Introduction to Computer Programming

Week 4.1: Functions

Functions

A **function** is a collection of operations that have been given a name

- These operations can involve variable assignment, mathematical operations, loops, if-else statements, etc

Functions are the building blocks of Python programs

- They can be thought of as mini-programs that carry out specific tasks (e.g. square a number)

Python comes with a number of built-in functions

- `max(L)` computes the maximum entry in a list of numbers called *L*

We can also write our own functions in Python

- We could write a function `is_prime(n)` that determines whether an integer n is prime

What are the benefits of writing our own functions?

- It reduces the need to copy and paste code that does the same operation, making code more reusable
- It makes programs easy to maintain, more readable, and easier to understand

An analogy with mathematical functions

In maths, we often work with functions of the form $y = f(x)$, where

- x is an input (e.g. a number)
- f is the function which carries out operations on x, such as x^2 or $\sin(x)$
- y is the output, that is, the result of carrying out the operations on x

Functions in Python work in the same way, but are much more powerful:

- Python functions take inputs which can be ints, floats, lists, dicts, etc!
- They can carry out multiple operations that aren't necessarily mathematical
- And they may produce no outputs, one output, or many outputs

Some programming terminology

Inputs to functions are called **arguments**

When we run or execute a function, we say that we are **calling** the function

Defining our own functions

Every function definition is of the form

```
def name_of_function(arg1, arg2, ...):
    # indented block of code
```

The key ingredients are:

- `def` is a keyword that tells Python we are defining a function
- *name_of_function* is the name of the function
- *arg1*, *arg2*, ... are comma-separated arguments (inputs) that we provide to the function
- Round brackets that surround the arguments, followed by a colon
- A block of indented code

Example: Let's write a function that doubles the value of a number x and prints the result:

In [ ]:
```

```

Once a function is defined, it can be used. For example:

In [ ]:
```

```

In [ ]:
```

```

Example: Write a function without any arguments that prints 'Hello'

In [ ]:
```

```

Example: Write a function that prints all of the entries in a list that is provided as an argument

In [ ]:
```

```

Creating output using `return`

We often want to save the result of a function by assigning it to a new variable.

This is possible using the `return` keyword

```
def name_of_function(arg1, arg2, ...):
    # indented block of code

    return val_to_output
```

When `name_of_function` is called, it **returns** (or outputs) the value of *val_to_output*, which can then be assigned to a new variable using

```
saved_output = name_of_function(arg1, arg2, ...)
```

Example: Write a function that doubles the value of a number x and returns the results

In [ ]:
```

```

Returning multiple outputs

It is possible to return multiple values by creating a tuple out of them

Example: Write a function called `sum_prod` that returns the sum and product of two numbers

In [ ]:
```

```

There are two ways we can run this function and save the output:

In [1]:
```
def sum_prod(x, y):
    return (x + y, x * y)

# save the output as a tuple


# save the output as two numbers
```

More about the `return` keyword

The `return` keyword is optional, but it can play two important roles in functions:

1. It is used to define the output of a function, if there is any
2. It is used to exit a function prematurely (similar to `break` in loops)

The second point means the `return` statement is useful in controlling the flow of functions that involve `if` statements

Consider the following function:

In [8]:
```
def my_function(x):
    print('x equals', x)
    return
    print('2x equals', 2 * x)
```

When the function is called, it proceeds through each statement until the `return` keyword is encountered, at which point the function terminates, and any values that follow `return` are returned as outputs

In [9]:
```
my_function(2)
```

```
x equals 2
```

Since there is nothing that follows `return` in this example, the function does not output anything

Example: Write a function that determines whether an integer is even. If so, the function returns the boolean True. Otherwise, the function returns the boolean False.

In [ ]:
```

```

Good programming practice - docstrings

Python programs often involve many user-defined functions, and it can be difficult to remember what they do and how they should be used.

A **docstring** is text in triple quotation marks placed below the name of the function that explains what it does.

In [10]:
```
def square(x):
    """
    Computes the square of a real number x and returns its value
    """
    return x * x
```

Python's `help` function can be used to print the docstring:

In [11]:
```
help(square)
```

```
Help on function square in module __main__:

square(x)
    Computes the square of a real number x and returns its value
```

Summary

- A function is a group of code that has been given a name
- They can take input and produce output
- Functions are defined using the `def` keyword
- Output is producing using the `return` keyword
- Docstrings are helpful for explaining what a function does and how to use it