Episode 6

Modules

Modules are very important in Python programming. In fact, every script you write is a module, whether you use it in such a manner or not. Modules enable manageable chunks of related code to be encapsulated. They make it easier and more effective for multiple programmers to work on a project. And finally, they encode certain principles of good software design, such as code reuse.

We will not discuss classes in this introductory tutorial series, but classes in Python are normally implemented within their own module, or at most a few closely-related classes are in a module. So it is important to understand modules for good Python programming practices. Furthermore, packages are bundles of modules, and there are hundreds, if not thousands, of packages available for you to use for all manner of programming tasks. Understanding how modules work is a requirement to be able to utilize these packages.

In order to access the functions in a module, we must `import` the module. Doing so defines a *namespace*. A namespace is an identifier for all the elements defined in the module. The default namespace of a module is the name of its file, without the `.py` suffix. For packages, the namespace will be controlled by the master module. So for example

```
import math
x = 257.3
z=math.sqrt(x)  #sqrt is a function within the math namespace
print z
import os
home_dir=os.getenv("HOME")  #or "HOMEPATH" for Windows
print home_dir
import numpy
A=numpy.zeros(200)
A[75:100] = 1.0
print A[88]
```

In this example, we import the `math` module using its default namespace. We then must prefix all the members of this namespace with `math` followed by a period. We also import the `os` module and use a function from it. Finally we import the NumPy master module (note that its native namespace is all lower case) and set up an array.

If we wish to change the namespace identifier, perhaps to avoid a lot of typing, we can change it when we import, but not thereafter.

```
import numpy as np
B=np.zeros(200)
```

We can import specified symbols, and only those symbols, with another form of the import statement.

```
from math import sqrt, exp
w=sqrt(19.)
y=exp(-2.)
```

When we use this form of import, we do not use a prefix with the items we imported.

```
w=sqrt(19.)
y=exp(-2.)
```

Finally, if we wish to import all the symbols in the module without requiring a prefix, use the form

```
from math import *
```

This should be avoided in most circumstances, because of the possibility of *namespace collisions* when two modules might contain functions with the same names. For example, both the math module and the NumPy module contain functions called `sin`, but the NumPy version has capabilities the math version does not have. We would generate an error if we tried to use the math version with a NumPy array argument. If it has no other way to distinguish a function name, the interpreter will use the one it saw last. It is frequently acceptable to import * with math, however. We could then use `sin` as the math version and e.g. `np.sin` as the NumPy version.

The namespace of the main module for your program is a special string `__main__` (two underscores on each side of the word `main`). The system assigns this namespace to whatever file is run directly through the interpreter. When you write a Python program, it is good practice to place all the general work that is not in other functions into a function conventionally called `main()`. We then invoke it with

```
if __name__=="__main__":
      main()
```

For example:

```
def main():
      print "Starting the program"
      # more code
      print "Program ended"
if __name__=="__main__":
      main()
```

Recall that we said any file you write is a module. If you include the

```
if __name__=="__main__":
      main()
```

clause, then anything in `main()` will be ignored if you import that file into another module.

Let's practice this on an example. We are going to implement one of the oldest numerical algorithms known, the "Babylonian method" for finding square roots. This method relies on the fact that if S is an overestimate of the square root of a nonnegative number x, then x/S is an underestimate, so their average is a better estimate than S was. Expressed as an algorithm, we start with a guess for $\sqrt{x}$ that we know is too large – we could use the number itself to get started—we then compute x/S and average it with S. Initially this gives us (x+1.)/2. as our next guess. In code, this looks like an update

S=0.5*(S+x/S)

We continue to update S until we reach the desired accuracy.

Our function will perform the following steps:

1. Initialize S_0
2. Loop while the difference between S_n and S_{n+1} is greater than the specified tolerance

3. Compute S_{n+1} from S_n

In code this becomes

```
def bab_sqrt(x):
    """This function implements the Babylonian method to find the
                                          square root"""
    tol=1.e-14
    old_sqrt=x
    new_sqrt=(x+1)/2.
    while abs(new_sqrt-old_sqrt)>tol:
        old_sqrt=new_sqrt
        new_sqrt=0.5*(old_sqrt+x/old_sqrt)
    return new_sqrt
```

Call the file with this function `test_bab_sqrt.py`. Now let's write a `main()` function

```
def main():
     tests=[0.,4.,9., 16., 25.]
     for x in tests:
         my_sqrt=bab_sqrt(x)
         print x,"      ",my_sqrt
if __name__=="__main__":
        main()
```

Run the code. All should seem fine. Now let's change tests to

```
tests=[2., 10.3, -2., 0.45]
```

We have a problem when we reach `-2`. You'll need to click the red button at the top of the iPython console to stop the runaway program. Our mistake was to try to use an algorithm that works only for nonnegative numbers on a negative value; in this case it will never converge.

We go back to our function to add code to make it check whether its input value is nonnegative. Make the first line

```
if (x<0): return None
```

If this tests `True` it will immediately return.

Now cut your bab_sqrt function and paste it into a file b_sqrt.py. At the end our b_sqrt.py file will contain

```
def bab_sqrt(x):
"""This function implements the Babylonian method to find the
     square root """
     if x<0: return None
     tol=1.e-14
     old_sqrt=x
     new_sqrt=(x+1)/2.
     while abs(new_sqrt-old_sqrt)>tol:
          old_sqrt=new_sqrt
          new_sqrt=0.5*(old_sqrt+x/old_sqrt)
     return new_sqrt
```

and our test_bab_sqrt.py will contain

```
def main():


    tests=[2., 10.3, -2., 0.45]

    for x in tests:
        my_sqrt=bab_sqrt(x)
        print "x:", x, "sqrt:", my_sqrt

if __name__=="__main__":
    main()
```

In order to use bab_sqrt in test_bab_sqrt.py we will need to add an import statement. Place this line below the import from math:

```
from b_sqrt import bab_sqrt
```

Let's do a more informative comparison of our result. We'll import `sqrt` from the math module and print the difference between our computation and the standard result. Now we can use the `is` operator to test in our main function (this is a slightly more “Pythonic” way to test than using the double equals as we did in our video):

```
from math import sqrt
from b_sqrt import bab_sqrt

def main():

    tests=[0.,-4.,9., 16., 25.]
    for x in tests:
        my_sqrt=bab_sqrt(x)
        if my_sqrt is None:
            my_sqrt = str(my_sqrt)+"\t"
            real_sqrt = sqrt(x)
            difference = real_sqrt
        else:
            real_sqrt=sqrt(x)
            difference = abs(real_sqrt-my_sqrt)
        print "x:", x, "  bab_sqrt:", my_sqrt, \
            "\tsqrt:", real_sqrt, "\tdiff: ", difference

if __name__=="__main__":
    main()
```

You have now written your own modules and imported one into another. Next episode we will discuss several important packages you can use in your programs.

Modules are important concepts in Python. Some good sources include
https://www.tutorialspoint.com/python/python_modules.htm
https://docs.python.org/2/tutorial/modules.html