Astronomical data analysis using Python

Lecture 5

The Python Module Ecosystem

There are three types of modules you will encounter in Python.

- Built-in Modules (come with all standard installations of Python)
- Third Party Modules (need to be installed separately)
- Your Own Modules (we will see how to make them soon)

Built-in Modules - the Python Standard Library

- sys - contains tools for system arguments, OS information etc.
- os - for handling files, directories, executing external programs
- re - for parsing regular expressions
- datetime - for date and time conversions etc.
- csv - for reading and writing CSV tables

and more than a hundred others that allow you to do many different things like text processing, networking and interprocess communication, internet data handling and much more. There are no built-in modules for advanced mathematics or big data handling.

https://docs.python.org/3/library/ (https://docs.python.org/3/library/)

Third Party Modules

These need to be installed separately. There are probably hundreds of thousands of modules in every imaginable area of computing. We are only going to learn about a handful of them.

- **numpy / scipy** - numerical plus scientific computing extensions to Python
- **matplotlib** - using Python for plots
- mayavi - for animations in 3D
- pandas - for tabular data analysis
- **astropy** - Python for Astronomers
- **astroquery** - access online astronomy data repositories from Python
- scikit-learn - machine learning and classification tools for Python

Third party modules will need to be separately installed via a program called `pip` See installation instructions at: https://docs.python.org/3/installing/index.html (https://docs.python.org/3/installing/index.html)

For a list of publicly available Python modules see: https://pypi.org/ (https://pypi.org/) which has more than 340k modules available as of Nov. 2021, incl. 651 astronomy packages.

Making your Own Modules

Very simple. Open a file, say, "MyModule.py"

Write code in the file.

If the file is in the present folder or in folders in the PYTHONPATH environment variable, the following will work.

```
import MyModule
MyModule.somemethod ...
```

- NOTE 1: **File name must have extension .py**
- NOTE 2: **When importing extension must be dropped.**
- NOTE 3: **Do not create modules with the same name as modules in the Python Standard Library**

Example Module - Example.py

```
"""
This is a custom module.
Containing some functions for the purpose of demonstration.
"""
def fun1():
    print "Inside fun1"

def fun2():
    print "Inside fun2"

pi = 3.14
e = 2.7

print ("I am a Custom Module")
```

The above code is stored in a file on my computer called Example.py. Let's see how to use it.

```
In [1]: import Example
```

```
I am a Custom Module
```

Notice the message printed by Example.py. This is to illustrate that any output generated by Example.py will appear on the screen.

```
In [40]: print (Example.pi)
```

```
3.14
```

```
In [2]: Example.fun1()
```

```
Inside fun1
```

```
In [3]: help(Example)
```

```
Help on module Example:

NAME
    Example

DESCRIPTION
    This is a custom module.
    Containing some functions for the purpose of demonstration.

FUNCTIONS
    fun1()

    fun2()

DATA
    e = 2.7
    pi = 3.14

FILE
    /home/yogesh/Dropbox/python_2021/Example.py
```

Notice the description. It is what you enclosed in the "docstring" at the beginning of the module.

assert Statement

```
In [5]: L= [0.1,3,5,7]
        A= 5.0
        # assert condition, description
        assert len(L) > 10, "not enough data points"
        assert type(A) is type(""), "requires a string"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-5-ca84793eb6e2> in <module>
      2 A= 5.0
      3 # assert condition, description
----> 4 assert len(L) > 10, "not enough data points"
      5 assert type(A) is type(""), "requires a string"

AssertionError: not enough data points
```

Use assertions liberally, they make debugging complicated code much easier. Particularly useful at interfaces e.g. just before a function is called. Very important component of defensive programming

Exception handling with try-except

```
In [13]: y=5
         x=0
         try:
             ratio = y/x
         except ZeroDivisionError:
             print ('Divisor = 0')
```

```
Divisor = 0
```

The structure of try-except

```
try:

    [do some processing]

except SomeError:

    [respond to this particular error condition]

    raise SomeError # now let something else handle the error
```

The try/except syntax has the advantage that what you want to do appears first, you don’t have to read past a lot of error trapping code to find out what a particular block of code is doing.

finally - block is executed even if an exception happens

```
f = open('thisfile.txt','r')
try:
    [do something with the file]
finally:
    f.close()
```

If you know what exception to expect

```
In [1]: try:
            f = open('is_it_there.txt')
        except FileNotFoundError:
            # Fallback code
            print("Your specified file is not found.")
```

```
Your specified file is not found.
```

The for-else contruct

```
In [8]: for i in [1, 2, 3, 4, 5]:
            if i == 3:
                break
        else:
            print("this block is only executed when no item of the list is equal to 3")
```

the `with` statement

```
In [4]: # using with statement
        with open('output_file', 'w') as file:
            file.write('hello world !')
            print('No need to explicitly close the file. The with has taken care of it'
        )
```

```
No need to explicitly close the file. The with has taken care of it
```

Variable Scoping

In [17]:
```
import math

def area(r):
    """Area of circle with radius r"""
    return math.pi * r**2 # Name math is known!

def volume(r, h):
    """Vol. of cylinder with radius r, height h"""
    return area(r) * h # Name area is known!

volume(1., 2.) # Everything should be known at call time
```

Out[17]:
```
6.283185307179586
```

Local variables inside functions

In [20]:

```
def f1():
    print (x) # Use variable x
def f2():
    x = "local x" # Assign variable x
    print (x)
x = "global x" # Same name x as in functions
f1() # "global x"
print (x) # "global x"
f2() # "local x"
print (x) # still "global x"
```

```
global x
global x
local x
global x
```

Function scope

Functions provide a nested namespace (scope). Name references search four scopes:

- L: the function’s local scope
- E: the scope of enclosing functions
- G: the (module’s) global scope
- B: the built-in scope (e.g. print() function)

Name assignments create local names unless you use the `global` statement

Because of these scoping rules, if you create a function in your main program that has the same name as a built-in function then your function (e.g. `type()`) will override the built-in function leading to unexpected issues. So please don't create functions with the same name as built-in functions.

`global` statement

In [23]:
```python
def f3():
    global x
    x = "local x"
    print (x)

x = "global x"
print (x) # "global x"
f3() # "local x"
print (x) # now "local x"
```

```
global x
local x
local x
```

Using global variables is almost always a bad idea. It makes debugging harder. Avoid them.

Passing rules

- Immutable arguments act as if passed by value
- When changing mutable arguments in place inside the function, the object is changed outside the function too!

Reminder:

- Numbers, strings, tuples are immutable
- Lists, dictionaries, numpy.arrays are mutable

`list` - a mutable function argument

```
In [27]: mylist= [1,2,3]
         def extendlist(var):
             var.extend([4,5])

         extendlist(mylist)
         print (len(mylist))
         print(mylist)
```

```
5
[1, 2, 3, 4, 5]
```

`float` an Immutable function argument

In [35]:

```
a=5.0
def floattostr(var):
    var = str(var)
    return var

floattostr(a)
print (type(a))
print (a)
print()
b = floattostr(a)
print (type(b))
print (b)
```

```
<class 'float'>
5.0

<class 'str'>
5.0
```

`del` keyword in Python

The `del` keyword in Python is primarily used to delete objects in Python. Since almost everything in python represents an object in one way or another, the del keyword can also be used to delete a list, slice a list, delete a dictionary, remove key-value pairs from a dictionary, delete variables, etc. e.g. `del a,redshift['M31'],mylist[2], otherlist[3:]`

The `del` frees up memory. Very useful if you have large arrays that are not going to be used in your analysis going forward. May need to run garbage collection explicitly.

The `is` keyword

```
In [7]: a=5
        b=5.0
        print (a is b)
        print (a==b)
```

```
False
True
```

How to choose variable names

Names of variables can contain upper and lower case English letters, underscores, and the digits from 0 to 9, but the name cannot start with a digit. Nor can a variable name be a reserved word in Python.

Choose descriptive variables names, i.e., names that explain the variable's role in the program. Well-chosen variable names are essential for making a program easy to read, easy to debug, and easy to extend. Well-chosen variable names also reduce the need for comments.

Unicode variable names are allowed in Python 3

```
In [54]: number = 5
         எண் = 7
         मेरी_सन्ख्या = 9
         यादी = [4,5,6,'नमस्कार']

         print (number + எண் + मेरी_सन्ख्या)

         for i in यादी:
             print (i)
```

```
21
4
5
6
नमस्कार
```

Reserved words in Python

These reserved words cannot be used as variable names: and, as, assert, break, class, continue, def, del, elif, else, except, False, finally, for, from, global, if, import, in, is, lambda, None, nonlocal, not, or, pass, raise, return, True, try, with, while, and yield . Besides these some special characters are used in Python programs **- : # {} () [ ]**

With this we have covered essentially all of Core Python. If you have reviewed the lectures and have practiced the notebooks, you can claim that you are now set to program in Python. The Python standard library and the vast world of third-party modules now awaits you. Congratulations!

Assignment 1

Assignment 1 is nearly ready and will be placed on the Moodle platform and on the website by tomorrow. Please solve the assignment. Preetish will conduct a tutorial session, most likely on 2 Dec, 2021 to discuss the assigment problems and their solution.

Remember programming is about the doing, not about the knowing.

Plans for the second half of the course

We are now halfway through the lectures. In the remaining 5 lectures, we will cover

- numpy + scipy
- matplotlib
- astropy
- astroquery

The second assignment will be more difficult than the first and will have real life (although very simplified) code interactions with real data.