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. Placing functions in modules allows for reuse in multiple programs, and will be discussed later in this chapter.

1.1.1 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
```

1.2 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')]
```

1.3 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.