Functions

Overview

- So far, our programs have run sequentially from start to finish.
    - Sometimes we used loops or ifs make code run repeatedly, or only under certain circumstances…
    - ...but we still ran from start to finish.
- When writing big programs, this could quickly get confusing.
- What if we want to reuse a small algorithm at various points in our program?
    - Copying it out over and over again would be a waste of time, effort, and space, and be hard to maintain.
- Functions are the solution!
- A function is a block of code, which is given certain inputs (called *parameters* or *arguments*), and can optionally *return* a value when it finishes.

Simple Functions

- The most basic syntax for a function, which takes no parameters and returns no value, is:
    - ```
      def function_name():
          ... code goes here ...
      ```
    - This is the 'code block' syntax we are familiar with - the indented code belongs to the function.
- The function code is not run on its own - it is ignored by the interpreter.
- To run the function, it must be *called*. The syntax to call a function is:
    - `function_name()`
    - Watch out! You can only call a function *after* you have defined it.
        - (Just like you cannot use a variable until you have defined it.)
        - Technical detail for the curious: More specifically, the interpreter, parsing from top to bottom, must have already come across the function definition by the time the function is called. This means that one function can call another function defined further down the program just fine, so long as the first function is only called by code *after* the second function's definition.
        - In summary: define all your functions *first*, before your main code, and everything will be fine.
- Example:
    - The following code defines a function that outputs a greeting.
    - ```
      def greet():
          print("Well, hello there!")
      ```
    - We could then call this function using:
    - `greet()`

Taking Parameters

- We will usually want to give our functions some inputs to work with.
- We can place parameters in the brackets of the function definition.
- Parameters are really just variables within the function!
- We can have as many or as few as we want.

    - `def function_name(param1, param2, etc...):`
      `... code goes here ...`
- When calling the function, we can *pass* parameters by placing them inside the brackets in the same order.
    - `function_name(value1, value2, etc...)`
- Example:
    - ```
      def greet(name):
          print("Hello, %s!" % (name))
      ```
    - `greet("Alex")`
- You will notice that this is very similar to `print()` and `input()` that we have seen previously. These are both functions built into Python!
- Exercise:
    - Write a function that takes two numbers, and prints their sum and their product.
    - Call this function several times with different values.
    - ```
      def sum_and_product(x, y):
          print("Sum: %d" % (x + y))
          print("Product: %d" % (x * y))

      sum_and_product(2, 3)
      sum_and_product(8, 12)
      sum_and_product(10, 10)
      ```
    - Observe how the interpreter does not just run `sum_and_product` immediately - it is only run when it is called.
    - Each time it is run, different parameters are used, so the results are different - the code is reused.

Returning Values

- Rather than outputting the results of our functions, we might want to store them in variables to use elsewhere - we need our functions to *return* a value.
- This is done with the `return` statement.
- When used on its own, it will simply cause the function to terminate.
    - ```
      def myfunction():
          print("Hi there!")
          return
          print("This message will never be shown...")
      ```
    - `myfunction()`
- When used with a value, that value is *returned*.
    - The code that called the function will evaluate to the returned value.
    - ```
      def add(x, y):
          return x + y
      ```
    - ```
      z = add(3, 4)
      print(z)        # Outputs 7
      ```
- Exercise:
    - Recall from the lesson on Expressions and Variables the following program to calculate properties of a circle:
      ```

        - ```
          r = float(input("Radius: "))
          pi = 3.14159
          circ = 2 * pi * r
          area = pi * r * r
          print("Circumference: %f" % circ)
          print("Area: %f" % area)
          ```
    - Rewrite this program to use two functions: one to calculate and return the circumference, and one to calculate and return the area. Take an input radius, pass it to both functions, and print the values returned.
    - ```
      pi = 3.15159

      def circ(r):
          return 2 * pi * r

      def area(r):
          return pi * r * r

      r = float(input("Radius: "))
      print("Circumference: %f" % circ(r))
      print("Area: %f" % area(r))
      ```
    - Note that the variable `pi` was available for use inside the functions. We will come back to the rules for this in a moment.
- What if we want to return multiple values?
- Remember, we can return any data type we want - this includes compound data structures such as lists and tuples.
    - ```
      def sum_and_product(x, y):
          return (x + y, x * y)

      (s, p) = sum_and_product(6, 7)
      # s = 13
      # p = 42
      ```

Variable Scope

- Not all variables can be accessed by all parts of the program.
- Each variable has something called a *scope* - which part of the program it exists in.
- Variables with *global* scope are accessible throughout the program.
    - Any variable defined in the main body of the program has global scope.
- Variables with *local* scope are accessible only within the function in which they were defined.
    - Any variable defined in a function has local scope within that function only.
- If a variable exists both within the current local scope, and at global scope, the locally scoped variable takes priority.
- For example, consider the solution to our previous exercise:
    - ```
      pi = 3.15159

      def circ(r):
      ```

```
    return 2 * pi * r

def area(r):
    return pi * r * r

r = float(input("Radius: "))
print("Circumference: %f" % circ(r))
print("Area: %f" % area(r))
```

- `pi` is declared in the main body of the program, so is a global variable. Whenever our functions use it, they are using that global value.
- We have three *different* variables called `r` in this program!
  - `r = float(input("Radius: "))` This is a global variable, since it is defined in the main program.
  - `def circ(r)` This is a local variable, since it is defined as a function parameter.
  - `def area(r)` This is also a local variable, but is local to *this* function.
  - When passing the value `r` in the print statements, the only available `r` is the global one, so this is used.
  - When within the functions, the relevant local `r` is available, so this is used instead of the global `r` (although, in this example, they both have the same value anyway).

<u>Advanced Parameter Usage</u>

- We can do fancier things with function parameters.
- These are all more advanced concepts to be aware of, but don't worry if you don't get them all.
- We can set default parameter values:
  - ```
    def greet(name = "User"):
        print("Hello, %s!" % (name))
    ```
  - ```
    greet("Alex")         # "Hello, Alex!"
    greet()               # "Hello, User!"
    ```
  - If no value is specified for that parameter, the default value is used.
  - Note that default parameters must be specified *after* parameters without default values - otherwise it would be ambiguous!
    - i.e.:
      - ```
        def good(a, b = 2):
            ...
        ```
    - Not:
      - ```
        def bad(a = 6, b):
            ...
        ```
- It is possible to specify individuals parameters by name, if we wanted to use them out of order for whatever reason:
  - ```
    def greet(greeting, name):
        print("%s, %s!" % (greeting, name))
    ```
  - Normally:
    ```

        - `greet("Hello", "Alex")`
    - But instead we could write:
        - `greet(name = "Alex", greeting = "Hello")`
- We can make our functions take a variable number of parameters, by specifying the final parameter in a particular way:
    - `def many_args(param1, param2, *otherparams):`
      `...`
    - The asterisk marks the final parameters, `otherparams`, as variable length.
    - It will be a tuple containing the remaining parameters.
    - The earlier parameters act like normal.
    - So:
        - The call:
            - `many_args(4, "hello", "x", "y", "z")`
        - Will set the parameters of `many_args` to:
            - `param1 = 4`
            - `param2 = "hello"`
            - `otherparams = ("x", "y", "z")`
- *Passing by reference* - something to watch out for:
    - In Python, values are passed to function parameters '*by reference*'.
    - This means that, even though the variables in your function are different to those you might have passed in, they both refer to *the same memory*.
    - If you reassign a variable inside your function, that is fine.
        - ```
          def myfunction(y):
              y = 5

          x = 2
          print(x)              # x is 2
          myfunction(x)
          print(x)              # x is 2
          ```
    - But if you carry out an operation that changes the memory itself, you will affect the original.
        - For instance, method calls on objects (subject of a future lesson):
        - ```
          def myfunction(y):
              y.append(5)

          x = [2, 4]
          print(x)              # x is [2, 4]
          myfunction(x)
          print(x)              # x is [2, 4, 5]
          ```

<u>Recursion</u>

- It is possible for one function to call another function - how this works should be fairly clear by now.
    - ```
      def myfunction():
          print("Hi!")
      ```
      ```

```
def otherfunction():
    myfunction()
    print("Other function")
```

- It is also possible for a function to call itself!
- This is called *recursion*.
- We must be careful that our recursive functions do not call themselves forever, or the program will run out of memory and crash.
    - ```
      def badfunction():
          print("Let's go on forever!")
          badfunction()
      ```
- We therefore will want to pass some parameter between each call of our recursive function, changing it each time, until it reaches some value at which we stop recursing.
- Example:
    - We could calculate integer powers really inefficiently using a recursive function.
    - ```
      def power(x, n):
          if n == 1:
              return x
          else:
              return x * power(x, n - 1)
      ```
- Exercise:
    - The *factorial* (!) of a number is defined as the product of every integer between 1 and that number.
    - e.g. 5! = 1 * 2 * 3 * 4 * 5 = 120
    - Write a recursive function find the factorial of an input.
    - ```
      def factorial(x):
          if x == 1:
              return x
          else:
              return x * factorial(x - 1)
      ```

Next lesson: Modules
      ```