we can modify our *draw* **function**, so that it can iterate over each object and apply *draw* **method** on it.

def draw(controls):

    for control in controls: //takes an iterable object as argument

        control.draw()

ddl = DropDownList()

tb = TextBox()

draw([ddl, tb]) //DropDownList TextBox

Using this approach we can render the user interface of an application. Imagine we have a form with bunch of TextBox, DropDownList, radio buttons etc.

We could have a list of all these objects and pass that lit to a function like draw and that function will take care of rendering entire form.

Note: The important thing here is draw function does not know what kind of control it is working with. This is determined at run time. It simply iterates over list of controls and calls the draw method of each control object.

This is what we call,

POLY (*many*) MORPHISM (*forms*)

In this example our draw method is taking many different forms and this is determined at run time.

**Duck Typing**:

In last example, in order to achieve polymorphism we started by defining a base class in which we defined a common behavior or common *method* (*draw*) that we need in its derivatives or children.

With this we achieve polymorphic behavior in our *draw* *function*.

So depending on type of control object we are working with at run time, this draw method takes a different form (*draw a TextBox or draw a DropDownList*). This is how polymorphism works in pretty much all languages that support *classes*.

But Python is a dynamically typed language, so we do not need UIControl as the base class to TextBox and DropDownList.

In other words *if we get rid of this base class we can still achieve polymorphism in python*.

class TextBox:

    def draw(self):

        print("TextBox")

class DropDownList:

    def draw(self):

        print("DropDownList")

def draw(controls): 🡪draw function has no relation with UIControl class

    for control in controls:

        control.draw()

ddl = DropDownList()

tb = TextBox()

draw([ddl, tb]) //DropDownList TextBox

We can pass any object to this *draw* function as long as it is iterable, python is happy.

By the same token, that iterable object should have a draw method. So python does not care if these objects derive from UIControl class or not.

As long as these objects has the draw method, python will be happy. This is what we call *duck typing*.

*If it walks like a duck and quacks like a duck, it is a duck*. This is how python works because it is a dynamically typed language and *it does not check the type of objects. It only looks for existence of certain methods in our objects*.

def draw(controls):

    for control in controls:

        control.draw() 🡪 only looks for draw method nothing else

Summary:

To achieve polymorphic behavior we do not necessarily need a base class because python supports duck typing.

But having UIControl as a base class is a good practice because it enforces a common interface or a common contract across all derivatives.

**Extending Built in Types**:

In python we can also use inheritance with the built in types.

For example, we can create a class called *Text* and have it inherit from built in *str* class.

class Text(str):

With this our built in *Text* class will *inherit all the features of python strings but we can also add additional features to it*. For example ability to summarize or duplicate and so on…

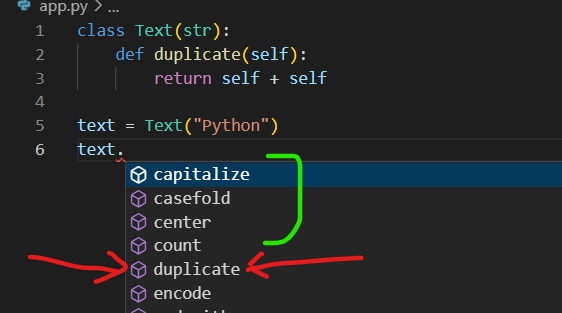
class Text(str):

    def duplicate(self):

        return self + self

Since self represents the current object (*in this case it is an instance of string class; in short… an object*). So here we are concatenating a string with itself.

Notice text object has all the methods available to a string as well as well as our *duplicate* method.



text = Text("Python")

print(text.lower()) //python

print(text.duplicate()) //PythonPython

As another example, we can also extend python lists.

class TrackableList(list):

    def append(self, object):

        print("Append called")

        super().append(object) //append method of base class *list*.

This TrackableList class inherits from built in *list* class. Here we are overriding the append method.

So that every time we want to append an item to this list, a message should be printed on terminal.

tl = TrackableList([1, 2, 3])

tl.append(4) // Append called

**Data Classes**:  
So we have learned that we can use classes to bundle data and functionality into one unit.

Now as we work on larger programs we might come across classes that do not have any behavior means without any methods and only data.

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

Example of such class is this *Point* class. With this we can create two point objects.

p1 = Point(1, 2)

p2 = Point(1, 2)

When we compare these two objects and print the results,

print(p1 == p2) //false

We get false because these two objects are stored in different places in memory. So by default python compares objects based on where they are stored in memory.

To get the memory address of where the object is stored we can use ***id*** method.

print(id(p1), id(p2)) // 2745245069328 2745245069584 different memory

So in order to solve the issue of comparing objects based on their attributes, we need to implement magic method called \_\_*eq*\_\_

    def \_\_eq\_\_(self, other):

        return self.x == other.x and self.y == other.y

Now we see *true*, after comparing objects

p1 = Point(1, 2)

p2 = Point(1, 2)

print(p1 == p2) //True

The issue is solved however writing all this code for data classes is a little bit tedious. *If you are dealing with classes that have no behavior, no methods, only data*, we can use a *namedtuple* instance.

from collections import namedtuple

We import this namedtuple function from collections module and as the first argument we specify the name for this new type we want to create.

namedtuple("Point")

As second argument we pass an array of field names or attribute names.

Point = namedtuple("Point", ["x", "y"])

We want our Point object to have two attributes x and y. Notice that Point is in Pascal naming convention (*as it represents the type of dataset*).

from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])

p1 = Point(x=1, y=2)

p2 = Point(x=1, y=2)

print(p1 == p2) //True…no \_\_eq\_\_ method required here.

The first improvement is that *our code is clearer and less ambiguous due to keyword arguments*.

Second benefit is that *we do not have to implement a magic method to compare two objects*.

Note: These namedtuple are better than regular tuples because here we have attributes in this point object just like the attributes we have in our classes.

print(p1.x) //1

These namedtuple are immutable which means once we create them we cannot modify them.

p1.x = 4 //AttributeError

We get AttributeError: can't set attribute because we cannot set the attribute of a namedtuple after we initialize it.