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.
- Related functions can also be placed in a module to be imported and invoked by other programs.
- Thus functions support code **sharing** and **reuse**.
- 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

- Let's write a function that converts Celsius to Fahrenheit.

```
def celsius2fahrenheit(c):
   f = c * 1.8 + 32
   print(f'That is {f:.1f} degrees Fahrenheit')

celsius = 21
celsius2fahrenheit(celsius)
```

```
That is 69.8 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 the `celsius2fahrenheit()` function.
- Variables which are created within a function are said to be *local* to the function.
- Local variables are created during the execution of the function, and do not survive after the invocation has completed.
- Values for parameters, supplied at invocation, are called *arguments* or *actual parameters*.

- We see that when we call the `celsius2fahrenheit()` function we pass to it an *argument*.
- The argument in this case is the `celsius` variable.
- What is the relationship between the argument `celsius` and the parameter `c`?
- Well, when the function is invoked the contents of `celsius` *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, change the values of variables, 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

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

celsius = 21
fahrenheit = celsius2fahrenheit(celsius)
print(f'That is {fahrenheit:.1f} degrees Fahrenheit')
```

```
That is 69.8 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 the variable `fahrenheit` to it.

Multiple return statements

```
def bigger(a, b):
   if a > b:
      return a
   return b

print(f'The bigger value is {bigger(3, 4)}')
```

```
The bigger value is 4
```

- 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. As soon as a function has an answer it can return it.

Returning multiple values

- A function can return only a single *object*.
- If we wish to return multiple values, such as in the example below 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 below the object returned is a tuple. The caller unpacks 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.

```
from math import pi

def sphere(r):
   v = (4.0 / 3.0) * pi * r**3
   sa = 4.0 * pi * r**2
   return v, sa # return a tuple

volume, area = sphere(1)
print(f'The volume is {volume:.1f} m^3')
print(f'The surface area is {area:.1f} m^2')
```

```
The volume is 4.2 m^3
The surface area is 12.6 m^2
```

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* to 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 disappear when the function has completed its execution for that invocation.
- The value of a local variable does not carry over to any following invocation of the function.
- Below we see a failed attempt to reference a local variable outside of the function where it resides.

```
def sphere(r):
   v = (4.0 / 3.0) * pi * r**3
   sa = 4.0 * pi * r**2
   return v, sa # return a tuple

sphere(1)
print(v)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [5], in <module>
      4    return v, sa # return a tuple
      6 sphere(1)
----> 7 print(v)

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

Global and local scope

- A variable that is visible across the program namespace (rather than being confined to a particular function's namespace) is called a *global* variable.

```
x = 42

def foo():
   print(x)

foo()
print(x)
```

```
42
42
```

Scope puzzle #1

- It is the act of assignment of a variable to a value that causes the variable to come into existence.
- Thus the assignment in the function below creates a *new* local variable `x` that is distinct from the global one.

```
x = 42

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

foo()
print(x)
```

```
33
42
```

Scope puzzle #2

- Below we confuse Python and it does not like it.
- The reference to `x` on line 4 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` on line 4 is an error since the local variable `x` has not yet been assigned to a value.

```
1  x = 42
2
3  def foo():
4     print(x)
5     x = 33
6     print(x)
7
8  foo()
9  print(x)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [8], in <module>
      5     x = 33
      6     print(x)
----> 8 foo()
      9 print(x)

Input In [8], in foo()
      3 def foo():
----> 4     print(x)
      5     x = 33
      6     print(x)

UnboundLocalError: local variable 'x' referenced before assignment
```

Scope puzzle #3

- 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 and Python is again unhappy (and says so).

```
1  x = 42
2
3  def foo():
4     x = x + 1 # same issue with x += 1
```

```
    print(x)

foo()
print(x)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
Input In [9], in <module>
      4     x = x + 1 # same issue with x += 1
      5     print(x)
----> 7 foo()
      8 print(x)

Input In [9], in foo()
      3 def foo():
----> 4     x = x + 1 # same issue with x += 1
      5     print(x)

UnboundLocalError: local variable 'x' referenced before assignment
```

Scope puzzle #4

- So how can a function update a global variable if the assignment creates a *new* variable?!
- By marking the variable as *global*.
- Line 4 marks `x` in this function as always a referring to the global variable `x`.
- Thus the assignment `x = x + 1` in this case does *not* create a new local variable and instead updates the global variable `x`. Phew!

```
x = 42

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

foo()
print(x)
```

```
43
43
```

Programs, modules, functions

- As the programs you write get longer and more complex you may want to group related functions into a file to facilitate maintenance and sharing.
- 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 related function definitions in a *module* from where we can *import* them into a *program*.
- (Going further we can group related modules into packages.)

- For example, below we import the `math` module before using its `cos` function and its `pi` definition.

```
import math

print(math.cos(math.pi))
```

```
-1.0
```

- Should we wish to import only particular functions or definitions from a module we can do that too.
- We can then reference them directly in our program without going through the module reference.

```
from math import sin, pi

print(sin(3*pi/2))
```

```
-1.0
```

Programs as modules

- The Python interpreter maintains a global variable `__name__`.
- We have some code in `hello.py`.
- If `hello.py` is *executed as a program* then `__name__ == '__main__'`.
- If `hello.py` is *imported as a module* then `__name__ == '__hello__'`.
- We can take advantage of this as follows.

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

- If `hello.py` is being executed then run `main`.
- Otherwise we are simply importing the module (so we can use its functions) and we do so without running `main`.
- That's why we always include this snippet of code in our programs/modules.

Module/program template

- The following template will work irrespective of whether you are asked to write a program or a module.

```
# Imports go here...

# Global variables go here...

# Function definitions go here...
```

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

# If I am a program call main
if __name__ == '__main__':
    main()
```