



PROGRAMMING 0 CHALLENGE 1

Dr Richard Holden

Table of Contents

Summary	2
Learning Outcomes:	2
Developing your coding skills: the power of independent thinking	2
Exercise 1: calling Python without arguments	3
Exercise 2: Explore the 'python' command without args (short for arguments).....	4
Exercise 3: Escape sequence experimentation.....	4
Exercise 4: moving into a script (first_string_commands.py)	5
Using 'main block' and 'what is the main block?'	5
Move previous code to a script following main block convention	6
Exercise 5: einstein.py.....	7
Exercise 6: DRY acronym.....	7
Exercise 7: word counting.....	7
Exercise 8: Einstein's equation.	8
Exercise 9: Solution.....	8

Summary

This document is structured into exercises that gradually introduce key programming concepts, focusing on Python's basic features and syntax.

Learning Outcomes:

- Develop independent problem-solving skills by attempting coding challenges on your own.
- Gain familiarity with running Python code in different environments (REPL and scripts).
- Understand and use escape sequences for string formatting in Python.
- Learn to create and execute Python scripts from the command line.
- Understand the DRY principle.
- Practice string manipulation techniques for word counting and text analysis.
- Perform simple calculations and embed numerical results in strings.

Developing your coding skills: the power of independent thinking

N.B. I will provide solutions to all lab tasks and other challenges. However, I hope you are motivated to set out on *your own* coding journey. So, to begin with I wanted to share some insights on how you can maximize your learning experience and truly become a proficient programmer. In your programming journey, it's crucial to strike a balance between independent problem-solving and seeking guidance from others. While it may be tempting to look for solutions immediately, I want to emphasize the value of trying to solve coding challenges independently.

Here's why this is an excellent idea:

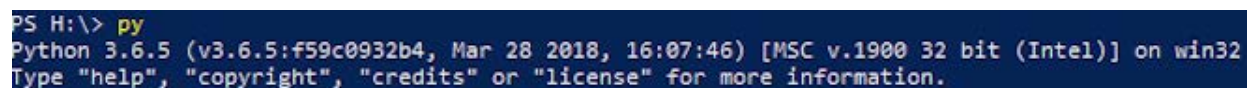
1. *Building problem-solving skills*: When you tackle a problem on your own, you're **actively engaging** your problem-solving skills. This process encourages critical thinking and creativity, which are essential traits of a successful programmer.
2. *Deep understanding*: Attempting a problem independently forces you to understand the underlying concepts and logic thoroughly. You'll find yourself gaining a more profound comprehension of the code you write. Don't just ChatGPT it.
3. *Personal growth*: Overcoming challenges independently can boost your confidence and self-esteem. It's incredibly rewarding to see your code working after **you** put in the effort.
4. *Effective learning*: Struggling with a problem before seeking a solution can make the learning experience more effective. The process of debugging and refining your code helps solidify your understanding.
5. *Developing resilience*: Coding isn't always straightforward. You'll encounter errors and roadblocks. Learning to overcome these challenges is absolutely part of programming. Debugging and getting used to seeing errors is absolutely part of the workflow of a programmer. Try not to get frustrated that things don't always work, and you can't always understand things.

Experiment, make mistakes, and learn from them. Write comments in your code to document your thought process. Reflect on what you've learned from the process. Only after you've genuinely given it your best shot should you seek help (either in tutorial or lab time) or consult solutions. When you do, you'll likely find that you can understand and appreciate the solutions more deeply because of your prior efforts. You will remember things. You will LEARN.

Remember that coding is not just about writing code; it's about becoming a better problem solver and thinker. Each challenge you face is an opportunity for growth. Programming isn't difficult, as long as you put in the work. I have no doubt that with your dedication and commitment, you'll become an exceptional programmer. Keep pushing your boundaries, keep learning, and always strive for your best.

Exercise 1: calling Python without arguments

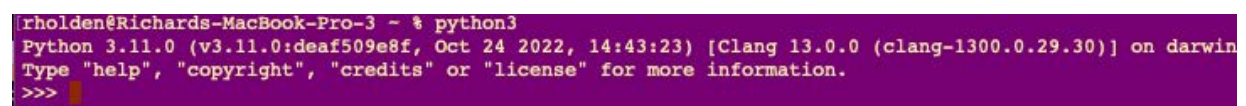
Depending on your installation or operating system, calling Python will look different. Often, on Windows you simply call 'py', like this:



```
PS H:\> py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
```

Figure 1: Calling python on Windows

On a Mac operating system, it's something like this:



```
rholden@Richards-MacBook-Pro-3 ~ % python3
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

Figure 2: Calling python on Mac

In these examples, the windows machine has Python version **3.6.5** installed, but the Mac is on a newer version **3.11.0**. If you are using your own machine just install the latest version. Any version that is **version 3** should be fine for this course. If you have a machine that is using version 2.##, **DON'T USE THAT FOR THIS COURSE**. On this course we are using **Python Version 3**. Python 2 is old and the release of version 3 came with significant improvements and syntax alterations. If you are trying to follow my code from version 2, you will encounter many problems!

Exercise 2: Explore the 'python' command without args (short for arguments)

Note the three adjacent '>' characters in Figure 2 that look like this >>>. This means you are inside the 'REPL' (Read Evaluate Print Loop) environment or what we will sometimes refer to as the inner Python shell:

```
>>> ENTER PYTHON CODE HERE
>>> ONCE YOU HAVE FINISHED YOU CAN
>>> quit()
```

We will very soon be creating scripts. The repl is only useful for playing with / testing very small codes. Make sure you can enter and quit(), then re-enter with the 'py' or 'python3' call, as above.

Exercise 3: Escape sequence experimentation

Inside the Python inner shell (repl) explore the use of escape sequences:

```
\n
\t
\b
```

... by embedding them into some strings of your choice and printing these strings by using the *print()* function. Note the different behaviour the escape sequences have on the printed output.

Look at the following quotes:

Quote 1

```
Python is the "most powerful language you can still read".
- Paul Dubois
```

Quote 2

```
Python is an experiment in how much freedom programmers need. Too much freedom and nobody
can read another's code; too little and expressiveness is endangered.
- Guido van Rossum
```

Quote 3

```
I suggested holding a "Python Object Oriented Programming Seminar", but the acronym was
unpopular.
- Joseph Strout
```

Try to reproduce the format of these by using the `\n` and `\t` escape sequences. Furthermore, try and look for examples of how to include a quote within a quote, and how to create an apostrophe so that you can accurately reproduce the **another's** in part of Quote 2.

Exercise 4: moving into a script (first_string_commands.py)

For each script, you need to follow a specific structure that includes a 'main block' and a 'main' function. While we will cover functions in more detail later in the course, it's important for now that you focus on setting up your scripts correctly, following this structure, even if you don't fully understand every aspect yet. This approach will help you build good habits as you progress. For Each Script, You Will:

- Implement a simple function. In the early examples, this function will simply be called 'main'
- Call that function from within a main block.

Using 'main block' and 'what is the main block?'.

The phrase 'main block' we will use to refer to the code that goes 'inside' this part of a script:

```
if __name__ == "__main__":
```

What is this thing called the main block?

The code `if __name__ == "__main__":` is a common and important construct in Python scripts. It is used to control the execution of code in a script and is particularly useful when you want to ensure that certain parts of the code are only run when the script is executed directly, not when it is imported as a module in another script. Here's how it works:

- `__name__`:
 - a special built-in variable in Python that represents the name of the module or script. When a Python file is run, Python sets this variable automatically.
- `"__main__"`:
 - This is a string that Python assigns to the `__name__` variable when the script is run directly (e.g., by using `python script.py` in the command line).
- `if __name__ == "__main__":`:
 - This condition checks if the `__name__` variable is equal to `"__main__"`. This will be `True` if the script is being run directly (from a command line). If it's `True`, the code block under this line will be executed, otherwise the code will not be executed.

Just use it for now. Don't worry too much about it's meaning.

It might seem a bit confusing or unnecessary at first. For now, it's okay if you don't fully understand what it does or why it's there. The important thing is to **use it as part of the structure of your scripts**. As you gain more experience with Python and programming in general, you'll start to appreciate its purpose and why it's considered good practice. It's like learning to drive a car; you don't need to understand how the engine works to start driving. Similarly, you don't need to fully grasp `if __name__ == "__main__":` to start coding effectively.

Therefore, suggested empty scripting structure:

```
def main():
    # put your code here, e.g. if you want to print "Hello World" to the console
    print("Hello World")

# If this script is being run directly (not imported as a module), enter
# this block, a.k.a the 'main block'
if __name__ == "__main__":
    main() # Call the main function
```

Move previous code to a script following main block convention

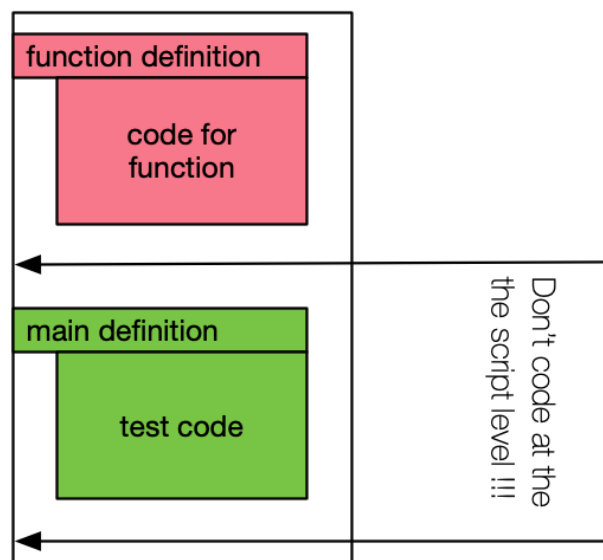


Figure 3: General 'main blocked' script structure.

Put your commands into a script called `<first_string_commands.py>`, ensure you create that script and call it from command line to produce the same output that you did from the repl.

Exercise 5: einstein.py

Create a script called *einstein.py*. Look at the following quotes:

We cannot solve our problems with the same thinking we used when we created them.

Learn from yesterday, live for today, hope for tomorrow. The important thing is not to stop questioning.

It's not that I'm so smart, it's just that I stay with problems longer.

Print each quote followed by attribution to Albert Einstein in the following form:

"Quote text, Quote text, Quote text, Quote text, Quote text, Quote text"
Albert Einstein

Exercise 6: DRY acronym

If you didn't already, inside your script ensure that the string literal "Albert Einstein" is defined in a variable called *<physicist>* which is used 3 times...

You will learn later on in the course that re-using code is generally a very good thing to do. Google the 'dry' acronym for programming and note what it means. I said take a look at the 'dry' acronym for programming and note what it means. I said take a look at the 'dry' acronym for programming and note what it means. I said...

... over time you will learn that repetition in code is also a waste of time, even though some beginners want to read code sequentially, line-by-line, like a book. However, in a lot of the scripts (especially in the early parts of the course) I repeat code, sequentially, to aid your reading of the code, and I provide extensive comments. Just note for now that this isn't the best way to code, but as you learn syntax, we will develop better, neater, and more modular, code, later in the course, code **re-use** being the longer-term goal and by the end of the course you will understand that code isn't sequentially read, but is 'encapsulated' in different areas, and used in other areas of the code, in a non-sequential fashion.

Exercise 7: word counting

Inside the *<einstein.py>* take each quote and create some code that counts the number of words in each quote. **Tip:** The functions: **len()** and **split()** ...might be useful here. Achieve the following kind output to the console, but for each quote, not just quote 3.


```
It's not that I'm so smart, it's just that I stay with problems longer.  
14 words long
```

Figure 4: Einstein quote output

Exercise 8: Einstein's equation.

In *einstein.py*, discover (by coding it) the most frequently occurring character in the variable `<physicist>`, recording that string in the variable `<eqn>` then concatenate this string:

`" = mc^2"`

Finally, assuming a mass of 80% (Einstein was quite small) of an 'average weight' (say 90kg) and constant speed *c* (look up the value of that), output the string:

According to the equation "`e = mc^2`", a famous physicist has mass `<mass>` and, therefore, energy `<number>`

...where numerical values are embedded in the string to achieve the following output:

```
result: e  
According to the equation e = mc^2, a famous physicist has mass <90> and therefore,  
energy <6480000000000.0>
```

Figure 5:Einstein equation output

Tip: the above **result: e** is output from my solution

Exercise 9: Solution

When you get the solution, you should go over the lab again, with the solutions open, look at the codes and understand how they address the exercises.