## Contents

## 1 Functions

In the previous labs, we've used many different functions that were written by someone else. For example, `print()`, `input()`, `math.sin()`, and `random.random()`, all of which perform some specific set of instructions.

In math, we have functions like $f(x) = \sin x$, which is a function of one variable. It takes in one input, $x$, and has one output, $\sin x$. We can also have functions which take in more than one input, for example, $f(x, y) = 2xy$.

In Python, or more generally programming, we can define our own functions. We can have as many or as few inputs as we want, and as many or as few inputs as we want. The table below lists a few examples. `print()` is somewhat special, being able to take 0 inputs, or (infinitely) many inputs.

| Function | Inputs | Outputs | Description |
|---|---|---|---|
| `print()` | - | 0 | Displays what is passed in |
| `input()` | 0 or 1 | 1 | Waits for and returns the user's input |
| `math.sin()` | 1 | 1 | Calculates the sine of the input |
| `random.random()` | 0 | 1 | Returns a random value in the interval $[0, 1)$ |

One thing to note is that you never once needed to know *how* the functions perform what it does, only that it will perform what you asked it to do, akin to a black box.

### 1.1 Calling/invoking a function

Over the past labs, we've implicitly called/invoked functions without explaining its syntax. A function is run when the *function name* is followed by a *pair of parentheses* containing the function's parameters. For example, calling `math.sqrt(4)` will invoke the named function, with a parameter value of 4.

The number of parameters must match what the function can accept, and when the function takes multiple parameters, the order of the parameters also matters.

### 1.2 Black box behaviour

To understand what is meant by 'a function being a black box', we will focus on one function: `math.sqrt()`, which calculates the square root of its input and returns it. How does it do so?

You may think of using Newton's method (which works), but it could use any other method to compute the square root. In fact, Python's `math.sqrt()` depends on the use of another programming language, C, and uses it to compute the square roots. The specific method used depends on the machine.

This 'hiding' of implementation **abstracts** the performance of an operation.

## 2 Defining a function (`def`)

We can define our own functions that we can use elsewhere in our programs. The syntax to define a function is simple, and uses the `def` keyword.

```python
def function_name(param1, param2):
  # code to run - here written with `pass` as a placeholder.
  # `pass` does nothing, it exists only to make the code syntactically
    valid.
  # without it, there would be a syntax error.
  # to see this, just erase the `pass` below.
  pass
```

The first line with `def` contains the **function signature**, which includes its *name* and the *parameters/arguments* that it takes and their order. These functions may optionally include a **`return`** statement, which Python interprets as the **output** of the function.

For example, the below describes a function that takes a single input and *returns* its value after multiplying by two.

```python
def f(value):
  return 2 * value

print(f(4))        # calls f with value = 4, returns 8
print(f("abc"))    # calls f with value = "abc", returns "abcabc"
f(1, 2)            # error, f() expects only a single input
print(value)       # error, variable `value` is not defined
```

The parameters that a function takes are used **only within the function itself**. For example, attempting to print `value` will cause an error as `value` is not defined outside the function.

### 2.1 Return values (`return`)

Above, we defined a function which outputs a value, using the **`return`** keyword. The **`return`** keyword can only be used *in a function definition*, and indicates that the expression that follows is the output/return value of the function. This means that a function could, for example, return the output of another function.

The below block, for example, computes $f(g(2))$, where $f(x) = x^2$ and $g(x) = 2x$.

```python
def f(x):
  return g(x)**2
def g(x):
  return 2*x

print(f(2))     # f(2) -> g(2)**2 -> (2*2)**2 -> 4**2 -> 16
```

### 2.2 Multiple return values

You can also return *multiple values* separated by a comma, which will be returned as a tuple. You can access each value by indexing the output tuple.

```
def f(x):
  return x, x+1

print(f(3))     # (3, 4)
print(f(3)[1])  # 4
```

### 2.3 Multiple `return` statements

We can use multiple `return` statements in a function, but only the first one that Python encounters will ever be executed.

```
def f(n):
  if n % 2 == 1:
    return "odd"
  else:
    return "even"

print(f(1), f(2))
```

Another thing to note is that the function execution ends when Python encounters the `return` statement, and all other lines after the `return` statement will never be run.

```
def f():
  return 12     # function terminates here
  print("I will never run.")
  return "this will never be reached"

print(f())      # 12
```

### 2.4 No `return` statements

Functions may also have no output value or no `return` statements. We may call such functions *procedures*. When a function has no return statement, it implicitly returns a `None` value. The `None` value is a special type, basically saying 'there is nothing here'. `print()` and `list.sort()` are examples of procedures.

```
outs = print()  # get the return value of print()
print(outs)     # and print it. None

def f():
  pass          # function does nothing, has no return statement
print(f())      # None
```

## 3 Function guidelines

Generally, when we define a function, we want it to be **reusable**. While we could defined a function to be long and complex, length and complexity affect readability, understanding, and reusability. A complex function with many steps used only once might as well not be a function. You may also want to consider a few questions:

- What is the function supposed to do? What should it return, if anything?
- What parameters does it need to take to do it?
- Can it perform its operation *with only the inputs it is passed*?

## 4 Ground-up example

Consider a script that counts the number of vowels in a string. We have done this before:

```
string = input("enter a string: ")

count = 0
for char in string:
  if char in 'aeiou':
    count += 1

print("there are", count, "vowels.")
```

To count the number of vowels in *another* string elsewhere, we have to repeat this computation, i.e. copy 4 lines. Instead, consider this as a function that has a single input, and returns the number of vowels in it:

```
def count_vowels(string):
  count = 0
  for char in string:
    if char in 'aeiou':
      count += 1

  return count

string = input("enter a string: ")
print("there are", count_vowels(string), "vowels.")

# do stuff... 100 lines later

string = input("enter a string: ")
print("there are", count_vowels(string), "vowels.")
```

The purpose and output of the function is clear and descriptive. All we need to know about the function is that it takes a string as an input and returns the number of vowels in it, and then we can freely use it, in any context that we need. In the context of a script that has only a single purpose, this is not as useful, but when the program has multiple functions or hundreds of lines of code, this abstraction is useful.