1 Custom Function and Modules

Python supports a wide range of programming styles including procedural (imperative), object oriented and functional. While object oriented programming and functional programming are powerful programming paradigms, especially in large, complex software, procedural is often both easier to understand and a direct representation of a mathematical formula. The basic idea of procedural programming is to produce a function or set of function (generically) of the form:

$$y = f(x).$$

That is, the functions take one or more inputs and produce one or more outputs.

1.1 Functions

Python functions are very simple to declare and can occur in the same file as the main program or a standalone file. Functions are declared using the `def` keyword, and the value produced is returned using the `return` keyword. Consider a simple function which returns the square of the input, $y = x^2$.

```
from __future__ import print_function, division

def square(x):
return x**2
# Call the function
x = 2
y = square(x)
print(x,y)
```

In this example, the same Python file contains the main program– the final 3 lines – aswell as the function. More complex function can be crafted with multiple inputs.

```
from __future__ import print_function, division
def l2distance(x,y):
return (xy)**
2
# Call the function
x = 3
y = 10
z = l2distance(x,y)
print(x,y,z)
```

Function can also be defined using NumPy arrays and matrices.

```
from __future__ import print_function, division
import numpy as np
```

```
def l2_norm(x,y):
d = x y
return np.sqrt(np.dot(d,d))
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z = l2_norm(x,y)
print(xy)
print("The L2 distance is ",z)
```

When multiple outputs are returned but only a single variable is available for assignment, all outputs are returned in a tuple. Alternatively, the outputs can be directly assigned when the function is called with the same number of variables as outputs.

```
from __future__ import print_function, division
import numpy as np
def l1_l2_norm(x,y):
d = x y
return sum(np.abs(d)),np.sqrt(np.dot(d,d))
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Using 1 output returns a tuple
z = l1_l2_norm(x,y)
print(xy)
print("The L1 distance is ",z[0])
print("The L2 distance is ",z[1])
# Using 2 output returns the values
l1,l2 = l1_l2_norm(x,y)
print("The L1 distance is ",l1)
print("The L2 distance is ",l2)
```

All of these functions have been placed in the same file as the main program.

1.1.1 Keyword Arguments

All input variables in functions are automatically keyword arguments, so that the function can be accessed either by placing the inputs in the order they appear in the function (positional arguments), or by calling the input by their name using keyword=value.

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p):
```

```
d = x y
return sum(abs(d)**p)**(1/p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
z1 = lp_norm(x,y,2)
z2 = lp_norm(p=2,x=x,y=y)
print("The Lp distances are ",z1,z2)
```

Because variable names are automatically keywords, it is important to use meaningful variable names when possible, rather than generic variables such as a, b, c or x, y and z. In some cases, x may be a reasonable default, but in the previous example which computed the Lp norm, calling the third input z would be bad idea.

1.1.2 Default Values

Default values are set in the function declaration using the syntax input=default.

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p = 2):
d = x y
return sum(abs(d)**p)**(1/p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
l2 = lp_norm(x,y)
l1 = lp_norm(x,y,1)
print("The l1 and l2 distances are ",l1,l2)
print("Is the default value overridden?", sum(abs(xy))==
l1)
```

Default values should not normally be mutable (e.g. lists or arrays) since they are only initialized the first time the function is called. Subsequent calls will use the same value, which means that the default value could change every time the function is called.

```
from __future__ import print_function, division
import numpy as np
def bad_function(x = zeros(1)):
print(x)
x[0] = np.random.randn(1)
# Call the function
bad_function()
bad_function()
bad_function()
```

Each call to `bad_function` shows that x has a different value – despite the default being 0. The solution to this problem is to initialize mutable objects to None, and then the use an if to check and initialize only if the value is None. Note that tests for None use the is keyword rather the testing for equality using ==.

```
from __future__ import print_function, division
import numpy as np
def good_function(x = None):
if x is None:
x = zeros(1)
print(x)
x[0] = np.random.randn(1)
# Call the function
good_function()
good_function()
```

1.2 Variable Number of Inputs

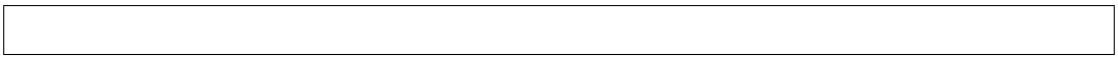
Most function written as an "end user" have an known (ex ante) number of inputs. However, functions which evaluate other functions often must accept variable numbers of input. Variable inputs can be handled using the *args (arguments) or **kwargs (keyword arguments) syntax. The *args syntax will generate tuple a containing all inputs past the required input list. For example, consider extending the Lp function so that it can accept a set of p values as extra inputs (Note: in practice it would make more sense to accept an array for p).

```
from __future__ import print_function, division
```

```
import numpy as np
def lp_norm(x,y,p = 2, *args):
d = x-y
print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
out = [sum(abs(d)**p)**(1/p)]
print('Number of *args:', len(args))
for p in args:
print('The L' + str(p) + ' distance is :', sum(abs(d)**p)**(1/p))
out.append(sum(abs(d)**p)**(1/p))
return tuple(out)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# x & y are required inputs and so are not in *args
lp = lp_norm(x,y)
# Function takes 3 inputs, so no *args
lp = lp_norm(x,y,1)
# Inputs with default values can be ignored
lp = lp_norm(x,y,1,2,3,4,1.5,2.5,0.5)
```

The alternative syntax, **kwargs, generates a dictionary with all keyword inputs which are not in the function signature. One reason for using **kwargs is to allow a long list of optional inputs without having to have an excessively long function definition. This is how this input mechanism operates in many matplotlib functions such as plot.

```
from __future__ import print_function, division
import numpy as np
def lp_norm(x,y,p = 2, **kwargs):
d = x y
print('Number of *kwargs:', len(kwargs))
for key in kwargs:
print('Key :', key, ' Value:', kwargs[key])
return sum(abs(d)**p)
# Call the function
x = np.random.randn(10)
y = np.random.randn(10)
# Inputs with default values can be ignored
lp = lp_norm(x,y,kword1=1,kword2=3.2)
# The p keyword is in the function def, so not in **kwargs
lp = lp_norm(x,y,kword1=1,kword2=3.2,p=0)
```



1.3 Anonymous Functions

Python support anonymous functions using the keyword lambda. Anonymous functions are usually encountered when another function expects a function as an input and a simple function will suffice. Anonymous function take the generic formlambda a,b,c,. . .:code using a,b,c. The key elements are the keyword lambda, a list of comma separated inputs, a colon between the inputs and the actual function code. For example lambda x,y:x+y would return the sum of the variables x and y.

Anonymous functions are simple but useful. For example, when lists containing other lists it isn't directly possible to sort on an arbitrary element of the nested list. Anonymous functions allow sorting through the keyword argument key by returning the element Python should use to sort. In this example, a direct call to `sort()` will sort on the first element (first name). Using the anonymous function lambda x:x[1] to returnthe second element of the tuple allows for sorting on the lastname. `lambda x:x[2]` would allow for sorting on the University.

```
>>> nested = [('John','Doe','Oxford'),\
... ('Jane','Dearing','Cambridge'),\
... ('Jerry','Dawn','Harvard')]
>>> nested.sort()
>>> nested
[('Jane', 'Dearing', 'Cambridge'),
('Jerry', 'Dawn', 'Harvard'),
('John', 'Doe', 'Oxford')]
>>> nested.sort(key=lambda x:x[1])
>>> nested
[('Jerry', 'Dawn', 'Harvard'),
('Jane', 'Dearing', 'Cambridge'),
('John', 'Doe', 'Oxford')]
```

1.4 Python Coding Conventions

There are a number of common practices which can be adopted to produce Python code which looks more like code found in other modules:

1. Use 4 spaces to indent blocks – avoid using tab, except when an editor automatically converts tabs to 4 spaces
2. Avoid more than 4 levels of nesting, if possible
3. Limit lines to 79 characters. The symbol can be used to break long lines 219

4. Use two blank lines to separate functions, and one to separate logical sections in a function.
5. Use ASCII mode in text editors, not UTF-8
6. One module per import line
7. Avoid from module import * (for any module). Use either from module import func1, func2 or import module as shortname.
8. Follow the NumPy guidelines for documenting functions

More suggestions can be found in PEP8.