Functions

Execution order

- When you load and run a Python module (file), the statements and definitions in the file are executed in the order in which they occur
- Executing a **`def`** ***defines*** a function, it doesn't run the function
  - Functions are only run when they are called
- A very small program might not define any functions at all, but just be a series of statements to be executed
- Most programs consist of a lot of function definitions, along with maybe a few *top-level* statements (statements not in functions)
- Usually one particular function is the "starting point" for running the program
  - By convention, this function is usually named **`main`**, and usually it has no parameters
  - Often a call to **`main()`** is the last top-level statement in a program
  - ```
    def main():
        print('Hello!')

    main()
    ```
    ```

Defining and calling functions

- Simple Python programs are just a collection of functions, where all the functions are ***top-level***, that is, not included in something else
- The syntax for ***defining*** a function is
  - `def` *function_name*`(`*parameter1*`, ..., `*parameterN*`):`
    `"""Comment describing the function"""`
    one or more statements
    - Inside a function, you can have one or more statements of the form `return` *value* or just `return` (with no *value*)
- The syntax for ***calling*** a function is
  - *function_name*`(`*argument1*`, ..., `*argumentN*`)`
  - *Usually*, you should call a function with as many arguments as the function has parameters
  - *Usually*, you should do something with the value returned from the function (save it in a variable, or use it in an expression or test)
- If the function has no parameters, the `()` parentheses are still required

Scope I

- The ***scope*** of a name is the part of the code in which other uses of that name refer to the same thing
    - A ***default*** is what you get if you don't specify otherwise
- By default, the scope of a variable name is ***local*** to the function in which it occurs
    - Every use of a name inside a function refers to the same thing
    - If the same name is used in two different functions, they refer to two different things
- ```
  def quadruple(x):
      return double(x) + double(x)

  def double(x):
      return x + x
  ```

- ```
  x = 10
  print(quadruple(x)) # prints 40
  ```

Example function

- ```
  def average(x, a):
      """Averages two numbers."""
      return (x + a) / 2
  ```
- Suppose `a = 7` and `b = 10`, and you execute the statement

  ```
  avg = average(a, a + b)
  ```
- Here's what happens:
  1. The value `7` (from `a`) gets assigned to the function's `x`.
  2. The value `17` (from `a + b`) gets assigned to the function's `a`.
  3. The function computes and returns the value `12.0`.
  4. The value `12.0` is assigned to `avg`. (outside the function, `a` is still `7`, and `b` is still `10`).
- Notice that the `a` inside the function is different from the `a` outside the function.
- Notice that the values of the arguments (`a` and `a + b`) are assigned to the function's parameters (`x` and a different `a`), but the parameters are *not* assigned back to the arguments
  ```

Returning a result

- ***Every*** function returns a result
- There are three ways to return from a function:
  1. Say **`return`** *value* -- the result of the function is *value*
  2. Say **`return`** -- the result of the function is **`None`**
  3. Reach the end of the function -- the result is **`None`**
- **`None`** is a special "uninteresting" value, but it *is* a value-- you can assign it, compare something to it, etc.
- Some functions, such as **`print`** and **`append`**, return **`None`**
- When you execute a **`return`** statement from within a loop, you don't finish the loop; the function is finished

Pure functions

- A function is ***pure*** if -
  - the only information it gets is from its parameters, and
  - the only information it produces is the value it returns
- Functions that get input or write output are not pure
- Functions that call impure functions are not pure (but they can call other pure functions)
- Functions that use global variables (next slide) are not pure

- Pure functions, being self-contained, add less to the overall complexity of a program than impure functions
- **Rule:** Minimize the number of impure functions

Global variables I

- A ***global*** variable is one whose scope is the entire program or entire module (file)
  - Global variables are sometimes used for very important values used in many places
    - For example, if you were writing a chess playing program, you might make the chessboard global
- Any function that uses global variables becomes dependent on whatever other functions might have done to those global variables
  - Consequently, global variables can complicate debugging and understanding of programs
- **Rule:** Global variables should be used sparingly, if at all

Global variables II

- You can make a variable global by giving it a value outside any function:
  - ```
    foo = 77

    def main():
        print(foo) # prints 77
    ```
- You can make a variable global by declaring it to be **`global`** and giving it a value inside a function
  - ```
    def bar():
        global foo
        foo = 33

    def main():
        bar()
        print(foo) # prints 33

    main()
    ```
- You cannot **`print(foo)`** before the call to **`bar()`** makes it global

Global variables III

- You don't need to do anything special to "read" (use the value of) a global variable within a function
  - ```
    foo = 77

    def main():
        print(foo) # prints 77
    ```
- However, if you **assign** to a variable with the same name as a global variable, that is a new ***local*** variable (the scope is the function)
  - ```
    foo = 77

    def main():
        foo = 1
        print(foo) # prints 1

    main() # calls main, which prints 1
    print(foo) # prints 77
    ```
- If you want to assign a global variable in a function, rather than to a local variable with the same name, you must declare it inside that function to be global

Nested functions

- In Python, you can declare a function inside another function (making it local to the outer function)
  - ```
    def average(x, y):
        def sum(x, y):
            return x + y
        return sum(x, y) / 2
    ```
    - The only reason you would do this is if the inner function is so specialized that it would have no use elsewhere
- In the inner function, you can get the values of variables in the enclosing function
  - ```
    def average(x, y):
        def sum(y):
            return x + y
        return sum(x) / 2
    ```
  - If this seems to you like poor coding, I agree!

Nonlocal variables

- In a nested function, you can get the values of variables in the enclosing function...
- ...but if you assign to a variable, it's local

- To be able to assign to a variable in the enclosing function, you need to declare it nonlocal

```
def average(x, y):
    sum = 0
    def add(x, y):
        nonlocal sum
        sum = x + y
    add(x, y) # this call changes sum
    return sum / 2
```

Default arguments

- You can supply default values for the last parameter or parameters in a function
  - ```
    def countdown(start=10):
        for i in range(start, 0, -1):
            print(i)
        print("Blast off!")
    ```
  - The range function is defined with a default step size of one:
    ```
    def range(start, stop, step=1):
        ...
    ```
- Since arguments are matched to parameters by position,
  - You can't put a parameter with a default value before one without a default value
  - You can't use the default value for one parameter and supply a value for a later parameter
    ```

Named arguments

- In a function call, you can use names instead of position to match up arguments to parameters
- ```
  def foo(a, b, c):
      print('a =', a, '  b =', b, '  c =', c)

  foo(b=2, a=1, c=3)
  ```
  prints
  ```
  a = 1   b = 2   c = 3
  ```
- You can even mix named and positional arguments in a function call, but the named ones must come first
  ```

Functions are values

- Functions are values, just like floats and booleans are values
- When you define a function, the name of the function is a variable whose value is that function
  - **Example:**

```
def double(x):
    return x + x

twice = double

print(twice(5)) # prints 10

def do_it(f, n):
    print(f(n))

print(do_it(double, 7)) # prints 14
```

Lambdas

- If a function (1) can be expressed as a single expression and (2) is only needed in a single place, then it can be written as a ***lambda expression***
  - **Syntax:** `lambda` *arg_1, ..., arg_N*`:` *expression*
  - **Example:**

```
import math

def do_and_show(fun, a, b):
    print(fun(a, b))

do_and_show(lambda x, y: math.sqrt(x * x + y * y), 3,
4)
```

- Lambda expressions are sometimes called ***anonymous functions***, because a typical use is as a parameter to some other function
  - However, you can certainly give a name to a lambda expression
  - **Example:**

```
hyp = lambda x, y: math.sqrt(x * x + y * y)

print(hyp(3, 4)) # prints 5.0
```

Extended example

- One assignment requires a number of functions such as **`is_prime`** and **`is_square`**
- My test code contains the following method:

```
def tell_about(self, fun, n, should_be):
    if fun(n) == should_be:
        return
    if should_be:
        self.fail(str(n) + " should be " + fun.__name__[3:])
    else:
        self.fail(str(n) + " should not be " +
fun.__name__[3:])
```

- And uses calls to this method such as the following:

```
self.tell_about(is_prime, 3, True)
self.tell_about(is_prime, 4, False)

self.tell_about(is_square, 3, False)
self.tell_about(is_square, 4, True)
```

Don’t rename built-in functions

- `str(x)` will return the string representation of *x*
  - `>>> str(123)`
    `'123'`
- `str` is just a variable name whose value is a function, so it is possible to give it a different value
  - `>>> str = 'Hello there'`
- Doing this is almost always a mistake, and should be avoided
  - `>>> str(123)`
    `Traceback (most recent call last):`
    `  File "<pyshell#105>", line 1, in <module>`
    `    str(123)`
    `TypeError: 'str' object is not callable`
- Don’t use names like `str`, `int`, `dict`, etc.

Generators

- When a function executes a **`return`** statement, that's it--the function is done
- But there's a way to leave a function, then come back to it at the same place you left off
  - Use **`yield`** instead of **`return`** in the function
  - Call the function *once* with any starting parameters you need, and save the result in a *variable*
  - Call **`next(`***`variable`***`)`** for the first and each successive result
- **Example:**

```
def powers(n):
    base = n
    yield n
    while True:
        n = base * n
        yield n

gen = powers(2)

for i in range(0, 11):
    print(next(gen))
```

The End

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

---C.A.R. Hoare, 1981 Turing Award lecture