**Modules and Packages**

In this section we briefly:

* code out a basic module and show how to import it into a Python script
* run a Python script from a Jupyter cell
* show how command line arguments can be passed into a script

The best online resource is the official docs: <https://docs.python.org/3/tutorial/modules.html#packages>

But I really like the info here: <https://python4astronomers.github.io/installation/packages.html>

**Writing modules**

**%%**writefile file1.py

​

**if** \_\_name\_\_ **==** '\_\_main\_\_': *#if we wanted some stateemnts to be only to main module then put them under cond*

print("This is the main part")

**def** myfunc1(x): *#main module*

**return** [num **for** num **in** range(x) **if** num**%**2**==**0]

**else**:

print("This is in the else part")

**def** myfunc(x): *#main module*

**return** [num **for** num **in** range(x) **if** num**%**2**==**0]

print(\_\_name\_\_)

*# list1 = myfunc(11)*

Overwriting file1.py

**file1.py** is going to be used as a module.

Note that it doesn't print or return anything, it just defines a function called *myfunc* and a variable called *list1*.

**Writing scripts**

**!** python file1.py

​

This is the main part

\_\_main\_\_

**%%**writefile file2.py

*# from file1 import list1,myfunc*

**import** file1 *# name of this module file1*

​

*#import file1 as f1 #aliasing*

​

*# list2 =file1.myfunc1(20) #will get error as fun1 is under main and will not get executed*

*# print(list2)*

list1 **=** file1.myfunc(10) *# either can be called using the module or can be called in the main module itself as shown in preious cell*

list1.append(12)

print(list1) *#main module*

Overwriting file2.py

**!** python file2.py

This is in the else part

file1

[0, 2, 4, 6, 8, 12]

**file2.py** is a Python script.

First, we import our **file1** module (note the lack of a .py extension)  
Next, we access the *list1* variable inside **file1**, and perform a list method on it.  
.append(12) proves we're working with a Python list object, and not just a string.  
Finally, we tell our script to print the modified list.

**Running scripts**

**!** python file2.py

[0, 2, 4, 6, 8, 10, 12]

Here we run our script from the command line. The exclamation point is a Jupyter trick that lets you run command line statements from inside a jupyter cell.

**import** file1

print(file1.list1)

[0, 2, 4, 6, 8, 10]

**%%**writefile one.py

*# file one.py*

​

​

​

​

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

print("one.py is being run directly")

print("top-level in one.py")

print(\_\_name\_\_)

​

**else**:

print("one.py is being imported into another module")

print(\_\_name\_\_)

**def** func():

print("func() in one.py")

Overwriting one.py

**!** python one.py

one.py is being run directly

top-level in one.py

\_\_main\_\_

**%%**writefile two.py

*# file two.py*

**import** one

​

print("top-level in two.py")

one.func()

​

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

print("two.py is being run directly")

**else**:

print("two.py is being imported into another module")

Overwriting two.py

The above cell proves that we never altered **file1.py**, we just appended a number to the list *after* it was brought into **file2**.

**!** python two.py

one.py is being imported into another module

one

top-level in two.py

func() in one.py

two.py is being run directly

**Passing command line arguments**

Python's sys module gives you access to command line arguments when calling scripts.

**%%**writefile file3.py

**import** sys

**import** file1

num **=** int(sys.argv[1])

print(file1.myfunc(num))

Overwriting file3.py

Note that we selected the second item in the list of arguments with sys.argv[1].  
This is because the list created with sys.argv always starts with the name of the file being used.

**!** python file3.py 30 20

[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

Here we're passing 21 to be the upper range value used by the *myfunc* function in **list1.py**

**Understanding modules**

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command.

To import a module, we use the import command. Check out the full list of built-in modules in the Python standard library [here](https://docs.python.org/3/py-modindex.html).

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import the math module, we simply import the name of the module:

*# import the library*

**import** math

**from** math **import** ceil

​

*# use it (ceiling rounding)*

math.ceil(2.4)

Out[8]:

3

**Exploring built-in modules**

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

We can look for which functions are implemented in each module by using the dir function:

print(dir(math))

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

help(math.ceil)

Help on built-in function ceil in module math:

ceil(...)

ceil(x)

Return the ceiling of x as an Integral.

This is the smallest integer >= x.

**Writing modules**

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

**Writing packages**

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

Each package in Python is a directory which MUST contain a special file called **\_*init\_*.py**. This file can be empty, and it indicates that the directory it contains is a Python package, so it can be imported the same way a module can be imported.

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the **\_*init\_*.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

*# Just an example, this won't work*

**import** foo.bar

*# OR could do it this way*

**from** foo **import** bar

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The **\_*init\_*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the **\_*all\_*** variable, like so:

\_\_init\_\_.py:

​

\_\_all\_\_ **=** ["bar"]

*#To import the modules from other different folder other than the one which are added in the sys path*

*#So we need to add the path where the modules to be searched to be added so that during import it can search for presence.*

**import** sys

sys.path.append('C:/Users/asha.t')

*#print(sys.path)*

**import** Classvariable

​

print(Classvariable.emp\_1.\_\_dict\_\_)