Ever so often, either while reading a question/answer on stack overflow or while going through documentations, you will come across the use of an underscore. These underscores come in all shapes and sizes, each of which have its own specific use case. All in all, there are about five different variations of these underscores that you can come across, and they are as follows:

1. Single leading underscore – Example “\_VariableName”
2. Single trailing underscore – Example “VariableName\_”
3. Double leading underscore – Example “\_\_VariableName”
4. Double leading and trailing underscore – Example “\_\_init\_\_”
5. Single underscore – Example “\_”

How this has benefited me?

So, let us go through each one at a time.

Single leading underscore – Example “\_VariableName”

Before we jump into this, let’s first take a look at a couple of examples of this specific use of underscores in python. Below, I have three different areas in which you might see the single leading underscore. These include a) before the name of a function b) before the name of a variable or class attribute and c) before the name of the class method.

def \_myfunction(myargument):

    return myargument

\_myagevariable = 26

class myclass():

    def \_myclassmethod(self):

        return self.age

The single leading underscore above is simply a convention that Python developers and certain frameworks maintain related to function, method and variable and attribute names. The convention is to use the single leading underscore for function, method and variable and attribute that are used for internal purpose only and are hence not client facing. This helps python developers uniquely identify such function, method and variable and attribute. It is important to remember that this is simply a convention used by Python developers and thereby is not enforced by the interpreter. This is to say that even if you end up using the single leading underscore for a client facing function, method and variable and attribute, the python interpreter will NOT mad and hence will NOT throw an error back at you.

There is one nuance that a developer needs to be aware of when it comes to the use of the single leading underscore, and it is related to importing function, method and variable and attribute from other modules. Lets imagine a situation in which you are trying to import functions from another python module (basically any other .py file) into your current python file.

The file that you are currently in is called main.py and you are importing from functions.py:

Functions.py

def \_internal():

    return "internal"

def external():

    return "external"

main.py

from functions import \*

>>> print(external())

"external"

>>> print(\_internal())

NameError: "name '\_internal' is not defined"

The reason why there is a NameError in this situation, even though we have imported everything from functions.py using the (from functions import \* wildcard statement), is because the internal function is defined with a single leading underscore. So do we get around this? Based on the python official documentation on this, we will need to define an \_\_all\_\_ list that overrides this behavior. In other words, if we add the following in our functions.py file, we will be able to import and use the “\_internal()” function:

\_\_all\_\_ = ['\_internal','external']

Single trailing underscore – Example “VariableName\_”

Single trailing underscores are simply used to break the name conflict with python keywords. For example, imagine a situation in which you would like to use the name “class” for a variable.

>>> class = 'myvariable'

If you attempt to run the above code in your python interpreter, you will be experience the following error:

SyntaxError: invalid syntax

This syntax error makes sense since class is a python keyword to define a class (object oriented programming paradigm), and therefore, we can not declare a variable called class and assign it a value.

However, if you still wanted to name your variable class, you can then use the single trailing underscore after the variable name and then be able to use it in python program without encountering this syntax error. See the following example:

>>> class\_ = 'myvariable'

>>> print(class\_)

myvariable

Double leading underscore – Example “\_\_VariableName”

The use of double leading underscore is something that can be used for both class methods as well as class attributes. Let us look at each one of these in an example:

Class attributes:

Double underscore (i.e. dunders) are used before the name of a class attribute to help prevent name conflict with attribute names of the subclasses. This is not something that is just a python convention that is used by Python developers, but rather this is something that is enforced by the python interpreter. This process is called name mangling. This concept is best understood with examples, so let’s get into one now:

class Bhains:

            def \_\_init\_\_(self):

                self.leg = 4

                self.\_taang = 2

                self.\_\_seengh = 2

Let us instantiate an object of the Bhains class, and then call the dir function in python to get more information about the attributes and methods available for this particular object.

>>> MyBhains = Bhains()

>>> dir(MyBhains) ['\_Bhains\_\_seengh', '\_\_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\_\_', '\_\_weakref\_\_', '\_taang', 'leg']

Now let’s look at this outputted list a little closely. We can clearly see both the “leg” and the “\_taang” attribute of this Bhains object. However, there appears to be no “\_\_seengh” attribute in this list. Upon further inspection, we can see that the first item in this list does indeed have the “\_\_seengh” part of the attribute, however, it also has “\_Bhains” (i.e. an underscore followed by the name of the class of the object upon which the dir function was just called). Essentially, this “\_Bhains\_\_seengh” attribute represent the “\_\_seengh” attribute of the Bhains class, and hence the MyBhains object.

As I had mentioned above, the use case of name mangling becomes apparent when extending classes. Therefore, lets suppose that we have now decided to extend the Bhains class as per the following:

class bachiya(Bhains):

                def \_\_init\_\_(self):

                    super().\_\_init\_\_()

                    self.leg = 'override'

                    self.\_teeth = 'check karke batata hun'

                    self.\_\_seengh = 0

The super() function above is simply performing the operations as defined in the parent class’s \_\_init\_\_ method. Subsequently, we are simply instantiating the values for various attributes of the bachiya class.

>>> Kaali\_gae = bachiya()

>>> Kaali\_gae.leg

'override'

>>> Kaali\_gae.\_teeth

'check karke batata hun'

>>> Kaali\_gae.\_\_seengh

Traceback (most recent call last):

File "<pyshell#15>", line 1, in <module>

Kaali\_gae.\_\_seengh

AttributeError: 'bachiya' object has no attribute '\_\_seengh'

Now, this is interesting. We can clearly see that the bachiya class has a “\_\_seengh” attribute and its value is equal to the integer 0. However, the python interpreter thinks otherwise. It is usually unwise to start doubting the interpreter, so we are going to take a step back and think through our code above and where we have made this mistake.

Now think through, the outputted list when we had called the dir function on the MyBhains object which was an object of the parent class of the bachiya class. Since we used a dunder when defining the seengh attribute, we found that the name of this attribute (as adjusted by the Python interpreter) had been changed to “\_Bhains\_\_seengh”. The format this followed was \_<class\_name>\_\_<attribute\_name>.

So, now with this new found reflection, let’s revisit the

Class methods:

Double leading and trailing underscore – Example “\_\_init\_\_”

This one is the most common use case of double underscores in Python (also referred to as dunders), which is that you have a double leading and trailing underscore.

If you are familiar with object oriented programming in Python, you would have surely come across both of the following:

class myclass():

    def \_\_init\_\_(self, age):

        self.age = age

    def \_\_call\_\_():

        return self.age

The first one is the \_\_init\_\_ method which is basically the method used to initialize a class object, this is also called the object constructors. The other is the use of the \_\_call\_\_ method which makes the object callable.

Single underscore – Example “\_”

Nothing too fancy. Basically this "\_" single underscore is used to indicate that this is a temporary variable.

The following are the two used cases of this:

1. Lets say you have a tuple with 4 values in it:

mytuple = ('Toyota', 'Red', '2000', '250km/h')

For us to access the specific values of this mytuple object, I would need to assign each value in it to a variable. However, I am only interested in its Brand name and the max speed. So, this is what I do:

Brand, \_, \_, Max\_speed = mytuple

Now, the "Brand" variable has a value of "Toyota" and the "Max\_speed" variable has a value of "250km/h"

b. Additionally, this can be used for Python IDEs to represent the last used variable.

Example:

>>> 20 + 3

23

>>> \_

23

>>> print(\_)

23