Week 6: Functions and methods

Intended learning outcomes

By the end of this class, you will be able to:

- define and call Python functions;
- use functions to make the structure of a program simpler;
- use in-built Python functions and methods.

1 Functions

Roughly speaking, a Python function is like a program within a program. Type in the following program:

```
1  def three_greetings():
2      """Say hello three times."""
3      for i in range(3):
4          print("(", i, ") Hello!", sep="")
5
6
7  print("\nBefore we say hello.")
8  three_greetings()
9  print("In the middle.")
10 three_greetings()
```

Let's analyse what this program does.

Line 1 uses the command **def** to **define a function** called `three_greetings()` consisting of the following indented code (lines 2–4). It does not execute this code yet – it will just define the function to be this mini-program.

Line 2 is a **docstring** (short for "document string") and is basically a comment explaining what the function does. Docstrings use three double quotation marks (`"""`) at each end. It is good practice to write clear docstrings for code, so that anyone using the code know what it does.

Lines 3 and 4 form the main body of the function and comprise a loop that prints "Hello!" three times. Note that this code is not executed yet. Line 4 is double-indented as it is inside the loop inside the function.

Lines 5 and 6 are blank to separate the function from the main program. (This is a style convention.)

Line 7 prints "Before we say hello."

Line 8 *calls* the function `three_greetings()`, which means that it executes the code contained in the function. So it prints "Hello!" three times.

Line 9 prints "In the middle."

Line 10 calls the function `three_greetings()` again, so we get "Hello!" printed another three times.

When calling a function, you can also *pass arguments* to it. Modify lines 1, 2, 3, 8 and 10 to obtain the following program:

```
1  def many_greetings(n):
2      """Say hello n times."""
3      for i in range(n):
4          print("(", i, ") Hello!", sep="")
5
6
7  print("\nBefore we say hello.")
8  many_greetings(2)
9  print("In the middle.")
10 many_greetings(5)
```

When you run this, you see that the first time `many_greetings()` is called the message gets printed twice, and the second time it is called the message gets printed five times. The key here is the `n` in the `def many_greetings(n):` this is called a **parameter** or an **argument**. When you call the function, you have to specify the value of the parameter, e.g. with `many_greetings(2)`, it would assign `n = 2` before executing the function. This is like with a mathematical function: $f(2)$ means a function $f(x)$ evaluated at $x = 2$.

Python functions can take more than one parameter, as illustrated by the following program. Guess what will happen when it is run. Then type it in and run it.

```
def print_sum(a, b):
    """Print the sum of the two numbers."""
    print(a, "+", b, "=", a + b)


print_sum(3, 4)
print_sum(7, 2)
```

Exercise 6.1. Write a program that defines a function `hamming_distance()`, which takes two strings as parameters and prints out their Hamming distance (you can copy the code from the Week 3 solutions on the website if you like). The program should use this function to print out the Hamming distance between 'great' and 'groan' and between 'alpha' and 'alpha'.

So far our Python functions such as `many_greetings()` have not behaved like mathematical functions, because they did not give you a specific value but instead did something. This behaviour can be mimicked by **returning a value**. You use the `return` command to do this. Compare the following program with the previous one:

```
def add(a, b):
    """Calculate the sum of two numbers."""
    return a + b


a = add(3, 4)
print("3 + 4 =", a)
print("7 + 2 =", add(7, 2))
```

Here the function doesn't print anything, but uses the `return` command instead. This means that for instance `a = add(3, 4)` will set `a` to be whatever value (in this case `7`) is returned by the `return` command. The `add()` function now looks a lot more like a mathematical function.

Exercise 6.2. Write a program that defines a function `square()`, which takes a single number as a parameter and returns the square of that number. The program should use this function to print out 3^2 and $(-27)^2$.

Functions can be used to make the structure of a program simpler. Suppose you want to write a program to get the user to enter a positive integer and then calculate its factorial. You can farm off both of these processes to functions to make it clearer what the program is actually doing. Here is an example. It might be overkill in such a simple program as this, but it can definitely make life easier as soon as your programs start getting more complex.

```
def factorial(n):
    """Calculate a factorial"""
    product = 1
    for i in range(1,n+1):
        product = product * i
    return product
```

```
7
8
9   def enter_positive():
10      """Get the user to enter a positive integer."""
11      n = -1
12      while n <= 0:
13          n = int(input("Enter a positive integer: "))
14      return n
15
16
17  number = enter_positive()
18  print(number, "! =", factorial(number))
```

2 Built-in functions and methods

We have already met several built-in Python functions, such as `int()`, `str()`, `max()` and `input()`. A full list can be found by googling. Next week we will see how to *import* more functions.

Python *methods* are very similar to functions, but have a different syntax. (Defining methods yourself is beyond the scope of this course; you need to learn about *object-oriented programming*.) We saw some methods for lists last week; here is an example.

```
>>> menu = ["spam", "egg", "beans"]
>>> i = menu.index("egg")
>>> print(i)
```

If `index()` were a function it would be written as `index(menu, "egg")`, it is telling us the index of the first occurrence of `"egg"` in the list `menu`. Methods always apply to a specific *type*. Here the `index` method is applied to an object of type `list` so we refer to it as the `list.index()` method, but when we use it we always replace the word `list` by the name of the list we want to apply it to, so we wrote `menu.index("egg")` in the code above.

We will look at some useful string methods here: `str.upper()`, `str.lower()` and `str.split()`. The methods `str.upper()` and `str.lower()` return versions of the string that are all lower and all upper case, respectively.

```
>>> greeting = "Hello!  How are you?"
>>> print(greeting.lower())
>>> print("I am well.".upper())
```

The method `str.split()` will return a *list* of the words in the string.

```
>>> print(greeting.split())
>>> print("1 3 5.5".split())
```

As you can see in the second example, what it actually does is split the string into a list of the parts that are separated by spaces.

Exercise 6.3. Write a function `string_to_vector()` which takes a string of numbers such as `"1 2 4 5.5 6.786"` as an argument and returns a list of *floats* such as `[1.0, 2.0, 4.0, 5.5, 6.786]`.

Homework

The mini-project will be launched this week, so there is no assessed homework for the next lab.

1. Finish the sheet.

2. (*Quick review.*) Figure out what the output of the following two programs will be. Then type them in and run them. (I have deliberately made the variable names meaningless to make them harder to understand.)

(i)

```
def function_1(s):
    """First mystery function."""
    for i in range(len(s)):
        if i % 3 == 0:
            print(s[i].upper(), end="")
        else:
            print(s[i].lower(), end="")
    print("!")

function_1("elephantine")
function_1("ENORMOUS")
```

(ii)

```
def function_2(s):
    """Second mystery function."""
    v = 0
    for i in range(len(s)):
        if s[i] in "aeiou":
            v = v + 1
    return v

print(function_2("Spain") * function_2("oceanic"))
```

3. (*Optional.*) Write a program that defines the `factorial()` function as above and a new function, `binomial()`, which takes two parameters n and k and returns the binomial coefficient $\binom{n}{k}$. The program should ask the user for a number and prints out that many rows of Pascal's triangle. So if '5' were entered, you would get something like the following:

```
                1
            1       1
        1       2       1
    1       3       3       1
1       4       6       4       1
```

[Hints: First, there was a triangle drawing program in last week's exercises that might be useful. Second, I imagine one problem will be figuring how many tab characters to insert. You can 'multiply' strings to get the same string repeated, so e.g. `"\t" * 7` will give `"\t\t\t\t\t\t\t"`.]

If you are feeling adventurous and want to see a pretty pattern emerging from Pascal's triangle then modify your program so that it prints an asterisk "*" if the entry in Pascal's triangle is odd and a space " " if the entry is even. So for the first five rows as above you would get

```
    *
   * *
  *   *
 * * * *
*       *
```

Your program should print the first 32 rows.