Double underscores in python:

These cause the python interpreter to rewrite the attribute name to avoid naming conflicts in subclasses.

This is aka name mangling

For example:

Class test:

Def \_\_init\_\_(self):

Self.foo = 11

Self.\_bar = 23

Self.\_\_baz = 23

In this case if we do dir(test), this will print out that test does not have an attribute \_\_baz but it has an attribute \_test\_\_baz instead

Now if we do

Class extendTest (test):

Def \_\_init\_\_(self):

Super().\_\_init\_\_()

Self.foo = ‘overridden’

Self.\_bar = ‘overridden’

Self.\_\_baz = ‘overridden’

Then I do dir (extendTest)

Now we get that foo and \_bar both have been modified but \_\_baz does not exist, instead, there is a attribute \_extendTest\_\_baz

Name mangling also applies to method name.

Leading and trailing double underscore:

Those are usually reserved as naming convention for methods defined by the python language

| **Pattern** | **Example** | **Meaning** |
| --- | --- | --- |
| **Single Leading Underscore** | \_var | Naming convention indicating a name is meant for internal use. Generally not enforced by the Python interpreter (except in wildcard imports) and meant as a hint to the programmer only. |
| **Single Trailing Underscore** | var\_ | Used by convention to avoid naming conflicts with Python keywords. |
| **Double Leading Underscore** | \_\_var | Triggers name mangling when used in a class context. Enforced by the Python interpreter. |
| **Double Leading and Trailing Underscore** | \_\_var\_\_ | Indicates special methods defined by the Python language. Avoid this naming scheme for your own attributes. |
| **Single Underscore** | \_ | Sometimes used as a name for temporary or insignificant variables (“don’t care”). Also: The result of the last expression in a Python REPL. |

What is \_\_name\_\_?

If the source file is executed as the main program, the interpreter sets the \_\_name\_\_ variable to have a value “\_\_main\_\_”. If this file is being imported from another module, \_\_name will be set to the module’s name.

So it is a built-in variable that evaluates to the name of the current module. And can be used to check whether the current script is being run on its own or is being imported somewhere else by combining it with if statements:

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    print ("File1 is being run directly")

**else**:

    print ("File1 is being imported")

what is \_\_main\_\_ ?

it is the name of the enbironment where top-level code is run. “top-level code” is the first user-specified Python module that starts running.