

PPiAI Coding Survival Guide: A Manual For The African Teacher



David Sciacca
Gourav Nayak
Vraj Patel

David Sciacca

Gourav Nayak, Vraj Patel

PPiAI Coding Survival Guide: A Manual For The African Teacher

Recommended for use in
primary and secondary schools
throughout sub-Saharan Africa

Public Policy in Africa Initiative (PPiAI)

Cameroon

Email: info@publicpolicyafrica.org

Website: www.publicpolicyafrica.org

First published: December 2021

Copyright © Public Policy in Africa Initiative. All rights reserved.

Our vision at the Public Policy in Africa Initiative is: Powering the African Economy. We achieve this via evidence informed decisions. A research study suggests that coding is a skill young people need to succeed in society. Our hope is that this book will be used in sub-Saharan Africa's primary and secondary schools to teach our children how to code. The aim of this manual is that by learning from it, our beloved and hardworking students and teachers will gain some basic coding skills and then develop those skills through a project of their own.

This book can be downloaded for free on our website.

Particular thanks are due to the authors who through their kindness and desire to add value to the African youth have accepted and devoted themselves to write this coding book. Thank you to Benjamin Njila Tchakounte Fields for editing this manual and Dela Sorkpor for the graphic design.

Yours sincerely,

Hugue Nkoutchou, PhD(Bath)

Founder & Chairperson at PPiAI

Contents

How to Install Python [Pycharm IDE]	1
1.1 How to install a Python IDE on Windows	1
1.2 How to Install PyCharm.....	3
1.3 Installing Python and PyCharm on other platforms	7
Data Types, Variables and Operators	8
2.1 Variables.....	8
2.1.0 Strings.....	8
2.1.1 Numbers	9
2.1.2 Boolean	9
2.2 Variables.....	10
2.3 Arithmetic Operators	10
2.4 Assignment Operators	11
2.5 Comparison Operators.....	12
2.6 Logical Operators.....	13
2.7 Identity Operators	13
2.8 Bitwise Operators	14
Conditionals and Data Structures Part I	18
3.1 Conditionals.....	18
3.2 Data Structures	20
3.3 Sets.....	20
3.4 Set Methods	21
3.5 The in Operator	22
Data Structures Part II	25
4.1 Arrays	25
4.2 Array Methods.....	26
4.3 Dictionaries.....	27
4.4 Dictionary Methods	28
Data Structures Part III	31
5.1 Lists	31
5.2 List Slicing	32
5.3 List Methods	32

5.4 Lists and Strings – a useful relationship!	33
5.5 Tuples	34
5.6 Tuple indexing and Slicing	34
5.7 Tuple Methods	35
Loops, Iterables and Iterators	38
6.1 Loops	38
6.2 for Loop	38
6.3 while Loop	39
6.4 Nested Loop	40
6.5 Iterable Functions	41
6.6 break statement	42
6.7 continue statement	42
Basic Input and Output, Files and Folders	44
7.1 Input And Output (I/O)	44
7.2 Input Via user	44
7.3 Input Via Files	45
7.4 Input Via Files – Syntax	46
7.5 Input Via Files – Write and Append mode	46
7.6 Files And Folders	46
7.7 Files And Folders – os.walk	47
7.8 Files And Folders – os	47
7.9 Files And Folders – replace	49
Functions in Python	52
8.1 What is Function?	52
8.2 Defining a Function	52
8.3 Invoking a Function	53
8.4 Parameters	53
8.5 Return Statement	55
8.6 Documentation Strings (Docstrings)	56
8.7 Scope	57
Classes and Error Handling in Python	60
9.1 Object Oriented Programming	60
9.2 Classes	60

9.3 The <code>__init__()</code> Method.....	61
9.4 Methods	62
9.5 The self Parameter	62
9.7 Errors and Exceptions	63
9.8 Exception Handling	64
Modules and Packages in Python	67
10.1 Module	67
10.2 Module – Import statement	67
10.3 Module – from statement.....	68
10.4 Package	69
10.5 Package – import	69
10.6 Multiple import – package and module.....	70
10.7 Why are packages and modules are needed?	70
10.8 Python Packages	70
10.9 pip	71
Unit Testing in Python	73
11.1 Testing in Python	73
11.2 assert Keyword	73
11.3 unittest Package	74
11.4 unittest Methods.....	76
11.4 Testing Custom Functions with unittest	76

Chapter 1

How to Install Python [Pycharm IDE]

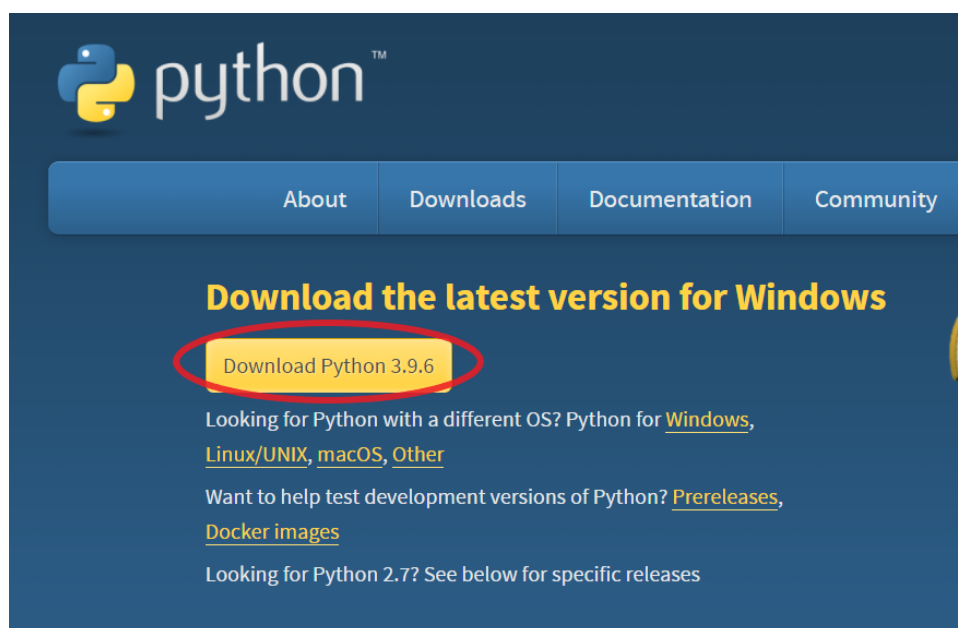
PyCharm is an integrated development environment (IDE) developed by JetBrains that is used in computer programming, specifically as a productive Python development tool.

Throughout the following chapter, we are going to cover installing Python 3 onto our workspace.

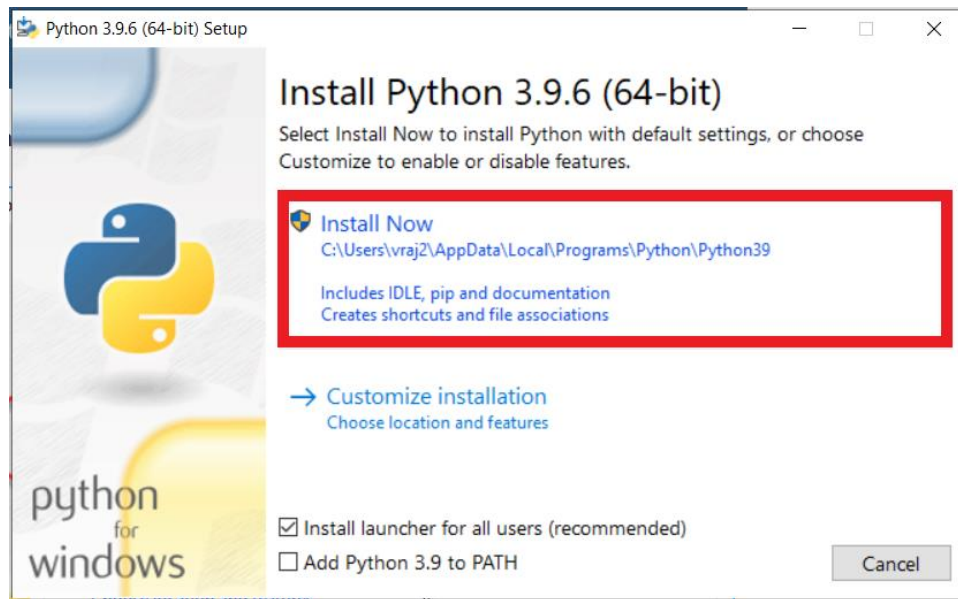
1.1 How to install a Python IDE on Windows

Below is a step-by-step guide on how to download and install Python on Windows:

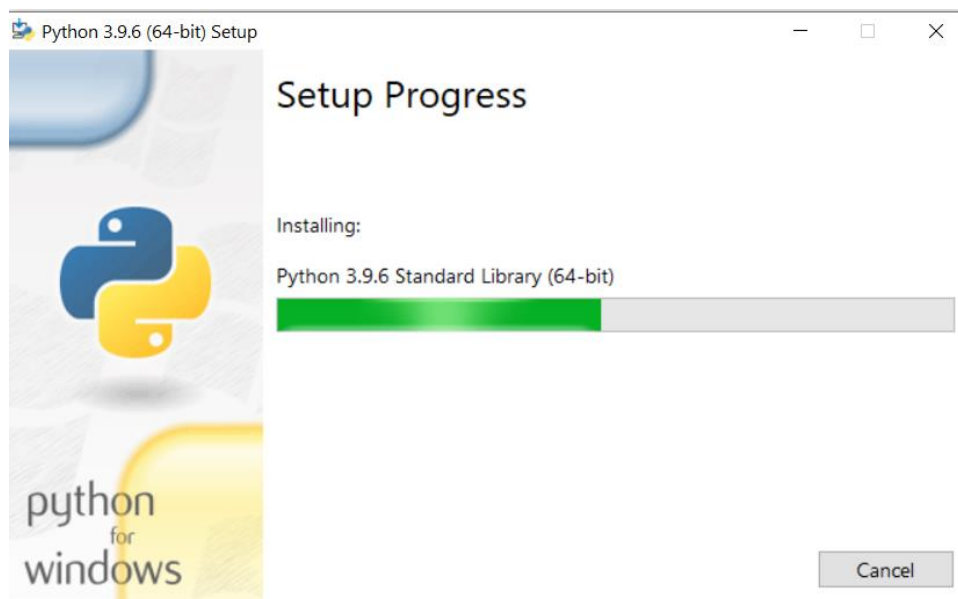
Step 1) Visit the official website of Python to download and install the latest version of Python 3 on Windows: <https://www.python.org/downloads/>.



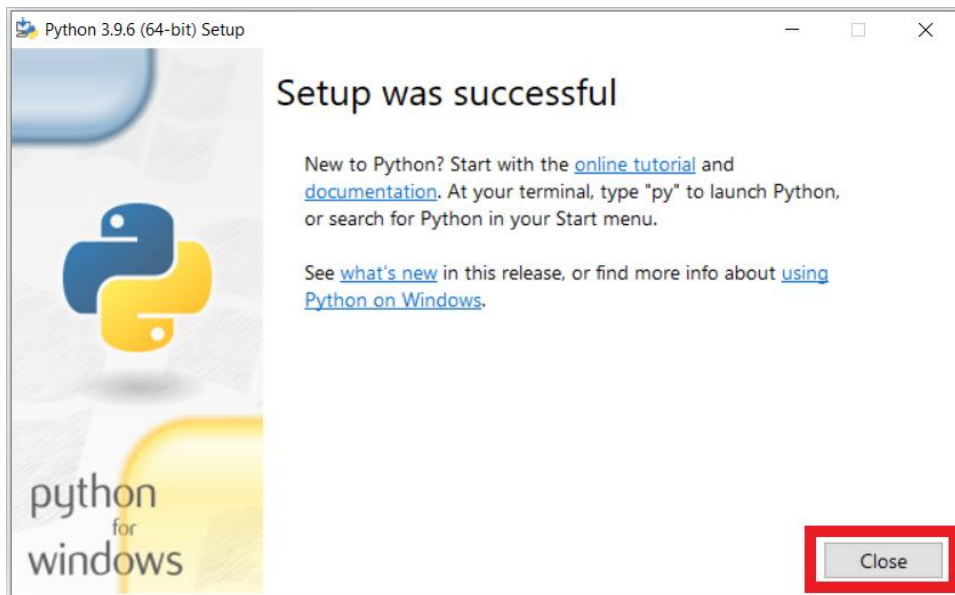
Step 2) Once the download has been completed, run the .exe file to install Python.



Disclaimer You may see Python installing at this point as seen below.



Step 3) When the installation is complete, you will be able to see a screen that says the "Setup was successful". You may now click on close.



1.2 How to Install PyCharm

Next, you will find below a step-by-step guide on how to download and install PyCharm IDE on Windows:

Step 1) Visit the official website of JetBrains to download and install the latest version of PyCharm on your workspace.

Download PyCharm

Windows macOS Linux

Professional

Full-featured IDE
for Python & Web
development

DOWNLOAD

Free trial

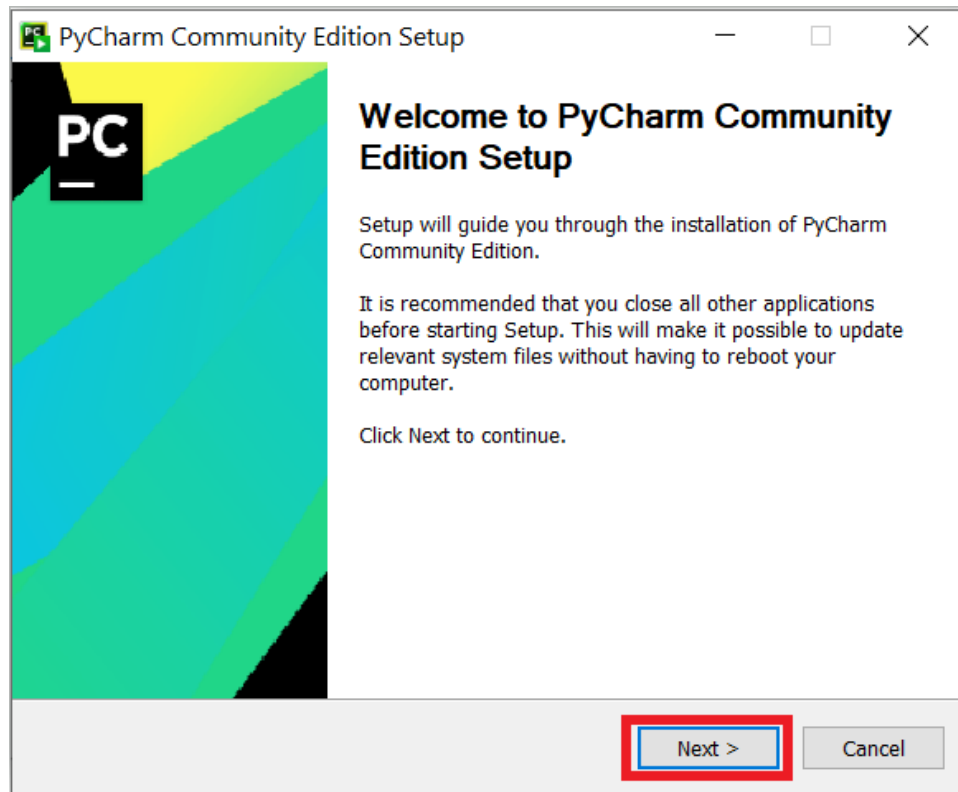
Community

Lightweight IDE
for Python & Scientific
development

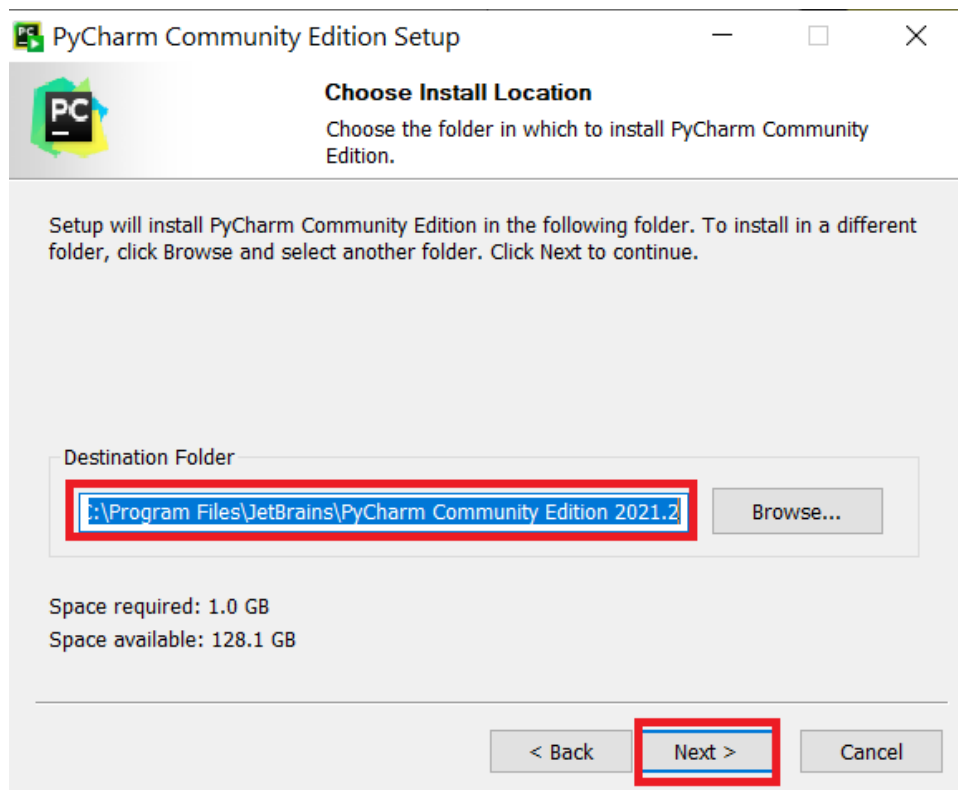
DOWNLOAD

Free, open-source

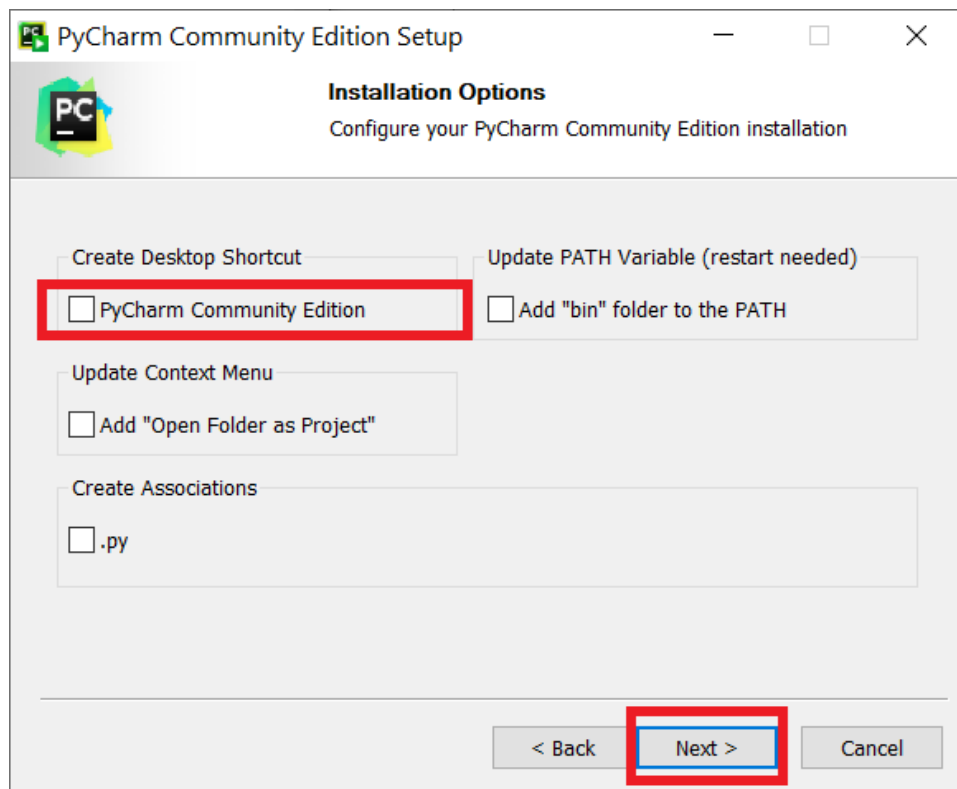
Step 2) Once the download has been completed, run the .exe file to install PyCharm.



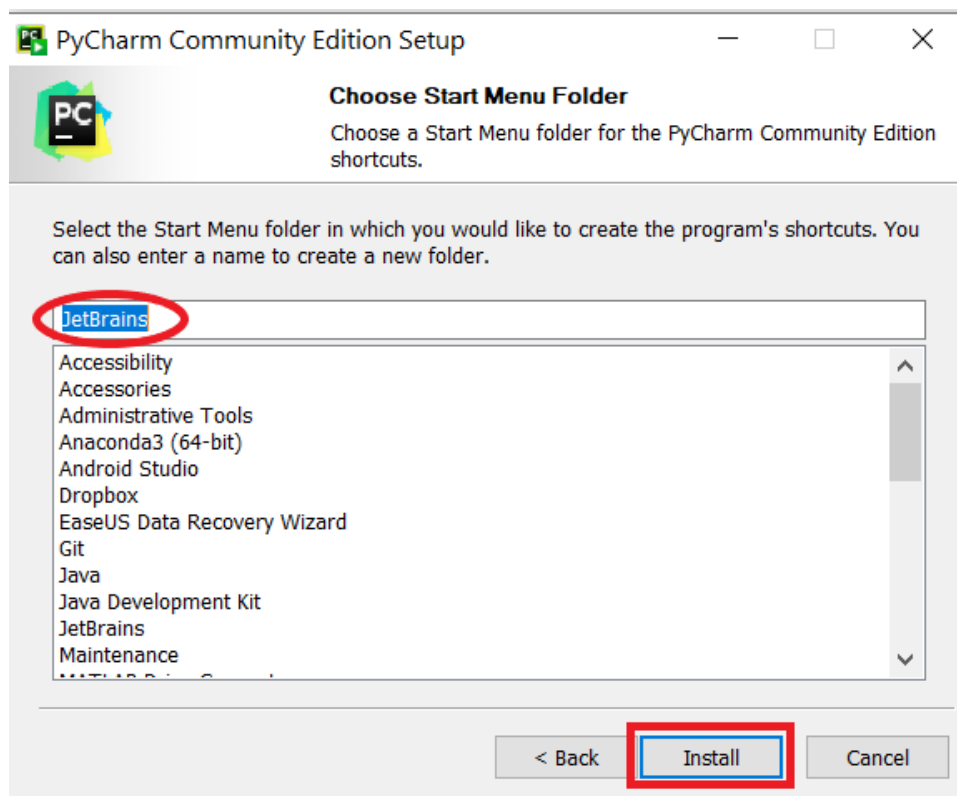
Step 3) Click on "Next" (you may have to change the installation path depending on your circumstances).



Step 4) Create a desktop shortcut and click on “Next”.

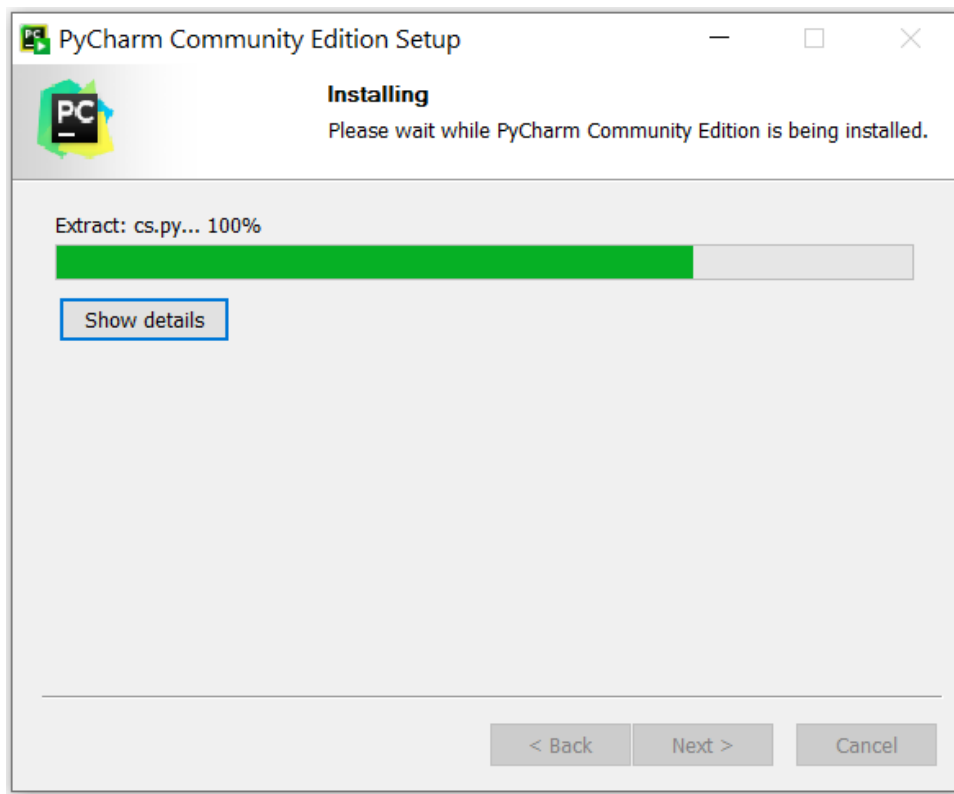


Step 5) Keep the selection as JetBrains and continue by clicking “Next”.

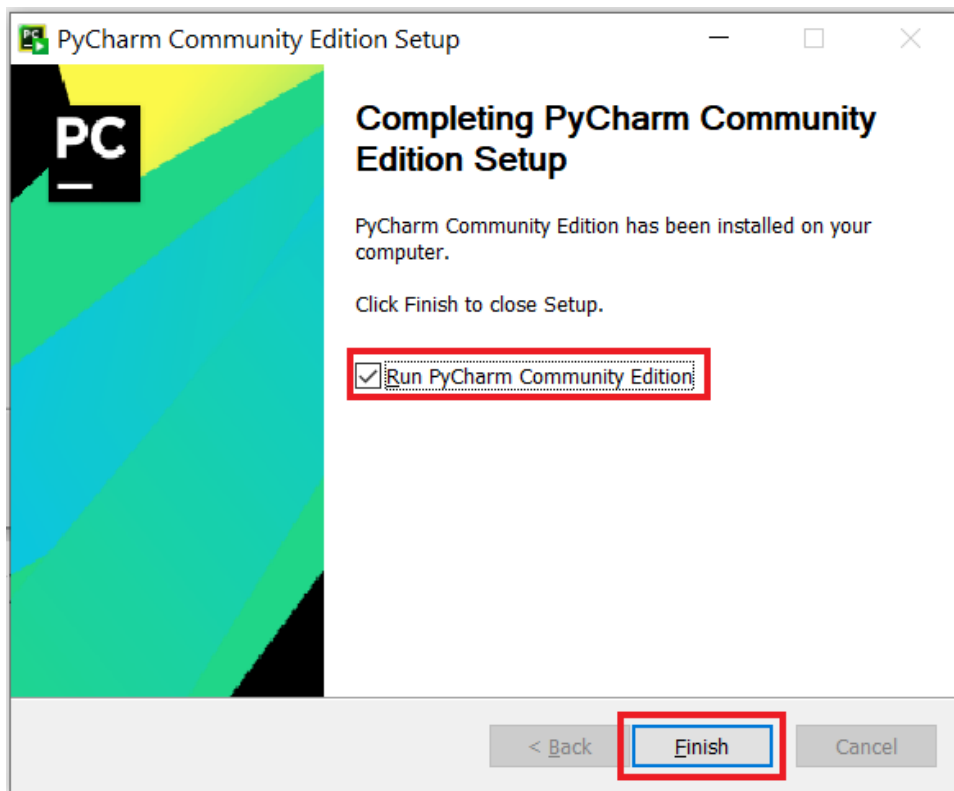


Disclaimer

You may see PyCharm installing at this point as seen below.



Step 6) Once the installation is completed, click on “Run PyCharm Community Edition” and click on “Finish” to run the application.



1.3 Installing Python and PyCharm on other platforms

If you are not a Windows user, use the following links to learn how to install Python for MacOS or Linux.

MacOS : <https://realpython.com/installing-python/#how-to-install-python-on-macos>

Linux : <https://realpython.com/installing-python/#how-to-install-python-on-linux>

Installation guide for PyCharm: <https://www.jetbrains.com/help/pycharm/installation-guide.html>

NOTE: There are minimum criteria your computer must meet to successfully install PyCharm; see the System Requirements section of the above link.

Assignment 1

Creating your First Python Program

- 1) Open the PyCharm editor and create a new project.
- 2) Select where you would like to save your project and give it a meaningful name.
- 3) Click on “File” from the menu bar and select New Python File.
- 4) Name your first Python file.
- 5) Write a simple program – **print(“Hello World from PPIAI!”)**.
- 6) Go up to the menu bar and select run to compile your first program.
- 7) You will be able to see an output of your program at the bottom of the screen.

Chapter 2

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://youtu.be/1x0pJoNJFN0>

Data Types, Variables and Operators

2.1 Variables

2.1.0 Strings

Strings are one of the fundamental data types in computer programming used to represent text. In other words, we may say Strings are considered as arrays of bytes representing Unicode characters.

Creating a string: A string is created by enclosing text in quotes. You may use either single quotes `'`, or double quotes `"`. Here are some examples:

```
>>> "Hello World!"
'Hello World!'
>>> 'Hello World!'
'Hello World!'
>>> """Hello World!"""
'Hello World!'
>>> '"Hello World!'"
'"Hello World!'"
```

An empty string `''` is a string with nothing in it.

```
>>> ' '
' '
>>> ''
''
```

The Python **str()** function converts the specified value into a string.

For example:

```
>>> str(90)
'90'
```

The **str()** method returns a string, which is considered to be a printable version that represents the given object.

2.1.1 Numbers

Python supports integers, floating-point numbers and complex numbers. They are defined using the functions **int()**, **float()**, and **complex()** in python. The presence or absence of a decimal point determines whether or not the input is an integer or floating point. For example, 2 is an integer whereas 2.0 is considered a floating-point number.

Integers (whole numbers) can be arbitrarily large. Examples of integers are numbers such as 1, -1337, 2540 and so on.

Floats (decimal numbers) are represented by floating-point numbers. Examples of floats are 1.25, -13.37E2, 1E-3, and so on.

Complex numbers are written in the form "X + Yi" where X is the real part (i.e. a real number like the integer 8) and Y is the imaginary part (an imaginary number containing an i after the integer). Examples of complex numbers may include 5i, -2.2i and so on.

```
>>> int("303")
303
>>> float(56)
56.0
>>> complex(43)
(43+0j)
>>> complex(56, 45)
(56+45j)
```

2.1.2 Boolean

A Boolean type is one the built-in data types that helps us represent the truth value of an expression. For example, the expression `2 < 6` is True, while the expression `6 > 9` is False.

The Python Boolean type has two possible values:

- 1) True
- 2) False

The function **bool()** allows us to evaluate a value that gives True or False in return depending on the situation.

```
>>> 29>9
True
>>> 29<9
False
>>> 29==9
False
```

2.2 Variables

Variables are containers that are used to store data values. A variable is created when a value is assigned to it.

```
>>> greeting = 'Hello, World'
>>> x = 5
>>> y = 12.0
>>> yes_no = x < y
>>> boolean = bool()
```

Variables in Python do not require themselves to be declared to any particular data type, and they can even change the data type after it has been set.

```
>>> x = 5
>>> x = "Vraj"
>>> print(x)
Vraj
```

The data type of a variable may be specified if required by use of casting.

```
>>> x = str(5)      # x will be '5'
>>> y = int(5)      # y will be 5
>>> z = float(5)    # z will be 5.0
```

2.3 Arithmetic Operators

Arithmetic operators are used with numeric values to perform common mathematical operations.

Operator	Meaning	Example
+	Addition	1 + 2

-	Subtraction	1 - 2
*	Multiplication	1 * 2
/	Division	1 / 2
%	Modulus	1 % 2
**	Exponentiation	1 ** 2
//	Floor division	1 // 2

2.4 Assignment Operators

Assignment operators are used to assign values to variables.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 6	x = x + 6
-=	x -= 3	x = x - 2
*=	x *= 2	x = x * 2
/=	x /= 2	x = x / 2
%=	x %= 2	x = x % 2

//=	x //= 2	x = x // 2
**=	x **= 2	x = x ** 2
&=	x &= 2	x = x & 2
=	x = 2	x = x 2
^=	x ^= 2	x = x ^ 2
>>=	x >>= 2	x = x >> 2
<<=	x <<= 2	x = x << 2

2.5 Comparison Operators

Comparison operators are used to compare two values.

Operator	Description	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y

<=	Less than or equal to	x <= y
----	-----------------------	--------

2.6 Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

2.7 Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location.

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

2.8 Bitwise Operators

Bitwise operators are used to compare (binary) numbers.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Assignment 2

Creating a Basic Financial State Calculator

Your client wants a program to generate a basic financial report that users can run regularly to understand their current financial state. This program should be able to store:

- The user's name
- Checking account balance
- Saving's account balance
- Investment account value
- Total utility bills
- Total credit card debt
- Credit card balance due
- Credit card minimum payment due
- Annual credit card interest rate
- Loan debt
- Loan payment due
- Annual loan interest rate
- Other asset value

When run, this program should generate the following report:

Hello <user's name>,

The total dollar value of assets you own is \$<total value of assets> and the total dollar value of your debt is currently \$<total debt>; therefore, your current net worth is: \$<total net worth>.

Your total bills due are \$<total value of bills due>. Once you make these payments, you will have \$<value remaining in the checking account after making payments> left in your checking account, and \$<total remaining money in bank> in your bank accounts overall. Additionally, your total credit card debt will be down to \$<credit card debt remaining after paying balance due> and your loan debt will be down to \$<loan debt after paying amount due and interest is applied on the remaining balance for the month> (including interest applied on the remaining balance after your payment). Therefore, your total debt will then be paid down to \$<total debt after making payments> and your net worth will be \$<net worth after making payments as specified>.

EXTRA CREDIT: If you'd like extra credit on your assignment, you can include the following passage in your output. Successfully including this will give you an extra one quarter worth of assignment credit:

If you instead choose not to pay off your full credit card balance due (\$<credit card balance due>) and only pay the minimum payment due (\$<credit card minimum payment due>) you will have \$<value remaining in the checking account after making minimum payments> left in your checking account, and \$<total remaining money in bank>

in your bank accounts overall. However, you will accrue \$<interest accrued on credit card balance for the month> in interest. Your total credit card debt will then be \$<total credit card debt after paying minimum payment and accruing interest on remaining balance due>. In this case, your total debt would instead be \$<total debt after making payments as specified> and your net worth will be \$<net worth after making payments as specified>.

Formula Sheet

Total value of assets = checking account balance + saving's account balance + investment account value + other assets value

Total Debt = total utility bills + total credit card debt + loan debt

Total net worth = total value of assets - total debt

Total Value of Bills Due = total utility bills + credit card balance due + loan payment due

Value remaining in the checking account after making payments = checking account balance - total value of bills due

Total remaining money in bank = Value remaining in the checking account after making payments + saving's account balance

Credit Card Debt After paying balance due = total credit card debt - credit card balance due

Loan debt after paying amount due and interest is applied on the remaining balance for the month = loan debt - loan payment due + ((loan debt - loan payment due) X annual loan interest rate \div 12)

Total debt after making payments = loan debt after paying amount due and interest is applied on the remaining balance for the month + credit card debt after paying balance due

Net worth after making payments specified = total value of assets - total value of bills due - credit card debt after paying balance due - loan debt after paying amount due and interest is applied on the remaining balance for the month

Minimum Bills Due = total utility bills + credit card minimum payment due + loan payment due

Value remaining in checking account after making minimum payments = checking account balance - minimum bills due

Total remaining money in bank (Paragraph 2) = checking account balance + saving's account balance - minimum bills due

Interest accrued on credit card balance for the month = (credit card balance due - credit card minimum payment due) X (annual credit card interest rate \div 12)

Total credit card debt after paying minimum payment and accruing interest on remaining balance due = total credit card debt - credit card minimum payment due + interest accrued on credit card balance for the month

Total debt after making payments as specified (paragraph 2) = total credit card debt after paying minimum payment and accruing interest on remaining balance due + loan debt after paying amount due and interest is applied on the remaining balance for the month

Net worth after making payments as specified (paragraph 2) = total value of assets - total utility bills - loan payment due - credit card minimum payment due - loan debt after paying amount due and interest is applied on the remaining balance for the month - total credit card debt after paying minimum payment and accruing interest on remaining balance due

Chapter 3

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://youtu.be/hBw7mc8XUVQ>

Conditionals and Data Structures Part I

3.1 Conditionals

Python if statement

Many times, we may need our program to do something provided something else is true. Python's **if** statement helps us in this situation.

Let's start with a simple example to demonstrate how this works.

```
if boolean1:
    print("boolean1 is True")

if boolean1 or boolean2:
    print("Atleast one of boolean1 and boolean2 is True!")
```

In the above example, we use two variables, `boolean1` and `boolean2`, which are used as a part of the `if` statement to test whether `boolean1` is initialized to true in the first case, or either `boolean1` or `boolean2` are initialized to true.

Python if ... else statement

The **if ... else** statement evaluates the variable and will execute the body below the `if` only when the test condition is True.

If the condition is False, the body below the **else** block will be executed.

Look below for an example: if `boolean1` is not initialized to true, the program moves to the **else** block for execution.


```

if boolean1:
    print("boolean1 is True!")

else:
    print("boolean1 is False!")

```

Python if ... elif ... else statement

The “**elif**” designation is short for else if. It allows you to check for multiple expressions.

If the condition for the **if** block is False, then it checks for the condition of the next **elif** block and so on.

If all the conditions are False, the body of the **else** block is executed.

Only one block among the several **if ... elif ... else** blocks is executed based on the conditions.

The **if** block can only have one else block. However, it may have several **elif** blocks.

In the following example, the program checks which **if ... elif ... else** blocks returns True before executing the body of the block. If **boolean1** is found to be True, it will not move further to the rest of the statements. However, if **boolean1** is False, then it will move on to the **elif** statement. If the **elif** statement returns True, then the block is executed, otherwise the **else** block is executed.

```

if boolean1:
    print("boolean1 is True!")

elif boolean2:
    print("boolean2 is True! Also, we know that boolean1 is False!")

else:
    print("Both boolean1 and boolean2 are False!")

```

Notice that Python relies on indentation to define the scope in the code.

If only one statement is to be executed, one for **if**, and one for **else**, we may put it all on the same line.

Shorthand **if**

```

if boolean1: print("boolean1 is True!")

```

Shorthand **if ... else**

```

if boolean1: print("boolean1 is True!") else: print("boolean1 is False!")

```

Conditionals can also be “nested”, meaning we may add a conditional within another conditional. We just need to be mindful of indentation:

```
if boolean1:
    if boolean2:
        if boolean3:
            print("boolean1, boolean2, and boolean3 are all True, but who knows about boolean4!")
        else:
            print("boolean1 and boolean2 are True, but boolean3 is False, and who knows about boolean4")
    else:
        print("boolean1 is True, and boolean2 is False, but who knows about boolean3 and boolean4!")
else:
    if boolean4:
        print("boolean1 is False, boolean4 is True, but who knows about boolean2 and boolean3!")
    else:
        print("boolean1 and boolean4 are False, but who knows about boolean2 and boolean3")
```

3.2 Data Structures

A data structure is essentially a means for storing and organizing a collection of data values.

- Generally composed of “primitive” data types such as Strings, Booleans, and Numeric data types, they can also include other data structures “nested” within them.
- Each has their own set of methods, or a defined action that can be performed on the data structure. Methods are called “on” a data structure using dot notation (i.e. data_structure.method())

There are many different kinds, each with their own relative benefits and uses.

- Sets
- Tuples
- Arrays/Lists
- Dictionaries
- Objects
- Etc.

3.3 Sets

Sets are an unordered collection of unique values that are immutable. Here are the different types of sets:

- Unordered - The elements within a set are not ordered. You can enter them in any order when creating the set, but the order is not maintained.

```
>>> {1, 4, 6, 8, 9, 3, 54, 7, 8, }
{1, 3, 4, 6, 7, 8, 9, 54}
```

- Unique - All elements in a set must be unique and duplicates are not allowed.

```
>>> {1,1,1,2,2,2,3,3,3,4,4,4}
{1, 2, 3, 4}
```

- Immutable - Once defined, you cannot change the values of the elements in the set, but you can add and remove elements from the set.

A set may be defined in one of the two ways:

- `my_set = set()`
- `my_set = {}`

3.4 Set Methods

There are a few built-in methods that can be used on sets:

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another specified set
discard()	Remove the specified item
intersection()	Returns a set that is the intersection of two or more sets
intersection_update()	Removes the items in this set that are not present in other specified set(s)
isdisjoint()	Returns whether or not two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element

symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update())	Inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with another set or any other iterable

3.5 The in Operator

The **in** operator can help us determine if an element already exists in a set by using it to form a Boolean expression:

```
>>> my_set = {'david', 'matthew', 'erik', 'sean'}
>>> 'david' in my_set
True
>>> 'josh' in my_set
False
```

Assignment 3

Creating an Activity Suggestion App

Your client is extremely indecisive with what to do in their free time and wants you to develop a simple program to make suggestions as to what they should do based on a few factors. This program should be able to store:

- A weekday indicator (whether or not it is a weekday) - This should be a Boolean value
- The current weather - this should be a String value
- The funds the user has available to spend - this should be a Numeric value

The client lives in a place where the weather is always either nice, raining, or freezing, so the program should verify that the weather entered matches one of these values.
(HINT: use a set along with the **in** operator!)

After verifying that the weather entered is valid, the program should do the following:

- In the case that it is a weekday:
 - The program should output a reminder stating: You may have work or school today, but here are some suggestions if not...
 - Then, in the case that the weather is nice:
 - The program should output: The weather is nice, so let's do something outside.
 - Then, in the case that the user's available funds are at least 100, the program should output: Let's go to an amusement park!
Otherwise, the program should output: Let's go for a walk in the park.
 - Otherwise, in the case that the weather is raining:
 - The program should output: It's raining outside, so let's do something indoors.
 - Then, in the case that the user's available funds are at least 50, the program should output: Let's go to a museum!
Otherwise, the program should output: Let's stay in and watch movies.
 - Otherwise:
 - The program should output: It is freezing outside! If you leave the house, make sure to bundle up!
 - Then, in the case that the user's available funds are at least 150, the program should output: Let's hit the slopes and go skiing!
Otherwise, the program should output: Let's stay in and drink hot chocolate while watching movies.
- Otherwise,
 - The program should output: It's the weekend! Wooo! Let's decide what to do...
 - Then, in the case that the weather is nice:
 - The program should output: The weather is nice, so let's do something outside.
 - Then, in the case that the user's available funds are at least 100, the program should output: Let's go to a concert at an outdoor venue!
Otherwise, the program should output: Let's go on a hike somewhere.

- Otherwise, in the case that the weather is raining:
 - The program should output: It's raining outside, so let's do something indoors.
 - Then, in the case that the user's available funds are at least 50, the program should output: Let's go to a comedy show somewhere.
Otherwise, the program should output: Let's go on some free tours of the local places of worship!
- Otherwise:
 - The program should output: It is freezing outside! If you leave the house, make sure to bundle up!

Then, in the case that the user's available funds are at least 150, the program should output:
Let's check out a local bar and/or arcade and play games!

Otherwise, the program should output: Let's just call some friends and see if they want to come over and play board games!

Chapter 4

Here's a link to a video on our YouTube channel covering the material in this chapter:

https://youtu.be/N7ztCYBi_28

Data Structures Part II

4.1 Arrays

Arrays are an ordered collection of non-unique values that are all a specified type and are mutable.

- Ordered - The elements within an array have a defined order; the order in which you enter them is the order in which they will remain.

```
>>> array('i', [3,2,1,5,7,6,8,0,9])
array('i', [3, 2, 1, 5, 7, 6, 8, 0, 9])
```

- Non-unique - Unlike a set, you can have multiple of the same value in an array.

```
>>> array('i', [1,1,1,1,1,1,2,2,2,3,3,3,4,4,5,5,6])
array('i', [1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4, 5, 5, 6])
```

- Specified type - When creating an array, you must specify the type of the elements within the array and all elements in the array as they must match that specified type.
- Mutable - You can alter the values of the elements in an array.

```
>>> my_array = array('i', [3,2,1,5,7,8,0,9])
>>> my_array
array('i', [3, 2, 1, 5, 7, 8, 0, 9])
>>> my_array[0] = 0
>>> my_array
array('i', [0, 2, 1, 5, 7, 8, 0, 9])
```

- An array is not built into Python, so it needs to be imported using the **array** module.

```
from array import *
```

- Accessing individual elements:
 - Elements of an array can be accessed via their index value.
 - Arrays are zero-indexed, meaning that the first element has index value 0, the second has index value 1, etcetera

```
>>> my_array = array('i', [1,2,3,4,5,6,7,8,9,10])
>>> my_array[0]
1
>>> my_array[1]
2
```

- You can use the concept of indexing to replace the value of an element with another value.

```
>>> my_array = array('i', [1,2,3,4,5,6,7,8,9,10])
>>> my_array[0] = 9
>>> my_array
array('i', [9, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

4.2 Array Methods

There are several built-in methods that may be used on arrays:

- **append()** - appends a specified element to the end of the array

```
>>> my_array.append(11)
>>> my_array
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

- **pop()** - removes the last element from an array

```
>>> my_array.pop()
11
>>> my_array
array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

- **insert(,)** - inserts an element into the array at a specified index

```
>>> my_array.insert(1,1)
>>> my_array
array('i', [0, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

- **remove()** - removes the first occurrence of a specified element from an array

```
>>> my_array = array('i', [1,2,3,4,3,2,1])
>>> my_array.remove(3)
>>> my_array
array('i', [1, 2, 4, 3, 2, 1])
```

- **index()** - returns the index of the first occurrence of a value in the array


```
>>> my_array.index(2)
1
```

4.3 Dictionaries

Dictionaries are an unordered collection of key-value pairs where values are mutable but keys are not.

- Unordered - Dictionaries have no particular order to them and thus can't be indexed like an array or list.

- Key-value pairs - Items of a dictionary are stored in key-value pairs and each item must have a key and a value, and these are separated by a colon.

```
>>> {'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
{'key1': 'value1', 'key2': 'value2', 'key3': 'value3'}
```

- Dictionaries can be defined using curly brackets or using the **dict()** constructor function.

```
>>> my_dictionary = {}
>>> your_dictionary = dict()
>>> my_dictionary == your_dictionary
True
>>> my_dictionary is your_dictionary
False
```

- Unlike in indexing, values in a dictionary can be referenced using their keys.

```
>>> fav_color_dict = {'David': 'blue', 'Jon': 'red', 'Jane': 'blue', 'Jill': 'purple', 'Paul': 'green'}
>>> fav_color_dict['David']
'blue'
```

- But we need to be careful! You'll get an error if you try to reference a nonexistent key.

```
>>> fav_color_dict['Jack']
Traceback (most recent call last):
  File "<pyshell#44>", line 1, in <module>
    fav_color_dict['Jack']
KeyError: 'Jack'
```

- Dictionaries can be nested such that the value of some key (or keys) in your dictionary is itself a dictionary.

```
>>> contacts_dict = {"David": {"phone": 17195552879, "email": "david@gmail.com", "address": "1234 Somewhere Dr., Denver, CO 80221"}}
>>> contacts_dict['David']['phone']
17195552879
```

- You can also add new key value pairs as a dictionary.

```
>>> fav_color_dict["Jack"] = 'pink'
>>> fav_color_dict
{'David': 'blue', 'Jon': 'red', 'Jane': 'blue', 'Jill': 'purple', 'Paul': 'green', 'Jack': 'pink'}
```

4.4 Dictionary Methods

There are several built-in methods that may be used on dictionaries:

- **get(<key>, <default>)** - Returns the value of the entered key if it exists; otherwise the default value will be returned if present. If no default is given, "none" is returned.

```
>>> contacts_dict.get('David')
{'phone': 17195552879, 'email': 'david@gmail.com', 'address': '1234 Somewhere Dr., Denver, CO 80221'}
>>> contacts_dict.get('Jane', "Jane not in contacts")
'Jane not in contacts'
```

- **keys()** - Returns a list of all the keys present in your dictionary

```
>>> contacts_dict.keys()
dict_keys(['David'])
```

- **values()** - Returns a list of all the values present in your dictionary.

```
>>> contacts_dict.values()
dict_values([{'phone': 17195552879, 'email': 'david@gmail.com', 'address': '1234 Somewhere Dr., Denver, CO 80221'}])
...
```

- **items()** - Returns a list of Tuples (see chapter 28) representing the key-value pairs of your dictionary.

```
>>> contacts_dict.items()
dict_items([('David', {'phone': 17195552879, 'email': 'david@gmail.com', 'address': '1234 Somewhere Dr., Denver, CO 80221'})])
```

Assignment 4

Searchable Address Book

Your client wants you to build them a searchable address book program which they can use to determine if they have contact information for someone and, if so, have the program list out all of the possible ways in which the person of interest could be contacted. This program should store:

- An arbitrary collection of contact methods - build this out yourself with fake values
 - Each contact should have one or more of the following contact details:
 - Home phone number
 - Mobile phone number
 - Work phone number
 - Personal Email address
 - Work Email address
 - Home address
 - Work address
- The name of the individual that the user wants to search for

When run, the program should do the following:

1. First, determine if the entered name exists **in** the collection of contacts
 1. In the case that the contact does exist, the program should output: Contact information found for `<name of contact entered>`!
 1. The program should then output each of the contact details available for that user in the format: `<name of contact entered>` can be reached via their `<contact detail name, for example home phone number>` at `<contact detail value>`
 - b. In the case that the contact doesn't exist, the program should output: No contact information found for `<name of contact>`! Be sure to ask them for their contact details next time you see them.

For example, say my address book contains the following contacts:

- David
 - Mobile phone number: 123-456-7890
 - Work phone number: 098-765-4321
 - Personal email address: david@home-email.com
 - Home address: 1234 Somewhere Dr., Denver, CO, 80221 USA
- Zofia
 - Work phone number: 456-123-7890
 - Work address: 4321 Workplace St., New York, NY, 10001 USA

If the name I enter into the program is David, the program would output:

```
Contact information found for David!
David can be reached via their mobile phone number at 123-456-7890.
David can be reached via their work phone number at 098-765-4321.
David can be reached via their personal email address at david@home-email.com.
David can be reached via their home address at 1234 Somewhere Dr.,
Denver, CO, 80221 USA.
```

Alternatively, if the name I enter into the program is Zofia, the program would output:

Contact information found for Zofia!
Zofia can be reached via their work phone number at 456-123-7890.
Zofia can be reached via their work address at 4321 Workplace St.,
New York, NY, 10001 USA.

Finally, if the name I enter into the program is Jonathan, the program would output:

No contact information found for Jonathan! Be sure to ask them for
their contact details next time you see them.

Chapter 5

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://m.youtube.com/watch?v=71IAh6C95jw&pp=sAQA>

Data Structures Part III

5.1 Lists

Lists are an ordered collection of non-unique values that are of any type and are mutable.

- Ordered - The elements within a list have a defined order, so the order in which you enter them is the order in which they will remain.

```
>>> [0.25, 0.5, 1, 2, 4, 8, 'Hello', 'World', True, False]
[0.25, 0.5, 1, 2, 4, 8, 'Hello', 'World', True, False]
```

- Non-unique - Unlike a set, you can have multiple of the same value in a list.

```
>>> [1,2,2,3,3,3,4,4,4,4,'hello','hello',True,False,True,False]
[1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 'hello', 'hello', True, False, True, False]
```

- Any type - Unlike an array, when creating a list, you do not need to specify the type of the elements within the list and the elements of the list can be of any type needed.
- Mutable – You can alter the values of the elements in a list.

```
>>> my_list = [1,2,3,4,5,6,7,8,9,0]
>>> my_list[9] = 10
>>> my_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Accessing individual elements:
 - Like arrays, elements of a list can be accessed via their index value.
 - Lists are zero-indexed, meaning that the first element has index value 0, the second has index value 1, etcetera.

```
>>> my_list[0]
1
>>> my_list[1]
2
>>> my_list[5]
6
```

- You can use this concept of indexing to replace the value of an element with another value.

```
>>> my_list = [1,2,3,4,5,6,7,8,9,0]
>>> my_list[9] = 10
>>> my_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- Lists can also be negatively indexed. Negative indexing is -1-indexed, meaning the last element in your list is at index -1.

```
>>> my_list[-1]
10
>>> my_list[-4]
7
```

5.2 List Slicing

List slicing is essentially a means for grabbing a portion, or slice, of a list. This is accomplished using indexes in combination with bracket notation and a colon.

- From some index onward - negative indexes allowed!

```
>>> my_list[3:]
[4, 5, 6, 7, 8, 9, 10]
>>> my_list[-3:]
[8, 9, 10]
```

- From some index to another (NOTE: first index is inclusive, second is exclusive!)

```
>>> my_list[3:5]
[4, 5]
>>> my_list[3:-1]
[4, 5, 6, 7, 8, 9]
```

- Up to some index (exclusive!)

```
>>> my_list[:5]
[1, 2, 3, 4, 5]
>>> my_list[:-3]
[1, 2, 3, 4, 5, 6, 7]
```

The slicing operator **mylist[i:j]** returns a new list containing the characters of mylist starting at index i (inclusive) up to index j (exclusive).

5.3 List Methods

Most array methods also apply to lists:

- **count()** - counts the number of elements with the passed value

```
>>> [1,1,1,2,2,2,2].count(2)
4
```

- **extend()** - extends the existing list by appending values of the passed list to the end

```
>>> my_list.extend([11,12,13,14])
>>> my_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

- **reverse()** - reverses the elements of the list

```
>>> my_list.reverse()
>>> my_list
[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- **sort(reverse=False)** - sorts the list into ascending order. One can set reverse=True to sort in a descending fashion. The default value of reverse is False.

```
>>> l = [4,5,2,5,7,9,6,3,2,5,7,8]
>>> l.sort()
>>> l
[2, 2, 3, 4, 5, 5, 5, 6, 7, 7, 8, 9]
```

```
>>> l = [4,5,2,5,7,9,6,3,2,5,7,8]
>>> l.sort(reverse=True)
>>> l
[9, 8, 7, 7, 6, 5, 5, 5, 4, 3, 2, 2]
```

5.4 Lists and Strings – a useful relationship!

Sometimes it can be useful to think of a string as a list of characters as you can apply many list concepts and even some methods (but not all) to a string.

- Slicing

```
>>> 'hello, world!'[:5]
'hello'
```

- Indexing

```
>>> 'hello, world!'[5]
','
```

- Methods – only some, not all!

```
>>> 'hello, world!'.count('l')
3
>>> 'hello, world!'.index('!')
12
```

For example, the above two methods work for both lists and strings.

```
>>> 'hello, world!'.sort()
Traceback (most recent call last):
  File "<pyshell#68>", line 1, in <module>
    'hello, world!'.sort()
AttributeError: 'str' object has no attribute 'sort'
```

Some methods such as *sort()* do not work the same for strings as they work for lists.

- For methods that don't work on a string, simply cast to a list, first, then join back together.

```
>>> my_str = list('hello, world!')
>>> my_str
['h', 'e', 'l', 'l', 'o', ',', ' ', 'w', 'o', 'r', 'l', 'd', '!']
>>> my_str.sort()
>>> ''.join(my_str)
'!,dehllloorw'
```

5.5 Tuples

Tuples are an ordered collection of non-unique values that are of any type and are immutable.

- Ordered - The elements within a tuple have a defined order, so the order in which you enter them is the order in which they will remain.

```
>>> (0.25, 0.5, 1, 2, 4, 8, 'hello', 'world', True, False)
(0.25, 0.5, 1, 2, 4, 8, 'hello', 'world', True, False)
```

- Non-unique - Unlike a set and more like a list, you can have multiple of the same value in a tuple.

```
>>> (1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 'hello', 'hello', True, False, True, False)
(1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 'hello', 'hello', True, False, True, False)
```

- Any type - Unlike an array, when creating a tuple, you do not need to specify the type of the elements within the tuple and the elements can be of different types.

- Immutable - You cannot alter the contents of a tuple once it's set.

```
>>> my_tuple = (1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
>>> my_tuple[9] = 10
Traceback (most recent call last):
  File "<pyshell#80>", line 1, in <module>
    my_tuple[9] = 10
TypeError: 'tuple' object does not support item assignment
```

5.6 Tuple indexing and Slicing

- Like lists, tuples can be indexed and sliced.


```
>>> my_tuple[9]
0
>>> my_tuple[-2]
9
>>> my_tuple[3:-3]
(4, 5, 6, 7)
```

- Tuples are immutable, so unlike lists you cannot modify the elements of a tuple.

```
>>> my_tuple = (1,2,3,4,5,6,7,8,9,0)
>>> my_tuple[9] = 10
Traceback (most recent call last):
  File "<pysHELL#93>", line 1, in <module>
    my_tuple[9] = 10
TypeError: 'tuple' object does not support item assignment
```

5.7 Tuple Methods

There are several built-in methods that may be used on:

- **index(<val>)** - returns the index of the first element of the tuple that matches the value passed

```
>>> my_tuple = (1,2,2,3,3,3,4,4,4,4,'hello', 'hello', True, False, True, False)
>>> my_tuple.index(4)
6
```

- **count(<val>)** - returns the number of instances of the value passed within the tuple

```
>>> my_tuple = ('hello','hello', 3,4,4,4)
>>> my_tuple.count(5)
0
>>> my_tuple.count('hello')
2
```

- Because tuples are immutable, many of the methods that lists and even sets have aren't possible on a tuple:
 - Can't add elements
 - Can't remove elements
 - Can't update the value of elements

Assignment 5

Complete the exercises below, placing the necessary code in the blanks to achieve the desired output

1. Given a list, display it in backward order.

```
my_list = [100, 200, 300, 400, 500]
```

```
_____
```

```
[500, 400, 300, 200, 100]
```

1. Insert the value "7000" at the end of the innermost list.

```
my_list = [10, 20, [300, 400, [5000, 6000], 500], 30, 40]
```

```
_____
```

```
[10, 20, [300, 400, [5000, 6000, 7000], 500], 30, 40]
```

1. Insert the elements 'h', 'i', and 'j' into the innermost list.

```
my_list = ["a", "b", ["c", ["d", "e", ["f", "g"], "k"], "l"], "m",  
"n"]
```

```
_____
```

```
['a', 'b', ['c', ['d', 'e', ['f', 'g', 'h', 'i', 'j'], 'k'], 'l'],  
'm', 'n']
```

1. Given any list, find the value "20" in the list and, if present, replace the first occurrence with 200. Write your code so that the same lines of code can be used for all use cases below.

```
my_list = [5, 10, 15, 20, 25, 50, 20]
```

```
_____
```

```
[5, 10, 15, 200, 25, 50, 20]
```

```
my_list = [6, 11, 16, 21, 26, 51, 21]
```

```
_____
```

```
[6, 11, 16, 21, 26, 51, 21]
```

```
my_list = [5, 10, 15, 21, 25, 50, 20]
```

```
_____
```

```
[5, 10, 15, 21, 25, 50, 200]
```

1. Retrieve the 5th through 8th elements of a list.

```
my_list = [11, 33, 55, 22, 66, 77, 3, 77, 8, 9, 4, 66, 7, 99]
```

```
_____
```

```
[66, 77, 3, 77]
```

1. Retrieve the last 6 elements of a list. Write your code so that the same lines will work for both of the below use cases.

```
my_list = [5, 10, 15, 20, 25, 50, 20]
```

```
_____
```

```
[10, 15, 20, 25, 50, 20]
```

```
my_list = [11, 33, 55, 22, 66, 77, 3, 77, 8, 9, 4, 66, 7, 99]  
            
[8, 9, 4, 66, 7, 99]
```

Chapter 6

Here's a link to a video on our YouTube channel covering the material in this chapter:

https://m.youtube.com/watch?v=TVr2_Lu7MGk&feature=youtu.be

Loops, Iterables and Iterators

6.1 Loops

Loops allow certain portions of the code to execute several times. This repeated execution of a set of statements is called an **iteration**. **Additionally**, loops enable iteration on python objects that are capable of returning its members one at a time.

Types of Loops: for loop, while loop, nested for loop

Iterable objects in Python: Arrays/Lists, Dictionaries, Tuples, Strings

A common function used in loops is **range()**

6.2 for Loop

A for loop is used to iterate over a sequence that may be either a list, a tuple, a dictionary, a set, or a string.

With the for loop, we may execute a set of statements, once of each item in a list, tuple, etc.

For example, this code will print all the items in student_names_list one by one.

```
#define list
student_names_list = ["David", "John", "Simon", "Jack"]
#for loop
for student_name in student_names_list:
    print(student_name)
```

The output is:

```
David
John
Simon
Jack
```

Remember: Strings are arrays of bytes.

This code iterates a string using Loop.

```
#initialize a string
welcome_msg_string = "Hello to Everyone!"
for msg_char in welcome_msg_string:
    print(msg_char)
```

It will iterate and print every character, including spaces, each on a new line.

```
H
e
l
l
o

t
o

E
v
e
r
y
o
n
e
!
```

6.3 while Loop

A **while** loop is used to execute a set of statements or iterate an object until the condition in the while statement is true.

The syntax for a while loop is as follows:

```
while condition:
    any code
    Some operation that eventually turns condition to false
```

Diagram annotations:

- Condition: This must be evaluated as True or False (Eventually!)
- Very Important! Otherwise the code will not stop

For example:

```
fruits = ["apple", "grapes", "bananas", "oranges"]
index = 0
while index < 4:
    print(fruits[index])
    index = index + 1
```

Diagram annotations:

- Condition
- Condition manipulator

Note: while loops require index to index documentation through the members of an object while for loops do n't.

6.4 Nested Loop

It is possible to nest loops.

You can have a for loop inside another for loop or a while loop inside another while loop or one of each inside another.

```
row_count = 1
Outer while loop { while row_count <= 6:
                   print("Row ", row_count)
                   for star in range(0, row_count):
                       print(star)
                   row_count = row_count + 1 } Inside for loop
```

Output is:

```
Row  1
0
Row  2
0
1
Row  3
0
1
2
Row  4
0
1
2
3
Row  5
0
1
2
3
4
Row  6
0
1
2
3
4
5
```

6.5 Iterable Functions

An iterable function in Python is capable of returning its members one at a time, allowing it to be iterated over in a for-loop.

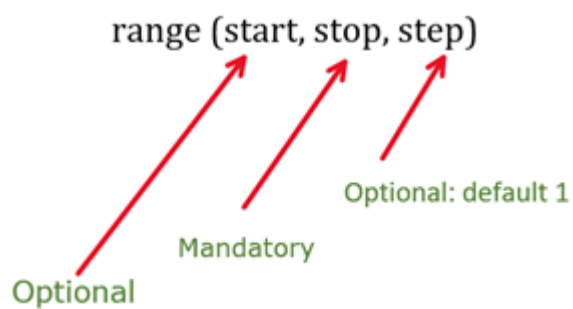
For example, the function **range()** returns a set of numbers:

```
0
1
2
for i in range(5): 3
    print (i)      4
```

```
1
for i in range(1,4): 2
    print (i)        3
```

```
1
for i in range (1,8, 2): 3
    print (i)            5
                        7
```

Syntax:



Similarly, there are **.keys()**, **.values()**, **.iterkeys()**, and **.items()** for iterating through a dictionary.

The function **enumerate()** is used to convert a list into an iterable list of tuples.

iter() is used to iterate through lists, arrays, strings, and others.

6.6 break statement

A **break** statement in Python terminates the current loop and resumes execution at the next statement.

```
items = ["orange", "banana", "toys", "grapes"]
for item in items:
    if "toys" == item:
        print ("I found toys in list of fruits, breaking")
        break
    print("Inside Loop ", item)
print("I am out of loop")
```

```
Inside Loop  orange
Inside Loop  banana
I found toys in list of fruits, breaking
I am out of loop
```

6.7 continue statement

A **continue** statement in python is used to skip the rest of the code inside a loop only for the current iteration. The loop doesn't terminate but continues with the next iteration.

```
items = ["orange", "banana", "toys", "grapes"]
for item in items:
    if "toys" == item:
        print ("I found toys in list of fruits, breaking")
        continue
    print("Inside Loop ", item)
print("I am out of loop")
```

```
Inside Loop  orange
Inside Loop  banana
I found toys in list of fruits, breaking
I am out of loop
```

```
<<< Notice how it did not
      print "Inside loop toys" but
      print "Inside loop grapes"
      before exiting from loop
```


Assignment 6

Q1: Write a program that accepts a number and outputs either “valid” if it is 5-numbers long or “invalid” if it is not.

Example 1: $x = 1234$

Ans: Invalid

Example 2: $x = 45321$

Ans: Valid for processing

Q2: Modify the program in Q1 that adds all the digits of the entered number and display it in the form of an equation using loops.

For example: $x = 56781$

Ans: $5+6+7+8+1 = 27$

Note: the left-hand side of the answer must be dynamic.

Q3: Modify the program that you have written in Q1 to capture the below scenarios:

1. If there is a number 2 in the entered number, then do not add.

For example: $x = 73251$

Ans: $7+3+5+1 = 16$

2. If there is a number 9 in the entered number, then multiply it by 2 and add.

For example: $x = 12949$

Ans: $1+2+18+4+18 = 43$

See how I multiplied 9 with 2 two times.

Chapter 7

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://m.youtube.com/watch?v=bzBjfEB9u9s&feature=youtu.be>

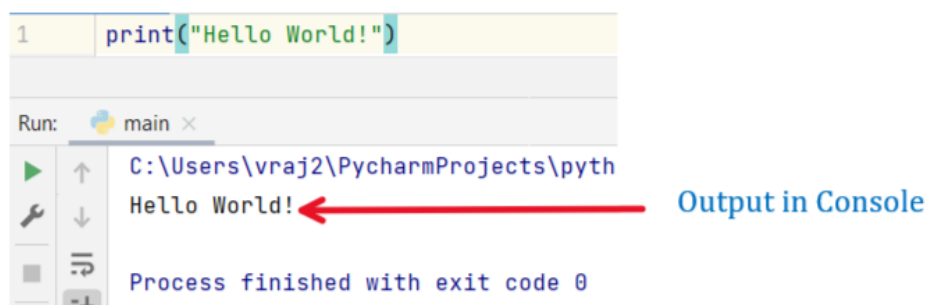
Basic Input and Output, Files and Folders

7.1 Input And Output (I/O)

Inputs are ways to accept information from the user whereas the output is to provide information back to the user.

The inputs and outputs both could be via command-line, user-interface, or files.

Remember the print function we have been using? Guess what, that's a classic example of output in the console.



7.2 Input Via user

Inputs can be accepted from users using the **input()** function in python.

The **input()** function stops the execution of code and gives the control of the program to the user. The user is prompted to enter the data (using the keyboard) and once the user enters the data, the control returns to the program and the function **input()** returns the entered data **in the form of a string** which can be processed further or displayed back to the user.

Always store the value
return from input



```
user_input = input("Enter the number: ")  
print(user_input)
```

```
main x  
C:\Users\vraj2\PycharmProjects\pythonPro:  
Enter the number: 50  
50
```

The **input()** function always treats the input as a string.

The **isInstance()** is a python function that checks the data type in argument 1 with argument 2 and returns the boolean result.

Use the **type()** function to find the arg 2 class type.

```
user_input = input("Enter the number: ")  
print("is User input data type a String?", isinstance(user_input, str))  
print("Type of data: ", type(user_input))
```

arg 1 arg 2
↓ ↓

```
main x  
C:\Users\vraj2\PycharmProjects\pythonProject\venv\Scripts\python.exe C:/U:  
Enter the number: 50  
is User input data type a String? True  
Type of data: <class 'str'>
```

7.3 Input Via Files

Input can also be read from files using the **open()** function.

The open function takes two parameters: *filename* and *mode*.

- filename: it is the path of a file in the system along with file name and it's extension
- mode: it could be

r	for reading the file, and returns "error" if it doesn't exist (default)
a	for appending data into the file and creating a file if does not already exist
w	for writing into a file and creating a file if not already exist
x	for creating a file returning error if it already exists

7.4 Input Via Files – Syntax

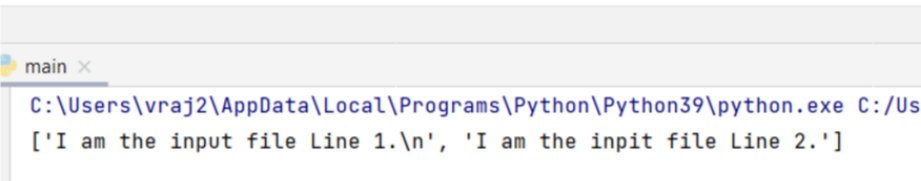
The following functions allow us to read open and read these files;

- **open()** is generally preceded by with keyword to handle file operations easily.
- **readlines()**: this function creates a list of each line in the file.
- **read()** : it will read the entire file and return the text. [Not a good choice if the file is huge]

file name mode Object name to access file

```
with open("example.txt", 'r') as file_obj:
    lines = file_obj.readlines()

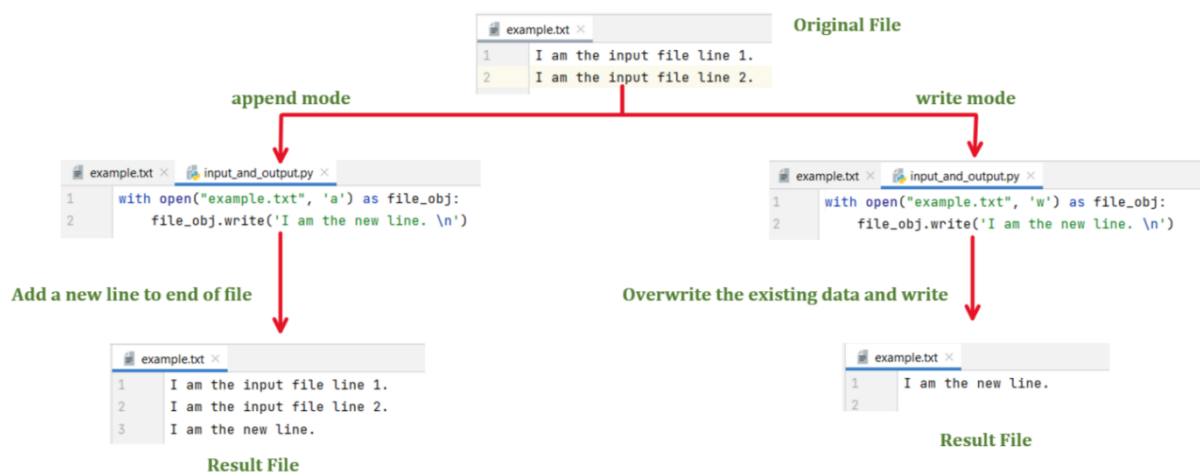
print(lines)
```



7.5 Input Via Files – Write and Append mode

The **append** mode will add the text into the existing file and won't overwrite anything. It can be used on an empty file as well.

The **write** mode will add the text into the existing file and erase everything which was there before.



7.6 Files And Folders

With our current knowledge, we can write the path of every file and perform operations for each of them. What should we do in case that there are 100's of folders and 1000's of files inside them?

- Python provides an easy way to iterate the folders and sub-folders in a path using library **os**.
- Import the library before using the functions.

7.7 Files And Folders – os.walk

```
import os

for root, folders, files in os.walk('C:\Users\vraj2\Desktop\Root folder'):
    print(root)
    print(folders)
    print(files)
```

root path → C:\Users\vraj2\Desktop\trila\Root folder

Folders in root path → ['Folder 1', 'Folder 2', 'Folder 3']

No files in root path → []

Files in folders of root path → C:\Users\vraj2\Desktop\trila\Root folder\Folder 1

['sub-folder 1a']

['file 1a.txt']

C:\Users\vraj2\Desktop\trila\Root folder\Folder 1\sub-folder 1a

['file 1a.txt']

C:\Users\vraj2\Desktop\trila\Root folder\Folder 2

['sub-folder 2a', 'sub-folder 2b']

C:\Users\vraj2\Desktop\trila\Root folder\Folder 2\sub-folder 2a

['file s2a.txt', 'file s2b.txt']

C:\Users\vraj2\Desktop\trila\Root folder\Folder 2\sub-folder 2b

['file s2a.txt', 'file s2b.txt']

C:\Users\vraj2\Desktop\trila\Root folder\Folder 3

- To check if a path exists or not with os, use **os.path.exists**
- To check if a path is to a file and not a directory, use **os.path.isfile**

```
import os

path = 'C:/random_path'
if os.path.exists(path):
    print('Directory path is correct')
else:
    print('Incorrect directory')
```

input_and_output ×

```
C:\Users\vraj2\PycharmProjects\pythonProject1\venv\Scripts\python.exe C:\Use
Incorrect directory

Process finished with exit code 0
```

→ To check if a file is empty with **os**, use **os.path.getsize()**

Below is an example where we first checked if a file is available in the path and then checked the size to avoid python exceptions.

```
import os

file_path = r'C:\Users\vraj2\Desktop\desktop\text file.txt'
if os.path.exists(file_path) and os.path.getsize(file_path) > 0:
    print('File is not empty')
else:
    print('Either file path is invalid or file is empty')
    if os.path.isfile(file_path):
        print('file path is valid and file is empty')
```

input_and_output ×

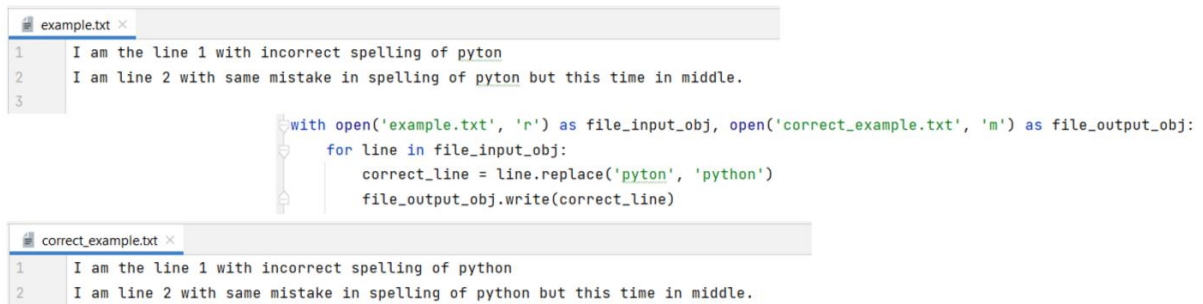
```
C:\Users\vraj2\PycharmProjects\pythonProject1\venv\Scripts\python.exe C
Either file path is invalid or file is empty
file path is valid and file is empty

Process finished with exit code 0
```

7.9 Files And Folders – replace

What if we need to replace a string with another and create a new file? use **replace()**

- **open()** can be used in a single line using 'with' loop to read and write at the same time



The screenshot shows a code editor with two tabs: 'example.txt' and 'correct_example.txt'. The 'example.txt' tab is active and shows two lines of text: 'I am the line 1 with incorrect spelling of pyton' and 'I am line 2 with same mistake in spelling of pyton but this time in middle.' Below the text, there is a Python code snippet that uses a 'with' statement to open 'example.txt' for reading and 'correct_example.txt' for writing. It then iterates over each line in the input file, replaces the word 'pyton' with 'python', and writes the corrected line to the output file.

```
1 I am the line 1 with incorrect spelling of pyton
2 I am line 2 with same mistake in spelling of pyton but this time in middle.
3
with open('example.txt', 'r') as file_input_obj, open('correct_example.txt', 'w') as file_output_obj:
    for line in file_input_obj:
        correct_line = line.replace('pyton', 'python')
        file_output_obj.write(correct_line)
```

The 'correct_example.txt' tab is also visible and shows the same two lines of text, but with the spelling corrected: 'I am the line 1 with incorrect spelling of python' and 'I am line 2 with same mistake in spelling of python but this time in middle.'

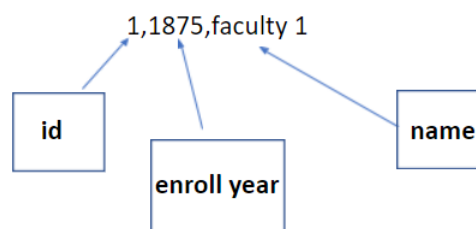
Assignment 7

[Click here to access the folder](#)

Background: There is a folder named “College” provided to you. The folder has many sub- folders and these folders may or may not have files inside them.

The folders inside College have the names “Stream 1”, “Stream 2”, and “Stream 3”.

These folders have files named as Student details.txt and faculty details.txt which has information for students and faculty respectively, given as :



Note: even if the folder has a file inside them, this file may be empty.

Question 1: Write a program that accepts a filename as user input and create a file inside the College folder if the user writes a valid name. Otherwise print “Invalid file name” in console and exit out of the program.

Hint: Users can press ‘Enter’ instead of writing file name and that is a failure condition.

Tip: `exit(0)` will terminate the code.

Question 2: Modify the program in question 1 to write a header inside the file if the user enters a valid name.

id,enroll_year,name,stream,student,faculty,source_file_path

Question 3: Write a program to iterate through all the folders inside the ‘College’ folder and read the data from the files and write them inside the file created in question 2.

Tip: **id**, **enroll_year** and **name** is given in the file.

Stream is the name of sub folder.

Student: if the data is inside the student details, write 'yes' for the student and 'no' for faculty.

Faculty: Similarly, if the file name from where the data is getting read is faculty details, write 'no' for students and 'yes' for faculty.

Source file path is the full path of the file from where the data is being read.

Expected format of output:

```
id,enroll_year,name,stream,student,faculty,source_file_path
1,1875,faculty 1,Stream 1,N,Y,\College\Stream 1\faculty details.txt
2,1879,faculty 2,Stream 1,N,Y,\College\Stream 1\faculty details.txt
3,1880,faculty 3,Stream 1,N,Y,\College\Stream 1\faculty details.txt
4,1862,faculty 4,Stream 1,N,Y,\College\Stream 1\faculty details.txt
1,1990,student 1,Stream 1,Y,N,\College\Stream 1\student details.txt
2,1994,student 2,Stream 1,Y,N,\College\Stream 1\student details.txt
3,1991,student 3,Stream 1,Y,N,\College\Stream 1\student details.txt
4,1999,faculty 4,Stream 1,Y,N,\College\Stream 1\student details.txt
5,1996,student 2,Stream 1,Y,N,\College\Stream 1\student details.txt
```

You may need to learn about **split() functions** in python.

Chapter 8

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://m.youtube.com/watch?v=E8SSABtgYNE&feature=youtu.be>

Functions in Python

8.1 What is Function?

A function can be thought of as a named and stored code block that runs only when it's called.

- Typically this is a block of code that you'd otherwise have to repeat in your program multiple times:
 - Say we didn't use a function for repeated code. What happens when you need to change a portion of that code block? You'd have to change it everywhere the code block appears! Functions help make it easy so that you only have to make the change in one place.

Functions are defined using the **def** keyword .

Functions can optionally accept inputs called parameters (or arguments).

Functions return data to the caller.

We've actually already been using built-in functions throughout the course:

- **print()** - Outputs whatever is entered as a parameter to the console.
- **str()** - Converts whatever is passed as a parameter to a String.
- **int()** - Converts whatever is passed as a parameter to an Integer if this conversion is logical; otherwise an error occurs.

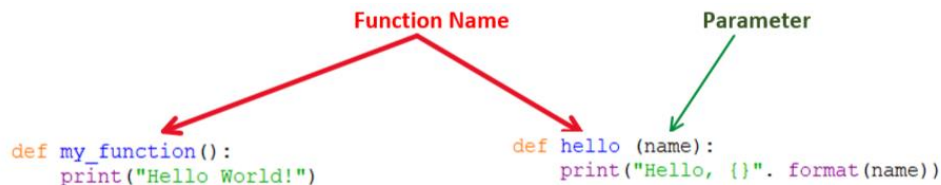
8.2 Defining a Function

Functions are declared using the **“def”** keyword followed by the function’s name, then parentheses, and a colon.

The code that defines the function, aka the function body, is indented underneath the function declaration.

Parameters (also called arguments), if present, are defined within the parentheses of the function declaration:

- These parameters can then be used within the function definition.



8.3 Invoking a Function

Functions are invoked, or called, using their name followed by parentheses.

```
>>> my_function()
Hello, world!
```

If the function requires a parameter (or parameters), you must pass the value(s) you want assigned to the parameter(s) in the parentheses, otherwise an error will occur.

```
>>> hello('World!')
Hello, World!
>>> hello('Class!')
Hello, Class!
```

Two red arrows point from the function calls in the code block above to the `hello` function definition below. One arrow points from the first call `hello('World!')` to the `hello` definition. The other arrow points from the second call `hello('Class!')` to the `hello` definition.

```
def hello (name):
    print("Hello, {}".format(name))
```


```
>>> hello()
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    hello()
TypeError: hello() missing 1 required positional argument: 'name'
```

8.4 Parameters

Parameters (arguments) are essentially inputs to a function that are used within a function.

Order matters - positional arguments

```
def display_exponentiation(base, exponent):  
    print("{}^{} = {}".format(base, exponent, base ** exponent))
```



```
>>> display_exponentiation(5, 2)  
5^2 = 25  
>>> display_exponentiation(2, 5)  
2^5 = 32
```

Parameters are named:

- Repeated names are not allowed

```
def invalid(x, x):  
    print(x)
```

duplicate parameter name

- Instead of invoking with positional arguments, you can use keyword arguments (i.e. parameter names)

```
>>> display_exponentiation(base=10, exponent=2)  
10^2 = 100  
>>> display_exponentiation(base=2, exponent=10)  
2^10 = 1024
```

Parameters can have default values


```
def say_hello(name='World'):  
    print("Hello, {}".format(name))
```

- If a parameter is given a default value, it becomes optional when invoking the function

```
>>> say_hello() ← No param passed, so use default value  
Hello, World!  
>>> say_hello('David!') ← Param passed, so use value of param  
Hello, David!!
```

- If creating a function with both required and optional (i.e. defaulted) parameters, you may define required parameters first and optional parameters last

```
def greet_user(greeting = 'Hello', name):
    print("{} {}".format(greeting, name))
```



```
def greet_user(name, greeting = 'Hello'):
    print("{} {}".format(greeting, name))
```

```
>>> greet_user ("David")
Hello, David!
>>> greet_user ("David", "Hey")
Hey, David!
```

8.5 Return Statement

As mentioned earlier, all functions return data to the caller. The return keyword allows us to specify what that data should be

- Functions without a return statement always return “None”

NOTE: Output printed to the console via the print() function != return_value!

```
>>> return_value = greet_user("David")
Hello, David!
>>> return_value is None
```

- Functions with a return statement, returns the value specified after the return keyword, which can be stored in a variable

NOTE: return statement is not printed to the console when it's stored in a variable, otherwise it is

```
def get_user_greeting(name, greeting="Hello"):
    return "{} {}".format(greeting, name)

>>> user_greeting = get_user_greeting("World")
>>> user_greeting
'Hello, World!'

>>> get_user_greeting("World")
'Hello, World!'
```

- ... but only if the return statement is reached, otherwise “None” is returned!

```
def get_user_greeting(name, greeting="Hello"):
    if greeting in {"Hello", "Hey", "Hi", "Good morning", "Good afternoon", "Good evening"}:
        return "{} {}".format(greeting, name)

>>> get_user_greeting("David", "Goodbye")
>>> user_greeting = get_user_greeting("David", "Goodbye")
>>> user_greeting is None
True
```

You can return multiple values with a single return by separating return values with a comma.

```
def get_integer_quotient_and_remainder(dividend, divisor):
    return dividend // divisor, dividend % divisor
```

- Returned in the form of a Tuple

```
>>> get_integer_quotient_and_remainder(10,3)
(3, 1)
```

- Can “unpack” by assigning to multiple variables

```
>>> quotient, remainder = get_integer_quotient_and_remainder (10,3)
>>> quotient
3
>>> remainder
1
```

8.6 Documentation Strings (Docstrings)

Documentation strings, or docstrings for short, are essentially comments used to document a function’s usage.

- Typically wrapped in triple quotes
- Contains one to four parts:
 - Description
 - Parameters (if applicable)
 - Return value (if applicable)
 - Errors/exceptions raised by the function (if applicable)

```
def get_user_greeting(name, greeting="Hello"):
    """
    Function to build a greeting to a user using

    Args:
        :param name: A string representation of a user's name, i.e. the user to greet
        :param greeting: The string greeting to use when greeting the user, defaults to "Hello"

    Return:
        :return: A greeting in the format "<greeting>, <name>!" If a valid greeting is entered, else None

    """
    if greeting in {"Hello", "Hey", "Hi", "Good morning", "Good afternoon", "Good evening"}:
        return "{}!, {}".format(greeting, name)
```

Functions have a special attribute, `__doc__`, with the value of the docstring that you entered when defining the function. Use this attribute to learn about new functions.

```
print(get_user_greeting.__doc__)
```



Function to build a greeting to a user using

Args:

:param name: A string representation of a user's name, i.e. the user to greet

:param greeting: The string greeting to use when greeting the user, defaults to "Hello"

Return:

:return: A greeting in the format "<greeting>, <name>!" If a valid greeting is entered, else None

8.7 Scope

The scope of a variable or function defines the context in which that variable or function is recognized.

```
x=10
print(x)

def my_func():
    print(x)

my_func()
```

C:\Users\vraj2\Downloads\venv\Script
10
10
Process finished with exit code 0

Variables we've defined and used in class so far have a global scope, meaning they can be referenced from anywhere in the code.

If a variable (or function) is defined within a function, it is part of the local scope of that function and cannot be accessed outside of that scope (i.e. outside of the function).

```
def my_func():
    locally_scoped_variable = "I am not accessible outside of my_func()!"
    print(locally_scoped_variable)
```

```
my_func()
print(locally_scoped_variable)
```

Unresolved reference 'locally_scoped_variable'

```

x=10

def my_func():
    x = 20
    print(x)

my_func()
print(x)

```

C:\Users\vraj2\Downloads\venv\Scripts
20
10
Process finished with exit code 0

You can have multiple variables with the same name, one in the global scope and others inside the scope of function(s): this is called shadowing.

```

def my_func():
    x = 20

```

Shadows name 'x' from outer scope

You can get a reference to a global variable from within a local scope using the global keyword.

```

x=10

def my_func():
    global x
    x = 20
    print(x)

my_func()
print(x)

```

C:\Users\vraj2\Downloads\venv\Scripts
20
20
Process finished with exit code 0

Assignment 8

For each of the below questions, write a function to solve the use case at hand and provide a few examples of using the function. The questions will get increasingly challenging both technically and in terms of being less descriptive as to exactly what is required to accomplish the task as you progress through the assignment. Make sure to document all of your functions thoroughly with docstrings!

1. Write a function that accepts two arguments: a numeric value as well as a list of numeric values. This function will be used to multiply each value in the list by the number passed and return the resulting list after this multiplication has been applied. For example, if I gave the inputs 2 and [1, 2, 3, 4, 5] the result would be [2, 4, 6, 8, 10]
2. Write a function that converts a temperature entered in Fahrenheit to Celsius.
3. Create a function that accepts three arguments corresponding to the current temperature in degrees Fahrenheit or Celsius, an indicator telling if the temp is in degrees Fahrenheit or Celsius, and the current weather forecast (must be one of "cloudy", "raining", or "sunny") and prints out what the caller should wear based on these values according to the following:
 1. First, if the temperature is in Fahrenheit, use the function created for question 2 to convert to Celsius
 2. If the temperature is below freezing, output it is probably best to stay in today
 3. Otherwise, output it's warm enough to go outside and...
 1. In the case the weather is sunny output and you should bring sunglasses
 2. In the case the weather is raining output but you should bring an umbrella
 3. In the case the weather is cloudy output you do not need an umbrella, and you may not need sunglasses
 4. Additionally, in the case the temperature is below 10 degrees, output and you should bring a jacket
 5. In the case the temperature is above 10 degrees but below 15, output and you should bring a sweatshirt
 6. In the case the temperature is 15 degrees or greater, output and you don't need any extra layers!!!

EXTRA CREDIT!!!

1. A factorial, denoted by an integer followed by an exclamation point, is the product of an integer and all the integers below it; e.g. factorial four (4!) is equal to $4 \times 3 \times 2 \times 1 = 24$. Note that $0! = 1$. Write a function that calculates and returns the factorial of a number. **Do not** use an existing function for this, come up with the logic yourself!
2. Write a function that can convert between U.S. customary units and metric units. Here is a link to various conversion tables: <https://www.mathsisfun.com/metric-imperial-conversion-charts.html>. Include the ability to convert between at least the following:
 - a. Feet/inches and meters (and vice versa)
 - b. Miles and kilometers (and vice versa)
 - c. Fluid ounces and milliliters (and vice versa)
 - d. Gallons and liters (and vice versa)

Chapter 9

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://www.youtube.com/watch?v=5CupjuosLkA>

Classes and Error Handling in Python

9.1 Object Oriented Programming

Just about everything in Python is considered an object. Objects have properties/attributes as well as methods.

- Properties are like variables that are a part of an object and they are only referenceable through the object using dot notation.

```
>>> example_class_object.property1
1
>>> example_class_object.property2
'a'
>>> example_class_object.property3
[]
```

- Methods are like functions that are defined within and called on an object using dot notation.

```
>>> my_list_object = [1,2,3,4]
>>> my_list_object.reverse()
>>> my_list_object
[4, 3, 2, 1]
```

9.2 Classes

A class is essentially a template that is used to define a custom object type, it has properties/attributes, its methods, and its behaviors.

Classes are defined using the class keyword.

Properties/attributes and methods of a class are indented underneath the class declaration.

```
class ExampleClass:
    property1 = 1
    property2 = 'a'
    property3 = []
```

Properties →

← **Class Name**

We can create instances of a class by calling the class name followed by parentheses.

```
>>> example_class_object = ExampleClass()
```

Similar to methods, we can access an instance of a class's attributes using dot notation.

```
>>> example_class_object.property1
1
>>> example_class_object.property2
'a'
>>> example_class_object.property3
[]
```

9.3 The `__init__()` Method

Every class has an `__init__()` function, which is called every time you instantiate, or create a new instance of a class.

```
class ExampleClass:
```

```
    def __init__(self):
        property1 = 1
        property2 = 'a'
        property3 = []
```

← **This is essentially the same as the class definition from the previous slide, but using `__init__()`**

We use the `__init__()` function to dynamically assign the values of a new object instance's properties.

Notice that we don't include the self parameter when instantiating a class

```
class ExampleClass:
    def __init__(self, a, b, c):
        self.property1 = 1
        self.property2 = 'a'
        self.property3 = []

>>> example_class_object2 = ExampleClass(2, 'b', {})
>>> example_class_object2.property1
1
>>> example_class_object2.property2
'a'
>>> example_class_object2.property3
[]
```

9.4 Methods

Methods are like functions that are defined within and called on an object using dot notation.

We define methods just like functions, except they are indented underneath a class.

```
class User:
    def __init__(self, name, age, occupation, location):
        self.name = name
        self.age = age
        self.occupation = occupation
        self.location = location
    def introduce_self(self):
        print("Hello, my name is " + self.name)
    def describe_self(self):
        self.introduce_self()
        print("I am {} years old and work as a {} in {}".format(self.age, self.occupation, self.location))
```

Properties {

Methods {

```
>>> user = User("David", 27, "Data Engineer", "Colorado, USA")
>>> user.introduce_self()
Hello, my name is David
>>> user.describe_self()
Hello, my name is David
I am 27 years old and work as a Data Engineer in Colorado, USA
```

9.5 The self Parameter

self refers to the current instance of the class and we use it throughout the class definition to access the classes properties and methods.

self must be the first parameter in every class method and can be optionally followed by other parameters.

9.6 Docstrings

Just like with functions, classes and methods can and should include docstrings.

```
class User:
    """
    Abstract representation of a user

    Properties:
    name: String name of the user
    age: Integer representing how old the user is
    occupation: String occupation of the user
    location: String approximate geographical location of the user
    """
    def __init__(self, name, age, occupation, location):
        self.name = name
        self.age = age
        self.occupation = occupation
        self.location = location

    def introduce_self(self):
        """
        Introduces the user object to someone or something

        Args:
        :param other: String name of the person or thing to introduce self to
        """
        print("Hello, my name is " + self.name)
```

Class docstring

Method docstring

9.7 Errors and Exceptions

A Syntax Error occurs when the syntax, or structure and composition, of the code written is invalid.

- You've likely seen these while attempting assignments and you forget a piece of punctuation (i.e. create a conditional or loop and you forget a colon), or if you misuse an operator. When this occurs, a traceback is returned in the output to the console to help point out the error.

```
>>> While True
...     print("Hello World!")
...
File "<input>", line 1
  While True
    ^
SyntaxError: invalid syntax
```

The traceback tells you the file name and line number (more useful when occurs in a script), the error that occurred, and uses the ^ to point to about where the error occurred

```
>>> if x = y:
...     print ("x and y are equal!")
...
File "<input>", line 1
  if x = y:
    ^
SyntaxError: invalid syntax
```

- Syntax Errors are detected before execution and are always “fatal” meaning you cannot successfully run a program that contains such an error, even if that line of code is never reached.

Even though line 46 is never executed, a SyntaxError still occurs

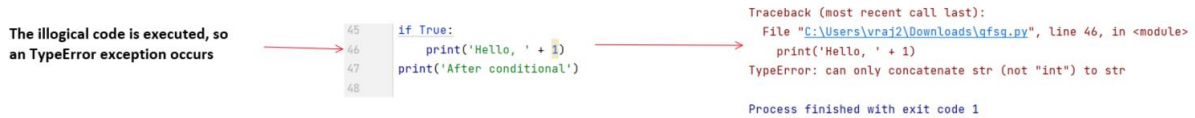
```
46 if False:
47     if x = y:
48         print('ayooo')
```

File "C:\Users\vraj2\Downloads\qfsq.py", line 46
if x = y:
^
SyntaxError: invalid syntax

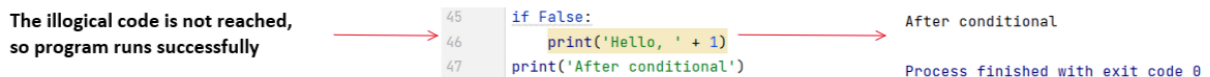
Process finished with exit code 1

Exceptions are essentially errors that occur due to flaws in the program logic.

- You've likely seen an exception when attempting an assignment if you've ever tried to concatenate a number to a string but forget to cast that number to a string first.



- Exceptions are only detected while the program is executing, so you can still run a program that may raise an Exception and you will only see an error occur if the code which produces the Exception is executed.



- There are many different built-in Exceptions that can occur, too many to go over in depth, but common ones include `TypeError`, `IndexError`, `KeyError`, `IndentationError` and `ValueError`.

9.8 Exception Handling

Up to this point, encountering an Exception has meant our program stops execution and cannot proceed further. We can get around this by integrating exception handling into our code base where we essentially anticipate potential errors and include the necessary logic to address these errors so that the program execution can continue.

- try ... except** is a control structure that allows us to try some lines of code and catch exceptions that might occur in that code so we can handle them as needed.



Notice we get exit code 0 meaning the program ran successfully, no exceptions

- We can include as many except blocks as needed in a **try ... except** for handling different exceptions differently. These are evaluated in order, with Exception being the broadest possible.

Exception type

Variable name for referencing the exception

```

x = '10'
try:
    print(9+x)
except TypeError as te:
    print("A TypeError occurred: " + str(te))
except ValueError as ve:
    print("A ValueError occurred: " + str(ve))
except Exception as e:
    print("An unexpected error occurred: " + str(e))

```

A TypeError occurred: unsupported operand type(s) for +: 'int' and 'str'

Process finished with exit code 0

NOTE: except Exception should always come last otherwise other except blocks will never be reached!

Say we want to actually fix the code and continue on in execution, we can do that in our except blocks:

```

x = '10'
try:
    print(9+x)
except TypeError as te:
    print("A TypeError occurred, x must not be a number, cast and retry")
    print(9 + int(x))

```

A TypeError occurred, x must not be a number, cast and retry

19

Process finished with exit code 0

Alternatively, say we do want the exception to interrupt execution after adding our logging statement, we can use the raise keyword to continue to raise the exception to the user's level.

```

44 #x = '10'
45 try:
46     print(9+x)
47 except TypeError as te:
48     print("A TypeError occurred: " + str(te))
49 except Exception as e:
50     print("An unexpected error occurred: " + str(e))
51     raise e

```

An unexpected error occurred: name 'x' is not defined

Traceback (most recent call last):

File "C:\Users\vraj2\Downloads\qfsg.py", line 51, in <module>

raise e

File "C:\Users\vraj2\Downloads\qfsg.py", line 46, in <module>

print(9+x)

NameError: name 'x' is not defined

Process finished with exit code 1

Assignment 9

Your client needs you to create a program that will allow the creation of users in their backend system. All the work of saving this to their system is already in place, but they need you to create the customer facing side of the application that will accept input from the user in order to build a User object that can then be stored in their system. They've provided the following program requirements:

1. The program must elicit input from the user for populating the User object attributes
2. The program must include data validation to ensure that values given by the user are correct per the property requirements (see properties under 3.a below)
3. User objects must be created using a User class.
 1. This class should have the following properties/attributes, each following the mentioned requirements belows:
 1. First name - cannot be an empty string
 2. Last name - cannot be an empty string
 3. Age - must be a valid number between 18 and 100. If the user is under 18, stop execution. Stored value must be in integer format
 1. Include exception handling for the possibility that the user enters a non-numeric string for this value. In the case that this occurs, print `Please enter your age in numeric format` to the console and raise the exception so the program exits
 4. Address - Must contain 6 parts: an address number, street name, city, province/state, ZIP code, and country - HINT: see `split()` method for strings
 5. Birth date - Optional
 6. Phone number - must contain at least 10 digits
 7. Email address - must contain at least 5 characters including an "@" **symbol** and a "."
 - b. This class must have a method for getting the User objects full name based on their first and last name.

Once the user object is created, it should print the following to the console:

Hello <users full name>,

Thank you for registering with the following information:

First Name: <first name>

Last Name: <last name>

Age: <age>

Address: <address>

Birthday: <birthdate> if given, otherwise output `Not provided`

Phone Number: <phone number>

Email Address: <email address>

Chapter 10

Here's a link to a video on our YouTube channel covering the material in this chapter:

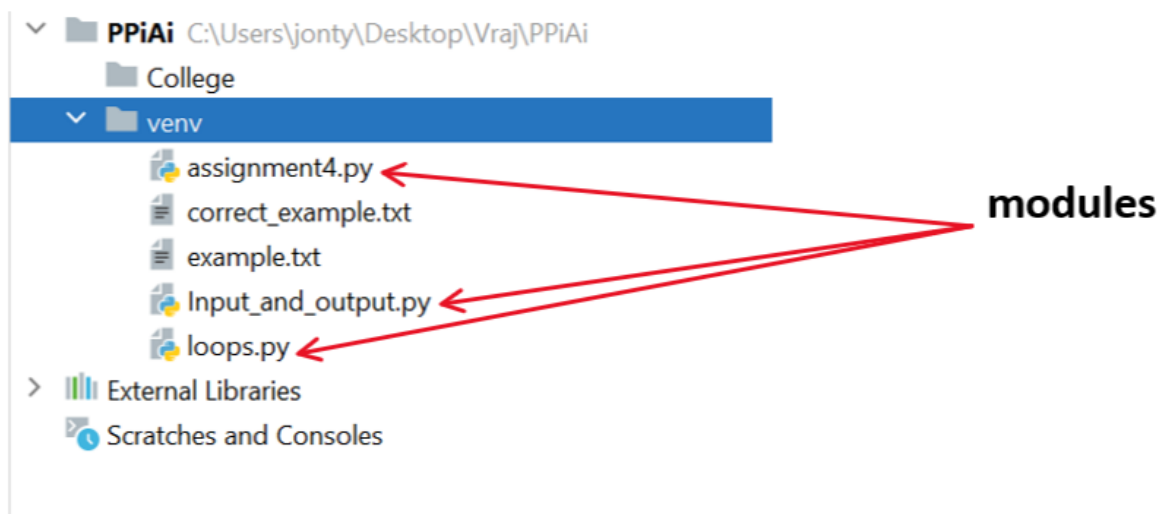
https://www.youtube.com/watch?v=R6p_maOzlxw

Modules and Packages in Python

10.1 Module

A module is a file which consists of python code. Eg: hello_world.py

A module allows you to organize your python code logically. It's a good programming practice to keep related code together in a file.



10.2 Module – Import statement

You can use any python file as a module using an import statement.

```
Input_and_output.py × loops.py × assignment4.py × final.txt ×
1 welcome_text_1 = "I am assignment.py module variable 1"
2 welcome_text_2 = "I am assignment.py module variable 2"
3
Input_and_output.py × loops.py × assignment4.py × final.txt ×
1 import assignment4
2 print(assignment4.welcome_text_1)
3
4
loops ×
C:\Users\jonty\anaconda3\python.exe C:/Users/jonty/Desktop/Vraj/PPiAi/venv/loops.py
I am assignment.py module variable 1
```

I can access the variable defined in another module using dot operator

When the interpreter encounters an import statement, it executes the entire code of the module.

```
Input_and_output.py × loops.py × assignment4.py × final.txt ×
1 welcome_text_1 = "I am assignment.py module variable 1"
2 welcome_text_2 = "I am assignment.py module variable 2"
3 print(welcome_text_2)
4
5
Input_and_output.py × loops.py × assignment4.py × final.txt ×
1 import assignment4
2 print(assignment4.welcome_text_1)
3
4
loops ×
C:\Users\jonty\anaconda3\python.exe C:/Users/jonty/Desktop/Vraj/PPiAi/venv/loops.py
I am assignment.py module variable 2
I am assignment.py module variable 1
```

Check the order. It firsts print the statement of the import module.

10.3 Module – from statement

Python's **"from"** keyword lets you import specific attributes from a module into the current namespace.

```
from assignment4 import welcome_text_1
print(welcome_text_1)
```

See how we don't need dot operator anymore.

Use **(*)** symbol to import all names from a module.

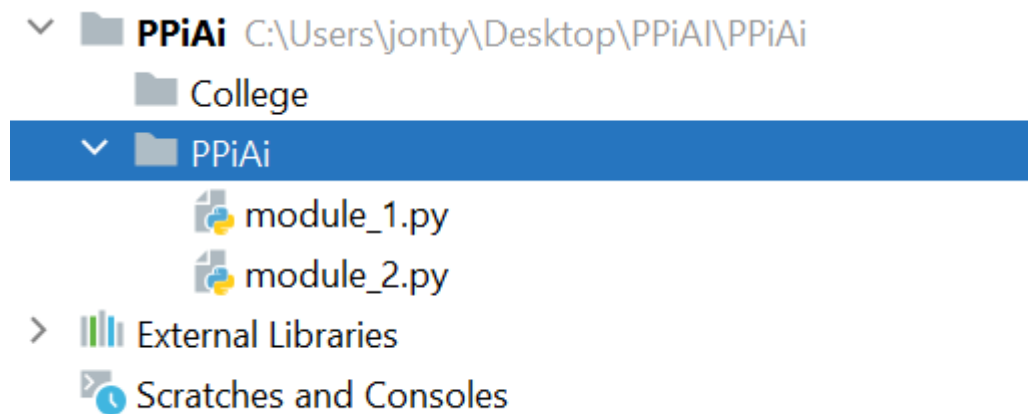
```

Input_and_output.py x loops.py x
1 from assignment4 import *
2 print(welcome_text_1)

```

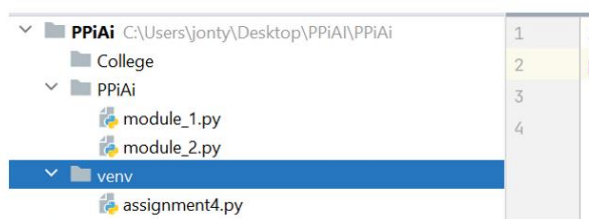
10.4 Package

A package consists of multiple python files. Basically, it is an entire folder structure that consists of modules.



10.5 Package – import

A package consists of multiple python files. Basically, it is an entire folder structure that consists of modules.



Note how can I access module_1 from module_2 because they are in same package

whereas I need to give the package name if the module is outside package

```

assignment4.py x module_1.py x module_2.py x
1 import module_1
2 print(module_1.welcome_text_1)

```

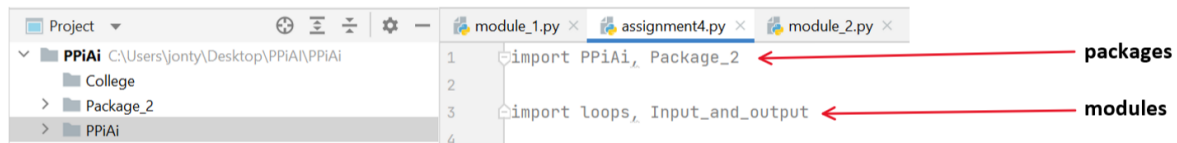
```

module_1.py x assignment4.py x module_2.py x
1 import PPiAi.module_1
2 print(PPiAi.module_1.welcome_text_1)
3

```

10.6 Multiple import – package and module

We can use comma (,) to import multiple modules or package in one line.



Use as keyword to refer to import modules as custom names.

```
import PPiAi.module_1 as my_module
```

```
print(my_module.welcome_text_1)
```

10.7 Why are packages and modules are needed?

The answer is “Modular Programming.”

Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual modules can then be combined like building blocks to create a larger application [1].

Advantages:

- Simplicity
- Maintainability
- Reusability
- Scoping

[1] <https://realpython.com/python-modules-packages/>

10.8 Python Packages

Python is very popular among developers nowadays and one of the reasons for that are mature pre-written, easily available packages to do the task.

- *Pandas* - open source data analysis and manipulation tool
- *Numpy* - library that supports multi-dimensional arrays and matrices and high-level math functions
- *Requests* - HTTP library for performing various HTTP actions with ease

- *Flask* - a micro web framework used for web development use cases and many more applications

Some packages come pre-installed when you install python, so you just need to import them

- *io* - contains functions for operating system level operations (eg. I/O)
- *math* - contains various predefined mathematical functions and values
- *sys* - contains functions and attributes of the python runtime environment
- *datetime* - contains date and time related functions
- etc...

10.9 pip

pip is a package-management system for python that facilitates installing and managing python packages

- It's name is a recursive acronym, pip = pip installs packages
- Comes pre-installed with most python distributions

If you need a package that isn't pre-installed with your python distribution, we typically use pip to install it

```
import requests
```

↓

```
(venv) davidsciacca@Davids-Macbook_Pro untitle %pip install requests
```

↓

```
|import requests
```

You can find and read about available packages and how to install and configure them on pypi.org

Assignment 10

A. Create a package in the project named geometry. Inside the package create 2 modules.

Q1. Create a module 'Areas.py' that has 3 functions in it.

Function 1: calculate the area of circle based on radius.

Function 2: calculate the area of rectangle based on length and width.

Function 3: calculate the area of square based on side length.

Q2. Create another module 'Volume.py' that has 3 functions in it.

Function 1: calculate the volume of cylinder based on radius and height.

Function 2: calculate the volume of cuboid based on length, width, and height.

Function 3: calculate the volume of cube based on side length.

B. Call the functions defined in both the modules from a python file outside the package.

Note: import math package to get the value of pi for calculating area of circle and volume of cylinder.

Chapter 11

Here's a link to a video on our YouTube channel covering the material in this chapter:

<https://www.youtube.com/watch?v=poCq5gUP4r4>

Unit Testing in Python

11.1 Testing in Python

Manual testing - manually running code w/ various inputs and observing outputs to ensure they match expectations

- Simple and intuitive - we're already doing this
- Time consuming - each test requires creating different inputs and digging through outputs to verify functionality. If you make a change, you need to manually test each scenario again .

Automated testing - test your code with code

- This takes up front work to write functions to exercise your code and programmatically verify outcomes.
 - If you make a change, retesting is as simple as rerunning your testing script.

Integration testing - tests how all component of a program work together (i.e. testing an entire program)

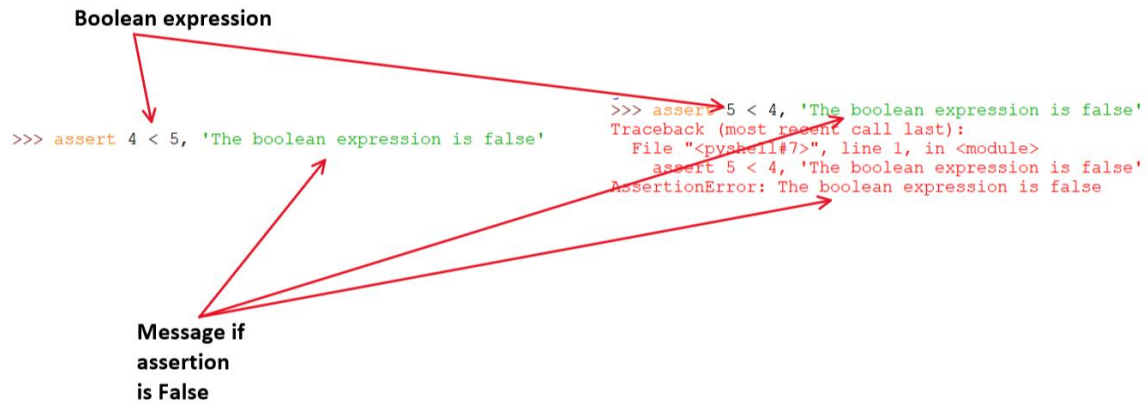
- If it is not working as expected, it can be difficult to know exactly where the issue is coming from.

Unit testing - tests individual components (i.e. units) of a program

- We do not test the entire program in a single unit test, but all of the individual parts that compose that program in distinct unit tests
 - Allows us to identify issues in individual aspects of our code so we can make targeted fixes
 - Typically done before integration testing

11.2 assert Keyword

The **assert** keyword is used to affirm that a boolean expression evaluates to True and raises an AssertionError with a specified message if it is False.



This can be used to verify that a function is returning an expected value

```
>>> assert sum([3,4,5]) == 12, "Result should be 12"
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    assert sum([3,4,6]) == 12, "Result should be 12"
AssertionError: Result should be 12
```

11.3 unittest Package

unittest is a unit testing framework inspired by unit testing frameworks from other languages. It often comes preinstalled with your Python distribution.

This is how you would use a package:

- Import it into your unit testing file as you do other packages (it should be its own file)

```
import unittest
```

- Once imported, create a class that is a subclass of `unittest.TestCase`

```
class TestListMethods(unittest.TestCase):
```

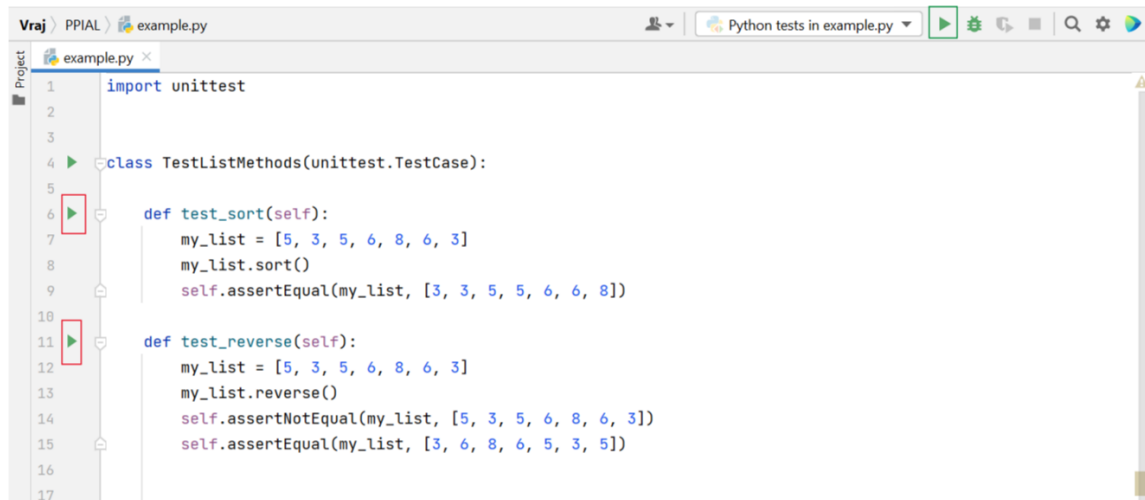
- Define methods within that class to perform various tests

```
def test_sort(self):
    my_list = [5,3,5,6,8,6,3]
    my_list.sort()
    self.assertEqual(my_list, [3,3,5,5,6,6,8])
```

```
def test_reverse(self):
    my_list = [5,3,5,6,8,6,3]
    my_list.reverse()
    self.assertNotEqual(my_list, [5,3,5,6,8,6,3])
    self.assertEqual(my_list, [3,6,8,6,5,3,5])
```


Once your tests are defined, you can run them in a few ways

- Run the file itself via your PyCharm IDE
- Selectively run individual tests using the run button near the file line numbers



- Run from the command line

```
(venv) davidsciacca@Davids-MacBook-Pro untitled % python3 -m unittest example.py
```

- When you run your tests, you'll get an output to the console indicating the results of your tests

```
..
Ran 2 tests in 0.000s
OK
```

- If any of your tests fail, the test report will say so

```
def test_reverse(self):
    my_list = [5, 3, 5, 6, 8, 6, 3]
    #my_list.reverse()
    self.assertNotEqual(my_list, [5, 3, 5, 6, 8, 6, 3])
    self.assertEqual(my_list, [3, 6, 8, 6, 5, 3, 5])

F.
=====
FAIL: test_reverse (example.TestListMethods)
-----
Traceback (most recent call last):
  File "/Users/davidsciacca/dev/src/kin_initiative/untitled/example.py", line 14, in test_reverse
    self.assertNotEqual(my_list, [5, 3, 5, 6, 8, 6, 3])
AssertionError: [5, 3, 5, 6, 8, 6, 3] != [5, 3, 5, 6, 8, 6, 3]

Ran 2 tests in 0.000s

FAILED (failures=1)
```

11.4 unittest Methods

The unittest package comes with a number of built-in methods to use in your unit tests :

- **assertEqual(<actual_value>, <expected_value>, <message>)** - asserts two values are equal
- **assertNotEqual(<actual_value>, <unexpected_value>, <message>)** - asserts values are not equal
- **assertFalse(<boolean_expression>, <message>)** - asserts boolean/boolean expression is False
- **assertTrue(<boolean_expression>, <message>)** - asserts boolean/boolean expression is True
- **assertIn(<value>, <container>, <message>)** - asserts value is a member of a container object
- **assertLess(<value>, <greater_eq_value>, <message>)** - asserts first value is less than or equal to second value
- **assertGreater(<value>, <less_value>, <message>)** - asserts first value is greater than second value
- **assertGreaterEqual(<value>, <greater_eq_value>, <message>)** - asserts first value is greater than or equal to second value
- **assertIsInstance(<value>, <obj_value>, <message>)** - asserts the first parameter is an instance of the object type passed in the second parameter

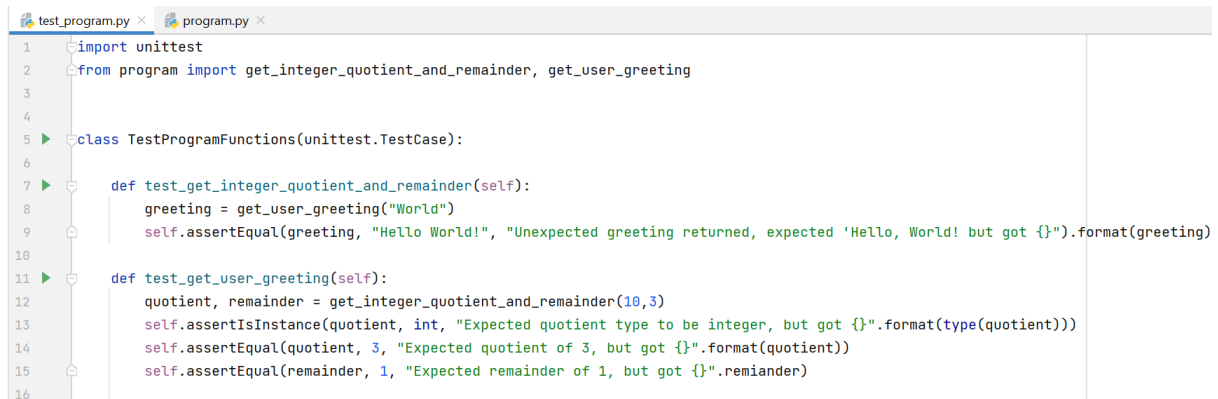
The message parameter is used to specify a custom message in the case that an AssertionError is raised.

11.4 Testing Custom Functions with unittest

You can test your own custom code via the **unittest** package by creating a testing file and importing the functions you'd like to test into the testing file.



Once you've imported your code into your testing file, you can use them within your testing class to validate their functionality.



```
1 import unittest
2 from program import get_integer_quotient_and_remainder, get_user_greeting
3
4
5 class TestProgramFunctions(unittest.TestCase):
6
7     def test_get_integer_quotient_and_remainder(self):
8         greeting = get_user_greeting("World")
9         self.assertEqual(greeting, "Hello World!", "Unexpected greeting returned, expected 'Hello, World! but got {}'.format(greeting)
10
11     def test_get_user_greeting(self):
12         quotient, remainder = get_integer_quotient_and_remainder(10,3)
13         self.assertIsInstance(quotient, int, "Expected quotient type to be integer, but got {}".format(type(quotient)))
14         self.assertEqual(quotient, 3, "Expected quotient of 3, but got {}".format(quotient))
15         self.assertEqual(remainder, 1, "Expected remainder of 1, but got {}".format(remainder))
16
```

Assignment 11

For the assignment this week we are going to go back to our assignment from Week 8 on Functions and create unit tests for the functions created in that assignment. Using your own submission from week 8, create unit tests to verify the functionality of the code you wrote making sure to cover the following test scenarios:

1. For the function created as the answer to question 1, verify that:
 1. your code works with different multiplier values minimally including 0, 2, and 5
 2. Your code works with different lists including lists with varying sizes
1. For the function created as the answer to question 2, verify that:
 1. 32F is converted to 0C
 2. 77F is converted to 25C
 3. At least one other test scenario
2. For the function created as the answer to question 4, verify that:
 1. 0! Equals 1
 2. 4! Equals 24
 3. At least 3 other test scenarios
2. For the function created as the answer to question 5, verify that:
 1. Units passed in feet are converted to meters
 2. Units passed in inches are converted to meters
 3. Units passed in meters are converted to feet
 4. Units passed in miles are converted to kilometers
 5. Units passed in kilometers are converted to miles
 6. Units passed in fluid ounces are converted to milliliters
 7. Units passed in milliliters are converted to fluid ounces
 8. Units passed in gallons are converted to liters
 9. Units passed in liters are converted to gallons
 10. Passing 1 foot returns 0.3048 meters
 11. Passing 12 inches returns 0.3048 meters
 12. Passing 3048 meters returns 10,000 feet
 13. Passing 1 mile returns 1.6039 kilometers
 14. Passing 1.6039 kilometers returns 1 mile
 15. Passing 1 fluid ounce returns 29.574 milliliters
 16. Passing 30 milliliters returns over 1 fluid ounce
 17. Passing 1 gallon returns 3.7854 liters
 18. Passing 4 liters returns over 1 gallon
2. **EXTRA CREDIT:** Update the function from Q3 to return instead of print recommendations, then verify that:
 1. Both 30F and -1C return "it is probably best to stay in today" - 0.25 pts extra credit
 2. Verify all testing scenarios you can think of (i.e. all combinations of temp and weather that would produce distinct outcomes) - 0.25 pts extra credit

Bibliography

1. Official Python website - <https://www.python.org/>
2. Python Notes for Professionals – can be downloaded for free from:
<https://books.goalkicker.com/PythonBook/PythonNotesForProfessionals.pdf>
3. W3 Schools – Python Tutorial - <https://www.w3schools.com/python/>
4. Learn Python platform - <https://www.learnpython.org/>
5. Python for Beginners - <https://www.pythonforbeginners.com/>
6. Google Python Learning Guide - <https://developers.google.com/edu/python>
7. Python Spot - <https://pythonspot.com/>
8. Python Guide - <https://docs.python-guide.org/>
9. Dive Into Python 3 - <http://www.diveintopython3.net/>