Lecture 5.1 : Functions

Introduction

- Functions facilitate a **divide-and-conquer** approach to problem-solving: when confronted with a complex problem we break it down into a number of simpler subproblems. The solution to each subproblem is implemented as a function. This approach allows us to create better quality, more readable code that is simpler to write and maintain.
- Inside a function we place code that typically takes some information (passed to it in the form of arguments), uses that information to calculate a result and returns that result to a caller. To use the function we do not need to know how it calculates its result, we merely need to know how to invoke the function. This hiding of implementation details is called **encapsulation**.
- Duplication of code is to be avoided. Once a function has been coded it can be called from anywhere in your program. Thus functions support code **reuse**. Functions can also be placed in a module to be imported and invoked by other programs. Thus functions support code **sharing**.
- Because of their simplicity, individual functions are more easily tested and verified compared to more complex, monolithic code blocks. Using functions thus produces more **secure** and **reliable** code.

Functions

```
import sys

def celsius2fahrenheit(c):
    f = c * 1.8 + 32
    print('That is {:.2f} degrees Fahrenheit'.format(f))

def main():
    temp = float(sys.argv[1])
    celsius2fahrenheit(temp)

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

```
$ python3 celsius_01.py 30
That is 86.00 degrees Fahrenheit
```

- Above we see the definition of a function `celsius2fahrenheit`. The `c` variable in the definition of the function is called a *parameter*. A parameter is essentially a variable that is *local* to the function: the `c` variable thus cannot be referenced outside of `celsius2fahrenheit` function.
- Variables which are created within a function are said to be *local* to the function. They are created during the execution of the function, and do not survive after the invocation has completed.
- Initial values for parameters, supplied at invocation, are called *arguments* or *actual parameters*. We see that when we call the `celsius2fahrenheit` function (from the `main` function) we pass to it an *argument*. The argument in this case is the `temp` variable. What is the relationship between the argument `temp` and the parameter `c`? Well, when the function is invoked the contents of `temp` *are copied into* `c`.

- By default, arguments and parameters are matched by position i.e. the first argument is copied into the first parameter, the second argument is copied into the second parameter, etc.
- Note how the `celsius2fahrenheit` function does not return any data to the caller. It calculates the temperature in Fahrenheit and prints it. Functions which do not return any value are called *procedures*. (It turns out that functions that lack a return statement do in fact return a value to their caller in Python, that value is `None`.)
- Procedures effect a change in the world. For example, they display something on a screen, or change the values of variables, or change the contents of a file, or delete a file from a disk. Functions on the other hand merely inspect the world without changing it. The result of their inspection is a value. We can use the function invocation anywhere an expression of the type returned by the function can be used.
- Another way to describe the difference between procedures and functions is to say that procedures are like complex *statements* while functions are like complex *expressions*.
- Suppose we want our function to make available, to the caller, the newly calculated temperature in Fahrenheit. How can we do that?

Return values

```
import sys

def celsius2fahrenheit(c):
    return c * 1.8 + 32

def main():
    temp = float(sys.argv[1])
    f = celsius2fahrenheit(temp)
    print('That is {:.2f} degrees Fahrenheit'.format(f))

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

```
$ python3 celsius_02.py 30
That is 86.00 degrees Fahrenheit
```

- Above we have added a `return` statement to our `celsius2fahrenheit` function. The effect is to *hand back* to the caller of the function the value of `c * 1.8 + 32`. Since the function returns a value its caller is expected to collect that value. Above we see the caller collects the returned value and assigns it to the variable `f`.

Multiple return statements

```
import sys

def biggest(x, y):
    if x > y:
        return x
    return y

def main():
    x = int(sys.argv[1])
    y = int(sys.argv[2])
    print('The biggest value is {}'.format(biggest(x, y)))
```

```python
if __name__ == '__main__':
    main()
```

- As illustrated above, a function may have more than one `return` statement. Execution of the function terminates and control returns to the caller as soon as the first `return` statement is executed however.

Returning multiple values

```python
import sys
import math

def sphere(r):
    v = (4.0 / 3.0) * math.pi * r**3
    sa = 4.0 * math.pi * r**2
    return (v, sa)

def main():
    radius = float(sys.argv[1])
    (v, sa) = sphere(radius)
    print('Volume: {:.3f} m^3'.format(v))
    print('Surface area: {:.3f} m^2'.format(sa))

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

```
$ python3 sphere.py 5
Volume: 523.599 m^3
Surface area: 314.159 m^2
```

- A function can return only a single object. If we wish to return multiple values, such as in the example above where we require a function to return both the volume and surface area of a sphere, then we must wrap up those values in a single object and return that object. In the case above the object returned is a tuple. The caller extracts the values from the tuple using multiple assignment and prints them separately.
- Note that although a tuple is used above to wrap the two returned values, any object capable of capturing the two values will do e.g. a list, dictionary, custom object, etc.

Variable scope

- When a function is executed it creates its own *scope*. Any variables that come into existence over the course of execution of the function belong to its *namespace* and are *local* to it. A variable comes into existence once it is *assigned* a value. Variables that are local to a function can only be referenced within that function and are inaccessible outside that function.

- If a function is invoked repeatedly, its local variables and parameters are created anew for each invocation, and they die when the function has completed its execution for that invocation. The final value of a local variable does not carry over to any following invocation of the function.

```python
import sys
```

```python
def celsius2fahrenheit(c):
    f = c * 1.8 + 32

def main():
    temp = float(sys.argv[1])
    celsius2fahrenheit(temp)
    print('That is {:.2f} degrees Fahrenheit'.format(f))

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

```
$ python3 scope_01.py 30
NameError: global name 'f' is not defined
```

- Above we see that a variable `f` is assigned inside the `celsius2fahrenheit` function and thus only comes into existence in that function and so is local to that function. The reference to `f` in `main` is therefore illegal since `f` does not exist outside of `celsius2fahrenheit` and an error is generated.
- Note that assignments *to* a parameter can never affect the associated argument. Assignments *through* a mutable parameter will however affect the argument and thus be visible to the caller.

Global and local scope

```python
x = 42

def foo():
    x = 33
    print(x)

def main():
    foo()
    print(x)

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

```
$ python3 globals_01.py
33
42
```

- It is the act of *assignment* that creates a variable. The assignment `x = 42` (outside of any function) creates a *global* variable called `x`. The assignment of `x = 33` inside the `foo` function creates a *new* variable `x` that is *local* to `foo`.

```python
x = 42

def foo():
    print(x)
    x = 33
    print(x)

def main():
    foo()
    print(x)
```

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

```
$ python3 globals_02.py
UnboundLocalError: local variable 'x' referenced before assignment
```

- The reference to `x` in the first `print(x)` is a reference to the *global* variable `x`. The assignment `x = 33` creates a *new* local variable `x`. Thus `x` is both local and global in the same function. Python does not permit this kind of ambiguity and deems `x` to be local throughout the function. Thus the reference to `x` in the first `print(x)` is an error since the local variable `x` has not yet been initialised.

```
x = 42

def foo():
    x = x + 1
    print(x)

def main():
    foo()
    print(x)

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

```
$ python3 globals_03.py
UnboundLocalError: local variable 'x' referenced before assignment
```

- This is a similar case to the one above. The reference to `x` on the right hand side of `x = x + 1` is a reference to the *global* variable `x`. The assignment `x = x + 1` however creates a *new* local variable `x`. Thus `x` is both local and global in the same function. Python does not permit this kind of ambiguity and deems `x` to be local throughout the function. Thus the reference to `x` on the right hand side of `x = x + 1` is an error since the local variable `x` has not yet been initialised.

```
x = 42

def foo():
    y = x + 1
    print(y)

def main():
    foo()
    print(x)

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

```
$ python3 globals_04.py
43
42
```

- There are no problems here. The reference to `x` in `y = x + 1` is to the global variable `x` and the assignment to `y` creates a new local variable called `y`.

```python
l = []

def foo():
    l.append(99)
    print(l)

def main():
    foo()
    print(l)

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

```
$ python3 globals_05.py
[99]
[99]
```

- There are no problems here. We write *through* the reference in *l* to update the underlying global list. Since there is no assignment no new local variables are created.

```python
x = 42

def foo():
    global x
    x = 33
    print(x)

def main():
    foo()
    print(x)

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

```
$ python3 globals_06.py
33
33
```

- The `global x` statement marks `x` in this function as always a reference to the global variable `x`. Thus the assignment `x = 33` in this case does *not* create a new local variable and instead updates the global variable `x`.

Programs, modules and functions

- As the programs you write get longer and more complex you may want to group related functions in separate files to facilitate maintenance. You may also have a handy function that you would like to use in several programs without having to copy its definition into each. In Python we place function definitions in a *module* from where we can *import* them into a *program*.

- For example, below we import the `random` module before calling its `random` and `select` functions:

```python
>>> import random
>>> random.random()
0.5367186947938044
>>> random.random()
```

```
0.9155009876145279
>>> random.sample([1,2,3,4,5],3)
[1, 5, 2]
>>> random.sample([1,2,3,4,5],3)
[3, 5, 1]
```

- Should we wish to import only particular functions from a module we can do so as follows (note how we can then reference such directly imported functions in our program without going through the module reference):

```
>>> from random import random, sample
>>> random()
0.6005839537277994
>>> random()
0.23831832488473992
>>> sample([1,2,3,4,5],3)
[3, 5, 4]
>>> sample([1,2,3,4,5],3)
[5, 2, 1]
```

Programs as modules

- Suppose we have the following program called `handy_functions.py` that includes some code to test the `count_vowels` function:

```
# handy_functions.py
vowels = 'aeiouAEIOU'
def count_vowels(s):
    return sum([1 for c in s if c in vowels])

test_words = ['apple', 'Orange', 'pineapple']
for word in test_words:
    print('{} : {}'.format(word, count_vowels(word)))
```

```
$ python3 handy_functions.py
apple : 2
Orange : 3
pineapple : 4
```

- Pleased with our function we would like to make it available to other programs for import: We would like our program to also serve as a module. However, look what happens when we try to import `handy_functions.py`:

```
>>> import handy_functions
apple : 2
Orange : 3
pineapple : 4
```

- We have a problem. When we import `handy_functions.py` any code it contains that is not in a function is executed. That leads to the unwanted situation above where importing `handy_functions.py` causes some output to be printed as a result of our test code.

- What we really want is two behaviours:

1. If `handy_functions.py` is being executed *as a program* then we want the test code to be executed.
2. If `handy_functions.py` is being imported *as a module* (i.e. as a set of functions) then we do not want the the test code to be executed.

- It turns out we can achieve the desired behaviour by taking advantage of the fact that within a module, the module's name (as a string) is available as the value of the global variable `__name__`.
- If `handy_functions.py` is *executed as a program* then `__name__ == '__main__'` while if `handy_functions.py` is *imported* then `__name__ == '__handy_functions__'`.
- Rewriting our code as follows achieves the desired behaviour:

```
# handier_functions.py
vowels = 'aeiouAEIOU'
def count_vowels(s):
    return sum([1 for c in s if c in vowels])

def main():
    test_words = ['apple', 'Orange', 'pineapple']
    for word in test_words:
        print('{} : {}'.format(word, count_vowels(word)))

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

- Running the code from the command line (i.e. as a program) causes the `main` function to be executed:

```
$ python3 handier_functions.py
apple : 2
Orange : 3
pineapple : 4
```

- Loading the code as a module skips the execution of the `main` function and simply makes the `count_vowels` function available to the importer:

```
>>> import handier_functions
>>> handier_functions.count_vowels('tangerine')
4
```

Module/program template

- Thus the following template will work irrespective of whether you are asked to write a program or a module:

```
# Module/program template

# Global variables go here...

# Function definitions go here...

def main():
    # Put here the code that calls the other functions...
```

```
    pass

# Am I a module or a program?
if __name__ == '__main__':
    main()
```