# M30299 – Programming

## Worksheet 1: Programming Essentials

This worksheet gets you started with the Python language and the programming environment **Thonny**. Work through it carefully at your own pace, making sure you understand each part before moving on to the next.

Read the worksheet and attempt as many of the exercises as you can. In the next practical session, we will give you feedback on your efforts and help you with any of the exercises that you could not complete.

If you are new to programming, you are encouraged to book a one-to-one session with the Academic Tutors (Simon Jones or Eleni Noussi) using their Moodle page. We additionally have a Discord channel for this module, follow the instructions on these slides to join.

Please start this first worksheet in your first practical session (not before). From week 2, worksheets will be released on Moodle early in the week for you to start as soon as you have completed the previous one.

## Using Python and an IDE

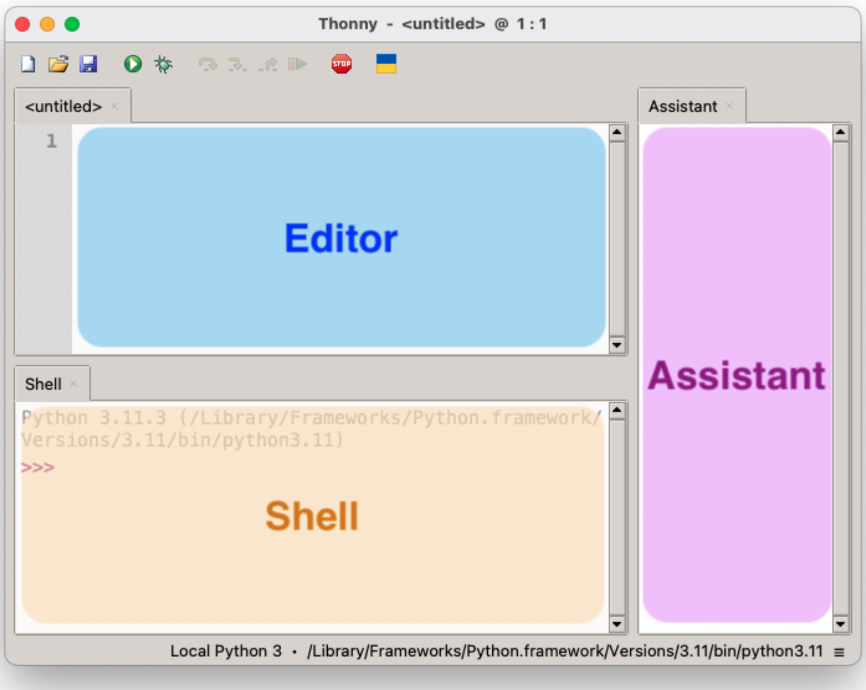
For the first part of the module, we use the programming language Python, and we recommend (especially for beginners) an easy-to-use Integrated Development Environment (IDE) called Thonny. We have provided a guide on how to set up Python and Thonny (and other IDEs) in this document.

If you are more familiar with programming, you may choose to use a more advanced IDE such as PyCharm. In that case, follow our document's installation and user guide to familiarise yourself with the IDE before proceeding.

All the instructions on this worksheet assume you are using Thonny.

## Hello Thonny!

When you start Thonny a window similar to that shown below will appear.



```
Python 3.11.3 (/Library/Frameworks/Python.framework/
Versions/3.11/bin/python3.11)
>>>
```

Local Python 3 • /Library/Frameworks/Python.framework/Versions/3.11/bin/python3.11

The area on the top left of the window (blue) is the **editor**, which we'll use later in this worksheet to write longer programs.

At the bottom left (orange), we have the Python **shell**. The shell allows you to interact with Python directly. Each line you type into the shell is interpreted (executed) immediately, with any results displayed. This provides a good way to learn the basics of the Python language and to test out new concepts and ideas.

On the right-hand side (purple), you have the **assistant** window. It will analyse your code as you type and display messages that explain what your code does, warn you about potential problems, or give you tips on how to improve your code. Feel free to minimise or close it.

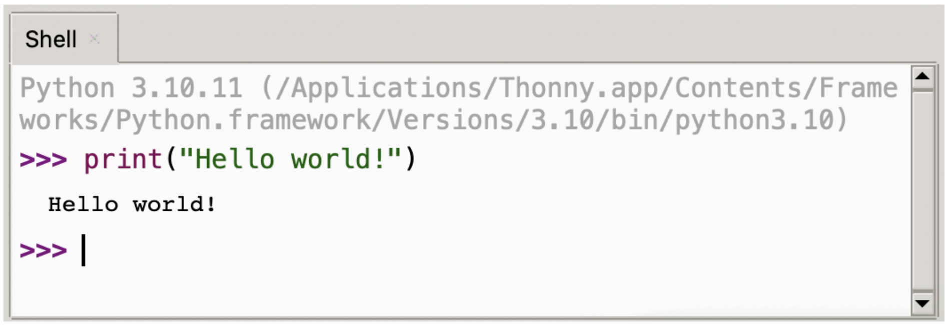
## Running statements in the shell

Let's start by experimenting in the shell (right-hand side) before writing Python scripts. Click on the shell and type the code in there. Then press enter to run it:

Shell

```
print("Hello world!")
```

You should get the outcome shown here:



Shell

```
Python 3.10.11 (/Applications/Thonny.app/Contents/Frame
works/Python.framework/Versions/3.10/bin/python3.10)
>>> print("Hello world!")
  Hello world!
>>> |
```

We are using the `print` function to display a text message “Hello world”. All we had to do was to place a message written between the quotation marks, inside the parentheses. The Python interpreter executes this line of code and outputs the message in the next line.

Each line of Python code is a **statement**, and this statement is known as a **print statement** since it uses the `print` function. This statement is a complete Python program!

The `print` function can display numbers as well as text. Enter the following in the shell:

Shell

```
print(7)
```

You can place arithmetic **expressions** inside the `print` function, and it displays the simplified value of the expression.

We will discuss operators later, but for now, remember that `/` (forward slash) is division and `*` (the star symbol) is multiplication. Take the following expression:

Shell

```
print(5 + 4 * 3)
```

How do you think the expression above is evaluated? A or B?

A. $(5 + 4) \times 3 = 9 \times 3 = 27$
B. $5 + (4 \times 3) = 5 + 12 = 17$

# Creating and updating variables

Variables are basic elements of a program in any programming language. You can think of a variable as a name with an arrow pointing to a value. Note that the variable's value can change during the execution of a program.

Let us experiment with some variables. To create a variable, we use an **assignment statement**. Type the following in the shell and press enter (you will not see any output):

Shell

```
x = 7
```

All assignment statements have this basic form:

- On the left of the = is a variable name (here x)
- On the right-hand side of = is an expression (here 7).

This assignment statement means: “Define a variable called x and assign it the value 7”. To check the value of your new variable, type:

Shell

```
print(x)
```

Or simply:

Shell

```
x
```

Let’s create another variable called y, using an assignment statement.

Shell

```
y = x + 1
```

The value assigned to y is equal to the value inside x plus 1. After entering this assignment, display the value of y:

Shell

```
y
```

Values of existing variables can be changed using assignment statements. Run the line below to change the value of the already defined variable `y` from 8 to 10:

Shell

```
y = 10
```

Remember the following about variable names:

- **Use unique names:** No two variables should have the same name.
- **Use meaningful names:** For example, use `first_name` for a variable storing someone's first name.
- **Variable names should start with alphabetical characters:** Don't start variable names with a number.
- **Use the `snake_case` naming convention:** words written in lowercase separated with underscores).

Let's discuss a slightly more complicated example. Below, we create a variable called `my_age` and assign it the value 21. Run the line below and print `my_age` afterwards:

Shell

```
my_age = 21
```

After displaying `my_age`, we are going to increment its value by 1. Run the following line, which takes the current value of `my_age` (which is 21) and adds 1 to it. The result, 22, is then assigned back to the `my_age` variable. Display `my_age` to verify this change.

Shell

```
my_age = my_age + 1
```

## Understanding errors

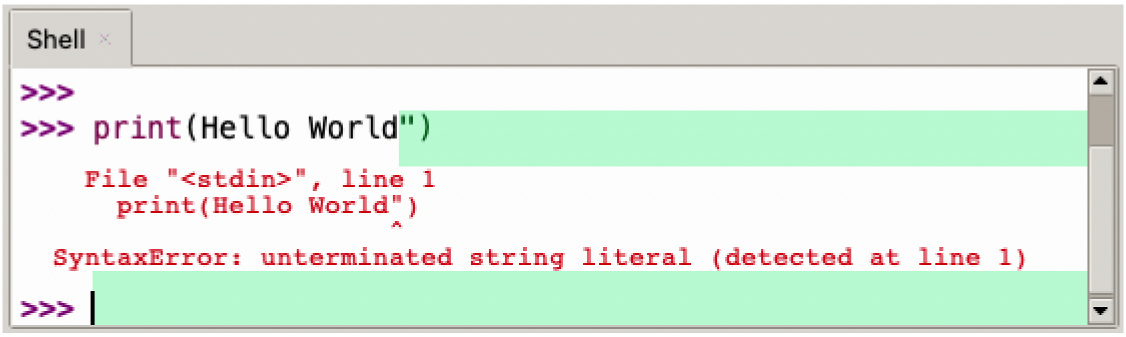
Like all programming languages, Python has strict rules and the interpreter produces error messages if you break them. If you are new to Python, you may come across **syntax errors**. These are mistakes in the structure of your code, and they are often caused by typos.

Syntax errors are usually quite easy to fix, and the error messages point out which part of your code needs to be fixed. For example, try the following print statement with a syntax error:

Shell

```
print(Hello World")
```

This should output the following error message in the shell:



```
>>>
>>> print(Hello World")
    File "<stdin>", line 1
      print(Hello World")
                       ^
  SyntaxError: unterminated string literal (detected at line 1)
>>> 
```

The error comes from the fact that the quotation mark at the end of the message is not matched with an opening quote.

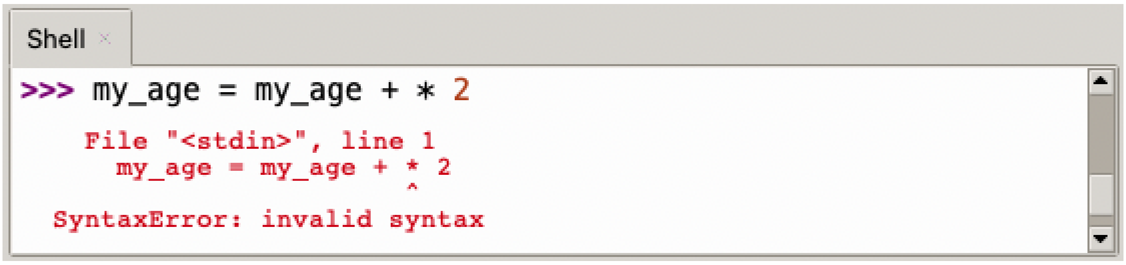
The shell has the history and stores the previously executed lines of code. Press the up-arrow key (🔼) on your keyboard to display the previously executed statement. You can then use the left arrow key on your keyboard (⬅️) to go back to the part that caused the error and fix it before rerunning it.

Next, try the following assignment statement that also has a syntax error:

Shell

```
my_age = my_age + * 2
```

This should output the following error message in the shell.



```
>>> my_age = my_age + * 2
    File "<stdin>", line 1
      my_age = my_age + * 2
                        ^
  SyntaxError: invalid syntax
```

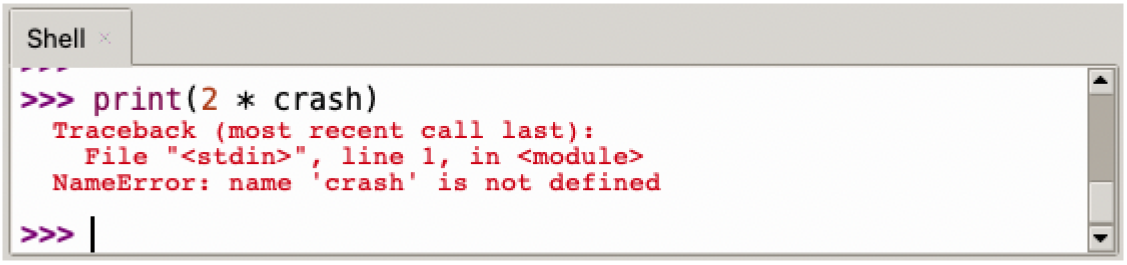
The error is thrown because the addition operator (+) expects an operand on the right-hand side. Instead, the interpreter found the multiplication operator (*).

Code without syntax errors may still have **semantic errors**. This means your code is written correctly, but the meaning of your code is incorrect. Try for example:

Shell

```
print(2 * crash)
```

Here, we call the `print` function to print two times the value stored in a variable called `crash`. This should output the following error message in the shell:



```
>>> print(2 * crash)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'crash' is not defined

>>> 
```

Read the explanation in the assistant window (right-hand side) if it is open. To fix this error, define the variable `crash` and give it a value (for example, 5) and re-run the above print statement.

Note that error messages can be several lines long. Always **look at the last line** of error messages: they may give you a clue as to what is wrong with your code.

## Writing longer programs using the editor

Whilst the shell is great for experimenting, it is only good for very short programs. We cannot save the programs that we write in the shell. A program written in the **editor** can be saved to a file and reused multiple times. In the editor, type your Hello World program:

Editor

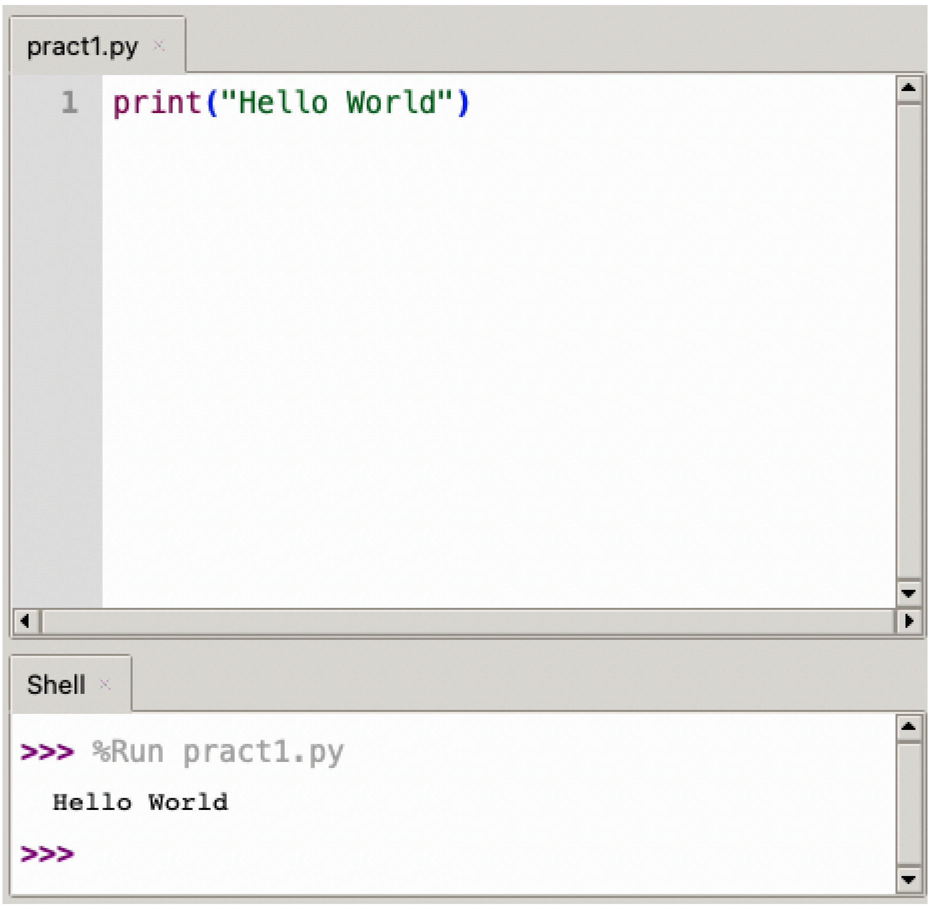
```
print("Hello World")
```

Now save your file using the save icon (💾) or use `Ctrl+S` (or `Cmd+S` on macOS).

Save your file under the name `pract1.py`. Make sure that your filename ends with **`.py`**. This is the file extension for Python scripts.

Remember to save your files in a folder called `programming`, in a location that is either accessible from other computers or synchronised with the cloud. For example, N drive on university machines, iCloud on macOS, OneDrive on Windows, or a folder synced with Google Drive for desktop).

Once you have saved pract1.py, click the run button (▶️) to run your program.  You'll get the output shown below in the shell:



```
print("Hello World")
```

```
>>> %Run pract1.py
  Hello World
>>>
```

Now change pract1.py so that it looks exactly as shown below. Pay close attention to the spaces at the start of the second line. You may have to use the tab key on your keyboard to **indent** it:

Editor

```
def say_hello():
    print("Hello World")
```

Save pract1.py and run it again using the run button. This **will not output anything**. Instead, the program defines a **function**. A function is a piece of code that is defined with the def keyword. Functions have a name and can be used multiple times.

In this case, our function is called say_hello. Function names, like variable names, need to be unique and **start with lowercase letters**. You should use snake case for functions.

The statements inside a function must be indented. They describe what the function does. In our case, say_hello prints Hello World.

Once you have run `pract1.py`, type the following in the shell and press enter:

Shell

```
say_hello()
```

This `say_hello()` statement is a function call. It uses (or "calls") the `say_hello` function. This should give the same output as before.

Let's add another function definition in our file. Back in the editor, write the definition of the `say_bye` function (highlighted below in green). `pract1.py` should look like our code below:

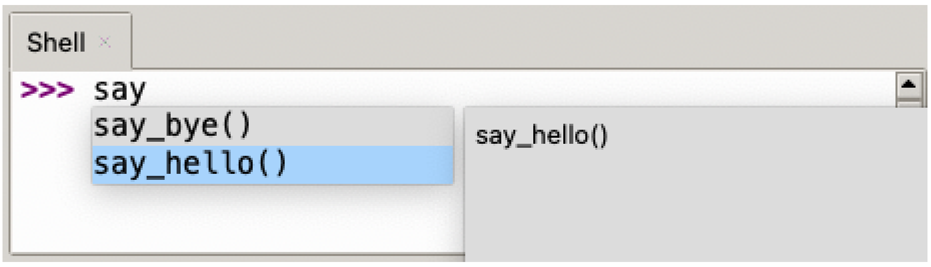
Editor

```
def say_hello():
    print("Hello World")

def say_bye():
    print("Goodbye Mars")
```

Pay attention to how the print statements are inside the functions because they are indented. Save the file again after your changes and run it using the green run button.

The shell has an auto-completion feature, which you can use by pressing the tab key. Simply type the first few letters of the function or variable you want to use in the shell, then use the tab key to autocomplete the word. For example, in the shell, type "say". Then press the tab key on the keyboard. It predicts that you are either trying to run the `say_hello` or the `say_bye` function. Now type the letter 'b', so you have "say_b" in the shell. Try the tab button again. Does it change to `say_bye()`?



```
>>> say
    say_bye()
    say_hello()
```

```
say_hello()
```

Call the `say_bye` function by entering the following in the shell. Remember that to call a function, you need to write the name of the function, and then add the `()` at the end.

Shell

```
say_bye()
```

Does it output the "Goodbye Mars" message as expected?

## Interacting with the user

Our programs, so far, have not communicated with the user in any way. It's time to write a more interactive program which asks the user to enter a value.

Add the `kilos_to_ounces` function (highlighted below) at the end of `pract1.py` in the editor. Save this file and run it again:

Editor

```
def say_hello():
    print("Hello World")

def say_bye():
    print("Goodbye Mars")

def kilos_to_ounces():
    kilos = float(input("Enter a weight in kilograms: "))
    ounces = 35.274 * kilos
    print("The weight in ounces is", ounces)
```

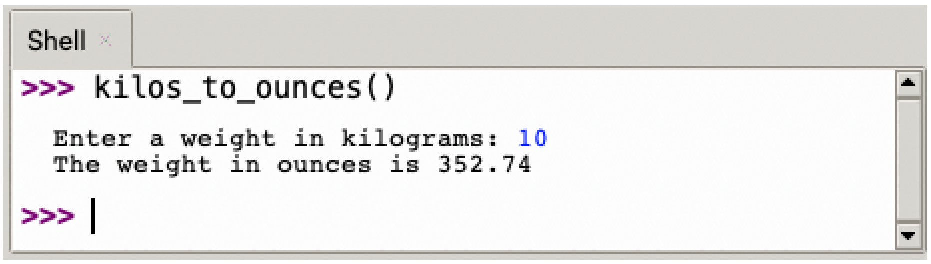
Don't worry if the code seems complicated. You will soon be familiar with everything inside `kilos_to_ounces`. Just call this function by entering the following in the shell:

Shell

```
kilos_to_ounces()
```

`kilos_to_ounces` asks the user to supply some values (using the `input` function). The program converts this value to a decimal point number (using the `float` function). It then saves this decimal number inside the variable `kilos`. This decimal number will then be converted to ounces and saved in the variable `ounces`. In the end, `ounces` is printed to the user.

Run `kilos_to_ounces` and enter 10. Make sure you have the same output as this:



```
>>> kilos_to_ounces()

  Enter a weight in kilograms: 10
  The weight in ounces is 352.74

>>> 
```

Let's make the `kilos_to_ounces` function a little clearer for people who read our code. We can do this by adding **comments** at the top of our functions.

Add the highlighted lines to `pract1.py` and save it:

Editor

```
def say_hello():
    print("Hello World")

def say_bye():
    print("Goodbye Mars")

# A simple kilograms to ounces conversion program
# It asks for a weight in kilograms (for example 10)
# and converts it to ounces (352.74)
def kilos_to_ounces():
    kilos = float(input("Enter a weight in kilograms: "))
    ounces = 35.274 * kilos
    print("The weight in ounces is", ounces)
```

Python will ignore any text that appears to the right of a `#` (hashtag symbol). So, this is how we document a program to make it more readable to yourself and other programmers.

There are other types of comments which we will cover later. But for now, execute the program again to see that the comments haven't changed what the program does.

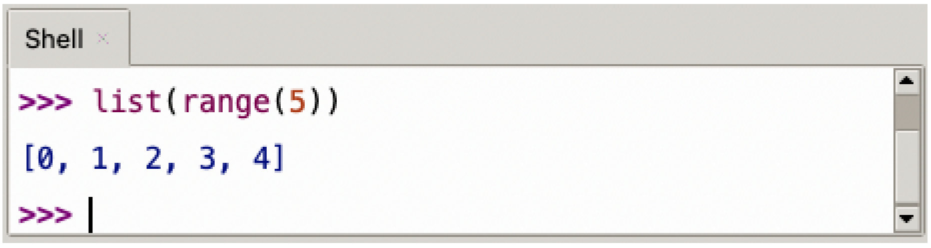
## Looking ahead: lists and for loops

Let's take a sneak preview of some of the programming concepts that we'll study in depth later in the module. First, using the shell, enter the following expressions:

Shell

```
list(range(5))
```

This should produce a list of numbers, 0 to 4, as shown below:



```
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> 
```

Let's use this range of numbers in a function that prints the values in the list displayed above. Add the `count` function highlighted below to `pract1.py`:

Editor

```
def say_hello():
    print("Hello World")
```

...

```
def kilos_to_ounces():
    kilos = float(input("Enter a weight in kilograms: "))
    ounces = 35.274 * kilos
    print("The weight in ounces is", ounces)

def count():
    for i in range(5):
        print(i)
```

The statement inside the `count` function is known as a **for loop**. For loops are used in programming to execute a piece of code repeatedly. Before we describe how it works, call the `count` function:

Shell

```
count()
```

In the loop, `i` is a variable, created and used by the loop itself. This variable is given each of the values in the list of numbers 0 to 4. The print statement inside the for loop (indented twice as a result) displays the current value of `i` at each iteration.

Let's experiment further with for loops. Update the `count` function as highlighted below:

Editor

```
def count():
    for number in range(10):
        print("Number is now: ", number)
```

Save `pract1.py` and call the `count` function again. What does it display now?

## Programming exercises

Each practical worksheet will include a set of programming exercises, and attempting these exercises will provide you with a foundation for some module assessments in later weeks. It is important therefore that you keep up-to-date with the practical exercises. As already mentioned, work at your own pace — you are not expected to complete the worksheet during a single practical session. You should, however, attempt to complete as much as you can in your own time before the following week's practical.

Add all your solutions to your `pract1.py` file, and show them to us for feedback in your next practical class. Remember to test your code by running your file and calling the function from the shell before moving on to the next exercise. Sometimes you will want to call the function several times to provide different input values.

1. Write a function called `say_name` that displays your name.
2. Write a `say_hello_2` function that uses two print statements to display the text below:

   ```
   hello
   world
   ```

3. Write a function called `dollars_to_pounds` which converts an amount in dollars entered by the user to a corresponding amount in pounds. Assume that the exchange rate is 1.35 dollars to the pound. (Be sure to test your solution carefully!)
4. Write a `sum_and_difference` function that asks the user to enter two numbers (using two separate questions), and outputs their sum and their difference (the first number minus the second number).
5. Write a `change_counter` function. This should ask the user how many 1p, 2p and 5p coins they have (using separate questions), and then display the total amount of money in pence. Remember that variable names cannot begin with a number (instead of `2pence` use `two_pence`).
6. Write a `ten_hellos` function that uses a loop to display "Hello World" ten times (on separate lines).
7. Write a `zoom_zoom` function that asks the user for a number and outputs "zoom" labelled from 1 to that number; for example, if the user enters 4, your function should output:

   ```
   zoom 1
   zoom 2
   zoom 3
   zoom 4
   ```

8. Write a `count_to` function that asks the user for a number and then counts from 1 to that number.
9. Based on your solution to the previous question, write a function called `count_from_to` that asks the user for two numbers. The first number is the start of the count and the second one is where the count ends.
10. [harder] Write a function `weights_table` that prints a table of kilogram weights (with values 10, 20, 30, ..., 100) and their ounce equivalents. (Don't worry too much about formatting the table neatly.)
11. [harder] Write a `future_value` function that uses a for loop to calculate the future value of an investment amount, assuming an annual interest rate of 3.5%. The function should ask the user for the initial amount and the number of years that it is to be invested, and should output the final value of the investment using compound interest, with the interest compounded every year. (There is a formula to work out compound interest, but you should not use this; instead, use a for loop to go through each year adding on the interest gained during the year.)