Lab 4: Debugging Python Code (2/20)

Learning Objectives:

- Identify types of objects
- Use Python help to learn to use unfamiliar functions
- Use Python debugger

Link to Lab worksheet.
Link to worksheet solution.

Objects and Types

Everything in Python is an object, and every object has a type. You can find the type of object `obj` using the command `type(obj)`.

Knowing the type is important because the type determines:

- How the object is constructed.
- What attributes it has. The syntax for accessing an attribute is `obj.attributeName`.
- What special operations can be done. Examples of special operations include:
- Calling (only works if the type is `function`). Syntax: `obj()`
- Indexing (only works if type is ordered and iterable, like `list`, `tuple`, `numpy.ndarray`, etc). Syntax: `obj[0]`
- Operators (i.e., plus, minus, multiple, comparison, etc). Syntax for + operator: `obj+obj2`

As an example, lists are created using syntax `l=[1,3,5]`. Lists have the attribute `append`, which is a function we learned (function attributes are also called methods). Lists cannot be called, can be indexed, and can be added to one another (list concatenation).

As another example, strings are created using syntax `s="abc"`. Strings have the attribute `split`, which is a function (aka method). Strings cannot be called, can be indexed, and can be added to one another (string concatenation).

As a third example, sets are created using the syntax `s={1,3,5}`. Sets have the attribute `add`, which is a function that, once called, adds an element to the set. Sets cannot be called, cannot be indexed, and cannot be added to one another.

As a final example, the built-in function `len` is an object with type `function`, has the attribute `__str__`, which is a function that converts the function into a string name (see example below). (These functions with have prefix and suffices `__` are meant to be internal system functions that users don't need to access, but you can access it still as below.) The object `len` can be called since it is a `function`, cannot be indexed, and cannot be added to one another.

```
[4]: len.__str__()
```

```
'<built-in function len>'
```

Functions

Functions are objects that can be called using the notation `functionName()`, where you can specify a list of inputs in the parenthesis.

Definition

Functions are defined using the syntax as below.

```
def func(x,y=2,z=5):
    s=x+y+z
    return s
```

This code creates the object `func` with type `function`. Based on this code, the function can take 3 inputs, with the first input `x` mandatory and the second two inputs `y` and `z` optional. The default setting of those inputs are given by the equal signs. This function returns the object `s` which is created inside the function and is equal to the sum of the three inputs.

Evaluating a function

After defining a function, we can call it using the notation `funcName( list of inputs )`. This will replace this expression with the returned value.

For example, `a=func(1,2,3)` will set `a` to 1+2+3=6, because the first input `x` is set to 1, the second input `y` to 2, and the third input `z` to 3.

However, specifying fewer inputs will activate the default values. For example, `b=func(1,3)` will set `b` to `1+3+5=9`, because the first input `x` is set to 1, the second input `y` is set to 3, and the third one `z` is set to the default value 5.

Finally `c=func(1,z=7)` will set `c` to 1+2+7=10. Note that we can specify only the third input `z` and not the second by the notation `z=7` in the list of arguments.

Scope of variables inside functions

The **scope** of a variable is the region of the program where it can be accessed. In the above function, the variables `x`, `y`, `z` are created every time the function is called and assigned the given input values. The variable `s` is created inside the function. All four of these variables are called **local variables** because they can only be accessed insided the function. Every Python function has its own **namespace**, which is an association of names to objects. Hence, the local variables only exist inside the namespace of the function, and cannot be accessed outside. For example, the following code will return an error because the name `s` is only known inside the function, but not outside.

```
[5]: def func(x,y=2,z=5):
         s=x+y+z
         return s
     func(1,2,3)
     print(s)
```

```
---------------------------------------------------------------------------

NameError                                 Traceback (most recent call last)

<ipython-input-5-7825bb074338> in <module>()
  3     return s
  4 func(1,2,3)
----> 5 print(s)
```

```
NameError: name 's' is not defined
```

Variables from outside the function can be accessed inside, but not the other way around. This is because a function inherits the namespace of its environment, but the outside environment does not know what goes inside a function. For example, consider the code:

```
[6]: l=5
     x=5
     def func(x,y=2,z=5):
         print('Inside the function','l={0} x={1} y={2}'.format(l,x,y))
         s=x+y+z
         return s
     y=func(1,2,3)
     print('Outside the function','l={0} x={1} y={2}'.format(l,x,y))
```

```
Inside the function l=5 x=1 y=2
Outside the function l=5 x=5 y=6
```

Here, inside the function, the variable `l` is inherited from the environment and set to 5. The variable `x` is overwritten with the input argument `x=1`, and the variable `y` comes also from the input.

Outside the function, the input parameters `x`, `y`, `z` in the function are unknown. Hence the variable `x` is equal to the value above, which is 5, and the value `y` is assigned the return value of the function. In other words, the `x` and `y` outside the function refer to completely different objects as those inside the function. This is the meaning of the sentence "every function has its own namespace," or its own association of names and objects.

The reason for this is so that we can start afresh with variable names inside every function, and don't have to worry about conflicting names in what other people's functions use.

Location of Functions

See the tree diagram in the lecture. The main idea here is that if `a=np.array([1,2,5])`, then the following are three different functions:

```
sum
np.sum
a.sum
```

The first one is a built-in function, the second one is an attribute of the module `np`, the third one is an attribute of the numpy array object `a`. You should think of these as completely different functions with different syntax and behavior. An analogy is that these are files with the same name in different directories of your computer.

To search for help for each, you would type

```
sum?
np.sum?
np.array.sum?
```

It is important to find out the location of the function you want to use so that you make sure you are using the right function (and not another function with the same name), and so you can search for the right help file.

Using the Python Debugger

There are two ways of using the debugger `pdb`, you can set a break point in the code using the command

```
import pdb # put anywhere before
pdb.set_trace() # put at the location of break point
```

This will pause Python at the location of the break point and enable you to examine variables and run code. See the lab solutions for examples of commands to run. You can also read the `pdb` documentation here: https://docs.python.org/3/library/pdb.html

The basic commands are: - `l` for listing code around current line - `p x` for printing expression `x`. In most cases, you can also simply type the expression `x`, unless for example the variable is named something that conflicts with `pdb` commands (i.e. variable also named `p`). - `s` to step one line (execute one line) - `c` for continuing execution until next break point - `u` for going up one level (go from inside of a function to outside). - `d` for going down one level (go from outside of a function to inside). - `q` for quiting

An alternative way of using the debugger is to run

```
%pdb on
```

somewhere in a Code cell, then every time there is an error, the program will enter the debugger and allow you to examine variables instead of exiting.

Note: sometimes the `pdb` debugger will make your Jupter notebook stall. To fix this problem, simply restart the kernel by going to "Kernel" in the menu above and click "Restart." Also, make sure to quit the debugger using `q` or else you will not be able to run any other cell.

Often you can understand the error by reading the error message or by printing intermediate outputs, as we've done in previous labs. The reason to use the debugger is that sometimes the error message is hard to understand or you don't want to print everything, but only where there is a problem. Moreover, entering the debugger at the point of error allows you to examine the specific variables in the program at that point of execution of the code, which gives more information than the error message itself.