PYTHON FROM ZERO TO ONE

FROM ABSOLUTE ZERO TO FUNCTIONAL PYTHON BEGINNER

LIU ZUO LIN

Introduction

Hey there! My name is Liu Zuo Lin, a Software Engineer based in Singapore. I've been tutoring Python as a side hustle alongside writing since early 2021, and have since worked with more than 70 unique students of all ages and backgrounds in a variety of areas:

1. Basic Python Programming
2. Object-Oriented Programming
3. Data Structures & Algorithms
4. Machine Learning & Data Science
5. Etc.

A significant portion of the students I've worked with were complete beginners at Python, and I've gathered a decent amount of experience working with students who start from absolute zero.

I don't claim that this book can make you a Python expert (it won't) – This book simply aims to elevate you from a complete beginner (zero) to a reasonably functional and competent beginner (one).

To do that, I've compiled a tried-and-tested beginner Python curriculum that I've used with many of my beginner students into 16 chapters in this book, along with detailed explanations and examples that I've refined over time with each new student.

At certain points in this book, there'll be 7 checkpoints containing a couple of Python questions (answers included). These questions aim to test you on the application of the Python concepts that we've gone through, and can be moderately challenging.

Without further ado, get your computer ready, and let's dive right in.

Content Page

1) Setting Up Python On Your Computer

1.1) Downloading Python

Download link: https://www.python.org/downloads/

Visit the link, and look for a Python 3 installation for your operating system (Windows, MacOS, etc). I'd recommend downloading Python 3.10.X, but using versions 3.9 or 3.8 shouldn't have too much of an impact.

Download Python, and run the .exe or .dmg file in order to install Python on your computer.

1.2) Getting an Integrated Development Environment (IDE)

An integrated development environment (IDE) is essentially an application that we can write and run our code in. Examples of Python IDEs include Visual Studio Code, PyCharm, Spyder etc etc.

For myself, I personally recommend using Visual Studio Code (VSCode) due to its simplicity of use, but feel free to use whatever you want. Examples of running code in Python will be for VSCode though.

Download link: https://code.visualstudio.com/

1.2.1) Setting up your VSCode workspace

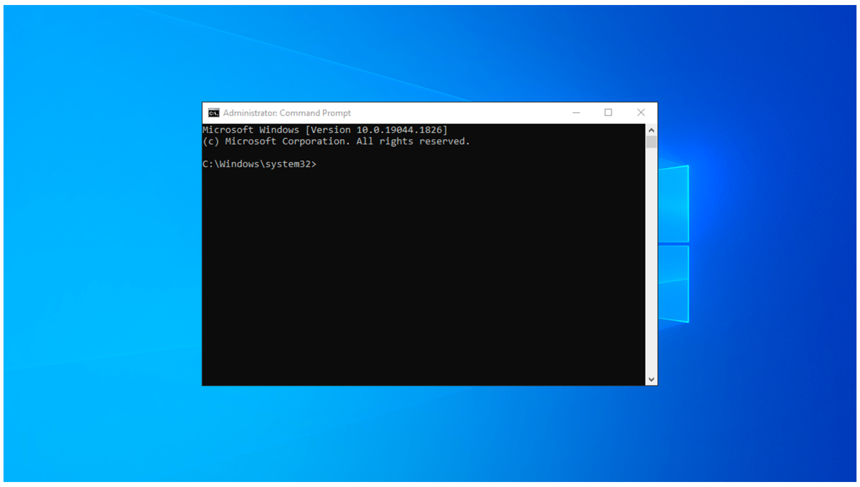
In VSCode, go to file → open folder → select your working folder (can be any folder). This opens VSCode inside your selected folder

Next, to open the terminal in VSCode, click on the 'terminal' tab in the top bar. This will open a terminal instance in VSCode. We need this to run our Python code later.

1.3) Verifying that Python works

1.3.1) Windows

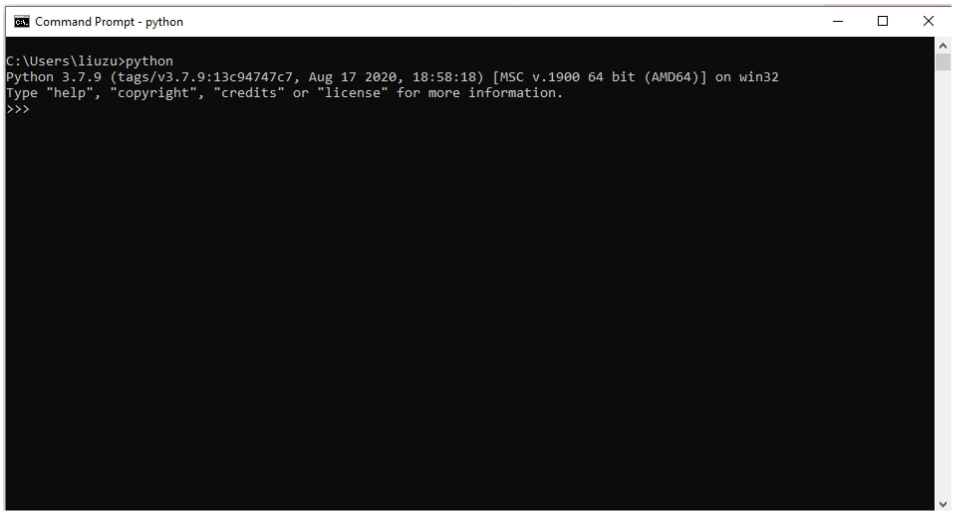
Firstly, open your command prompt (Windows search → Type 'cmd' → Enter). You should see something like this:



```
Microsoft Windows [Version 10.0.19044.1826]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\system32>
```

The black box is your command prompt. Next, type either 'python' or 'py' inside your command prompt, and hit Enter. You should see something like this if Python is working:



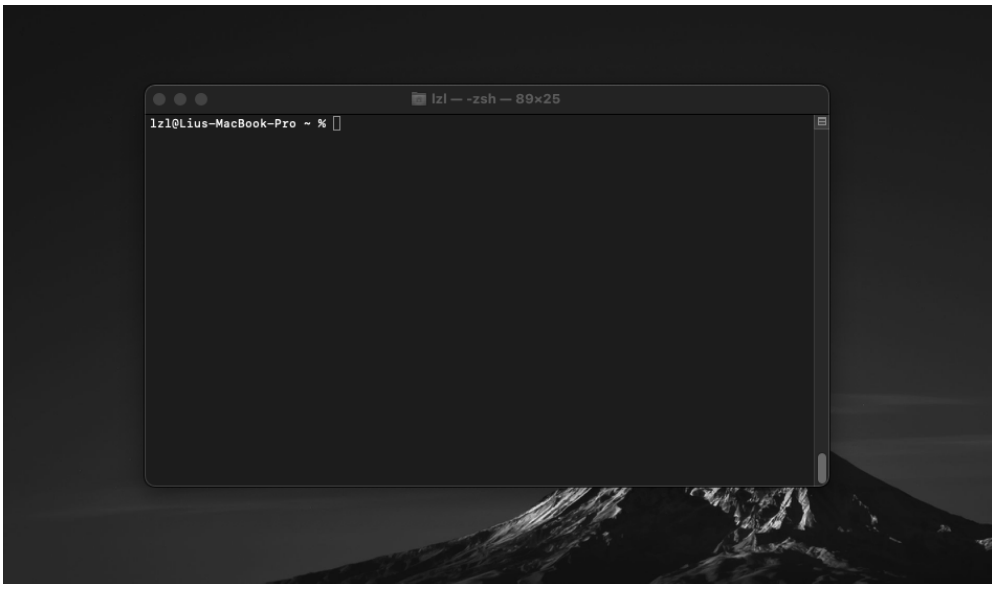
```
C:\Users\liuzu>python
Python 3.7.9 (tags/v3.7.9:13c94747c7, Aug 17 2020, 18:58:18) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You should see 'Python 3……' followed by 3 arrows >>>. This means that Python has been successfully installed on your computer, and is working.

Note – for some Windows machines, 'py' is the correct command, while for others, 'python' is the correct command

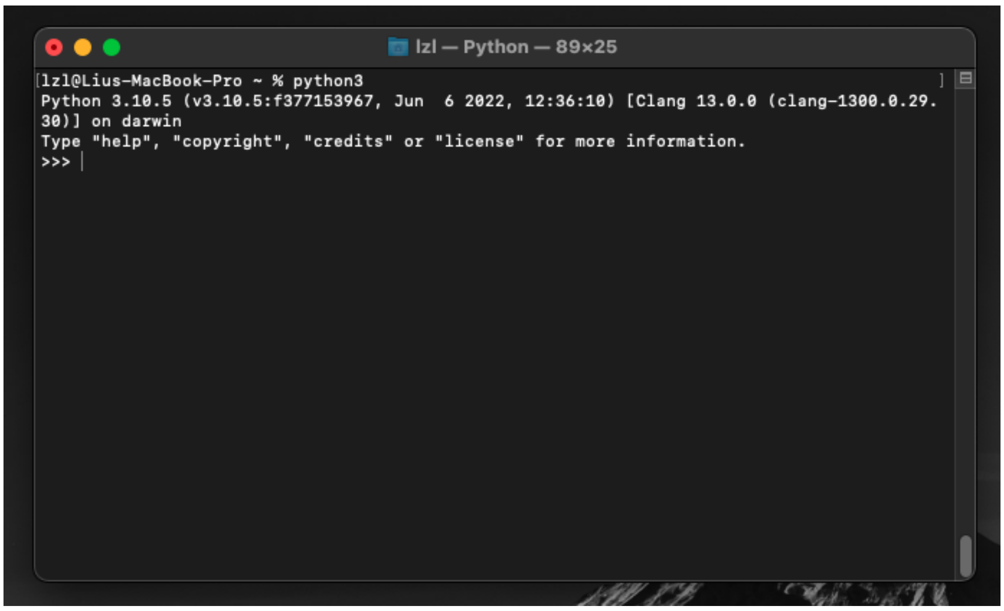
1.3.2) MacOS

Firstly, open your terminal (Spotlight search → Type 'terminal' → hit Enter). You should see something like this:



```
lzl@Lius-MacBook-Pro ~ % 
```

Next, type 'python3' into terminal, and hit Enter. You should see something like this:



```
[lzl@Lius-MacBook-Pro ~ % python3                                         ]
Python 3.10.5 (v3.10.5:f377153967, Jun  6 2022, 12:36:10) [Clang 13.0.0 (clang-1300.0.29.
30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

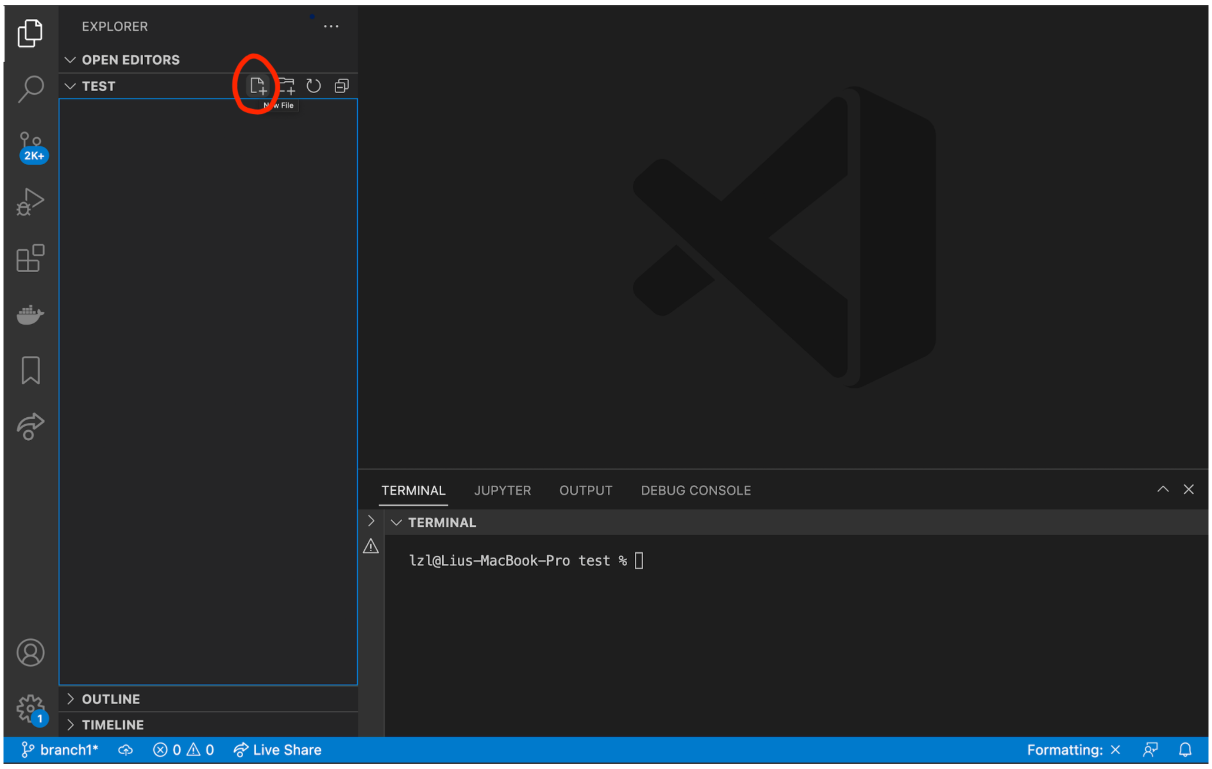
You should see 'Python 3.X.X ……' followed by 3 arrows >>>. This means that Python 3 has been installed on your computer, and is working fine.

1.4) Writing your first Python Script

1.4.1) What is a Python script?

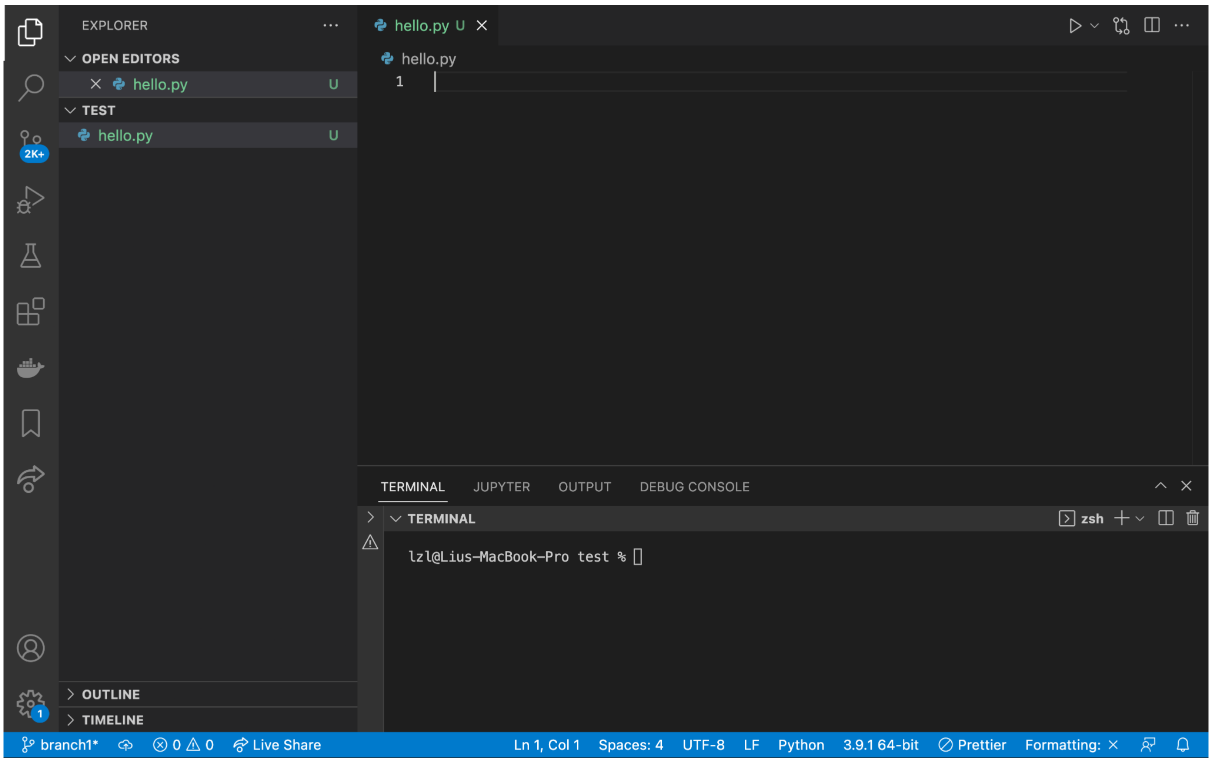
A Python script is a file that ends with a .py extension – we usually write Python code in Python scripts (.py files).

1.4.2) Creating a Python script (VSCode)



```
lzl@Lius-MacBook-Pro test % 
```

Click on the 'new file' button (circled in red). This creates a new file in the folder that we are in, and will prompt us to give our file a name. Let's name it 'hello.py' for now. After creating hello.py, you should see something like this:


EXPLORER
OPEN EDITORS
hello.py U
TEST
hello.py U
OUTLINE
TIMELINE
hello.py U
TERMINAL JUPYTER OUTPUT DEBUG CONSOLE
TERMINAL
zsh
branch1* 0 0 Live Share Ln 1, Col 1 Spaces: 4 UTF-8 LF Python 3.9.1 64-bit Prettier Formatting: ✕

```
lzl@Lius-MacBook-Pro test % 
```

At this point, your Python script hello.py has been created. We just haven't written any Python code inside it yet.

1.4.3) Writing some Python code

Add the following code into your new Python script hello.py

```python
print("hello world")
```

Make sure that the open and close brackets, as well as the open and closed inverted commas are all there – else we'll get an error from Python later on.

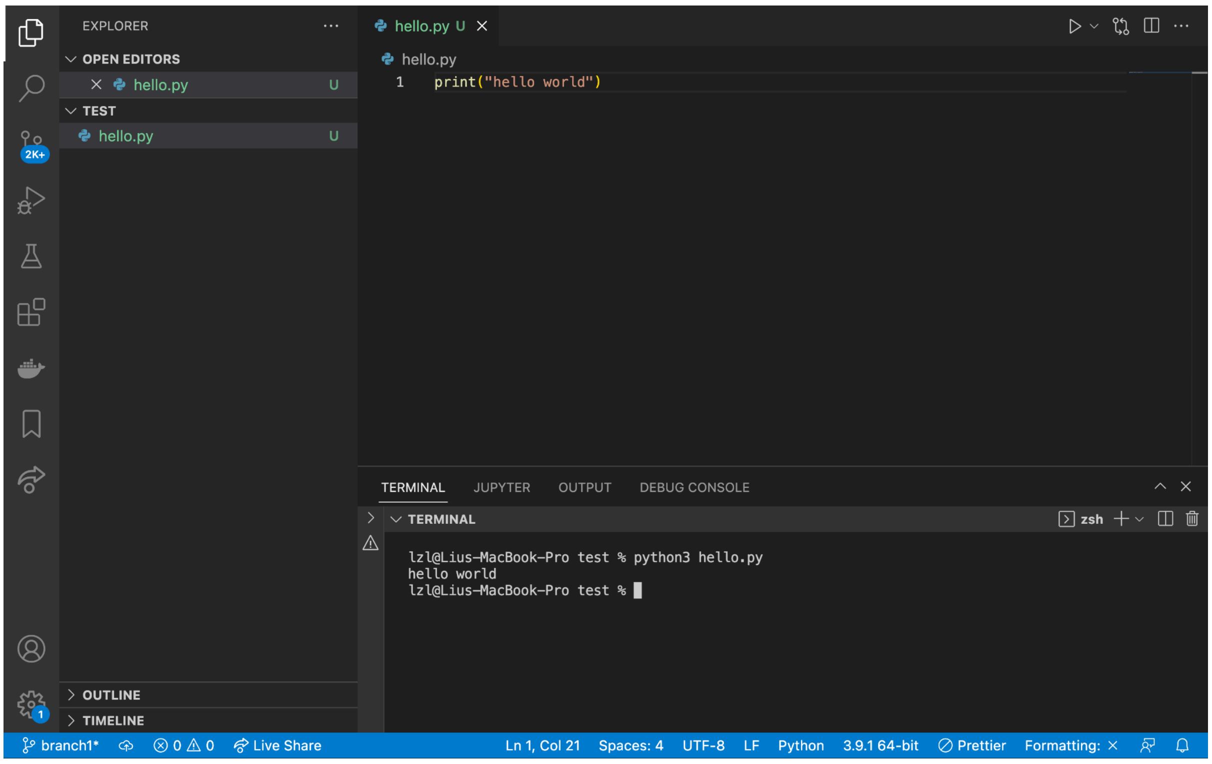
1.4.4) Running our Python script

Depending on your operating system, type the following into the terminal at the bottom.

For Windows, type either 'python hello.py' or 'py hello.py' depending on which one works.

For MacOS, type 'python3 hello.py'.

Next, hit Enter. You should see 'hello world' printed inside the terminal. Congratulations! You've just written and run your first Python script. Here's a screenshot of what your screen should roughly look like:



```python
print("hello world")
```

```
lzl@Lius-MacBook-Pro test % python3 hello.py
hello world
lzl@Lius-MacBook-Pro test %
```

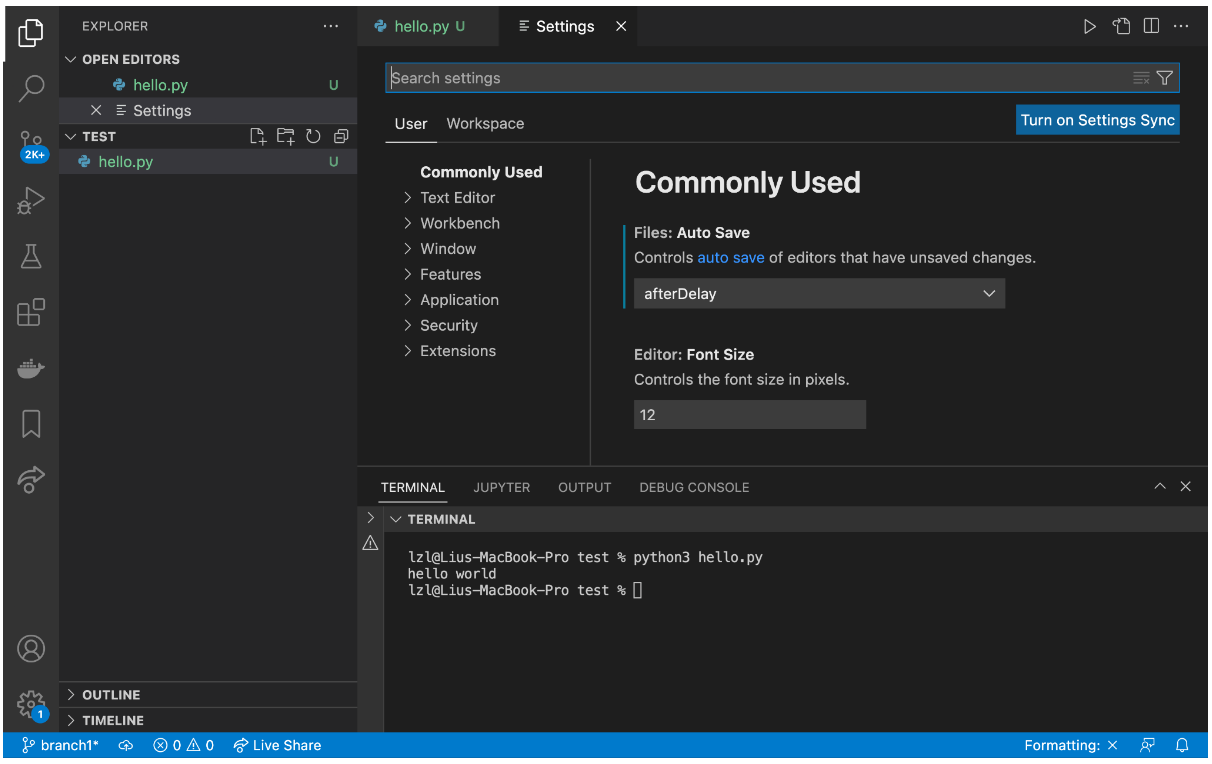
Once you are able to run your Python script, you are good to go!

1.4.5) Configuring Autosave for VSCode

VSCode will not autosave by default. Unless you want to keep hitting save, I recommend setting up autosave in VSCode

1. Go to settings (bottom left corner)
2. The autosave setting will (likely) be the first setting you see
3. Set its value to 'afterDelay'

Your settings page should look something like this:



Commonly Used

Files: Auto Save
Controls auto save of editors that have unsaved changes.
afterDelay

Editor: Font Size
Controls the font size in pixels.
12

```
lzl@Lius-MacBook-Pro test % python3 hello.py
hello world
lzl@Lius-MacBook-Pro test % 
```

Now, whenever you make a change to any file, it should automatically save itself after 1 second of you being idle.

NOTE – This is the case at the time of writing (Aug 2022) but please disregard this and look for a workaround if the user interface has somehow been updated by the time you read this.

2) Basic Syntax

2.1) The print() Function

The print() function displays whatever we give it to our screen. Here, we pass the string value "hello world" into the print function between the open and close brackets. When we run our Python program, it prints "hello world" for us.

```
print("hello world")
```

```
hello world
```

Another example – we pass the integer value 123 into the print function, also between the open and close brackets. Our Python program thus prints 123 for us.

```
print(123)
```

```
123
```

If we put multiple values inside the brackets (separated by comma), Python will print everything with 1 space in between.

```
print(100,200,300)
```

```
100 200 300
```

Note – we'll explain what string values and integer values are in the next chapter!

2.2) Comments In Python

Comments in Python are ignored, and are not executed. They begin with a hash #, and are used to explain Python code.

```python
# this is a comment, and will not be executed
print("hello world")
```

```
hello world
```

Notice that Python completely ignores the comment behind the #

2.3) Assignment Operator (=)

The assignment operator, or the = sign, is used to assign a variable to a value.

```python
x = 4
print(x)

# 4
```

Here, x is the variable, and it's being assigned a value of 4.

```python
y = "hello world"
print(y)

# hello world
```

Here, y is the variable, and it's being assigned to a value of "hello world". Don't worry about the data types yet – we'll get to them in the next chapter

2.4) Valid Variable Names

In Python, there are some rules to naming variables. Variable names:

1. contain only letters, numbers or the underscore character
2. must start with either a letter or an underscore character
3. cannot be a reserved Python keyword
4. are case sensitive

Reserved Python keywords:

```
False   await     else     import   pass
None    break     except   in       raise
True    class     finally  is       return
and     continue  for      lambda   try
as      def       from     nonlocal while
assert  del       global   not      with
async   elif      if       or       yield
```

The words above are reserved Python keywords, and cannot be used as variable names at all. Note that your IDE (I'm using Visual Studio Code) will usually colour the reserved keywords differently from other variables.

Examples of valid variable names:

```
x
y
myvariable
my_variable
_my_variable
__
my_variable_12345
```

Examples of invalid variable names:

```
123_my_variable   # cannot start with a number
my-variable       # can only contain letters, numbers & underscores
pass              # cannot be a reserved python keyword
```

2.5) Reassigning Variables

```
x = 4   # x is assigned to 4
x = 5   # x is 5 now
```

In mathematics, this is completely invalid as x cannot be equal to 4 and 5 at the same time. However, in Python, = refers to the assignment operator, and x is simply *reassigned* to 5. Here, the second line x=5 simply overrides the previous line.

2.6) Prompting The User For Input

```
name = input("what is your name? >>> ")
print(name)
```

When we run this, our program prompts us to enter something with the string value "what is your name? >>> ". We need to enter something, then hit <ENTER>, and whatever we type will be assigned to the variable *name.* Our output:

```
what is your name? >>> bob
bob
```

Note that the underlined bob above is what we enter into the terminal or CMD

2.7) Switching Variables

```
a = 4
b = 5
```

Here, we aim to switch the values of the variables a and b such that a=5 and b=4. Here's how we can do this with a temporary variable:

```
x = a   # x=4, a=4, b=5
a = b   # x=4, a=5, b=5
b = x   # x=4, a=5, b=4
```

At this point, a=5 and b=4, and they have been successfully switched.

3) Basic Data Types

In the previous chapter, we learnt that variables can be assigned certain values. Each value has a data type. Here are 4 basic data types that we need to know:

1. int → integer (whole numbers)
2. float → float (decimal numbers)
3. str → string (sequence of characters)
4. bool → boolean (True or False)

There are many more data types in Python, but let's just deal with these basic 4 for now.

3.1) Integers (int)

An integer is essentially a whole number (no decimal places), and it can be negative.

```
a = 5
print(type(a))
# <class 'int'>
```

```
b = 1000
print(type(b))
# <class 'int'>
```

```
c = -8
print(type(c))
# <class 'int'>
```

Note – to check the type of a variable/value x, we simply print type(x)

3.2) Floats (float)

A float is essentially a decimal number (it can be negative)

```
a = 3.14
print(type(a))
# <class 'float'>
```

```
b = -3.14159265
print(type(b))
# <class 'float'>
```

3.3) Strings (str)

A string is essentially a sequence of characters, and we can think of them as words or sentences. We put our characters (any character) in between single or double inverted commas (either is fine – there's no difference).

```
a = 'hello'
print(type(a))
# <class 'str'>
```

```
b = "apple orange pear"
print(type(b))
# <class 'str'>
```

```
c = "apple !@#$%^&*()"
print(type(c))
# <class 'str'>
```

When we prompt for user input, whatever the user enters will *always* be a string. Much more on strings will be covered in a later chapter, but for now, let's accept it as it is.

3.4) Booleans (bool)

A boolean value is either a True or a False (remember to capitalize the first letter). Note that there are only 2 possible boolean values.

```
a = True
print(type(a))
# <class 'bool'>
```

```
b = False
print(type(b))
# <class 'bool'>
```

3.5) Other Data Types

There are many more data types apart from the above 4:

```
None list dict tuple set etc
```

But let's get familiar with the basics first. We'll deal with all these later on.

3.6) Type Casting

Type casting refers to converting a value's type to another type. To convert a value to a new type, we simply need to place the value inside the new type like this:

```
newvalue = newtype(oldvalue)
```

Converting string to integer/float:

```
x = "100"     # x is a string of value "100"
x = int(x)    # x is now an integer of value 100

y = "3.14"    # y is a string of value "3.14"
y = float(y)  # y is now an float of value 3.14
```

Converting integer/float to string:

```
x = 150       # x is integer of value 150
x = str(x)    # x is now string of value "150"

y = 3.14      # y is float of value 3.14
y = str(x)    # y is now string of value "3.14"
```

Converting float to integer (this simply removes the decimal places):

```
x = 3.14159   # x is float of value 3.14159
x = int(x)    # x is now an integer of value 3
```

Note also that an error will be raised if a type cannot be properly converted:

```
x = "apple"   # x is a string of value "apple"
x = int(x)    # "apple" cannot be converted to integer, so error raised
```

4) Numerical Operators

Numerical operators in Python allow us to perform mathematical operations. Here are the 7 numerical operators in Python:

1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
5. Floor division (//)
6. Modulus (%)
7. Exponentiation (**)

4.1) Addition (+)

The addition operator + is used to add numbers (integer or float) together

```
x = 4 + 5
print(x)   # 9
```

```
a = 3
b = 0.14
x = a + b
print(x)   # 3.14
```

4.2) Subtraction (-)

The subtraction operator - is used to subtract one number from another

```
x = 10 - 7
print(x)   # 3
```

```
x = 5 - 9
print(x)   # -4
```

4.3) Multiplication (*)

The multiplication operator * (NOT x) is used to multiply numbers together

```
x = 4 * 5
print(x)   # 20
```

```
x = 10 * -5
print(x)   # -50
```

4.4) Division (/)

The division operator / is used to divide a number (int or float) by another number (int or float). Note that this operator always outputs a float number.

```
x = 10 / 3
print(x)   # 3.3333333333333
```

```
x = 10 / 5
print(x)   # 2.0
```

```
x = 5 / 10
print(x)   # 0.2
```

Note that dividing by 0 (invalid math operation) will raise a ZeroDivisionError

```
print(10/0)

# Traceback (most recent call last):
#   File "/some/path/run.py", line 1, in <module>
#     print(10/0)
# ZeroDivisionError: division by zero
```

Also, floating point inaccuracy is normal and we can simply ignore it:

```
x = 10 / 5
print(x)   # 2.00000000001
```

4.5) Floor Division (//)

The floor division operator // divides 2 numbers and outputs the quotient. You can think of floor division as dividing normally, then converting to an integer.

```
x = 10 // 3
print(x)   # 3, as 10 / 3 = 3 with remainder 1
```

```
x = 13 // 5
print(x)   # 2, as 13 / 5 = 2 with remainder 3
```

```
x = 76 // 7
print(x)   # 10, as 76 / 7 = 10 with remainder 6
```

4.6) Modulus (%)

The modulus operator divides 2 numbers, and outputs the remainder.

```
x = 10 % 3
print(x)   # 1, as 10 / 3 = 3 with remainder 1
```

```
x = 13 % 5
print(x)   # 3, as 13 / 5 = 2 with remainder 3
```

```
x = 76 % 7
print(x)   # 6, as 76 / 7 = 10 with remainder 6
```

Note that we can use the modulus operator % to check if a number is odd or even, but more on this later.

4.7) Exponentiation (**)

The exponentiation operator ** (NOT ^) raises one number to the power of another.

```
x = 5 ** 2
print(x)   # 25 as 5 * 5 = 25
```

```
x = 2 ** 3
print(x)   # 8 as 2 * 2 * 2 = 8
```

```
x = 2 ** 0.5
print(x)   # 1.4142135 - same as the square root function
```

4.8) BOSMAS/PEDMAS Rule

We've probably learnt this before in math class, but this applies to Python too. Numerical operators are executed in this order (first to last).

1. Brackets
2. Exponentiation
3. Multiplication and Division
4. Addition and Subtraction

```
x = 1 + 2 * 3 + 4
print(x)   # 11   as 1 + (2*3) + 4 = 11
```

```
x = 1 + 2**2 + 3**2
print(x)   # 14   as 1 + (4) + (9) = 14
```

4.9) Strings Concatenation/Addition (+)

This chapter has discussed numerical operators working mainly with numbers, but strings can use them too. More specifically, strings can use the addition (+) and multiplication (*) operator. String concatenation is a fancy word for joining 2 strings together one after another, and you don't have to remember this word if you don't feel like it.

```
x = "apple" + "pie"
print(x)    # applepie
```

```
a = "orange"
b = "juice"
x = a + b
print(x)    # orangejuice
```

Note that the order of strings in the expression matters.

```
x = "pie" + "apple"
print(x)    # pieapple
```

IMPORTANT – A string can only be added together with another string. If we attempt to add a string with anything else, we get a TypeError

4.10) String Multiplication (*)

We can multiply a string with an integer to repeat the string pattern multiple times.

```
x = "a" * 3
print(x)    # aaa
```

```
x = "apple" * 3
print(x)    # appleappleapple
```

```
x = "apple " * 3    # additional space character behind string
print(x)    # apple apple apple
```

IMPORTANT – A string can only be multiplied with an integer (else we get TypeError)

4.11) Adding/Multiplying Incompatible Data Types

```
a = "3"
b = 4
```

Let's say we really want to add a and b together. Here, we cannot simply do a + b, as we cannot add an integer with a string. We need to convert either 1) string to integer or 2) integer to string before we can add them together.

If we want to add them as integers to get 3 + 4 = 7:

```
x = int(a) + b   # int(a) is 3
print(x)   # 7
```

If we want to add them as strings to get "3" + "4" = "34":

```
x = a + str(b)   # str(b) is "4"
print(x)   # "34"
```

4.12) Case Study – The input() Function

Remember in chapter 2 where we talked about the input() function, which enables us to prompt the user for input? This function always returns a string type, even if we type numbers. If we want to use the user input as a number, we need to do some type casting.

For instance, let's say we want to write a program that prompts the user for 2 integer numbers, and later adds them together as integers. The code:

```
first = input("Enter the first number >>>")
second = input("Enter the second number >>>")
x = int(first) + int(second)
print(x)
```

Let's say we enter 4, then 5. The variable *first* is assigned to the string "4", while the variable *second* is assigned to the string "5". In order to add them together as integers, we first need to use type casting to convert them into integers

4.13) Compound Assignment Operators

```
x = 4
```

Let's say we have an existing variable *x.* In this example, it has the integer value 4. We wish to increment *x* by 1 (in this case to 5). We can use this syntax to achieve this:

```
x = x + 1
```

Here, *x* has the value 4, so *x* + 1 has the value 5. The variable *x* is thus reassigned to 5. This works for all other numerical operators too.

```
x = 4
x = x + 1   # x was 4, x + 1 is 5, x is now 5
x = x - 2   # x was 5, x - 2 is 3, x is now 3
x = x * 2   # x was 3, x * 2 is 6, x is now 6
x = x / 3   # x was 6, x / 3 is 2.0, x is now 2.0
```

Compound assignment operators allow us to write this in short form. The compound assignment operator in question is the += below:

```
x += 1   # this is the exact same as x = x + 1
```

Similarly, this applies to all numerical operators. The compound assignment operators in question are +=, -=, *= and /=:

```
x = 4
x += 1   # same as 'x = x + 1'   # x is now 5
x -= 2   # same as 'x = x - 1'   # x is now 3
x *= 2   # same as 'x = x * 1'   # x is now 6
x /= 3   # same as 'x = x / 1'   # x is now 2.0
```

Checkpoint 1

Test your understanding of the previous couple of chapters.

1) Adding numbers

Write a Python script that prompts the user for 2 numbers (using the input function). It then prints the sum of the 2 numbers. Example run (user input is underlined):

```
Enter the first number: 4
Enter the second number: 5
9
```

2) Division

Write a Python script that prompts the user for 2 numbers. It then prints 1) the result (decimal number) 2) the quotient and 3) the remainder when the first number is divided by the second number. Example run:

```
Enter the first number: 11
Enter the second number: 3
result: 3.6666666666667
quotient: 3
remainder: 2
```

3) Repeated String

Write a Python script that prompts the user for a string, then a number n. It then prints the string, but repeated n times. Example run:

```
Enter a string: hello
Enter a number: 3
hellohellohello
```

Answers (Checkpoint 1)

1) Adding numbers

```
number1 = input("Enter the first number: ")
number2 = input("Enter the second number: ")
total = int(number1) + int(number2)
print(total)
```

Remember that the input function ALWAYS returns a string, so if we want to add them as integers, we need to first cast them as integers using int().

2) Division

```
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
print("result:", number1 / number2)
print("quotient:", number1 // number2)
print("remainder:", number1 % number2)
```

Remember that 1) the true division operator / returns a decimal number 2) the floor division operator // returns an integer and 3) the modulus operator % returns the remainder.

3) Repeated String

```
string = input("Enter a string: ")
n = int(input("Enter a number: "))
print(string * n)
```

To get a repeated string, we need to multiply a string value with an integer, so we can only convert n into an integer using type casting. Afterwards, we can simply multiply them together to get our answer.

5) Conditional Statements (If-Else Loops)

Conditional statements, or if-else loops, run certain code when a certain condition is met. Before we dive right into the syntax, let's take a look at what comparison operators are.

5.1) Comparison Operators

Comparison operators are used to compare variables, and evaluate to a boolean value:

1. == → equals
2. != → not equals
3. < → less than
4. <= → less than or equal to
5. > → more than
6. >= → more than or equal to

5.1.1) The Equals Operator (==)

This operator checks if 2 values are equal to one another. It returns True if 2 values are equal, and False otherwise

```
print(1 == 1)   # True
print(1 == 3)   # False

print("apple" == "apple")   # True
print("apple" == "orange")   # False
```

IMPORTANT – we DO NOT use the assignment operator (=) to check for equality

5.1.2) The Not Equals Operator (!=)

The operator is the opposite of the equals operator, and it returns True if 2 values are NOT equal, and False if the 2 values are equal.

```
print(1 != 1)   # False
print(1 != 3)   # True

print("apple" != "apple")   # False
print("apple" != "orange")   # True
```

5.1.3) The Less Than Operator (<)

The less than operator checks if the first value is less than the second value. For strings, whether one string is 'less than' another depends on its alphabetical order.

```
print(4 < 6)   # True
print(4 < 3)   # False
print(4 < 4)   # False

print("b" < "c")   # True
print("b" < "a")   # False
```

Note that 4 < 4 returns False, as *4 is not smaller than 4*

5.1.4) The Less Than Or Equals Operator (<=)

The less than or equals to operator returns True if 1) the first value is less than the second value or 2) the first value is equal to the second value.

```
print(4 <= 6)   # True
print(4 <= 3)   # False
print(4 <= 4)   # True

print("b" <= "c")   # True
print("b" <= "a")   # False
```

Note that 4 <= 4 returns True now

5.1.5) The Greater Than Operator (>)

The greater than operator returns True if the first value is greater than the second value, and False otherwise. Similarly, string comparison is based on alphabetical order.

```
print(4 > 2)   # True
print(4 > 5)   # False
print(4 > 4)   # False
```

Note that 4 > 4 returns False, as 4 is not greater than 4.

5.1.6) The Greater Than Or Equals Operator (>=)

The greater than or equals to operator returns True if 1) the first value is greater than the second value or 2) the first value is equal to the second.

```
print(4 >= 2)   # True
print(4 >= 5)   # False
print(4 >= 4)   # True
```

Similarly, 4 >= 4 evaluates to True, as 4 is equal to 4.

5.2) If Else Loops Syntax

There are 3 keywords here that we need to be familiar with – 1) if 2) elif and 3) else. Note that only the 'if' keyword is compulsory when writing conditional statements in Python – the 'elif' and 'else' keywords are optional

5.2.1) If Loop Syntax

```
score = 65
if score > 50:
    print("pass")

# pass
```

What we are telling Python to do: if *score* is above 50, print "pass". Otherwise, do nothing. Notice that:

1. After our condition (score > 50), there is a colon. The colon must be there.
2. The print("pass") line has some spaces on its left. This is known as an indent, and tells Python that this line is *part of the* if loop that is above it.

NOTE – we can use either 1) 4 spaces 2) 2 spaces or 3) 1 tab as an indent in Python. For this book, we'll just be using 4 spaces

```
score = 65
if score > 50:
    print("pass")   # only runs if score > 50

print("hello")       # runs regardless as it's not part of if block
```

5.2.2) If Else Loop Syntax

```
score = 45
if score > 50:
    print("pass")   # runs if score > 50
else:
    print("fail")    # runs otherwise

# fail
```

Here, we've added an else loop, and take note of the similar syntax (colon + indent). The 'else' loop can only exist if there is an 'if' loop - it executes if the condition given in the 'if' loop evaluates to False.

In this case, since *score* is 45, the expression 'score > 50' evaluates to False. Our 'if' block doesn't run, and instead our 'else' block runs, and "fail" is printed.

5.2.3) If Elif Else Loop Syntax

```
score = 67
if score >= 80:     # runs if score is 80 or more
    print("A")
elif score >= 65:   # runs if score is 65 to 79
    print("B")
elif score >= 50:   # runs if score is 50 to 64
    print("C")
else:               # runs if all above conditions are False
    print("D")
# B
```

Here, we've added another block called the 'elif' block, known as an 'else if' block. When writing conditional statements:

1. we MUST have 1 'if' block
2. we can have 0 to infinity 'elif' blocks
3. we can have 0 or 1 'else' block.

When writing 'elif' blocks, remember to add a condition after the 'elif' keyword. An 'elif' block will execute if 1) all conditions before it are False and 2) its condition is True. Here, "B" will be printed.

5.3) The Short-Circuit Nature Of Conditional Statements

Consider the following chunk of conditional statements

```
choice = "c"
if choice == "a":
    print("your choice is a")
elif choice == "b":
    print("your choice is b")
elif choice == "c":
    print("your choice is c")
elif choice == "d"
    print("your choice is d")
else:
    print("your choice is neither a, b, c nor d)
```

For conditional statements in Python, once we meet a True condition, we ignore all other conditions below regardless of whether they are True or False. In this case, choice = "c".

1. choice == "a" evaluates to False, so we move on to the next statement.
2. choice == "b" evaluates to False, so we move on to the next statement.
3. choice == "c" evaluates to True, so we print "your choice is c"
4. All other conditional statements below are ignored

```
choice = "e"
```

Let's say now that choice = "e". Let's run through the conditional statements again.

1. choice == "a" evaluates to False, so we move on to the next statement.
2. choice == "b" evaluates to False, so we move on to the next statement.
3. choice == "c" evaluates to False, so we move on to the next statement.
4. choice == "d" evaluates to False, so we move on to the next statement.
5. We are now at the 'else' statement, as all previous conditions have been False
6. We hence print "your choice is neither a, b, c nor d".

5.4) Logical Operators

Logical operators in Python are used to combine multiple conditions together. There are 3 logical operators that we need to get familiar with:

1. and
2. or
3. not

5.4.1) The 'and' Logical Operator

The logical operator 'and' can be used to join 2 or more conditions, and the entire expression is True only if every single condition is True.

```
True and True    # True
True and False   # False
False and False  # False

True and True and True and True    # True
True and True and True and False   # False
```

Let's say a student wishes to get a prize. To get the prize, he needs to:

1. Have an excellent conduct
2. Have an average score of 80 or higher

```
avg_score = 83.5
conduct = "excellent"

if avg_score >= 80 and conduct == "excellent":
    print("you get the prize")
else:
    print("no prize")
```

Here, "you get the prize" is only printed if BOTH conditions 'avg_score >= 80' and 'conduct == "excellent" ' evaluate to True.

5.4.2) The 'or' Logical Operator

The logical operator 'or' can be used to join 2 or more conditions, and the entire expression is True if at least one of the conditions is True.

```
True or True    # True
True or False   # False
False or False  # False

True or True or True or True    # True
True and True or True or False   # False
```

Let's say a student can now get a prize if any one of his scores is 90 or more.

```
english = 87
math = 91
science = 79

if english >= 90 or math >= 90 or science >= 90:
    print("you get the prize")
else:
    print("no prize")
```

Here, the student only needs EITHER of his scores to be 90 or more to get the prize. If either english, math or science is 90 or more, the ENTIRE expression evaluates as True

5.4.3) The 'not' Logical Operator

The logical operator 'not' reveses the boolean value of whatever expressions that comes after the 'not' operator.

```
not True    # False
not False   # True
```

For instance, let's say that a passing grade is given if the score is NOT below 50

```
score = 49
if not score < 50:
    print("you pass")
```

The 'not' logical operator here essentially reverses the boolean value of the expression 'score < 50' – True becomes False and False becomes True

5.5) De Morgan's Law

Remember in math class where we had to expand brackets?

- - (4 + 5) becomes - 4 - 5 when we expand the brackets
- - (6 - 2) becomes - 6 + 2 when we expand the brackets

De Morgan's Law tells us how to do this when expanding brackets for conditional statements. When we expand the brackets with a 'not' in front, 'and' becomes 'or' and vice versa.

```
not (X or Y)

# is the same as

not X and not Y
```

```
not (X and Y)

# is the same as

not X or not Y
```

For instance, let's say a student is eligible for the dean's list only if:

1. he does not have more than 3 days in detention in total
2. his conduct is not "poor"

```
detention_days = 4
conduct = "ok"

if not (detention_days > 3 or conduct == "poor"):
    print("eligible for dean's list")

# is the same as

if (not detention_days > 3) and (not conduct == "poor"):
    print("eligible for dean's list")
```

6) Functions

A function is a reusable block of code that runs only when we call (use) it.

6.1) Some Built-In Python Functions

Let's take a look at some existing basic in-built Python functions.

6.1.1) The print function

```python
print("hello world")
```

Yes, *print* is a function, and it displays whatever we give it in our terminal. Here, the *input* or *argument* is the string "hello world", and we pass it to the print function inside brackets.

6.1.2) The input function

```python
name = input("what is your name?")
```

The input function prompts the user for some input, and returns whatever we enter. Here, the argument is the string "what is your name?", and the output is whatever we enter into the terminal/cmd when prompted.

6.1.3) The range function

```python
range(5)
```

The range function takes in a number, and returns a bunch of numbers to us. Here, the *input* or argument is the number 5. We'll see more of this built-in range function later on in chapter 7) For Loops.

6.1.4) The abs function

The abs function (short for 'absolute') takes in a number and forces it to be positive.

```python
x = abs(-5)    # x is now 5
y = abs(5)     # y is still 5
```

6.2) Defining Your Own Function – Syntax

Here's how we generally define a function:

```python
def function_name(parameter1, parameter2):
    # do stuff
    return function_output
```

- The function name must come after the 'def' keyword
- The function name can be anything as long as it is a valid variable name
- Functions can take in 0 to infinity parameters (inputs). The above example has 2.
- The output of the function follows the 'return' keyword
- There must be a colon after the function name and parameter definition
- The function body must be *indented*

Let's take a look at a couple of examples.

6.2.1) Example 1 – add10

```python
def add10(n):
    return n + 10
```

- function name is 'add10'
- this function 'add10' takes in *1* argument/parameter, denoted by n
- this function outputs n + 10

6.2.2) Example 2 – add

```python
def add(a, b):
    return a + b
```

- function name is 'add'
- this function takes in 2 arguments/parameters, denoted by a and b
- this function outputs the sum of a and b

6.2.3) Example 3 – sayhello

```python
def sayhello():
    print("hello")
```

- function name is 'sayhello'
- this function takes in *zero* arguments/parameters.
- this function does not output anything (absence of return statement)
- this function simply prints "hello" when used/called

6.2.4) Example 4 – average_of_3

```python
def average_of_3(a, b, c):
    total = a + b + c
    return total / 3
```

- function name is 'average_of_3'
- this function takes in 3 arguments/parameters denoted by a, b and c.
- this function computes the average of a, b and c, and outputs the average.
- Note that the entire function body needs to be *indented.*

6.2.5) Indentation In Functions

In functions, the function body *must* be indented once. Any code that is *not indented* will not be part of the function body.

```python
def hello():
    print("hello")     # part of function body
    print("world")     # part of function body
```

```python
def hello():
    print("hello")     # part of function body

    print("world")     # part of function body

print("testing")       # NOT part of function body

    print("world")     # NOT part of function body - error raised
```

6.3) Calling (Using) Functions

A function defined on its own does absolutely *nothing.* We have to call the function (use the function) for it to do anything. We call the function by typing the function name, and adding 2 brackets behind.

NOTE – our arguments/parameters go in between the 2 brackets, and the number of parameters must match that of the function definition

```
def function_name(parameter1):   # defining the function
    return parameter1 + 10

x = function_name(100)           # calling the function

y = function_name(4, 5)          # wrong number of parameters -> error
```

6.3.1) Example 1 – add10

```
def add10(n):
    return n + 10

x = add10(7)   # x is 17
```

The 'add10' function takes in only 1 parameter n, so we can only pass in 1 parameter when we are calling the 'add10' function.

6.3.2) Example 2 – add

```
def add(a, b):
    return a + b

x = add(4, 5)   # x is 9
```

The 'add' function takes in 2 parameters a and b, so we must pass in 2 parameters when we are calling the 'add' function.

6.3.3) Example 3 – sayhello

```python
def sayhello():
    print("hello")

sayhello()   # "hello" is printed
```

The 'sayhello' function takes in 0 parameters, so we cannot pass in any parameters when calling 'sayhello()'.

6.3.4) Example 4 – average_of_3

```python
def average_of_3(a, b, c):
    total = a + b + c
    return total / 3

x = average_of_3(4, 5, 6)   # x is 5.0
```

The 'average_of_3' function takes in 3 parameters, so we must pass in 3 parameters.

6.4) Calling (Using) A Function In Another Function

A function is essentially a reusable block of code, and we can use a function in another function as long as the function we wish to use is defined beforehand.

```python
def add10(n):
    return n + 10

def add17(n):
    nplus10 = add10(n)
    return nplus10 + 7

x = add17(2)   # 19
```

Checkpoint 2

1) Larger

Write a function *larger(a, b)* that takes in 2 numbers, a and b, and returns the larger number between a and b.

```
larger(4, 5)   # 5
larger(5, 4)   # 5
```

2) Largest

Write a function *largest(a, b, c)* that takes in 3 numbers, a, b and c. It returns the largest number between a, b and c. Hint – use the *larger(a, b)* function from before.

```
largest(4, 7, 5)   # 7
largest(7, 4, 5)   # 7
largest(5, 4, 7)   # 7
```

3) Dean's List

Write a function *dean(english, math, science, conduct)* that takes in 4 parameters – english score, math score, science score and conduct. A student can get into the dean's list if 1) his average score is 80 or more and 2) his conduct is "excellent". This function prints "you get into dean's list" if the student gets into the dean's list, and "no dean's list" otherwise.

```
dean(81, 85, 94, "excellent")   # you get into dean's list
dean(81, 85, 94, "good")        # no dean's list
dean(81, 82, 75, "excellent")   # no dean's list
```

Note that *both* conditions need to be fulfilled before the student can get into the dean's list.

Answers (Checkpoint 2)

1) Larger

```python
def larger(a, b):
    if a > b:
        return a
    else:
        return b
```

Here, we can write a simple conditional statement to return the larger number of a and b.

2) Largest

```python
def largest(a, b, c):
    larger_ab = larger(a, b)
    return larger(larger_ab, c)
```

Here, we *reuse* the *larger(a, b)* function from part 1. The variable *larger_ab* is the larger number between a and b. We then use the *larger(a, b)* function again to find the larger number between *larger_ab* and *c,* and this will be the largest number.

3) Dean's List

```python
def dean(english, math, science, conduct):
    avg = (english + math + science) / 3
    if avg >= 80 and conduct == "excellent":
        print("you get into dean's list")
    else:
        print("no dean's list")
```

Here, we first need to compute the student's average score. We add them all up, divide by 3, and assign the average score to the variable *avg.* Since 2 conditions need to be fulfilled for the student to get into the dean's list, we need to use the 'and' logical operator here to join the 2 conditions.

7) For Loops

For loops allow us to repeat an action multiple times in Python. For instance, if we need to print the string "apple" 5 times, we can choose to write 5 print statements.

```
print("apple")
print("apple")
print("apple")
print("apple")
print("apple")
```

This gets the job done, but what if we are asked to print "apple" 1000 times? Do we copy and paste the print statement 1000 times? Or, we can choose to be efficient, and use a for loop.

```
for i in range(5):
    print("apple")
```

The code that uses the for loop does the exact same thing as the one with 5 print statements. If we want to print "apple" 1000 times, we simply change 5 to 1000. If we want to print "apple" 1,000,000 times, we simply change 1000 to 1,000,000.

7.1) Basic For Loop Syntax

```
for iterator_variable in range(integer_number):
    # for loop body
    # do something with iterator variable
```

Similarly to functions, the body of the for loop must be indented. Here's a basic example of a for loop in action:

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

Here, the iterator variable can be any variable as long as it is a valid variable name. We normally use single letter variables like *i* as the iterator variable. Also, the 'print(i)' statement is repeated multiple times as a result of the for loop (5 times to be exact)

7.2) The range function

Yes, the 'range' in the for loop is a function, and we need to know how it works. Note that argument and parameter are used interchangeably here (just accept it for now)

7.2.1) The range function with 1 argument

```
range(end)
```

If we pass in 1 integer to range(), it generates all integers from 0 to our input integer, EXCLUDING our input integer.

```
range(4)
# numbers generated -> 0, 1, 2, 3 (EXCLUDING 4)
```

```
range(11)
# numbers generated -> 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 (EXCLUDING 11)
```

7.2.2) The range function with 2 arguments

```
range(start, end)
```

If we pass in 2 arguments into the range function, the first argument becomes the *start* and the second argument becomes the *end.* The range function then generates numbers from *start* to *end,* EXCLUDING *end.*

```
range(1, 5)
# start = 1, end = 5
# numbers generated -> 1, 2, 3, 4 (EXCLUDING 5)
```

```
range(4, 9)
# start = 4, end = 9
# numbers generated -> 4, 5, 6, 7, 8 (EXCLUDING 9)
```

```
range(-4, 5)
# start = -4, end = 5
# numbers generated -> -4, -3,  -2, -1, 0, 1, 2, 3, 4 (EXCLUDING 5)
```

7.2.3) The range function with 3 arguments

```
range(start, end, step)
```

If we pass in 3 arguments into the range function, the first argument becomes *start,* the second argument becomes *end,* and the third argument becomes *step.* The range function then generates numbers from *start* to *end*, EXCLUDING *end,* and incrementing by *step.*

```
range(1, 10, 2)
# start = 1, end = 10, step = 2
# numbers generated -> 1,3,5,7,9 (EXCLUDING 10)
```

```
range(5, 26, 5)
# start = 5, end = 26, step = 5
# numbers generated -> 5, 10, 15, 20, 25 (EXCLUDING 26)
```

```
range(-3, 10, 3)
# start = -3, end = 10, step = 3
# numbers generated -> -3, 0, 3, 6, 9 (EXCLUDING 10)
```

7.2.4) The range function with 3 arguments, the last one being negative

```
range(start, end, step)   # but step is negative
```

If we pass in 3 arguments into the range function, the step argument (the last one) being negative, we can make the range function generate numbers backwards. But we need to make sure that *start* is larger than *end.*

```
range(5, 1, -1)
# start = 5, end = 1, step = -1
# numbers generated  -> 5, 4, 3, 2 (EXCLUDING 1)
```

```
range(5, 0, -1)
# start = 5, end = 0, step = -1
# numbers generated  -> 5, 4, 3, 2, 1 (EXCLUDING 0)
```

```
range(10, -1, -2)
# start = 10, end = -1, step = -2
# numbers generated  -> 10, 8, 6, 4, 2, 0 (EXCLUDING -1)
```

7.3) Using the range() function

Now that we know how the range() function behaves, let's see how we can put it into action.

```
for i in range(1, 10, 2):
    print(i)

# range(1, 10, 2)   -> start=1, end=10, step=2
# numbers generated -> 1, 3, 5, 7, 9 (EXCLUDING 10)
# iterator variable i takes on these generated values
```

Here, the range function generates the numbers 1 3 5 7 9. The variable that refers to these numbers is the iterator variable i. Here's a run-through of the above code.

```
value of i     action
1              print(1)
3              print(3)
5              print(5)
7              print(7)
9              print(9)
```

7.3.1) Doing more stuff with our iterator variable

```
for i in range(1, 6):
    square = i ** 2
    print(square)
```

We can add more lines of code into the for loop body too. For instance, in this above example, we aim to print the *squares* of numbers from 1 to 5.

```
# range(1, 6) generates numbers 1, 2, 3, 4, 5

i    square   action
1    1        1 is printed
2    4        4 is printed
3    9        9 is printed
4    16       16 is printed
5    25       25 is printed
```

7.4) The 'break' keyword in for loops

If we ever want to prematurely stop our while loop, we can use the 'break' keyword.

```
for i in range(1, 100):
    print(i)

    if i >= 3:
        break   # stop the for loop immediately
```

Notice that we have a 'break' keyword inside our for loop here (under the if block). The 'break' keyword forces the for loop to stop immediately, and is executed if the iterator variable *i* >= 3.

```
# range(1, 100) generates numbers 1 to 99

i   action    i>=5
1   print 1   False
2   print 2   False
3   print 3   True -> break
```

When our iterator variable i reaches 3, the expression 'i >= 3' evaluates to True, and the 'break' keyword is executed. The for loop is forcefully stopped at this point.

8) While Loops

Like for loops, while loops also allow us to repeat an action multiple times. However, while loops continue to run forever and ever until a certain condition is met.

8.1) Basic while loop syntax

```
while condition:
    # do something
```

Similar to for loops, the body of the while loop must be indented. The condition here is known as the *looping condition,* and the while loop will run forever and ever *as long as the looping condition is True.* Here's a basic example of a while loop in action:

```
number = 1
while number < 5:
    print(number)
    number += 1    # increase number by 1 (compound assignment operator)
```

Here, 'number < 5' is the *looping condition* of the while loop. As long as number is *less than 5,* the while loop will keep running forever. At every iteration:

1. number is printed
2. we increment number by 1
3. this brings the *looping condition* 1 step closer to becoming False

```
number   action     number<5    number+=1
1        print 1    True        number is now 2
2        print 2    True        number is now 3
3        print 3    True        number is now 4
4        print 4    True        number is now 5
5        print 5    False -> while loop stops
```

When writing while loops, we need to incrementally bring our *looping condition* one step closer to becoming False, or else our while loop will run forever and ever.

8.2) Infinite while loops and how to deal with them

```
while True:
    print("hello")
```

This here is an infinite while loop. This means that the while loop will run forever and ever. This is because of the looping condition – True will always remain as True, and will never become False.

If we run this, our program will repeatedly print "hello" again and again and again. It won't ever stop unless we forcefully stop it, and that is what we must do! To forcefully stop an infinite while loop, we need to hit CONTROL-C.

8.3) The 'break' keyword in while loops

Similarly to for loops, if we wish to prematurely stop our while loop, we can use the 'break' keyword. The 'break' keyword forcefully stops our while loop when executed.

```
n = 0
while True:
    print(n)
    n += 1

    if n > 3:
        break
```

Here, we have a seemingly infinite while loop (while True), but notice the conditional 'break' statement at the bottom. Here, if n > 3, 'break' is executed, and we forcefully get out of the while loop.

```
n   action     n+=1          n>3
0   print 0    n is now 1    False
1   print 1    n is now 2    False
2   print 2    n is now 3    False
3   print 3    n is now 4    True -> break
```

Notice how when n becomes 4, we forcefully break out of the while loop

8.4) When to use for loops vs while loops

If you need to repeat an action multiple times, depending on our exact task, it's better to use a for loop in some situations, and a while loop in other situations. As a rule of thumb:

8.4.1) When to use a FOR loop

- when we know for sure how many times we want our code to repeat
- when we want our code to run X times (and we know what X is)

8.4.2) When to use a WHILE loop

- when we DON'T know how many times we want our code to repeat
- when we want our code to stop ONLY if a certain condition is met

8.5) Looping condition VS breaking condition

Looping condition – the while loop will continue to run as long as this condition is True
Breaking condition – we break the while loop when this condition is True

```
i = 0
while i < 3:
    print(i)
    i += 1

# 0
# 1
# 2
```

```
i = 0
while True:
    print(i)
    i += 1
    if i > 2:
        break

# 0
# 1
# 2
```

These 2 blocks of code do exactly the same thing, and have exactly the same output. The first one uses a looping condition, while the second uses a breaking condition. Which method to use? Both are correct and this is up to your preference.

Checkpoint 3

1) Odd numbers from 1 to 100

Write a Python script that prints all odd numbers from 1 to 100. Do this first using a for loop, then a while loop.

```
# 1
# 3
# 5
# ...
# 99
```

2) Decreasing Numbers

Write a Python script that prints all numbers from 3 to -3 (both inclusive). Do this first using a for loop, then a while loop.

```
# 3
# 2
# 1
# 0
# -1
# -2
# -3
```

Answers (Checkpoint 3)

1) Odd numbers from 1 to 100

```
# for loop
for i in range(1, 101, 2):
    print(i)

# while loop
i = 1
while i < 100:
    print(i)
    i += 2
```

For the for loop, we use range(1, 101, 2) – start=1, end=101, step=2. The iterator variable i hence starts from 1 and increments by 2 at every step until it reaches 100 (101 itself is excluded).

For the while loop, we achieve the exact same logic but without using the range function.

2) Decreasing Numbers

```
# for loop
for i in range(3, -4, -1):
    print(i)

# while loop
i = 3
while i > -4:
    print(i)
    i -= 1
```

For the for loop, we use range(3, -4, -1) – start=3, end=-4, step=-1. The iterator variable i hence starts from 3, and increments by -1 at every step (basically decreases by 1) until it reaches -3 (-4 itself is excluded).

For the while loop, we achieve the exact same logic but without using the range function.

9) Strings

Previously, we've learnt that strings are a sequence of characters that we can think of as a 'word' or 'sentence'. In this chapter, we'll go into detail about strings.

9.1) Basic syntax – defining strings

```
x = "hello"
y = "my name is happy"
z = 'testing 123 !@#?'
```

When defining strings, we simply put the characters inside either 1) double inverted commas or 2) single inverted commas (both are completely fine)

9.2) String addition (concatenation) and multiplication

```
x = "apple " + "tree"
# x is now "apple tree"

y = "apple " + "tree " + "12345"
# 7 is now "apple tree 12345"
```

Concatenation is a fancy word for addition, and simply refers to adding 2 or more strings together using the '+' operator. Strings that are added together simply get joined one after another.

```
x = "a" * 5
# x is now "aaaaa"

y = "apple" * 3
# y is now "appleappleapple"
```

When we multiply a string by an integer n, the string simply gets repeated n times.

9.3) String indexing

String indexing allows us to extract single characters from a string.

```
x = "abcde"

print(x[0])   # a
print(x[1])   # b
print(x[2])   # c
print(x[3])   # d
print(x[4])   # e
```

9.3.1) Indexes in a string

In Python, we start counting from 0 (not 1). Characters inside a string are given a certain position, starting from 0.

- the 1st character in the string is at position 0
- the 2nd character in the string is at position 1
- the 3rd character in the string is at position 2
- the 4th character in the string is at position 3
- the 5th character in the string is at position 4
- etc

9.3.2) Indexing a string

```
x = "abcde"

# character "a" is at index 0
# character "b" is at index 1
# character "c" is at index 2
# character "d" is at index 3
# character "e" is at index 4
```

To index a string, we use this syntax:

```
first_letter = x[0]     # a   1st letter at position 0
second_letter = x[1]    # b   2nd letter at position 1
third_letter = x[2]     # c   3rd letter at position 2
fourth_letter = x[3]    # d   4th letter at position 3
fifth_letter = x[4]     # e   5th letter at position 4
```

9.3.3) Index out of range error

If we try to index a string with an index that does not exist, we get an index out of range error. For instance:

```
x = "abcde"
letter = x[5]
```

Here, the last letter of the string is "e" at index 4, but we are trying to index 5 instead, which does not exist. As such, the line 'x[5]' will cause an index out of range error

```
Traceback (most recent call last):
  File "/some/path/a.py", line 2, in <module>
    letter = x[5]
IndexError: string index out of range
```

More examples of code that causes an index out of range error:

```
x = "apple"
letter = x[10]   # error as maximum index of "apple" is 4

x = "orange"
letter = x[6]    # error as maximum index of "apple" is 5

x = "a"
letter = x[1]    # error as maximum index of "apple" is 0
```

9.3.4) Negative indexes

When we index strings normally, we start counting from the left. However, if we wish to start counting from the right of the string instead, we can use negative indexes.

```
x = "abcde"

minus1 = x[-1]   # "e"   last letter
minus2 = x[-2]   # "d"   second last letter
minus3 = x[-3]   # "c"   third last letter
minus4 = x[-4]   # "b"   fourth last letter
minus5 = x[-5]   # "a"   fifth last letter
```

Note that if we try indexing x[-6], and index out of range error will occur.

9.4) String slicing

While string indexing lets us extract single characters, string slicing allows us to extract a sub-sequence within a string (multiple characters).

NOTE – string slicing uses the same indexing system as string indexing – the first letter is at index 0, the second letter is at index 1, the third letter is at index 2 and so on.

9.4.1) Basic string slicing syntax

```
string[start:end]
```

NOTE – like the range function, the *end* here is EXCLUDED.

```
x = "abcdefg"
slice = x[1:4]    # performing string slicing

# slice is now "bcd"
```

Here, we do a slice from index 1 to index 4 (EXCLUDING index 4 itself). Effectively, our slice will contain the letters at indexes 1, 2 and 3 (EXCLUDING 4 once again), and hence returns "bcd". A couple more examples:

```
x = "abcdefg"
slice1 = x[0:3]   # "abc"    (characters at indexes 0, 1, 2)
slice2 = x[3:6]   # "def"    (characters at indexes 3, 4, 5)
slice3 = x[5:7]   # "fg"     (characters at indexes 5, 6)
slice3 = x[5:9]   # "fg"     (characters at indexes 5, 6)

x = "i am happy"
slice1 = x[0:3]   # "i a"     (characters at index 0, 1, 2)
slice2 = x[0:4]   # "i am"    (characters at index 0, 1, 2, 3)
slice3 = x[5:10]  # "happy"   (characters at index 5, 6, 7, 8, 9)
```

NOTE – string slicing does not cause index out of range errors. Indexes that are out of range are simply ignored.

9.4.2) String slicing with empty arguments

```
string[:end]      # first character to end (EXCLUDING end)
string[start:]    # start to last character
string[:]         # the entire string
```

If we leave the *start* argument empty, we are essentially slicing from the absolute start, and if we leave the *end* argument empty, we are slicing to the absolute end.

```
x = "abcdefg"
slice1 = x[:2]    # "ab"        (indexes 0,1)
slice2 = x[:4]    # "abcd"      (indexes 0,1,2,3)
slice3 = x[:6]    # "abcdef"    (indexes 0,1,2,3,4,5)

slice4 = x[2:]    # "cdefg"     (indexes 2,3,4,5,6)
slice5 = x[4:]    # "efg"       (indexes 4,5,6)
slice6 = x[6:]    # "g"         (indexes 6)
```

9.4.3) String slicing with negative arguments

Yes, like string indexing, we can use negative indexes too to signify that we should start counting from the right of the string.

```
x = "abcdefg"
slice1 = x[-4:-1]   # "def"   (indexes -4,-3,-2)
slice2 = x[-6:-3]   # "bcd"   (indexes -6,-5,-4)
```

We can combine positive and negative indexes too.

```
x = "abcdefg"
slice3 = x[3:-1]   # "def"  (indexes 3, 4, 5)
slice3 = x[1:-2]   # "bcde" (indexes 1, 2, 3, 4)
```

To work out which index the negative integer is referring to, we can simply add the length of the string to the negative integer.

```
x = "abcdefg"
slice3 = x[3:-1]   # SAME AS x[3:6]   as 7-1=6
slice3 = x[1:-2]   # SAME AS x[1:5]   as 7-2= 5
```

9.4.4) String slicing with 3 arguments

```
string[start:end:step]
```

Previously, we've only dealt 2-argument string slicing. If we add a 3rd argument, we are specifying the *step* – how much to increase our index by at each iteration. If we choose not to specify the *step* argument, it defaults to 1.

```
x = "abcdefg"
slice1 = x[0:7:2]   # "aceg"   (indexes 0,2,4,6)
slice2 = x[0:7:3]   # "adg"    (indexes 0,3,6)
slice3 = x[1:7:2]   # "bdf"    (indexes 1,3,5)
```

9.4.5) String slicing with 3 arguments + Empty arguments

Similarly, if we leave the *start* or *end* empty, we are essentially telling Python to slice from the *absolute start* or the *absolute end* of the string.

```
x = "abcdefg"
slice1 = x[:5:2]    # "ace"    (indexes 0,2,4)
slice2 = x[3::2]    # "df"     (indexes 3,5)
slice3 = x[::2]     # "aceg"   (indexes 0,2,4,6)
```

Note that if we leave *step* empty, it defaults to 1

9.4.6) String slicing with 3 arguments + Negative step

When we make our *step* argument negative, we slice the string backwards

```
x = "abcdefg"
slice1 = x[5:1:-1]   # "fedc"    (indexes 5,4,3,2)
slice2 = x[5:0:-2]   # "fdb"     (indexes 5,3,1)
slice3 = x[::-1]     # "gfedcba" (indexes 6,5,4,3,2,1,0)
slice4 = x[::-2]     # "geca"    (indexes 6,4,2,0)
```

Note that if 1) our step is negative and 2) we leave either start or end empty, the start automatically refers to the end of the string, while the end refers to the start of the string (they are reversed!)

9.5) Common string functions and methods

9.5.1) len(string) – finding the length of the string

```
x = "abcdefg"
length = len(x)   # 7
```

Here, the length of the string simply refers to the number of characters inside the string.

9.5.2) string.upper() – converting to uppercase

```
string = "hello world!"
x = string.upper()   # "HELLO WORLD!"
```

This method essentially converts the entire string into uppercase (letters are converted, and everything else remains the same)

9.5.3) string.lower() – converting to lowercase

```
string = "HELLO world 12345"
x = string.lower()    # "hello world 12345"
```

This method converts the entire string in lowercase (letters are converted, everything else remains the same)

9.5.4) string.strip() – removing whitespace characters at both ends

```
string = "   hello world, it's me   \n   "
x = string.strip()   # "hello world it's me"
```

This method removes all whitespace characters (spaces, newlines, tabs) at the start and end of the string. Whitespace characters in the middle of the string remain untouched.

9.5.5) string.count() – counting a character in the string

```
string = "hello world"
x = string.count("o")   # 2
```

This method takes in a substring, and counts the number of times the substring appears inside the string. Here, the substring "o" appears inside the string twice, so 2 is returned.

9.5.6) string.find() – finding index of character in string

```
string = "hello world"
x = string.find("o")   # 4
```

This method takes in a substring, and returns the first-found position of the substring. Here, the substring "o" is at index 4, so 4 is returned.

```
string = "hello world"
x = string.find("z")   # -1
```

If the substring cannot be found, -1 is returned.

9.5.7) string.rfind() – finding index of character in string from behind

```
string = "hello world"
x = string.rfind("o")   # 7
```

This method is the same as .find(substring), just that it returns the first-found position of the substring from the end of the string. Here, the last index of the substring "o" is 7, so 7 is returned. If the substring cannot be found, -1 is returned.

9.5.8) string.replace() – replacing pattern with another pattern

```
string = "apple"
x = string.replace("a", "b")   # "bpple"
```

This method takes in 2 strings, and replaces all instances of the first string with the second string. In this case, our function helps to replace all "a"s with "b"s.

9.6) Common string boolean methods

Here are some commonly used string methods that return a boolean value (True or False).

9.6.1) substring in string – check if string contains substring

```
"a" in "apple"     # True
"b" in "apple"     # False

"app" in "apple"   # True
"ppa" in "apple"   # False (order is not correct)
```

The 'in' keyword here checks whether a string contains a substring (the order must be correct), and returns True if it does and False otherwise

9.6.2) string.isupper() – check if all letters are uppercase

```
"APPLE".isupper()      # True
"APPLE 123".isupper()  # True

"Apple".isupper()      # False
"APPLe".isupper()      # False
```

The .isupper() method returns True if *all* letters inside the string are uppercase, and returns False if even *one* letter is lowercase.

9.6.3) string.islower() – check if all letters are lowercase

```
"apple".islower()       # True
"apple 123".islower()   # True

"Apple".islower()       # False
"APPLe".islower()       # False
```

The .islower() method returns True if *all* letters inside the string are lowercase, and returns False is *even one* letter is uppercase.

9.6.4) string.isalpha() – check if string is all alphabets

```
"abcde".isalpha()      # True
"abcde ".isalpha()     # False (cus of space)
"abcde1".isalpha()     # False (cus of number)
```

The .isalpha() method returns True if every character in a string is an alphabet, and returns False if *even one* character is not an alphabet.

9.6.5) string.isnumeric() – check if string is numeric

```
"12345".isnumeric()    # True
"123.45".isnumeric()   # False (cus of .)
"12345e".isnumeric()   # False (cus of letter)
```

The .isnumeric() method returns True if every character in the string is a number, and returns False if *even one* character is not a number.

9.6.6) string.isalnum() – check is string is alphanumeric

```
"abc123".isalnum()   # True
"abc 123".isalnum()   # False   (cus of space)
"abc.123".isalnum()   # False   (cus of .)
```

The .isalnum() method is a combination of the .isnumeric() method and the .isalpha() method, and it returns True if every character in the string is alphanumeric (either a letter or a number), and False otherwise.

9.7) Whitespace and escape characters

Escape characters refer to special characters in strings, and usually start with a backslash \ (NOT a normal slash /). Here are some escape characters that we need to be aware of:

```
" "     # normal whitespace character
"\n"    # newline character
"\t"    # tab character
"\'"    # single quote character
"\""    # double quote character
"\\"    # backslash character
```

9.7.1) whitespace character " "

```
print("apple orange pear")
```

The output:

```
apple orange pear
```

The whitespace character " " is a character on its own, and we use the spacebar to type it.

9.7.2) newline character \n

```
print("apple\norange\npear")
```

The output:

```
apple
orange
pear
```

The newline character \n forces the next character to start from a new line.

9.7.3) tab character \t

```
print("a\tb\tc")
```

The output:

```
a	b	c
```

9.7.4) single quote character \'

```
print('I don\'t like shellfish')
```

The output:

```
I don't like shellfish
```

If we use single quotes to surround our string, and wish to have single quote characters inside the string itself, we need to use the escape character \' to represent a single quote.

9.7.5) double quote character \"

```
print("My name is \"rocky\"")
```

The output:

```
My name is "rocky"
```

Similarly, if we use double quotes to surround our string, and wish for our string to contain double quote characters, we need to make use of the escape character \".

9.7.6) backslash character \\

```
print("This is ONE backslash \\")
```

The output:

```
This is ONE backslash \
```

As the backslash character is used to escape certain characters, we need to use double backslashes \\ to represent 1 backslash.

9.8) Formatted strings

The usual way of printing variables in strings:

```
name = "bob"
age = 50
wife = "mary"
num_children = 3

string = "hello my name is " + name + ", I'm " + str(age) + " this year,
my wife is " + wife + " and we have " + str(num_children) + " children."

print(string)
```

The output:

```
hello my name is bob, I'm 50 this year, my wife is mary and we have 3
children.
```

However, if we don't wish to write so many + characters and convert so many numbers to strings, we're probably better off using a formatted string.

9.8.1) Basic formatted string (f-string) syntax

```
age = 20
string = f"I am {age} years old"
```

1. We need to put an 'f' in front of the string. This signifies that it is a formatted string.
2. We can put variables inside the string itself, within { } brackets.
3. The variables we add into our formatted string can be of ANY data type

The output:

```
I am 20 years old
```

9.8.2) More complicated f-string example

```
name = "bob"
age = 50
wife = "mary"
num_children = 3

string = f"hello my name is {name}, I'm {age} this year, my wife is
{wife} and we have {num_children} children."

print(string)
```

The output:

```
hello my name is bob, I'm 50 this year, my wife is mary and we have 3
children.
```

Note that in f-strings, variables can be any data type, and we don't have to deal with type casting and conversion. That's one headache out of the way!

9.8.3) The string.format() method

```
string = "We can use {0} and {1} many times! {0} {1} {0} {1}"
string = string.format("apple", "orange")

print(string)
```

If certain variables appear multiple times, we can also choose to use the string.format method. Here in the .format method, "apple" is the first argument, while "orange" is the second argument.

In the string itself, {0} refers to the first argument "apple", while {1} refers to the second argument "orange". Here's what the output looks like:

```
We can use apple and orange many times! apple orange apple orange
```

Do make sure that there is a corresponding argument for each {number} that you use, or Python will raise an error!

9.9) Iterating through strings

9.9.1) Iterating through a string itself

```
string = "apple"
for letter in string:
    print(letter)
```

The output:

```
a
p
p
l
e
```

A string is an iterable, which means that we can loop through it using a for loop. Here, if we loop through our string, the iterator variable *letter* essentially takes the value of every single character inside the string.

```
letter   action
"a"      print "a"
"p"      print "p"
"p"      print "p"
"l"      print "l"
"e"      print "e"
```

9.9.2) Iterating through a string using the range function

```
string = "apple"
for i in range(len(string)):
    letter = string[i]
    print(letter)
```

The output:

```
a
p
p
l
e
```

Here, instead of iterating through the string itself, we use the range function to iterate and indx the string.

```
len(string) is 5 ("apple" has 5 letters)
range(5) generates the numbers 0, 1, 2, 3, 4
iterator variable i hence takes the values 0, 1, 2, 3, 4

i    string[i]   action
0    "a"         "a" is printed
1    "p"         "p" is printed
2    "p"         "p" is printed
3    "l"         "l" is printed
4    "e"         "e" is printed
```

While this might be a much longer way to iterate through a string, it has its merits.

9.9.3) Iterating directly through a string VS using range function

```
# iterating directly through a string
for letter in string:
    print(letter)
```

```
# iterating through string using the range function
for i in range(len(string)):
    letter = string[i]
    print(letter)
```

In usual cases, we'll probably use the first method (iterating directly through a string) as it is shorter and we have to write less code. However, we need to use the second method (iterating through the string using the range function) if we need to know the index of each character in the string.

For instance, let's say we wish to print every group of 2 characters in the string.

```
string = "apple"
for i in range(len(string)-1):
    print(string[0], string[1])
```

The output:

```
a p
p p
p l
l e
```

How this works:

```
len(string) - 1   # 4
range(4) generates the numbers 0, 1, 2, 3

i    string[0]   string[1]
0    a           p
1    p           p
2    p           l
3    l           e
```

Since we need the index at each iteration, we cannot achieve this if we iterate through the string directly using 'for letter in string'.

9.10) Adding a letter to a string

```
string = "abcd"
string += "e"

# string is now "abcde"
```

Here, we have an existing string "abcd", and we wish to add something after it eg. "e". We can achieve this by using the += compound assignment operator. This is the same as:

```
string = "abcd"
string = string + "e"

# string is now "abcde"
```

Note that we are actually creating a new string "abcde", and reassigning the variable *string* to our new string.

9.11) The string.split() method

The string.split() method is used to separate a string into a list of strings. (Don't worry – we're covering lists next)

```
string = "apple orange pear"
lis = string.split()        # lis is ["apple", "orange", "pear"]

string2 = "apple,orange,pear"
lis2 = string2.split(",")   # lis2 is ["apple", "orange", "pear"]
```

By default, the .split() method separates the string by the whitespace character. However, we can specify the separator, and make the .split() method separate a string using other characters eg. comma.

10) Lists

In Python, lists are a built-in data structure that can store multiple items.

10.1) Basic syntax – defining lists

```
list0 = [ ]                     # empty list
list1 = [1, 2, 3]               # list of integers
list2 = [1.0, 2.5, 3.14159]     # list of floats
list3 = ["apple", "orange"]     # list of strings
list4 = [1, "apple", True]      # list of mixed data types
```

We can define lists using square brackets [ ] – we need to separate the elements inside a list using commas. Note that elements inside a list can be of ANY data type.

10.2) List addition (concatenation) and multiplication

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]
list3 = list1 + list2

# list3 will be [1, 2, 3, 4, 5, 6]
```

We can use the + operator to join (concatenate) 2 lists together. Like strings, the 2 lists simply add together one after another to form a longer list.

```
list1 = [1, 2]
list2 = list1 * 3

# list2 = [1, 2, 1, 2, 1, 2]
```

When we multiply a list with an integer using the * operator, we essentially repeat the list multiple times.

10.3) List indexing

List indexing behaves kinda the same way as string indexing.

10.3.1) Indexes in a list

```
list1 = ["apple", "orange", "pear", "durian"]
```

Similarly to strings, we start counting from 0 instead of 1.

- the 1st element "apple" is at index 0
- the 2nd element "orange" is at index 1
- the 3rd element "pear" is at index 2
- the 4th element "durian" is at index 3

10.3.2) Indexing a list

```
list1 = ["apple", "orange", "pear", "durian"]

list1[0]   # "apple"
list1[1]   # "orange"
list1[2]   # "pear"
list1[3]   # "durian"
```

To index a list, we add square brackets after our list variable (same as string indexing).

10.3.3) Index out of range error

```
list1 = ["apple", "orange", "pear", "durian"]
list1[5]   # index out of range error
```

Similarly, if we try to index an index that does not exist, we get an error

10.3.4) Negative indexing

```
list1 = ["apple", "orange", "pear", "durian"]

list1[-1]   # "durian"
list1[-2]   # "pear"
```

Similarly to strings, we can use negative integers as indexes if we wish to count from the back of the list.

10.4) List slicing

List slicing behaves the same way as string slicing.

10.4.1) Basic list slicing

```
list[start:end]
```

Similar to string slicing, we can specify a *start* and an *end*, and every character from start to end (EXCLUDING end itself) will be returned in our slice.

```
lis = ["apple", "orange", "pear", "durian", "pineapple"]
slice = lis[1:4]   # ["orange", "pear", "durian"]
```

10.4.1) List slicing with empty arguments

```
list[:end]     # first element to end (EXCLUDING end)
list[start:]   # start to last element
list[:]        # the entire list
```

If we leave the *start* argument empty, we are essentially slicing from the absolute start, and if we leave the *end* argument empty, we are slicing to the absolute end.

```
lis = ["apple", "orange", "pear", "pineapple", "durian"]
slice1 = lis[:2]   # ["apple", "orange"]
slice2 = lis[:3]   # ["apple", "orange", "pear"]

slice3 = lis[2:]   # ["pear", "pineapple", "durian"]
slice4 = lis[3:]   # ["pineapple", "durian"]
```

10.4.1) List slicing with negative arguments

Slicing lists with negative elements is similar to that of strings.

```
lis = ["apple", "orange", "pear", "pineapple", "durian"]
slice1 = lis[-3:-1]   # ["pear", "pineapple"]
slice2 = lis[-4:-1]   # ["orange", "pear", "pineapple"]
```

10.4.1) List slicing with 3 arguments

```
list[start:end:step]
```

If we specify the *step* argument, we essentially specify how much to increase our index by at each iteration. Once again, if we leave step empty, it defaults to 1.

```
lis = ["apple", "orange", "pear", "pineapple", "durian"]
slice1 = lis[0:5:2]   # ["apple", "pear", "durian"]
slice2 = lis[1:5:2]   # ["orange", "pineapple"]
slice3 = lis[0:5:3]   # ["apple", "pineapple"]
```

10.4.1) List slicing with 3 arguments + Empty arguments

```
lis = ["apple", "orange", "pear", "pineapple", "durian"]
slice1 = lis[:4:2]   # ["apple", "pear"]
slice2 = lis[1::2]   # ["orange", "pineapple"]
slice3 = lis[::2]    # ["apple", "pear", "durian"]
```

If we leave *start* empty, we essentially slice from the absolute start. If we leave *end* empty, we slice from the absolute end. If we leave both *start* and *end* empty, we slice from the absolute start to the absolute end.

10.4.1) List slicing with 3 arguments + Negative step

When we use a negative *step* argument, we essentially slice the string backwards.

```
lis = ["apple", "orange", "pear", "pineapple"]
slice1 = lis[::-1]   # ["pineapple", "pear", "orange", "apple"]
slice1 = lis[::-2]   # ["pineapple", "orange"]
```

10.5) Adding new elements to a list

```
lis = ["apple", "orange", "pear"]
lis.append("pineapple")

# lis is now ["apple", "orange", "pear", "pineapple"]
```

We can add a new element to an existing list using the *.append(element)* method.

- Note that the element we wish to add can be of ANY data type.
- Note also that we don't need the assignment operator = here.
- Note also that we can only append ONE element at a time

```
lis = []   # empty list
lis.append("apple")
lis.append("guava")
lis.append(12345)

# lis is now ["apple", "guava", 12345]
```

10.6) Replacing existing element in a list

```
existing_list[index_to_replace] = new_element
```

Let's say that we have an existing list, and we wish to replace the element at a certain index (say index 0). We can do that using this syntax:

```
lis = ["apple", "orange", "pear"]
lis[0] = "pineapple"   # set index 0's value as "pineapple"

# lis is now ["pineapple", "orange", "pear"]
```

Note that the old value "apple" is replaced by "pineapple".

10.7) Common List Functions & Methods

10.7.1) len(list) – finding the length of a list

```
len([])                          # 0
len([3,4])                       # 2
len(["apple", "orange", "pear"])  # 3
```

The length of a list is the number of elements it contains. To find its length, we can use the built-in len() method.

10.7.2) element in list – checking if element exists in list

```
lis = ["apple", "orange", "pear"]
"apple" in lis      # True
"banana" in lis     # False
```

We can use the 'in' keyword to check if a list contains an element. This returns True if the list contains the element, and returns False otherwise.

10.7.3) list.index(element) – finding index of element

```
lis = ["apple", "orange", "pear", "orange", "pear"]
lis.index("apple")    # 0
lis.index("orange")   # 1 (first-found index)
lis.index("banana")   # ValueError as "banana" cannot be found
```

We can use the .index(element) method to find the *first-found* index of an element, but note that a ValueError will be raised if we pass it an element that does not exist in the list.

10.7.4) list.count(element) – counting occurrence of element in list

```
lis = ["apple", "orange", "pear", "orange", "pear"]
lis.count("apple")    # 1
lis.count("orange")   # 2
lis.count("banana")   # 0
```

We can use the .count(element) method to count how many times an element appears inside the list.

10.8) Iterating through lists

10.8.1) Iterating through the list directly

```
lis = ["apple", "orange", "pear"]
for element in lis:
    print(element)
```

Here, the iterator variable *element* is assigned to every element in the list.

```
element     action
"apple"     print("apple")
"orange"    print("orange")
"pear"      print("pear")
```

10.8.2) Iterating using range(len(list))

```
lis = ["apple", "orange", "pear"]
for i in range(len(lis)):
    element = lis[i]
    print(element)
```

Here, the iterator variable *i* is assigned to the index of every element in the list. We then use this iterator variable to index every element, and print them.

```
i   element     action
0   "apple"     print("apple")
1   "orange"    print("orange")
2   "pear"      print("pear")
```

We use this way over the first if we need to access the index of the elements.

Checkpoint 4

1) Extracting the email name

Write a function *getname(email)* that takes in a string *email.* Assume that *email* will always be in a 'name@domain.com' format. This function returns the email name, or the string before the '@' character in the email.

```
getname("hello@gmail.com")                # hello
getname("pythonprogrammer@gmail.com")     # pythonprogrammer
getname("hellowaorld@googlemail.com")     # helloworld
```

2) String Slicing

We are given this string:

```
string = "abcdefg"
```

Using just string slicing, get the following string:

```
"aceg"
```

3) List Slicing

We are given a list:

```
lis = ["apple", "orange", "pear", "pineapple", "banana", "durian"]
```

Using just list slicing, get the following list:

```
["durian", "pineapple", "orange"]
```

Answers (Checkpoint 4)

1) Extracting the email name

```python
# method 1
def getname(email):
    lis = email.split("@")
    return lis[0]

# method 2
def getname(email):
    at_index = email.find("@")
    return email[:at_index]
```

In method 1, we use the built-in .split() function to separate the email by the "@" character. The first element in this new list will be the email name, and we simply access it using list indexing.

In method 2, we find the first index of the "@" character, and use string slicing to get every character up til the "@" character.

2) String Slicing

```python
string = "abcdefg"
new = string[::2]

# "aceg"
```

Here, we use empty start and end arguments to signify that we wish to slice the entire string, and as step=2, the index increments by 2 each time.

3) List Slicing

```python
lis = ["apple", "orange", "pear", "pineapple", "banana", "durian"]
new = lis[::-2]

# ["durian", "pineapple", "orange"]
```

Similarly, here we use empty start and end arguments to signify that we wish to slice the entire list. As we specify *step* to be -2, we start slicing from the end of the string, and the index decreases by 2 each time.

11) Dictionaries

In Python, dictionaries are a built-in data structure that can store multiple key-value pairs.

11.1) Basic syntax – defining dictionaries

```python
d0 = { }              # empty dictionary
d1 = {1:2, 3:4, 5:6}
d2 = {"apple":4, "orange":5}
```

Some rules when defining dictionaries

1. We can define dictionaries using curly brackets { }
2. We separate key-value pairs from one another using commas ,
3. We must have a colon : between each key and value
4. Each key must have a value, and each value must have a key
5. Keys in dictionary must be unique

11.1.1) Example 1

```python
d1 = {1:2, 3:4, 5:6}
```

Here, we have 3 key-value pairs (separated by commas)

- 1:2 – key is 1, and corresponding value is 2
- 3:4 – key is 3, and corresponding value is 4
- 5:6 – key is 5, and corresponding value is 6

11.1.2) Example 2

```python
d2 = {"apple":4, "orange":5}
```

Here, we have 2 key-value pairs (separated by commas)

- "apple":4 – key is "apple", and the corresponding value is 4
- "orange":5 – key is "orange", and the corresponding value is 5

11.1.3) Negative example

```python
d3 = {"apple":4, "orange":5, "apple":6}
```

Remember that keys in a dictionary MUST be unique. Here, we have 2 "apple" keys, so the second key overrides the first key, and the "apple":4 key-value pair is lost. There will be a total of 2 key-value pairs

- "apple":6 – key is "apple", and corresponding value is 6
- "orange":5 – key is "orange", and corresponding value is 5

11.2) Validity of dictionary keys/values

11.2.1) Concept of Mutable and Immutable data types

IMMUTABLE data types:

1. integers
2. floats
3. strings
4. tuples
5. boolean values
6. None

MUTABLE data types:

1. lists
2. dictionaries
3. sets

Immutable values CANNOT be changed in any way after we create them, while mutable values can be changed after we create them

11.2.2) Data types that can be dictionary keys

As a rule of thumb, only IMMUTABLE data types can be used as dictionary keys:

1. integers
2. floats
3. strings
4. boolean
5. None
6. tuples (we'll cover this later on)

Mutable data types CANNOT be used as dictionary keys:

1. lists
2. dictionaries
3. sets

11.2.2) Data types that can be dictionary values

Every single data type in Python can be a dictionary value – no restrictions here.

11.3) Accessing values in dictionary using keys

```
value = dictionary[key]
```

Let's say we have an existing dictionary, and we want to access a certain value. Here, we need to use the key to access the value using this syntax:

```
d = {"apple":4, "orange":5, "pear":6}
x = d["apple"]    # 4
y = d["orange"]   # 5
z = d["pear"]     # 6
```

Note that if we try to access a key that doesn't exist, we get a KeyError

```
d = {"apple":4, "orange":5, "pear":6}
x = d["pineapple"]

# KeyError occurs
```

11.4) Adding new key-value pairs into an existing dictionary

```
dictionary[newkey] = newvalue
```

Let's say we have an existing dictionary, and wish to add a new key-value pair to it. We can achieve this using the above syntax.

```
d = {"apple":4, "orange":5, "pear":6}
d["pineapple"] = 7

# d is now {"apple":4, "orange":5, "pear":6, "pineapple":7}
```

11.5) Replacing existing key-value pair in dictionary

```
dictionary[oldkey] = newvalue
```

We can use the same syntax to replace an existing key-value pair.

```
d = {"apple":4, "orange":5, "pear":6}
d["apple"] = 100

# d is now {"apple":100, "orange":5, "pear":6}

d["orange"] = 4000

# d is now {"apple":100, "orange":4000, "pear":6}
```

11.6) Updating existing value in dictionary

```
d = {"apple":4, "orange":5, "pear":6}
```

Let's say we want to increment the value of "apple" by 1. We can do it this way:

```
d["apple"] = d["apple"] + 1
```

Or we can use the compound assignment operator += (this does the exact same thing):

```
d["apple"] += 1
```

A working example:

```
d = {"apple":4, "orange":5, "pear":6}

d["apple"] += 1    # d is now {"apple":5, "orange":5, "pear":6}
d["apple"] += 1    # d is now {"apple":4, "orange":5, "pear":6}
d["orange"] += 10  # d is now {"apple":5, "orange":15, "pear":6}
d["pear"] += 100   # d is now {"apple":5, "orange":15, "pear":106}
```

11.7) Common dictionary methods and functions

11.7.1) len(dict) – finding length of a dictionary

```
len({})                                  # 0
len({"apple":4})                         # 1
len({"apple":4, "orange":5})             # 2
len({"apple":4, "orange":5, "pear":6})   # 3
```

The length of a dictionary is the number of key-value pairs it contains. We can use the built-in len() method to find its length.

11.7.2) key in dict – checking if key exists in dictionary

```
d = {"apple":4, "orange":5, "pear":6}

"apple" in d      # True
"banana" in  d    # False
4 in d            # False (only checks keys)
5 in d            # False (only checks keys)
```

We can use the 'in' keyword to check if a dictionary contains a key. Note that this syntax above only works for KEYS and not values.

11.7.3) value in dict.values() – checking if value exists in dictionary

```
d = {"apple":4, "orange":5, "pear":6}

4 in d.values()   # True
5 in d.values()   # True
7 in d.values()   # False
```

To check if a value exists in a dictionary, we need to use the dict.values() method.

11.7.4) dict.keys() – getting dictionary keys in a list

```
d = {"apple":4, "orange":5, "pear":6}
keys = list(d.keys())

# keys is ["apple", "orange", "pear"]
```

To extract the dictionary keys into a list, we can use the dict.keys() method. This method itself returns a 'dict_keys' data type, so do remember to convert it into a list.

11.7.5) dict.values() – getting dictionary values in a list

```
d = {"apple":4, "orange":5, "pear":6}
keys = list(d.values())

# keys is [4, 5, 6]
```

To extract the dictionary values into a list, we can use the dict.values() method. Similarly, this method returns a 'dict_values' data type, so do remember to convert it into a list.

11.7.6) dict.items() – getting key-value pairs in a list

```
d = {"apple":4, "orange":5, "pear":6}
keys = list(d.items())

# keys is [("apple",4), ("orange",5), ("pear", 6)]
```

To extract both keys and values together, we can use the dict.items() method. This method returns a list of tuples (we'll cover this in a bit) representing each key-value pairs. Each tuple contains 1 key and 1 value.

11.8) Iterating through dictionaries

```python
d = {"apple":4, "orange":5, "pear":6}

for key in d:
    value = d[key]
    print(key, value)

# apple 4
# orange 5
# pear 6
```

We can use a simple for loop to iterate through the dictionary itself – but note that this gives us the KEYS inside the dictionary. We can then use the keys to access the values through indexing inside the for loop.

```python
d = {"apple":4, "orange":5, "pear":6}

for key, value in d.items():
    print(key, value)

# apple 4
# orange 5
# pear 6
```

Alternatively, we can use the dict.items() method to generate both keys and values at one go. This does the exact same thing as the previous example – we simply save one line of code.

12) Tuples

Tuples are essentially IMMUTABLE lists – lists that cannot be changed after we create them.

12.1) Basic syntax – defining tuples

```
tup = ()                           # empty tuple
tup = ("apple",)                   # tuple of length 1
tup = ("apple", "orange")          # tuple of length 2

tup = (1, 2, 3)                    # tuple of integers
tup = (1.11, 2.22, 3.33)           # tuple of integers
tup = ("apple", "orange", "pear")  # tuple of strings
```

We define tuples using normal brackets ( ) instead of square brackets (used for lists), and we separate elements inside a tuple using commas. Similarly to lists, a tuple can contain elements of ANY type.

12.1.1) Defining tuples without brackets

```
tup = 1,          # tuple of length 1 (1,)
tup = 1, 2        # tuple of length 2 (1, 2)
tup = 1, 2, 3     # tuple of length 1 (1, 2, 3)
```

We can also define tuples without using brackets if we want to.

12.2) Why we even use tuples

But what's the point of using tuples if they are simply lists that cannot be changed?

12.2.1) Immutability – tuples can be used as dictionary keys

Tuples are IMMUTABLE, which means we cannot change it after we create it.

```
tup = ("apple", "orange", "pear")
tup.append("pineapple")     # ERROR
```

Lists cannot be used as dictionary keys because lists are mutable. Conversely, tuples CAN be used as dictionary keys, because tuples are IMMUTABLE.

```
d = {(1,2):"apple", (3,4):"orange", (5,6):"pear"}
```

^ a valid dictionary with tuples as keys.

12.2.2) Tuple unpacking

Tuple unpacking enables us to assign multiple variables to multiple values in ONE line.

```
a, b = 4, 5
# a is 4, and b is 5

a, b, c, d = 6, 7, 8, 9
# a=6, b=7, c=8, d=9
```

Note that the number of variables on the left and right must match, or else we get an error.

```
a, b, c = 4, 5
# ValueError: not enough values to unpack (expected 3, got 2)

a, b, c = 4, 5, 6, 7
# ValueError: too many values to unpack (expected 3)
```

12.3) Common Tuple operations

12.3.1) Tuple indexing

Exactly the same as lists – the first element is at index 0, the second at index 1 and so on.

```
tup = ("apple", "orange", "pear")

x = tup[0]   # "apple"
y = tup[1]   # "orange"
z = tup[2]   # "pear"
```

12.3.2) Tuple addition & multiplication

Addition and multiplication works exactly the same as lists

```
x = ("apple", "orange") + ("pear", "pineapple")
# x is ("apple", "orange", "pear", "pineapple")

y = ("apple", "orange") * 3
# y is ("apple", "orange", "apple", "orange", "apple", "orange")
```

12.3.3) Tuple slicing

This works exactly the same way as lists and strings.

12.3.4) Adding a new element to a tuple

As a tuple is IMMUTABLE, we cannot mutate the tuple itself. However, we can create another tuple containing the new element, and reassign our variable to it.

```
tup = ("apple", "orange")
tup += ("pear",)

# tup will be ("apple", "orange", "pear")
```

Here, we are not mutating our tuple at all – we are creating an entirely new tuple ("apple", orange", "pear") and assigning the variable tup to this new tuple.

12.3.5) len(tuple) – finding length of tuple

This works exactly the same way as lists – we use the len() function to find the length, which refers to the number of elements inside the tuple.

```
tup = ("apple", "orange", "pear")
x = len(tup)      # 3
```

12.3.6) element in tuple – checking if tuple contains element

This works exactly the same way as lists.

```
tup = ("apple", "orange", "pear")

"apple" in  tup        # True
"pineapple" in tup     # False
```

12.4) Iterating through tuples

Same as lists – we can either iterate through the tuple directly, or use range(len(tuple)).

```
tup = ("apple", "orange", "pear")

for fruit in tup:
    print(fruit)

# apple
# orange
# pear

for i in range(len(tup)):
    fruit = tup[i]
    print(i, fruit)

# 0 apple
# 1 orange
# 2 pear
```

13) Sets

In Python, sets are unordered collections that can only contain unique values.

13.1) Basic syntax – defining sets

```
s1 = set()                          # empty set
s2 = {1, 2, 3}                      # set containing 3 integers
s3 = {"apple", "orange", "pear"}    # set containing 3 strings
```

Notice that we use the same curly brackets as dictionaries { }, but there are no colons that separate key-value pairs. This is because there are only values in sets, no keys.

13.2) Valid set values

Only IMMUTABLE values can be added inside a set – int, float, bool, str, tuple etc. Mutable values (lists, dictionaries, sets) CANNOT be added inside a set.

13.3) Adding new values into a set

We can use the .add(element) method to add a new value into a set.

```
s = {"apple"}

s.add("orange")      # s is now {"apple", "orange"}
s.add("pear")        # s is now {"apple", "orange", "pear"}
s.add("pineapple")   # s is now {"apple", "orange", "pear", "pineapple"}
```

13.3.1) Adding duplicate values into a set

A set cannot contain duplicate values. If we try to add a duplicate value into a set, nothing happens and our set remains the same.

```
s = {1, 2, 3}

s.add(4)   # s is now {1, 2, 3, 4}
s.add(2)   # s is still {1, 2, 3, 4}
s.add(4)   # s is still {1, 2, 3, 4}
s.add(1)   # s is still {1, 2, 3, 4}
```

13.4) Removing a value from a set

To remove a value from a set, we can simply use the .remove(element) method.

```
s = {1, 2, 3, 4}

s.remove(3)   # s is now {1, 2, 4}
s.remove(2)   # s is now {1, 4}
```

Note that if you pass in a value that doesn't exist, an error is raised.

```
s = {1, 2, 3, 4}
s.remove(5)   # KeyError
```

13.5) Common set functions

13.5.1) len(set) – finding the length of a set

```
s = {1, 2, 3, 4}
len(s)   # 4
```

Similarly, the length of the set refers to the number of values that it has. We can use the len() function to find the length of the set.

13.5.2) Intersection & Union between 2 sets

```
s1 = {1, 2, 3, 4}
s2 = {3, 4, 5, 6}

x = s1.intersection(s2)   # {3, 4}
y = s1.union(s2)          # {1, 2, 3, 4, 5, 6}

# NOTE - s1.union(s2) is the SAME as s2.union(s1)
```

- The intersection of 2 sets refers to a set of values that appear in BOTH sets
- The union of 2 sets refers to a set of values that appear in EITHER set.

To find the intersection and unions of 2 sets, we can use the .intersection(otherset) and .union(otherset) methods respectively.

14) File Reading & Writing

This chapter involves using Python to 1) read stuff from a file and 2) write stuff to a file. A file in this context refers to a text file (or whatever extension you want eg. csv).

14.1) File reading

Let's say we have a simple text file – fruits.txt – and we wish to use Python to read the contents of fruits.txt

```
apple
orange
pear
```

14.1.1) Reading the entire file at once

```python
with open("fruits.txt") as f:
    print(f.read())

# apple
# orange
# pear
```

Here, the *with* keyword is used to ensure that a resource (the open file in this case) is cleaned up after we use it. Let's just accept this for now. The variable *f* refers to our file, and *f.read()* will read the contents of our file at once.

14.1.2) Reading the file line by line

```python
with open("fruits.txt") as f:
    for line in f:
        line = line.strip()
        print(line)

# apple
# orange
# pear
```

Instead of reading the entire file at once, we can choose to read it line by line using a for loop. Note that we use *line.strip()* to remove the newline \n character at the end of each line. We can then print each line at each iteration.

14.2) File Writing

Let's say that we want to write some string to a new file out.txt. We can choose to do this using the 1) overwrite mode or 2) append mode. We can specify these modes inside the *open* function.

14.2.1) Writing in overwrite mode

```
with open("out.txt" ,"w") as f:
    f.write("hello")
```

Notice here that we have an additional argument *"w"* in our *open* function. This "w" specifies that we are writing in "overwrite" mode, which means that everything in out.txt will be completely erased before we write "hello" inside.

Note – if out.txt does not exist before this, it will be automatically created.

14.2.2) Writing in append mode

```
with open("out.txt", "a") as f:
    f.write("hello")
```

Here, notice that instead of a "w", we have an "a" in our *open* function. This "a" specifies that we are writing in "append" mode, which means that any existing content currently inside our file will not be touched. Whatever we write will be added *after* the existing content.

Note – if out.txt does not exist before this, it will also be automatically created.

14.2.3) Other important things to note

1. The *.write* method takes in only STRINGS.
2. If you intend on writing multiple lines to a file, remember to add newline characters \n to the back of your string. This will NOT be done automatically for you.
3. If you open a file in overwrite mode "w", everything will be erased IMMEDIATELY. Be careful not to accidentally erase important documents.

Checkpoint 5

1) Counting Fruits

You are given a text file, fruits.txt, with the following content:

```
apple
orange
pear
apple
apple
orange
```

Write a Python script to read fruits.txt, count the number of times each fruit appears, and display it in a dictionary.

```
{"apple":3, "orange":2, "pear":1}
```

2) Writing Fruits

You are given a dictionary:

```
d = {"apple":3, "orange":2, "pear":1, "pineapple":5, "durian":4}
```

Write a Python script to write these fruits into a text file, out.txt, in the following format:

```
apple,3
orange,2
pear,1
pineapple,5
durian,4
```

Answers (Checkpoint 5)

1) Counting Fruits

```python
d = {}
with open("fruits.txt") as f:
    for line in f:
        fruit = line.strip()   # removing newline character
        if fruit not in d:
            d[fruit] = 1
        else:
            d[fruit] += 1
print(d)
```

Here, we first open the fruits.txt file, and loop through each line inside the file. We need to remember to call the .strip() function on each line in order to remove the newline character at the end.

To count the number of fruits, we need to use a dictionary, and we thus need to define a dictionary *before* the for loop. At each iteration, we then check if the fruit is already inside the dictionary. If it isn't, we set its value to 1, but if it is, we increment its value by 1.

2) Writing Fruits

```python
d = {"apple":3, "orange":2, "pear":1, "pineapple":5, "durian":4}

with open("out.txt", "w") as f:
    for key in d:
        value = d[key]
        string = f"{key},{value}\n"
        f.write(string)
```

In order to write a text file, we need to call the *open* function with an additional argument – the "w" string, which tells Python that we want to write in overwrite mode. Writing in overwrite mode essentially erases everything in a text file at first, then begins writing in it.

We first iterate through our given dictionary using a for loop – we get the dictionary keys by iterating through the dictionary directly, and we get the dictionary values by indexing the dictionary using the key. We then construct the string we wish to write for each line using a formatted string – remember to include the newline character at the end in order to actually create a new line. We finally write each line to the file using the .write function.

15) Accumulation Using A Loop

Once you're kinda comfortable with the basic concepts from the previous chapters, this concept becomes one of the most important beginner concepts in Python.

```
define collection outside of loop
for iterator_variable in iterable:
    add iterator_variable to collection
```

If this sounds like alien language, let's take a look at a couple of examples

15.1) Accumulating into a number (addition)

Let's say we want to find the sum of numbers from 1 to 5 using Python. Here, we know that our final answer is a number. We thus need to start with an *empty* number, which is essentially 0.

```
total = 0
for i in range(1, 6):
    total += i
print(total)   # 15
```

We thus first define a variable total *outside* of the for loop, *before* the for loop. We then use a for loop to generate and iterate through the numbers that we are interested in (1 to 5), and add *i* to *total* at every iteration.

```
i   total
1   1  (0 + 1)
2   3  (1 + 2)
3   6  (3 + 3)
4   10 (6 + 4)
5   15 (10 + 5)
```

We thus end up with total being 15 (1+2+3+4+5).

15.2) Accumulating into a number (multiplication)

Let's say we want to find the product of numbers from 1 to 5. Similarly, since we know that our final answer is a number, we need to start with an empty number. In this case, as we are doing multiplication, our 'empty number' needs to be a 1 (everything multiplied by 0 is 0).

```
total = 1
for i in range(1, 6):
    total *= i
print(total)   # 120
```

Similarly, we define total *outside of and before* the for loop. At every iteration, we accumulate each number (multiply) into total.

```
i   total
1   1    (1 * 1)
2   2    (1 * 2)
3   6    (2 * 3)
4   24   (6 * 4)
5   120  (24 * 5)
```

We thus end up with total being 120 (1*2*3*4*5)

15.3) Accumulating into a string

Let's say we wish to remove the vowels (aeiou) from an existing string. As usual, since we know that our final answer is a string, we need to start with an empty string "".

```
string = "apple"
output = ""
for letter in string:
    if letter not in "aeiou":
        output += letter
print(output)   # ppl
```

Here, we need to create an empty string (output) *before* our for loop. At every iteration, we use a conditional statement – if a letter is not a vowel, we add it to the back of output.

```
letter   is_vowel   output
a        True       ""        # no action
p        False      "p"       # "p" added to back of output
p        False      "pp"      # "p" added to back of output
l        False      "ppl"     # "l" added to back of output
e        True       "ppl"     # no action
```

15.4) Accumulating into a list

Let's say we wish to create a list containing the squares of numbers from 1 to 5. Similarly, as we want to end up with a list, we need to first start with an empty list.

```
lis = []
for i in range(1, 6):
    lis.append(i**2)
print(lis)   # [1, 4, 9, 16, 25]
```

How this works:

```
i    i**2    lis              what happens
1    1       [1]              # 1 appended to lis
2    4       [1,4]            # 4 appended to lis
3    9       [1,4,9]          # 9 appended to lis
4    16      [1,4,9,16]       # 16 appended to lis
5    25      [1,4,9,16,25]    # 25 appended to lis
```

15.5) Accumulating into a dictionary

Let's say we wish to create a dictionary where keys are numbers from 1 to 5, and values are the squares of the keys. Same thing – we need to first start off with an empty dictionary.

```
d = {}
for i in range(1, 6):
    d[i] = i**2
print(d)   # {1:1, 2:4, 3:9, 4:16, 5:25}
```

How this works:

```
i   i**2   d                              what happens
1   1      {1:1}                          # 1:1 added to d
2   4      {1:1, 2:4}                     # 2:4 added to d
3   9      {1:1, 2:4, 3:9}                # 3:9 added to d
4   16     {1:1, 2:4, 3:9, 4:16}          # 4:16 added to d
5   25     {1:1,2:4, 3:9, 4:16, 5:25}     # 5:25 added to d
```

15.6) Accumulating into a dictionary with duplicate keys

Let's say we have a list of strings, and we wish to count how many times each string appears.

```
lis = ["apple", "orange", "pear", "apple", "apple", "orange"]

# what we want to achieve
{"apple":3, "orange":2, "pear":1}
```

Remember that dictionary keys must be unique – we hence need to write some conditional statements to make sure we don't accidentally overwrite existing keys. Similarly, as we want to end up with a dictionary, we first create an empty dictionary.

```
d = {}
for fruit in lis:
    if fruit not in d:
        d[fruit] = 1    # set count to 1
    else:
        d[fruit] += 1   # increment count by 1

print(d)   # {"apple":3, "orange":2, "pear":1}
```

At every iteration, we need to first check if the fruit already exists inside the dictionary. If it does not exist, it means that we've not seen the fruit yet, so we set its count to 1. If fruit already exists inside the dictionary, it means that we've seen it before, so we increase the existing count by 1 instead of setting it to 1. The code run-through:

```
fruit     (fruit not in d)   d
"apple"   True               {"apple":1}
"orange"  True               {"apple":1, "orange":1}
"pear"    True               {"apple":1, "orange":1, "pear":1}
"apple"   False              {"apple":2, "orange":1, "pear":1}
"apple"   False              {"apple":3, "orange":1, "pear":1}
"orange"  False              {"apple":3, "orange":2, "pear":1}

# we thus end up with {"apple":3,  "orange":2, "pear":1}
```

Checkpoint 6

1) Sum of odd numbers

Write a function *sumodd(start, end)* that takes in 2 integers start and end, and computes the sum of the *odd* numbers between start and end (inclusive of both).

```
sumodd(1, 4)   # 4    # (1+3)
sumodd(1, 6)   # 9    # (1+3+5)
sumodd(1, 7)   # 16   # (1+3+5+7)
```

2) Add odd but subtract even

Write a function alternate(lis) that takes in a list of numbers, adds the odd numbers, subtracts the even numbers, and returns the final result.

```
alternate([1,2,3,4,5])   # 3    # (1-2+3-4+5)
alternate([1,3,5,7,2])   # 14    # (1+3+5+7-2)
alternate([2,4,6,8,1])   # -19    # (-2-4-6-8+1)
```

3) Uppercase Strings

Write a function uppercase(lis) that takes in a list of strings, and returns a new list containing uppercase versions of the strings in the original list.

```
uppercase(["apple", "orange", "pear"])    # ["APPLE", "ORANGE", "PEAR"]
uppercase(["pineapple", "durian"])        # ["PINEAPPLE", "DURIAN"]
uppercase(["apple123", "orange456"])      # ["APPLE123", "ORANGE456"]
```

4) Reverse Strings

Write a function reverse(lis) that takes in a list of strings, and returns a dictionary. Keys in this dictionary should be the same as the strings in the list, while values in this dictionary are the *reverse* of their keys.

```
reverse(["apple", "orange"])    # {"apple":"elppa", "orange":"egnaro"}
reverse(["pear", "pineapple"])  # {"pear":"raep", "pineapple":"elppaenip"}
reverse(["air", "boy", "cat"])  # {"air":"ria", "boy":"yob", "cat":"tac"}
```

Answers (Checkpoint 6)

1) Sum of odd numbers

```
def sumodd(start, end):
    total = 0
    for i in range(start, end+1):
        if i%2 == 1:
            total += i
    return total
```

Here, as we wish to end up with a number, we need to start with an empty number. As such, we set total=0 *before* the for loop. We then use a for loop to iterate through numbers from start to end, check if they are odd, and add only the odd numbers:

```
sumodd(1, 4) -> start=1, end=4

i   i%2==1   action     total
1   True     add        1
2   False    dont add   1
3   True     add        4
4   False    dont add   4      # 4 is then returned
```

2) Add odd but subtract even

```
def alternate(lis):
    total = 0
    for i in lis:
        if i%2 == 1:
            total += i
        else:
            total -= i
    return total
```

Here, we wish to end up with a number, so we start with an empty number, and define total=0 *before* our for loop. Next, we iterate through all numbers in our list, and use a conditional statement to check if each number is odd or even:

```
lis = [1, 2, 3, 4, 5]

i   (i%2==1)   action       total
1   True       add 1        1
2   False      subtract 2   -1
3   True       add 3        2
4   False      subtract 4   -2
5   True       add 5        3         # 3 is then returned
```

3) Uppercase Strings

```
def uppercase(lis):
    output = []
    for string in lis:
        upperstring = string.upper()
        output.append(upperstring)
    return output
```

Once again, we wish to end up with a list, so we have to start with an empty list, and define output=[] *before* the for loop. We then iterate through the original list, convert each string to uppercase, and add it to output.

```
lis = ["apple", "orange", "pear"]

string     upperstring   output
"apple"    "APPLE"       ["APPLE"]
"orange"   "ORANGE"      ["APPLE", "ORANGE"]
"pear"     "PEAR"        ["APPLE", "ORANGE", "PEAR"]
```

4) Reverse Strings

```
def reverse(lis):
    d = {}
    for string in lis:
        d[string] = string[::-1]
    return d
```

Here, we wish to end up with a dictionary, so we need to start with an empty dictionary. We hence define d={} before our for loop. When we loop through the list using a for loop, and add a new key-value pair at each iteration, the key being the string itself, and the value being the reverse of the string.

```
lis = ["apple", "boy", "cat"]

string    string[::1]   d
"apple"   "elppa"       {"apple":"elppa"}
"boy"     "yob"         {"apple":"elppa", "boy":"yob"}
"cat"     "tac"         {"apple":"elppa", "boy":"yob", "cat":"tac"}
```

16) Nested Loops

16.1) A simple example

A nested loop is a loop inside another loop (for/while loop). Let's first take a look at a simple example:

```
fruits = ["apple", "orange"]
numbers = [4, 5, 6]

for fruit in fruits:
    for number in numbers:
        print(fruit, number)

# apple 4
# apple 5
# apple 6
# orange 4
# orange 5
# orange 6
```

For each iteration in the outer for loop (for fruit in fruits), the inner for loop (for number in numbers) runs once. In the first iteration of the outer for loop, where fruit = "apple", the *entire* inner for loop runs once – number takes on the values 4, 5 then 6.

```
                  action
fruit = "apple"
    number = 4    prints "apple 4"
    number = 5    prints "apple 5"
    number = 6    prints "apple 6"

fruit = "orange"
    number = 4    prints "orange 4"
    number = 5    prints "orange 5"
    number = 6    prints "orange 6"
```

16.2) A more complicated example

Let's say we need to write a function that takes in an integer number, and print the following patterns depending on the number passed in.

```
n = 3
1
12
123

n = 4
1
12
123
1234
```

Here's how we can write this function using a nested for loop.

```
def pattern(n):
    for i in range(1, n+1):
        for j in range(1, i+1):
            print(j, end="")
        print()
pattern(3)
# 1
# 12
# 123
```

How this works:

```
n = 3
range(1, n+1) generates the numbers 1, 2, 3

i=1 -> range(1, 1+1) generates the numbers 1
    j=1 -> 1 is printed

i=2 -> range(1, 2+1) generates the numbers 1, 2
    j=1 -> 1 is printed
    j=2 -> 2 is printed

i=3 -> range(1, 3+1) generates the numbers 1, 2, 3
    j=1 -> 1 is printed
    j=2 -> 2 is printed
    j=3 -> 3 is printed
```

16.3) Triple Nested Loops & More

```python
list1 = ["apple", "orange", "pear"]
list2 = [4, 5, 6]
list3 = [True, False]

for fruit in list1:
    for number in list2:
        for boolean in list3:
            print(fruit, number, boolean)

# apple 4 True
# apple 4 False
# apple 5 True
# apple 5 False
# apple 6 True
# apple 6 False
# orange 4 True
# orange 4 False
# orange 5 True
# orange 5 False
# orange 6 True
# orange 6 False
# pear 4 True
# pear 4 False
# pear 5 True
# pear 5 False
# pear 6 True
# pear 6 False
```

Here's an example of 3 nested for loops. For each iteration in the outermost for loop (for fruit in list1), the *entire* inner for loop (for number in list2) runs once. For each iteration in the inner for loop (for number in list2), the *entire* innermost for loop (for boolean in list3) runs once. And that's why we have so many iterations.

Checkpoint 7

1) Pyramid of numbers

Write a function pyramid(n) that takes in an integer n, and prints the following pattern based on the input integer n.

```
n=3
1
21
321

n=5
1
21
321
4321
54321
```

2) Reverse pyramid of numbers

Write a function rpyramid(n) that takes in an integer n, and prints the following pattern based on the input integer n.

```
n=3
123
12
1

n=5
12345
1234
123
12
1
```

3) Reverse pyramid of numbers 2

Write a function rpyramid2(n) that takes in an integer n, and prints the following pattern based on the input integer n.

```
n=3
321
32
3

n=5
54321
5432
543
54
5
```

4) Numbers that add up to 10

Write a function addto10(lis) that takes in a list of integers, and prints every combination of numbers within the list that add up to 10.

```
addto10([3,4,5,6,7])
3 7
4 6
5 5
6 4
7 3

addto10([5,7,8,2,6])
5 5
8 2
2 8
```

Answers (Checkpoint 7)

1) Pyramid of numbers

```
def pyramid(n):
    for i in range(1, n+1):
        for j in range(i, 0, -1):
            print(j, end="")
        print()
```

Let's say that n=5

```
The outer for loop generates numbers 1 2 3 4 5

      inner for loop
i=1   range(1, 0, -1) which generates 1
i=2   range(2, 0, -1) which generates 2 1
i=3   range(3, 0, -1) which generates 3 2 1
i=4   range(4, 0, -1) which generates 4 3 2 1
i=5   range(5, 0, -1) which generates 5 4 3 2 1
```

2) Reverse pyramid

```
def rpyramid(n):
    for i in range(n, 0, -1):
        for j in range(1, i+1):
            print(j, end="")
        print()
```

Let's say that n=5

```
The outer for loop generates numbers 5 4 3 2 1

      inner for loop
i=5   range(1, 6) which generates 1 2 3 4 5
i=4   range(1, 5) which generates 1 2 3 4
i=3   range(1, 4) which generates 1 2 3
i=2   range(1, 3) which generates 1 2
i=1   range(1, 2) which generates 1
```

3) Reverse pyramid 2

```
def rpyramid2(n):
    for i in range(n, 0, -1):
        for j in range(n, n-i, -1):
            print(j, end="")
        print()
```

Let's say that n=5

```
The outer for loop generates the numbers 5 4 3 2 1

      inner for loop
i=5   range(5, 0, -1) which generates 5 4 3 2 1
i=4   range(5, 1, -1) which generates 5 4 3 2
i=3   range(5, 2, -1) which generates 5 4 3
i=2   range(5, 3, -1) which generates 5 4
i=1   range(5, 4, -1) which generates 5
```

4) Numbers that add up to 10

```
def addto10(lis):
    for i in lis:
        for j in lis:
            if i+j==10:
                print(i, j)
```

Example code run-through:

```
lis = [3, 4, 5, 6, 7]

      values of j   j where i+j==10
i=3   3 4 5 6 7     7
i=4   3 4 5 6 7     6
i=5   3 4 5 6 7     5
i=6   3 4 5 6 7     4
i=7   3 4 5 6 7     3
```

Conclusion

Congratulations – you've made it to the end. Hopefully the concepts and explanations in the various chapters were clear, and the checkpoint questions were mentally stimulating enough. Of course, we don't magically go from zero to one just by finishing this book – we still need to practice, revise, apply and re-apply that stuff that we've learnt.

From my experience as both a Python programmer and a tutor, the best way to effectively learn Python and retain most of what we learn is to apply it to a real world project, no matter how small or stupidly simple. After all, progress is still progress no matter how miniscule, and it is the little bits of progress that compound into something greater in the long run.

Some Python projects (no libraries required) I've done in the past when I was still new:

1. Number-guessing game
2. Tic-tac-toe game
3. Find-four game
4. Maze game

Some Python projects (some libraries required) that I've done:

1. Script to delete screenshots (check out the 'os' library)
2. Calling public APIs (check out the 'requests' library)
3. Web scraping sites like kickstarter.com (check out 'bs4')
4. Web automation (check out 'selenium')

While the content in this book alone is probably insufficient to confidently tackle all the above projects, no one book will probably be sufficient (unless it's 2000 pages long maybe). As programmers, search engines (and StackOverflow) are our best friend, and we can definitely figure things out if we search hard enough. I highly recommend you to give some of these mini-projects a go if they interest you – I promise that you'll learn something some way or another!

One again, I sincerely hope that this book has been useful in bringing you from zero to one in Python (or at least a 0.7 or 0.8). Thanks for reading – it's been fun (and kinda therapeutic) compiling and putting these concepts together into 16 chapters.

Cheers,
Liu Zuo Lin