Content

Exercise 0 (Set up)

Some believed programming was meant for some particular individuals with some particular skills, which before felt almost true, due the complexity of the tokens used to instruct the computer at the time. Guido van Rossum and the community, made it possible and simpler, through Brian Kernighan and Dennis Ritchie, for anyone with the least interest in learning how to instruct a computer to perform some basic task, the opportunity to do so. It is our dream that many would also learn how to code by introducing them to programming in Python from ground zero.

Aim

After reading this book and doing almost all the exercises and projects, the reader is guaranteed to muster, at the end, the sense and skill in programming using Python and would be able to adapt or translate the concept to other programming languages.

Who is the book meant for

This book is suited for individuals from any work of line, novice to expert, first timers - noobs to veterans. We trust anyone who gets hold of this book could speak, read and understand English - can hold a conversation in English language as this book is written in English.

Prerequisite

We expect the reader to be patient, determined, committed, genuine and not necessarily passionate but interested in programming. The reader must have a platform for coding, such as a laptop, desktop, which we recommend greatly, a tablet or a smart phone. Whatever experience the reader may have is a plus.

Set up environment

The code presented in this book would run on any major operating system. The reader may need, a development environment which we shall use to write and test the code.

There are basically three ways to set up and either is okay as far as the reader is confident in the environment chosen. We assume the user is using a PC.

Installing Python

Install Python from the Python website. Windows user may make reference to this video for assistance.

Unix and Linux users

Unix and Linux user may have Python already installed on their PC. To test whether you have Python installed, open the terminal (command-line) and type, `python` then press enter. For the Linux users, also do for `python3` since we are interested in the features of `python3` . Something like,

```
Python 3.8.2 (default, Apr 27 2020, 15:53:34)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> |
```

then we are good to go. Just make sure that Python 3.x.x used. This also goes for the Windows users too.

Copy and past, `"Hello there.."` and press enter. What did you see?

IDE: Pycharm

IDE stands for Integrated Development Environment. It comprises of a text editor and the interpreter bundled and shipped together. We recommend Pycharm. Download it here.

IDLE: Python's - windows

Windows users may have access to the default IDLE that comes packaged with Python.

Text Editor and command-line: vscode

At this option, the user has terminal (command-line, shell, ...) and a text editor such as notepad, notepad++, sublime, vscode, nano, etc. So user makes use of the editor and interpreter through the terminal. We recommend vscode as it has the terminal inside the text editor so no need to go back and forth. Install vscode from here.

Note

It is a little bit hard to navigate through the directories as a noob and if user insists on using the command-line and text editor, we recommend the watch this video for windows user and video. There is more also here. You can also reference our commandline presentation, using powershell as administrator on windows

Structure of the book

This book uses exercise instead of chapter, which should be the same. It is majorly divided into five parts.

1. Exercise 1 to 8 covers the fundamentals.
2. Exercise 9 to 13 covers some operators, loops and functions.
3. Exercise 14 to 17 covers some built-in data structures.
4. Exercise 18 to 22 covers some exceptions, file handling, OOP, modules and unit testing.
5. Exercise 23 to 25 covers Git, SQL and SQLite3.

How to read this book

We recommend the reader reads through each exercise based on the major divisions. Like any other skill, it requires time to horn thus, spend as much time as needed to complete each exercise diligently. It is sometimes recommended to glance through a book before actually reading it. Do that, it may help.

- First timers really do require the fundamentals
- Experienced but new to Python, should skim through the fundamentals

How to get assistance reading this book

Get assistance reading this book from Swift Python GitHub page or send us an email at our email, with the subject as swift-python. There are other programming platforms on telegram, Facebook or even copy and pasting the error into google or just googling the problem, checking stackoverflow and other coding platforms.

Feed back

We have worked diligently through this book and a typo or misinterpretation of a concept may be lurking somewhere. Thus readers may send us a feed back through our email or comment on it on Swift Python GitHub page, open an issue. Readers may also send a pull request too. Therefore we are opened to suggestions on improvement in any form.

#

Exercise 1 (Creating a variable)

A variable is a placeholder (a name, a container) given to an address (a memory location) of a value in memory. You can think of a variable as a box, which has a name. This box will have a value and whenever we reference (call) this variable (name of the address), we then get the value stored at that address. To create a variable and assign it a value, all we have to think about is an equation. `Eg: x = 2` . `x` is the variable (the placeholder) that points to the address of the value, `2` . So `2` is the value. A single (one) equal-to sign, `=` , is known as the assignment operator. It takes what is on the right (the value) and assigns (puts) it into what is on the left (the variable). It takes the value and puts it into the variable. In Python, the data type of a value is determined or controlled dynamically (meaning as the program runs) so we don't have to worry about it.

More examples:

```
# variable assignment
x = 3
city = "Albuquerque"
age = 45
weight = 23.7
grade = "A+"
```

All these are examples of variable assignment (varaible creation) of different data types.

Practicals

- Create 10 variables and assign them any values suitable (perhaps, as above).

Summary:

- Variable represents the name given to the memory address.
- Calling the variable returns the value
- The Value stored in a variable can change anytime - thats why we call it a variable
- To create a variable, think about an equation, LHS = RHS. LSH is the variable_name, RHS is the value . `Eg: fullname = "John Doe"`

Exercise 2 (Data types)

A data type is basically, the kind of value (data) a variable can hold or store. Think of them as the kind of data we use in our program. More technically, a data type dictates the size of memory to be allocated for a value.

There are basically two types of data which are written in different forms. We have Numbers and Text (broadly speaking). Have a brief read on C's data type - by the way, Python was implemented in C.

For the numbers we have `integer` also know as `int` and we have `float` which is also know as a `float` . Floats are basically decimal numbers - reals. For the Text we have `string` also known as `str` . A string is a collection of zero or more characters, enclosed in single or double quote.

Examples

```
Eg of int: -3, -5, 2, 58, ...
Eg of float: 1.0, 2.5, -3.000232, ...
Eg of str: "", "John", 'Github', 'Python', "swift Python", '3', '-3.464', '+32-1', ...
```

Note

- `""` or `''` denote an empty string
- '2' is not the same as 2. Why? this is because `'2'` is a `string` and `2` is an `integer` . `'2'` is a string because it is enclosed in a quote.
- A string is basically a text of any characters or just a sentence
- If we want an output say, `this is mashud's car` , then what we have to do is to use double quote. `Eg: "Mashud's car"` or we have to escape it. `Eg: 'Mashud\'s car'` . We encourage the former.

When to use `int` or `float`

Use `int` when the value of interest is an integer, a discrete data, such as age, number of people, number of babies, etc. For any value that can be counted, use `int` .

```
my_age = 45
number_of_babies = 3
```

Use `float` when the value of interest is a real, a continues value (a real or decimal value) such as the weight, heaight, time, speed, etc. Any value that can be measured, use `flaot` .

```
my_weight = 125.5
pi = 3.14
```

Type hinting

Python has a way of hinting the type of data passed into a variable. This is done by annotating the variable. It follows this format, `var_name:type=value` . Note that this is not neccearily something we really should do - not compulsory. It will help catch some error though. We the developers (the team) decide whether we would like to use it or not.

```
age:int = 34
name:str = "Veldora"
weight:float = 120.50
```

Variable name rules

Consider these whenever we want to create a variable.

- The variable must begin with a letter, `[a - z, A - Z]` or an underscore `_`
- followed by any other character(s) such as a letter `[a - z, A - Z]` , numbers `[0 -9]` or an underscore `_`
- These must exclude any other character since these character may have special meanings in python
- varible name are case sensitive thus `name` and `Name` or any generic of `name` are not the same
- A variable name must not be a reserved key words
- PEP-8 has more namings

Practicals

1. Look into `Exercise 1 ( creating a variable)` , and for the 10 varaibles we created earlier, state their data types
2. What will be the data type of the following values, `'University of github', 'swift-Python', 100, 12.0, 12, 'B+', '1 + 1', 345.0, 360, 3.1432, "sipping ice cream from a linux cup"`
3. Now create a variable with a suitable name (the name of the variable should be in a way that describes or gives an information about what the value of interest is) `Eg: user_ip = '123.123.100.134' and user_name = 'John Doe'`

Summary:

- Data type refers to the kind of values we make use of in our program
- The Basic data types we have are the integer (int), float (float) and string (str) but there are also boolean values.
- A string made up of zero or more charaters enclosed in a single and a double quote. `name = ''` is an empty string.
- Use int for counting values and float for measuring or continous values.
- Once in a while go through PEP-8

[md2pdf][https://www.markdowntopdf.com/]

Exercise 3 (More on data types and comments)

There are other data types apart from those mentioned earlier in `Exercise 2 (Data types)` . We shall discuss `boolean` data type.

This data type can only be `True` or `False` . Boolean values are generated when values are compared or when there is a condition. `Eg 1 < 2 is True, 1 > 2 is False` .

Examples

```
is_online = True
is_swift = True
is_human = False
```

Note

Boolean values are case sensity. Upper-case `T` for `True` and `F` for `False` , in python.

Comment

A comment is a piece of text that mostly should tells why or what we are trying to achieve. The `#` symbol, placed (inserted) at the beginning of the line, comments the line (the line is ignored by the interpreter).

Examples

```
# Hello world program
# Display Hello world for the user to see
# Useing the print function
print("Hello world")
```

The first 3 lines starting with `#` are comments and they are ignored during execution. You can also comment out some lines during debugging.

Note

Whatever that comes after the `#` would be ignored. `print('hello world') # greetings` , *greetings* would be ignored.

Praticals

1. In `Exercise 2 (Data types)` , add comments to your solutions to give the why/what the line is doing or meant to do (remember that not all lines needs comment if the code is discriptive itself).
2. - How many comments are there in the code below?
   - Which parts (line numbers) would be displayed and why?

```
1 # John Doe - profile
2 print("My name is John Doe") # yes that's my name
3 print("He washes my hair")
4 # i like to dance in the rain print("This costs $25.00")
5 my_name = "Danny Doe Dan"
```

Summary

- A boolean value is `True` or `False` .
- These values are generated during comparison.
- A comment is supposed to explain, "the" what or why in the code (line)
- We can also use it to state how we want to reach our solution
- Use `#` to create a comment

Exercise 4 (Arithmatic Operators)

Arithmetic operators are reserved symbols that are used for performing mathematical operations (calculations).

Examples

operators	symbols	use	return type
Addition	+	1 + 3	int
Subtraction	-	3 - 1	int
Multiplication	*	3 * 2	int
Exponent	**	3 ** 2	int
Float division	/	3 / 2	float
Integer division	//	3 // 2	int
Modulo	%	3 % 2	int

Note

- If one of the operands is a float, then the resulting value is casted (converted) into float. `Eg: 1.0 + 1 = 2.0 and 1 + 1 = 1` .
- `//` , returns the whole number part (quotient) of the division. So, `given: 22.0 // 3 = 7.0 and 22 // 3 = 7` .
- `/` , returns the quotient and the remainder as a float, together. `Eg: 22 /3 = 7.333333333333333 and 0.25 / 0.5 = 0.5` .

Casting

Casting means, converting or changing from one type to another. To know the type of a value, use the `type(obj)` function. `Eg: type(2) and type('2') will return <class 'int'> and <class 'str'>` respectively. Meaning that, 2 is an integer and '2' is a string.

```
# casting
x = 2 # x is an int
y = float(x) # we cast x to a float and passed the value to y
z = str(y) # y is a float, changed to a string and the value assigned to z
```

Note

The values of x and y doesn't change after the casting, except that we do, `y = float(y)` .

Practicals

write a program to evalute and print the results of the following given that `a = 2` and `b = 5` :

1.
    - a * (2 * b) - 5
    - 2 * (b - a) + b
2. (-(a * b) ** 2 - (4 * a * b)) / ((b // a) // 3 + (16 / a / b))
3. ((a ** 2) - (b ** 2)) // ((b - a) ** 2)
4. ((a + b) % 2) - ((b % a) + 1)

Summary

- `+, -, *, **, /, //, %` are reserved for mathematical operations.
- Rule of precedence is `(), **, *, //, /, +, -` .
- Use parentheses to change the precedence.

Exercise 5 a (Analysis of a problem)

In this exercise, we shall look into creation of a solution to a given problem. Let us assume a very basic problem to do with. We will analyse and design an algorithm for the solution.

Sample problem

```
'''
In a test of five students, the marks, 40, 78, 91, 59 and 12 were obtained.
If the test score was over 100. Find the:

    1. sum of the scores obtained by the student
    2. average of the score
    3. number of students who scored above the average score
    4. number of students who scored below the average score

'''
```

Analysis

To analyse the sample problem above, there are basically three things we need to lookout for in the given problem. These are:

1. the input
2. the output
3. the process

The input

This is simply the parameters that are given in the problem, and for the above problem, they are:

1. the total number of students, which is five.
2. the individual scores, which are 40, 78, 91, 59 and 12.
3. the overall score, which is 100.

The output

This is what the program is expected to do - the expected outcome, what the user would see, finally after the process. Usually, we look at the process (calculations before the output). We are expected to compute the values for the:

1. sum of the scores
2. average of the scores
3. number of students who scored above the average score
4. number of students who scored below the average score

Note

Usually the desired output dictates how to compute on the input to obtain the output.

The process

This is also known as the computation. From the problem, we are to compute the sum of the scores and the others. Our focus here is "the how" the computation is or will be done. Most often, there would be a straight forward formular to use, else we have to find it. So now, all we have to think about, is how to compute the :

1. sum of the scores
2. average of the scores
3. number of students who scored above the average score
4. number of students who scored below the average score

Note

The result from the computation is what becomes our output though not all becomes the output.

> continuation in `exercise 5 b (Design of a solution)`

Exercise 5 b (Design of a solution)

> continaution of `exercise 5 a (Analysis of a solution)`

In this phase, we try to put together the information we gathered from the Anaylsis we made. Here we may provide a human readable solution that would easily be used to implement the solution. Again, we decide what data types and data structures that would be used. So here, we could let the average score be an integer or a float and keep all the test scores in an array (here a list). What should come in mind is a pseudocode (a.k.a falsecode). From the analysis we can say:

```
'''
The inputs:
    totalNumberStudent: int = 5
    listOfScore: list = 40, 78, 91, 59 and 12
    overallScore: int = 100

The processes:
    sumOfScores: float = sum of all the elements in listOfScore
    averageOfScores: float = sumOfScores / totalNumberStudent
    numberAboveAverage = number of elements greater than the average
    numberBelowAverage = number of elements less than the average

The output:
    sumOfScores: float
    averageOfScores: float
    numberAboveAverage: int
    numberBelowAverage: int

'''
```

> `label: type` we used is what we discussed in `Exercise 2 ( Data types) - Type hinting`

Note

- Most of these stages are done together, because this is a small problem - which we could even implement straight forward
- Remember, it is absolutely easier to implement a solution after we have analyzed and chose which approach of the solution gave us the desired outcome, efficiently. This means, there could be more than one solution to solve the problem.

Practicals

> Tip: look out for the given inputs (the available parameters), the desired outputs (what is expected), and the process (how to get the output).

Analyse and design a solution for the following problems.

1. Given that a quadratic equation is of the form, `Ax^2 + Bx + C = 0` , where `A` , `B` , `C` are real parameters such that `A` , `B` is not zero. `A` and `B` are the co-efficients of `x` , of second and first degrees respectively and `c` a constant. Find and output the roots of the quadratic equation.
2. Find the Area of a circle of radius, `r` , given that `PI` is `3.143` .
3. A shop keeper sold an item of cost, `$340.00` at `$372.99` including a tax of `$2.99` .find the
    - selling price
    - profit made
    - profit percentage
    - tax percentage

Summary

- To analyse the problem, look out for the input, output and the process
- In the design stage, we choose what data type a value should be or be returned and a data structure, suitable to hold these values

- We choose a solution that best gives the desired outcome
- The best design is simple and readable

Exercise 5 c (Implementation of a solution)

In this stage we write the code based on the design and then go further on to test and debug our code. We are going to change the design to a valid Python code.

```
'''
The inputs:
    totalNumberStudent: int = 5
    listOfScore: list = 40, 78, 91, 59 and 12
    overallScore: int = 100

The processes:
    sumOfScores: float = sum of all the elements in listOfScore
    averageOfScores: float = sumOfScores / totalNumberStudent
    numberAboveAverage = number of elements greater than the average
    numberBelowAverage = number of elements less than the average

The output:
    sumOfScores: float
    averageOfScores: float
    numberAboveAverage: int
    numberBelowAverage: int
'''
```

Code 1

```
 # The inputs
totalNumberStudent = 5

# the scores of the five students
# our listOfScore
s1 = 40
s2 = 78
s3 = 91
s4 = 59
s5 = 12

overallScore = 100

# sum of all the elements in listOfScore
sumOfScores = s1 + s2 + s3 + s4 + s5

# the average
averageOfScores = sumOfScores / totalNumberStudent

# the number of student who had a score greater than the average
# initialize two variables, numberAboveAverage and numberBelowAverage to zero
# compare each student score with the average
# if the score is greater than the average, add one to numberAboveAverage else
# (it may mean it equal to or less than) do nothing here.

numberAboveAverage = 0

if s1 > averageOfScores:
    numberAboveAverage += 1

if s2 > averageOfScores:
    numberAboveAverage += 1

if s3 > averageOfScores:
    numberAboveAverage += 1

if s4 > averageOfScores:
    numberAboveAverage += 1

if s5 > averageOfScores:
    numberAboveAverage += 1

numberBelowAverage = 0

if s1 < averageOfScores:
    numberBelowAverage += 1

if s2 < averageOfScores:
    numberBelowAverage += 1

if s3 < averageOfScores:
    numberBelowAverage += 1

if s4 < averageOfScores:
    numberBelowAverage += 1

if s5 < averageOfScores:
    numberBelowAverage += 1

# the outputs
print("The sum of all the scores is", sumOfScores)
print("The average score is", averageOfScores)
print(numberAboveAverage, "scored above average")
print(numberBelowAverage, "scored below average")
```

Code 2

```
 # The inputs
totalNumberStudent = 5

# the scores of the five students
# here we use a data structure know as a list
# compare this to the other, s1, s2, s3, s4, s5
# which is simpler
listOfScore = [40, 78, 91, 59, 12]

overallScore = 100

# sum of all the elements in listOfScore
# We will use a loop and initialize sumOfScores to zero
sumOfScores = 0

for score in listOfScore:
    sumOfScores += score

# the average
averageOfScores = sumOfScores / totalNumberStudent

# the number of student who had a score greater than the average
# initialize two variables, numberAboveAverage and numberBelowAverage to zero
# compare each student score with the average
# if the score is greater than the average, add one to numberAboveAverage else
# (it may mean it equal to or less than) so we check if it is less than else it
# will be equal, which is of no interest so do nothing.
# we will use a loop here also

numberAboveAverage = 0
numberBelowAverage = 0

for score in listOfScore:
    if score > averageOfScores:
        numberAboveAverage += 1

    if score < averageOfScores:
        numberBelowAverage += 1

# the outputs
print("The sum of all the scores is", sumOfScores)
print("The average score is", averageOfScores)
print(numberAboveAverage, "scored above average")
print(numberBelowAverage, "scored below average")
```

Practicals

Implement the design from the practicals in `exercise 5 b (Design of a solution)`

Summary

- Implementation phase is where we code the design from the analysis
- Implementaion is not the last phase of software developement
- We have to test, maintain and document our programs to actually make them software

Exercise 6 (Input and output)

We shall dicuss in this exercise, Inputs and outputs.

Input

To take input from the user, we use the `input(prompt)` function. The `prompt` is a string - message, "prompting" the user of what is required of or it could be empty string - i.e nothing is passed in the parentheses.

Note

The value from the `input()` is always a string, so we have to *cast* it to the desired type.

> Refer to `Exercise 4 ( Arithmatic Operators) - Casting` , where we discussed in brief, casting.

Example

```
# prompt user for first name
first_name = input("Enter first name: ")

# first name as we require is already a string so no need to cast
print("First Name: ", first_name)
```

Output

The `print()` funtion is used to output information on the screen or write into files. This is the print function in full, `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

Parts of the print function

- `*objects:` these are comma seperated objects we want to display or output. The `*` indicates there is more than one object. `Eg: print('I am', 34, 'years old')` . So we display three objects.
- `sep=' ':` By default, all the objects `'I am', 34 and 'years old'` are seperated by a single space. We could change it to any character we desire by passing `sep=desired_character`
- `end='\n':` By default, the output from the print function adds `\n` - newline - to the objects so the pointer goes to the next line.
- `file=sys.stdout:` We see, always that, our output is displayed (sent) to the screen, it because of `file=sys.stdout` . This is basically a file and as such could be any other file. So if we want to output data into another file, then we change the default value of the `file` to the name of the file object of interest.
- `flush=False:` By default doesn't forcibly flush the buffer to the screen, and its *False*. Change to *True* to do otherwise.

> Read more on Python Flush [Here][flush-resource-site-1] and then [Here][flush-resource-site-2] or [Google it][google-site]
>
> Make references at [Programiz][Programiz-site] and The Pydoc

Example 1

```
# example 1
# all of these strings appear on the different lines because, by default, end='\n'
print("hello")
print("world")

# we make end='' - an empty string
print('John', end='')

# the next object after this print would be on the same line as this
print(' Doe')

# so when we set, `end=' - GVR\n'` , then the object will end with
# the string, ` - GVR` then add a newline.
print('Hello world', end=' - GVR\n')

# print() - without any object would print a newline instead
```

Example 2

Let us write a simple program that takes the name, age and weight of the user then we display the values to the screen

```
 # A simple program that takes the name, age and weight
# of the user then display the values to the screen

current_year = 2020
first_name = input('Enter our first name: ')
last_name = input('Enter our last name: ')
age = int(input('Enter our age: '))

# we can write the above line seperately as below
# age = input('Enter our age: ') # age is a string
# age = int(age) # we convert the age to an integer

weight = float(input('Enter our weight in kg: '))

print("First Name:", first_name)
print("Last Name:", last_name)
print("Age:", age, " - year of Birth:", current_year - age)
# I am subtraction the age, int, from
# current_year, an int, to get the year of birth
# If we didn't cast the age, then we'd get an error
# Try it and see
```

Practicals

1. write a program that finds the sum then the average of five numbers, by asking the user to enter then as floats. Display all the inputs, the sum and then the average. [Use descriptive outputs]
2. Write a simple program to simulate an interview for Python developers. Display the anwsers in an essay form, with heading, `Your responds` .

Summary

- Use `input(prompt)` to take input from the *std.in*, `promt` is the message we pass across to the user, as a guide.
- Use `print(objects)` to display comma seperated objects to the screen.
- We have to cast the value of an input to the desire value since by default, it is a string.

flush-resource-site-1 flush-resource-site-2 google-site Programiz-site pydoc-site

Exercise 7 (String operations and function)

A string is a sequence of at least zero characters enclosed within (delimiter by) an opened and closed single (`''`) or double (`""`) quotation marks. This include any character, even a space or numbers. `Eg: 'swift Python', "Alan Turing"`

String operations

These are operations - manipulations, we can do on strings objects. These are `string concatenation` and `string repetition` . We use the addition, `+` , and multiplication, `*` , operators, repectively, to cancatenate (*join*) and repeat strings.

Example

Concatenation, in this context is add one string to another string.

```
 # String concatetion
first_name = 'Daniel'
last_name  = "Doe"
full_name = first_name + " " + last_name

# we added the first name, space and then last name to
# produce a full name. The " " is an empty string but
# with space between the delimeters
# output => Daniel Doe
# Without the space, we'd have, DanielDoe as the output

# String repetition
# Lets say we want to repeat a particular string for n
# number of times, we basically do, string * n.
# Lets try to repeat "Hello" three time
h = "Hello"
print(h * 3)
# output=> HelloHelloHello
```

String function

These function will help us format the string. There are a lot of this function. The Python docs has more.

Calling the function

A function as we would discuss later is a piece of code that **does** something - performs certain computation. This done this way. `my_string.function()` .

```
########### String functions ###########

name = "John"

# change of a name to lower case
print(name.lower())

# change of a name to upper case
print(name.upper())

# capitalize of a name - the first character becomes upper
# and the rest lower
print(name.capitalize())

# return string as a title format
print(name.title())

# remove (strip) 'h' from the name
print(name.strip('h'))

# find the length or size (number of characters) in name
print(len(name))

# split a string by some char
str_split = "hello world"
str_list = str_split.split(' ')  # ['hello', 'world']
print(str_list)

# join
list_str = ['hello', 'world']
new_str = " ".join(list_str)
print(new_str)  # "hello world"

# There is a whole lot on the Python home page
```

Practical

- Write a program to take input from the user, print the input, seperated by a short-dash, `-` , and the length of the string. `Eg: input='Doe', output=Doe - 3`
- Write a program that reads data from the user and converts the data to lower case, upper case, title and capitalize.

Summary

- String concatenation - add (join) one string onto another. `Eg: name = 'Future' + ' ' + 'Dann'.`
- String repetition - repeat a string for *n-times*. `Eg: print("Green tea" * 2).`
- Do `my_string.function_name()` to call a string function.

#

Exercise 8 (String formatting)

There are a lot of ways to format our output which makes manipulation of our output more readable. We'd go with the `f"... {some_value}"` approach. The `{}` provides the means to insert values into our string as we write it out, `some_value` is the value you pass into the string and the `f` means format. The string must, begin with the `f` .

Example

```
 # Consider a simple program to take the users
# name and age and then output them to the screen

user_name = input("Enter name: ")
user_age = input("Enter age: ")

output = f"User name: {user_name}\nUser age: {user_age}"

print(output)

# Another sample code
# A program to find the area of a triangle
# We'd skip, Analysis and Design

height = float(input("Enter height of triangle: "))
base = float(input("Enter base of triangle: "))
area = 0.5 * base * height

print(f"The area of a triangle of height, {height} and base, {base} has an area of {area}")
```

Practicals

Using the new concept in this exercise and the white space character, newline - `\n` , write a program that accepts from the user, inputs, name, age, sex and hobby. Output a descriptive essay of the values taken.

> Be curious and creative, take as many inputs as possible.

Summary

- One may use `f"... {}"` to format a string
- `f` - means format
- `{}` allows us to insert values directly into the string

Exercise 9 (Logical and relational operators)

We advice skimming through or revisiting `Exercise 3 ( More on data types and comments)` .

How are boolean values generated? Simple, logical operators with the use of Relation Operators generate Boolean Values.

Relational Operators

Relational Operators, a binary operators just like the Arithmatic Operators. These are used for comparison. `Eg: <, >, <=, >=, ==, !=` . Thus are sometimes referred to as Comparison Operators.

Table of relational operators

Operator	Name	Use case	Return value
<	less than	`3 < 2`	`False`
>	greater than	`3 > 2`	`True`
<=	less than or equal to	`3 <= 2`	`False`
>=	greater than or equal to	`3 >= 2`	`True`
==	equal to	`3 == 2`	`False`
!=	not equal to	`3 != 2`	`True`

Example

```
 # let a and b be two non-zero integers of value 5 and 7 repectively
a = 5
b = 7

# Pay much attension to the Truth values generated

# greater than
print(a > b)
print(b > a)

# less than
print(a < b)
print(b < a)

# greater than or equal to
print(a >= b)
print(b >= a)

# greater than or equal to
print(a <= b)
print(b <= a)

# equal to
print(a == b)
# Not the double equal to. Unlike the assignment operator, which is just a character.

# not equal to
print(a != b)
# `!` , means `not`
```

Logical Operators

Logical operators, combine two or more expressions to generate a boolean value. `Eg: and, or, not` . This combines more relational expressions to generate a truth value.

The truth tables

The truth table simplifies what truth values are gebreated when use any of the logical operator.

Assume `t` as `True` and `f` as `False`

AND table

For an `AND` table, the truth value becomes true only when both components are true.

a	b	a and b
t	t	t
t	f	f
f	t	f
f	f	f

OR table

For an `OR` table, the truth value becomes true on when either components are truth, or we could say, the truth value becomes false only when both components are false.

a	b	a or b
t	t	t
t	f	t
f	t	t
f	f	f

Not table

For a `NOT` table, when the value is true it becomes false and when it is false then it becomes true. `NOT` here is the same as negation in some context. It is true and when it is not true then it is false.

a	not a
t	f
f	t

Example

```
# let a and b be two non-zero integers of value 5 and 7 repectively
a = 5
b = 7

# and
print(a <= 10 and b >= 10)

# or
print(a <= 10 or b >= 10)

# not
print(not a <= 10)
print(not b >= 10)

# compound with logical operators
print((a <= 10) and (b >= 10))

print((a <= 10) or (b >= 10))

print((a <= 10) or (b >= 10))
```

Note

- Any value that is None (null), empty, zero, etc is valued to False otherwise True.
- So all empty structures are evaluated to False by Python.

- `not` is uninary, thus `True or not True => True or False => True`
- order of precedence, `not, and, or`

Practicals

- Find the Truth Value of the following:
    - `True and not False`
    - `not True and not False`
    - `True and False and True or False`
    - `True or False and True or False`
- Write a simple program that makes use of all the relational and logical operators.

Summary

- Relational operators are used for comparison
- Locagical operators are use to combine simple relational expressions
- `not` is a uninary operator

Exercise 10 (Conditions)

In the previous exercise, `Exercise 9 (Logical and relational operators)` , we discussed relational and logical operators, and in this exercise we shall make use of Truth values.

This exercise is about decision making, conditions.

One would like to display certain output or take input or even terminate the program based on a certain condition. In Python we have the `if, elif and else` , statements which makes it easier to do some comparison.

If statement

Basically, this is the structure of an `if` statement.

```
if condition:
    # some code
```

This starts with the `if` keyword, followed by a condition to evaluate then a colon, `:` . On a newline, we indent beyound the `if` keyword and add the code to execute when the condition is `True` .

Example

```
# A simple program to check if a driver is driving above
# the speed limit

MAX_SPEED = 120  # this is a constant

# get the drivers speed
drivers_speed = float(input('Enter Vehicle speed: '))

if drivers_speed >= MAX_SPEED:
    print("Please slow down, think about your life and family first.")
```

Note

Nothing happends when the drivers speed is below speed limit

Else statement

Perhaps We want to alert the user to do something when the condition fails or evaluates False, then one must add an `else` statement.

```
if condition:
    # some code
else:
    # do something else
```

Example

```
# A simple program to check is a driver is driving abpve the speed limit

MAX_SPEED = 120
drivers_speed = float(input('Enter Vehicle speed: '))

if drivers_speed >= MAX_SPEED:
    print("Please slow down, think about our life and family first.")
else:
    print("waw, very responsible being.. You are one of a kind")
```

Note

Python uses indentation for structuring or creating block codes. Consider the above example on `if else` statement, all the code in the indentation below the `if` statement is the body or block of the `if` . Those that are outside the ifs indentation forms another block of code. So make proper use of the indentations and use it correctly.

Elif statement

So after the if statement failed, one may want to check for another condition, before the the `else` block, then we use the `elif` which is like, `else if` . Also there are certain instances where by a condition is neither `True` nor `False` but something closer. Consider when we take an integer input from a user, the input can be less than some constant, greater than that constant or even equal to that constant. Thus an `elif` statement becomes useful here.

Example

```
# A program to check the speed limit on the high way
# lets assume that the minimum and maximum speed limit
# is between 40 and 120 km/h.. something like this.

MIN_SPEED = 40
MAX_SPEED = 120

drivers_speed = float(input('Enter speed: '))

# method 1
if drivers_speed < MIN_SPEED:
    print(f"Please drive at least {MIN_SPEED}km/h")
else:
    if drivers_speed > MAX_SPEED:
        print(f"Please drive at most {MAX_SPEED}km/h")
    else:
        print("Rock on man")

# method 2, we use an inbuilt approach with combines `else if`
if drivers_speed < MIN_SPEED:
    print(f"Please drive at least {MIN_SPEED}km/h")
elif drivers_speed > MAX_SPEED:
    print(f"Please drive at most {MAX_SPEED}km/h")
else:
    print("Rock on man")
```

Nested Conditions

We may have a nested `if else` statements as many as We please

Example

```
 # this code has something to do with the above
# but here we check if the driver has parked
# or is over speeding senselessly
# assume reverse for a negative value

PARK_SPEED = 0
MIN_SPEED = 40
MAX_SPEED = 120
OVER_SPEED = 200

drivers_speed = float(input('Enter speed: '))

if drivers_speed < MIN_SPEED:

    # check if the vehicle is parked on the highway
    if drivers_speed == PARK_SPEED:
        print("Please don't park on the high way.. It's deadly man.")
    else:
        print(f"Please drive at least {MIN_SPEED}km/h")

elif drivers_speed > MAX_SPEED:

    # check if the vehicle is really light speed
    if drivers_speed > OVER_SPEED:
        print("Please We are not on a racing track, We are over speeding")
    else:
        print(f"Please drive at most {MAX_SPEED}km/h")

else:

    # check if the vehicle is reversing
    if drivers_speed < PARK_SPEED:
        print("what the heck man, no reversing on the highway")
    else:
        print("Rock on man")

# As We may see, we can have even more nested if and else and
# elifs as much as we can provided they don't make the code hard to read
```

Note

White spaces, such as, spaces, newlines and indentation only makes our code pretty to read but has no effect on the code we write.

Compound statements

Remeber relational and logical operators? We have made use of relational, what about logical?

Example

```
 # this code is the same as above but just serving some different concept
# we know it is bad to park or reverse on the high way,
# kind of we can evaluate these two together
# the maximum speed limit and the overspeeding limit are all the same
# we can either use a relational or a logical, either would do
# for the relational, over speeding limit is greater than the maximum speed
# thus we may check for maximum speed

PARK_SPEED = 0
MIN_SPEED  = 40
MAX_SPEED  = 120
OVER_SPEED = 200

drivers_speed = float(input('Enter speed: '))

# we use `<=` to compound reversing and parked
# we use `or` to compund the previous and minimum speed
if (drivers_speed <= PARK_SPEED) or (drivers_speed < MIN_SPEED):
    print(f"Please drive at least {MIN_SPEED}km/h")
elif drivers_speed > MAX_SPEED:
    print(f"Please drive below {MAX_SPEED}km/h")
else:
    print("Rock on man")
```

Note

```
 # consider, when given
MAX_SPEED  = 120
OVER_SPEED = 200
drivers_speed = 210

# if We would like to catch OVER_SPEED, We check for OVER_SPEED first else it won't be catched
if drivers_speed > OVER_SPEED:
    print("Over speed")
elif drivers_speed > MAX_SPEED:
    print("Max speed")
else:
    print("Rock on")

# unlike this example, OVER_SPEED would never be reached
# because OVER_SPEED is greater than MAX_SPEED
# even when drivers speed is 3000km/h
if drivers_speed > MAX_SPEED:
    print("Max speed")
elif drivers_speed > OVER_SPEED:
    print("Over speed")
else:
    print("Rock on")
```

Practicals

- Write a program that checks if a given integer input is a multiple of 2, 3 or both 2 and 3 and then print what multiple it is with the input.

```
# the code should behave this way

# input = 4
# output = 4 is a multitple of 2

# input = 6
# output = 6 is a multitple of 2 and 3

# input = 9
# output = 9 is a multitple of 3

# input = 18
# output = 18 is a multitple of 2 and 3
```

Summary

- `if else` and `elif` statement are used to create conditional statements.
- use `:` to create a block, followed by a consistent indentation
- the body of the `if` block is reached only if the condition evaluates to `True`
- We may have compound conditions in the `if` and `else` statement

Exercise 11 (Loops)

The need to repeat a certain process for a particular number of time or for as long as a particular condition holds arises. Here we'd look into loops (also know iterations). There are basically two types of loops in Python, the `for` and the `while` loop.

Already, we have seen a list, briefly. A list is a built-in structure for holding data (a collection of these data actually). A list can be looped upon.

Example of a list

```
# a list is a comma seperature collection of objects
# (numbers, strings, other structure, etc), delimited by
# open and close square bracket, `[]`
# which permits looping upon its elements
num_list = [1, 2, 3, 4, 5]
str_list = ['John', 'Mathew', 'zack', 'Doe']
list_obj = ['python', '3', 0.6]
```

For a given list object, such as any of the above or the likes, our interest would be to compute upon the elements of the list or even to check if the list object contains some particular elements.

For loop

Structure of a `for` loop

```
for element in some_structure:
    # do something
```

Structure looping with for

The `element` is an arbituary word we have used to refer to the current object (element) in the `some_structure` . `some_structure` is an object with can be iterated upon. Some example of iterables are `set` , `list` , `dict` , `str` , etc.

Example

```
# add 2 to each element of a list
num_list = [1, 2, 3, 4, 5]
for number in num_list:
    print(f"{number} + 2 = {number + 2}")

# looping through a list of strings and using the built-in function, `len()`
# to find the length of the string element
str_list = ['John', 'Mathew', 'zack', 'Doe']
for str_element in str_list:
    print(f"{str_element} has a length of {len(str_element)}")
```

Range looping

Now `some_structure` can be a built-in function known as `range()` . `range` is an iterating object, thus permits looping. The `range` funtion is of the form , `range(start, end, step)` . By default, it starts at `0` and steps at `1` . We may just pass `end` into the `range` function and it will work fine from `0` to `end` exclusive, `step` 1.

Note

A list is an indexed object, thus we may access the elements in the list using the index of the element in the list. The index always starts from `0` up to `n - 1` , where `n` is the size of the list object.

```
# str_list = ['John', 'Mathew', 'zack', 'Doe']
# their index:   0      1         2       3
# size of str_list, `n` = 4
# index of last element = `n` - 1 = 4 - 1 = 3
```

Example

```
 # loop with `range(start, end, step)`
# add 2 to each element of a list
# using the range function and the index of the element
num_list = [1, 2, 3, 4, 5]

# get the length of num_list
len_num_list = len(num_list)

for index in range(len_num_list):
    element = num_list[index]
    print(f"{element} + 2 = {element})

# find the length of the strings element in the list
# using the range, thus index and the len function
str_list = ['Alan', 'Theresa', 'JJ', 'Fowler']
len_str_list =  4 # len(str_list)

for str_index in range(len_str_list):
    str_el = str_list[str_index]
    print(f"{str_el} has a length of {len(str_el)}")

# looping in range of 0 to 10 (by default 10 would not be
# display but the looping occurs 10 times)
# because the loop is exclusive of `end`
# for i in range(0, 10):
for i in range(10):
    print(i)

# looping in range of 1 to 10
for i in range(1, 10):
    print(i)

# looping in range of 1 to 10, stepping 2
for i in range(1, 10, 2):
    print(i)
# output [1, 3, 5, 7, 9]

# print even numbers less than 20
for i in range(0, 20, 2):
    print(i)

# print odd numbers less than 20
for i in range(1, 20, 2):
    print(i)

# print elements in the reverse order
str_list = ['John', 'Mathew', 'zack', 'Doe']
len_str_list = len(str_list)

# last element, length of list minus one, thus start = len_str_list - 1
# end = -1
# step = -1

for index in range(len_str_list - 1, -1, -1):
    print(f"{str_list[index]}")

# assuming end = 0, implies that we only displace size minus one elements
```

While loop

This loop does something (executes its body) until a condition is met or as far as a condition is still valid.

Structure of a `while` loop

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

Reference `Exercise 10 (Conditions)` here.

Structure looping with while

Usually we find the length of the structure and we'd use it as we did with the range. Consider the example below.

Example

```
# looping through a list of numbers and adding 2 to each element
# and displaying it

num_list = [1, 2, 3, 4, 5]
length_of_list = len(num_list)

base = 0
# base is some arbituary variable
# we use base, as in base case
# our interest is to access the elements in the list using their index
# the first element is 0, second is 1 and so on.
# sure the last item will have an index of (length_of_list - 1)
# the index of the last element is less than the length of the list

while base < length_of_list:
    element = num_list[base]
    print(f"{element} + 2 = {element + 2}")
    base += 1
```

Note

- In the above code, without `base += 1` , the loop becomes an infinite loop. A loop that would never terminate and we'd be print only the first element.
- Also assume we just say, `while True:` , will also not terminate.

Alift off program

```
 # A simple program the simulates a lift off

# version 1
lift_off_time = 5  # 5sec
while lift_off_time >= 0:
    print(lift_off_time)
    lift_off_time -= 1

print("Lift off")

# version 2
lift_off_time = 5  # 5sec
while lift_off_time >= 0:
    if lift_off_time == 0:
        print("Lift off")
    else:
        print(lift_off_time)

    lift_off_time -= 1

# version 3
lift_off_time = 5  # 5sec
while lift_off_time >= 0:
    if not lift_off_time == 0:
        print(lift_off_time)
    else:
        print("Lift off")

    lift_off_time -= 1

# version 4
lift_off_time = 5  # 5sec
cur_time = 0
while cur_time <= lift_off_time:
    print(lift_off_time)
    lift_off_time -= 1

print("Lift off")
```

Practicals

- write a lift off program using a for loop
- Given a list of alphabets, from `a - z` , use a loop to print out the vowels (Tips: `list of vowels, loop, condition`)
- Write a program that simulates the rolling of a dice. A fair dice has 6 sides numbered from 1 to 6. Take an integer input from the user as the number of times the dice must be rolled and this value must be greater than 50. Using a `loop` (any `loop`) and an `if` , `elif` and `else` statement (if necessary) and then `print` the number of times a particular number was obtained when the dice was rolled.

> Use all that we have learnt up to this exercise

Summary

- A loop is used for repetition
- There are two types of loops, `for` and `while` loop
- `for` loop is best used when we know the range
- `while` loop is best when the repetition is based on a condition
- One can loop (iterate) through a structure such as a list or any object with an iterator built-in

Excercise 12 a (Functions)

A function is simply a block of code, with a unique name and maybe, has some arguments, that performs a specific task. The use of functions prevents one from repeating a particular piece of routine/procedure over and over again.

Structure a function

```
def function_name():
    # some code
```

Example

Let say we want to ask five users their name and print it with some string.

```
some_str = "$"

# first person
first_person = input("Enter name: ")
print(first_person + some_str)

# second person
second_person = input("Enter name: ")
print(second_person + some_str)

# third person
third_person = input("Enter name: ")
print(third_person + some_str)

# fourth person
fourth_person = input("Enter name: ")
print(fourth_person + some_str)

# fifth person
fifth_person = input("Enter name: ")
print(fifth_person + some_str)
```

The code above works well, we achieved our goal but we waisted a lot of time rewriting all these for five times. A simple solution would be to use a loop.

Example

```
# reading and outputting 5 users name using a loop

for i in range(5):
    person_name = input("Enter name: ")
    print(person_name  + some_str)
```

Well this code also works well. The problem is that, what if we don't want it `n` times any more but we want it when we need it?

Now another approach is the mudular (functioanl approach).

```
 # create a function to reading and outputting 5 users name
# by calling the function 5 times

def get_and_print_name():
    person_name = input("Enter name: ")
    print(person_name  + some_str)

# to make this function work, we have to call it
# just like we do to variables
# but here we add `()` to it.

get_and_print_name()
get_and_print_name()
get_and_print_name()
get_and_print_name()
get_and_print_name()

# well isn't there some repetition here? Yes there is.
# or we can use a loop
for i in range(5):
    get_and_print_name()
```

Advantage of using a function

- Reduce code redundancy
- It is easier to catch and fix bug in our code
- It can be plugged into another code

Example

```
 # A program that calculates and prints the area of a triangle taking the
# base and height as inputs

def calc_area():
    base = float(input("Enter base: "))
    height = float(input("Enter height: "))

    area = 0.5 * base * height

    print(f"The area of a triagle of base, {base} and height, {height} is {area}")

# call the function here
calc_area()
```

Excercise 12 b (Functions)

> This is a continuation of `excercise 12 a (Functions)`

In `Excercise 12 a (Functions)` , we created the function

```
def get_and_print_name():
    person_name = input("Enter name: ")
    print(person_name  + some_str)
```

We can make it better by passing some values to the function.

A function with argument

An argument is basically a value (some times a reference of the value) we pass to a function so that the function may make use it to reach an end.

```
def get_and_print_name(name, some_str):
    print(f"{name} {some_str}")

get_and_print_name('John Doe', "$_$")
```

A better version of the `calc_area` code from `Excercise 12 a (Functions)` would be that we are able to pass argument to it just as we did above. It will be better when we can dictate what the base or height can be. With this we can modify the functionality of the function to return a particular area based on the arguments passed.

Example

```
# A program that calculates and prints the area of a
# triangle passing the base and height as arguments

def calc_area(base, height):
    area = 0.5 * base * height

    print(f"The area of a triagle of base, {base} and height, {height} is {area}m^2")

# take the base and height from the user
base   = float(input("Enter base: "))
height = float(input("Enter height: "))

# call the function here
calc_area(base, height)
```

Note

There is what we call argument and also a parameter. They are almost the same basically. So an argument is a parameter. Use put the paramter into the function when creating the function and the argument is what we pass to the function when we are calling it.

```
def calc_area(base, height):
    # code

calc_area(3, 4)

# base and height in def calc_area(base, height) are paramters
# 3 and 4 are arguments, also we can assign 3 and 4 to a variable and pass the variables as argument instead.
```

Function that returns a value

Sometimes, one may want a value from a function to make use of it in one way or another. To achieve this we return the value rather printing it out in the function.

```
# this function returns the area of a triangle, taking the base and height as arguments

def calc_area(base, height):
    return 0.5 * base * height

# take the base and height from the user
user_base   = float(input("Enter base: "))
user_height = float(input("Enter height: "))

# call the function here
area = calc_area(user_base, user_height)

print(f"The area of a triagle of base, {user_base} and height, {user_height} is {area}")
```

Example - a sorting function

```
# this is a function that take list, an iterable as an argument and them sorts it
# how this sorting function works
# given a list of size, `n`
# having 2 loops, nested actually,
# the first loop, loops through the function `n` times
# the second does, `n - 1` time
# in the second loop we check if the the first value is greater than the second value
# if it is we `swap` the values and then we compare the second with the third and then
# the third and fourth and so on until we reacch the end of the list
# then the first loop moves ( steps 1) then the sequence begins again

def sort_func(my_ite):
    """ This function takes a list as an argument and returns a sorted version of it """
    length_of_ite = len(my_ite)

    for i in range(length_of_ite):
        for j in range(1, length_of_ite):
            if my_ite[j - 1] > my_ite[j]:
                my_ite[j - 1], my_ite[j] = my_ite[j], my_ite[j - 1]

    return my_ite

print(sort_func([6, 3, 2, 4, 1]))
print(sort_func(['w', 't', 'a', 'i']))
```

Excercise 12 c (Functions)

> This is a continuation of `excercise 12 b (Functions)`

Function with many arguments

Some time we would like to pass plenty argument into a function and thus one is forced to give the function numerous parameters on creation but there is a very simple approach in Python.

Example

```
# A function the takes a number of strings as argument and returns their length

def many_args(s1, s2, s3, s4, s5):
    print(len(s1))
    print(len(s2))
    print(len(s3))
    print(len(s4))
    print(len(s5))

many_args('sandy', 'jude', 'mani', 'desmond', 'peter')

# what the heck, what if there were about 1000's of args?

# better version is to use the tuple argument, *arg_name - the takeaway is `*`
# all the argument passed is seen as a tuple object thus iteration is feasible

def many_args(*s):
    for i in s:
        print(len(i))

many_args('sandy', 'jude', 'mani', 'desmond', 'peter', 'sandy', 'jude', 'mani', 'desmond', 'peter', 'sandy', 'jude', 'mani', 'desmon
```

Note

- You can do `some_name = func_name` then do, `some_name()` .this will work just like `func_name()`

Practicals

- Given a list, whose elements are also list (talking about nested list), write a function that sorts this list and it list elements if possible

Summary

- A function is simply a block of code than can be called and arguments be passed to it
- function definition

```
def function_name(some_args):
    # some code
```

- we can call the function by doing `func_name(some_args)`
- A function allows resue of code
- A function can be used in any part of our code
- paramter are passed into the function when creating the function
- argument is what we pass to the function when we are calling it
- `return` exits a function and returns a value from the function
- use the *arg - tuple argument to collect more arguments
- A function may be called as many times as possible

Exercise 13 (Some built-in functions)

> We really recommend you check out the python doc, see the library reference and click on the built-in functions.

In the previous exercise, `Exercise 12 (Functions)` , we looked into function and we created a couple of our own. Python comes packaged with some function and these functions are known as the built-in functions.

Some built-in functions

1. abs(x)
2. divmod(a, b)
3. float(x)
4. int(x)
5. input(prompt)
6. len(x)
7. list(x)
8. max(x)
9. min(x)
10. pow(a, b)
11. print(a)
12. range(start, end, step)
13. reversed(x)
14. round(x)
15. sorted(x)
16. str(x)
17. sum(x)
18. type(x)
19. chr(i) and ord(s)

Examples

```
 # abs - returns an absolute value of a number
print(abs(2.34), abs(-23.4))

# round - take a number x, rounds it to y decimal places
print(round(23.23567, 2))

# divmod - Take two args and return their quotient and remainder
print(divmod(23, 6))

# pow - take 2 args and returns x raised to the power y
print(pow(2, 3))

# max, min, sum, sorted, reversed
# the above functions are used on iterables
my_list = [7, 2, 4, 5, 1]

print(f"The largest number is: {max(my_list)}")
print(f"The smallest number is: {min(my_list)}")

print(f"The sum of the numbers is: {sum(my_list)}")

print(f"sorted list: {sorted(my_list)}")

print(f"reversed list: {reversed(my_list)}")

# these function does not alter the object

# chr and ord
# chr - returns a character when a number is passed as arg
print(chr(65))  # this are unicode related

# ord does the opposite of chr
print(ord('A'))

# float, int, str, list, dict, set
# these converts objects to their types
# type returns the ( data) type of an object
# len, range
# we have seen these two before, for the size and also looping

# input, print
# we have also seen them before
```

Practicals

- Implement a function known as `all(iterable)` . This function returns `True` if all of the elements of the `iterable` is `True` or the `iterable` is empty, else `False` .
- Implement a function known as `any(iterable)` . This function returns `True` if any of the elements of the `iterable` is `True` . If the iterable is empty, return False.
- Implement the `abs(x)` function where `x` , is a number.
- Implement a function that returns the minimum and maximum numbers in given list. Don't use the built-in function `min` and `max` .

Summary

- Built-in functions make working in python much more easier and flexible without you having to implement your own version of any of those function.
- Built-in functions does not change the object.

#

Exercise 14 (List)

We have had our fill with lists but here we kind of go much into it.

As already know, a `list` is a collection of comma seperated objects, where by the collection is delimitered by an opened and closed square bracket.

Examples

```
num_list = [1, 2, 3, 4, 5]

str_list = ['1', '2', 'Jonas', 'maiduguri', 'samoa']

bool_list = [True, False, False, True]
```

kinds of list

There are basically two kinds of list in Python, the one dimensional list - a single list and then the multi-dimensional list or nested list - these are 2D, 3D, etc

One Dimensional list

```
# sample of 1D list
num_list = [1, 2, 3, 4, 5]
```

Multi-dimensional list

```
# samples of XD list - form some int x >= 1

# 2d list
list_list = [
    [1, 2, 3, 4, 5],
    ['1', '2', 'Jonas', 'maiduguri', 'samoa']
]

# 3d list
list_list = [
    [1, 2, 3, 4, 5],
    ['1', '2', 'Jonas', 'maiduguri', 'samoa'],
    [True, False, False, True]
]
```

Accessing elements of a list

The elements of a list are indexed from `0 to (n - 1)` , where `n` is the size of the list. To access an element from the list, use the elements index.

```
# index =  0    1      2        3
my_list = [1, 'gnu', 'swift', 'kickass']

print(my_list[0])  # 1
print(my_list[1])  # gnu
print(my_list[2])  # swift
print(my_list[3])  # kickass

# using negative indices 0 as if moving backwards from `0`
print(my_list[-1])  # kickass
print(my_list[-2])  # swift
print(my_list[-4])  # 1
```

For a multi-dimensional list of say `x` , we provide `x` indices instead.

```
 # this is a 2D list
my_list = [
    [1, 2, 3, 4],
    ['go', 'py', 'js', 'kt']
]

# we use 2 indices
# the first goes into the main list
# the second the branch list

print(my_list[0])  # -> [1, 2, 3, 4]
print(my_list[1])  # -> ['go', 'py', 'js', 'kt']

print(my_list[0][0])  # -> 1
print(my_list[1][2])  # -> 'js'
```

Note

Try thinking in rows and columns with multi-dimensional lists

```


my_list = [
    [1, 2, 3, 4],
    ['go', 'py', 'js', 'kt']
]

# `4` is in row `1` column `4` .
# knowing well that index starts from `0` , we substract `1` from the rows and cols
# thus `4` can be indexed with `[1-1][4-1]=[0][3]`
```

List slicing

List slicing allows us to sublist the list object - slice the list from one `index` to another. Think about slicing of an actual bread.

Just like indexing, but here we provide a range, a `start` and an `end` with a colon, `:` , Something similiar to, `list_obj[start:end]` . List slicing works like the `range(start, end)` function, the `end` is exclusive.

```
 my_list = [1, 'gnu', 'swift', 'kickass']

print(my_list[1:3]) # [gnu, swift]
print(my_list[0:3]) # [1, gnu, swift]
print(my_list[:3]) # [1, gnu, swift]
print(my_list[1:]) # [gnu, swift, kickass]
```

List Operations

The addition operator, `+` operator is used to add (concatenate) lists.

```
 # Adding one list to another

my_list = [1, 2, 3]
s_list = ['e', 'u', 'o']

f_list = my_list + s_list

print(f_list)
```

The multiplication operator, `*` on a list object multiplies the list `n` time, for some integer value `n` > 1.

```
b_list = [2]
f_list = b_list * 4

print(f_list)
```

The `in` operator, checks if an object exists in a list (or an iterable object).

```
my_list = [1, 2, 3]

# check if the list contains 4
# or 4 is in the list

if 4 in my_list:
    print('The list has a four')
else:
    print('well, there is no four')
```

Some list functions

Function	Description
append(obj)	adds object to the end of the list object
extends(obj)	adds object to the end of the list object as a whole
index(obj)	returns the index of obj in the list object
insert(index, obj)	insert obj at index of list object
pop(index)	removes the element at index or the last element when no index is passed
remove(obj)	removes the first occurence of obj in the list object
reverse()	reverses the list in place (it changes the object)
count(obj)	counts the number of obj in list object
sort(reverse=True)	by default sorts list object in numerical order and can also reverse the sorted object by passing the keyword revere=True
clear()	removes all the elements of the list
del list_obj[index]	deletes object at index

```
name_list = ['john']

# append
name_list.append('Doe')

# extends
ext_list = ['dev', 'ubuntu']
name_list.extends(ext_list)

# index
third_el = name_list[2]

# insert
name_list.insert(0, third_el)

# pop
name_list.pop()  # removes last object
name_list.pop(len(name_list) - 1)

# remove
name_list.remove('Doe')

# reverse
name_list.reverse()

# count
name_list.count('Doe`)

# sort
name_list.sort()

# sort reverse
name_list.sort(reverse = True)

# clear
name_list.clear()

# or just set name_list to an empty list
name_list = []

# del
# an alternative to remove
del name_list[1]
```

Note

`del name_list` would delete `name_list` from memory.

Practicals

> this is supposed to be fun

- Create a function for each of the following, using any means possible without cutting corners.(no using of built-in function - we have go to try harder)
    1. `addition` - this function takes two objects as argument, and returns their sum if they are numbers, that is a float or an int.
    2. `subtraction` - this function takes two objects as argument, returns the resut of subtracting the second from the first.
    3. `division` - this function takes two objects as argument, returns the result from dividing the first by the second. Remember that zero division is not allowed thus check if the second is zero.
    4. `multplication` - this function takes two objects as argument, returns the product of the two.
- Write a function, that takes a list of various objects as an argument, return a list of all the objects that are numbers (that is integer and float).
- Write a function taking a list of various objects as argument, return the number of each object in the list.
- Write a function that takes a list of integers as an argument, remove (delete) any element that has the same parity as its `index + 1` .(If the `index + 1` is even and the element is even, remove the element. If number is odd and `index + 1` is odd, remove element, except when the `index` is 0), looping `n -1` times, where `n` is the size of the list.

```
s = [2, 6, 18, 11, 4]
# 6 is removed - loop 1

s = [2, 18, 11, 4]
# 18 is removed - loop 2

s = [2, 11, 4]
# None is removed - loop 3

s = [2, 11, 4]
# None is removed - loop 4

s = [2, 11]
```

- This practical is the same as the above but in the above as we remove the elements which passes the condition, the list is changed too. We don't want to remove the element as the list chnages - that in a way does really suffices. Consider the code below:

```
# i = index + 1
s = [2, 6, 18, 11, 4]
i = [1, 2,  3,  4, 5]

# only 6 will be removed because it is in parity with its index + 1
s = [2, 18, 11, 4]
```

Summary

- A `list` is a collection of comma seperated objects
- A literal list is created, `name_of_list_object = [a, b, c, ... ]`
- There can be a nested list
- Pass the index of the element of interest into a square bracket after the name of the list object. `Eg: list_obj[1]` returns the element at that index
- We can use `+` operator to concatenate one list to another
- We can use `*` operator to repeat the list n time. `Eg: list_obj * 3`
- We can check if an object is in a list by using the `in` keyword. `Eg: obj in list_object` , this returs a boolean value
- Use the dot operator to call a list function (aka method). `Eg: list_object_name.function_name(some args)`

Exercise 15 (Tuple)

A tuple is a comma seperated values and by conversion delimited by an open and closed brackets - parentheses, `()` . A tuple is just like a `list` but a tuple is immutable - can not be altered after creation unlike a list.

Structure of a Tuple

```
sample_tuple = 1, 2, 3

# a tuple with brackets
tuple_with_bracket = (1, 2, 3)

# thus an empty tuple
empty_tuple = ()  # an empty list, []

# empty tuple object
tup_object = tuple()

# to verify this try type(sample_tuple) and type(tuple_with_bracket)
# we should see some with tuple
```

Note

A single element tuple can be created by simply ending the statement with a comma

```
# this is also a tuple
single_element_tuple = 1,

# or
single_element_tuple = (1,)
print(type(single_element_tuple))

# but this is not a tuple
not_single_element_tuple = 1

# or
not_single_element_tuple = (1)
print(type(not_single_element_tuple))
```

Pros and Cons of a Tuple

Most of the thing we wish to do to a list, we may do to a tuple. Some of the things we can't do to a tuple is to mutate it - change it's content after initialization.

```
my_tuple = (1, 2, 3)

# indexing
first_element = my_tuple[0]

# reassigning
# TypeError: 'tuple' object does not support item assignment
my_tuple[0] = 4

# len, max, min
tuple_size = len(my_tuple)

# sequence unpacking
# this is another way to unpack the tuple
# this is also feasible for a list
first_el, second_el, third_el = my_tuple

# this is just like a multiple assignment
first_el, second_el, third_el = 1, 2, 3

# nested tuple
nest_tuple = (my_tuple, ('john', 'mic', 'Dorris'), 'New zealand')

# can not append nor extend
# AttributeError
sample_tup = 1,
sample_tup.append(2)
sample_tup.extend((2,3))

# but we can contatenate with +=
sample_tup += 2, 3
print(sample_tup)  # (1, 2, 3)
# What happended was that we concatenated 1, and 2, 3 and
# assigned it to sample_tup
```

Casting

we may cast - convert any iterable - a sequential object such as a `list` and `string` to a tuple but not an integer because we can not loop over an integer. This can be done using `tuple(sequence)` .

```
# casting a list to a tuple
my_list = [1, 2, 3]
my_tuple = tuple(my_list)

my_str = 'Hello world'
tuple_str = tuple(my_str)
print(tuple_str)
# output-> ('H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd')
```

Practicals

Try to implement the practicals in `Exercise 14 ( List)` .

Summary

- A tuple is an immutable list, delimetered by parentheses
- Sample tuple, `my_tuple = (1, 2, 3)`
- Tuple doe not have the `append` and `extend` method
- Make a sequence a tuple by casting it. `tuple('I am a string')`

Exercise 16 (Set)

A set is a mutable sequence with no duplicates. Just like a list, but no duplicate. A set is delimited by open and close curly brackets.

Structure of a set

```
# empty list
empty_list = set()  # not {}, this is a `dict`

# set structure
sample_set = {1, 2, 3}
```

Casting

We may cast any sequence to a set by passing the sequence as an argument to `set()` . This returns a unique elements of the sequence.

```
# str to set
name = "Gangliona Messian"
set_str = set(name)

print(set_str)
# output-> {'e', 'o', 'i', 'l', 'M', 'g', 'n', ' ', 's', 'a', 'G'}

my_list = [1, 2, 3]
my_set = set(my_list)

print(my_set)
# output-> {1, 2, 3}
```

Set operations and functions

Assume we have two sets, A anf B.

Operator	Function	description
`&`	intersection	returns elements in both sets
`|`	union	returns elements in either set
`-`	difference	returns elements in set A that are not in B

Operator	Function	description
`^`	symmetric_difference	returns elements that are not in both sets
`>=`	issuperset	returns True if A is a super set of B, else False
`<=`	issubset	returns True if A is a subset set of B, else False
	disjoint	returns is both sets have nothing in common
	add	adds an element to the set, just like append in list
	update	adds a sequence to the set, just like extend in list
	discard	removes an element from the list
	remove	just like discard but returns an error when the element doesn't exist

Examples

```
 # Let our two sets be
set_a = {1, 2, 3, 6}
set_b = {3, 4, 5, 6}

# intersection -> {3, 6}
fintersec_ab = set_a.intersection(set_b)
ointersec_ab = set_a & set_b

print('intesection: ', fintersec_ab == ointersec_ab)

# union -> {1, 2, 3, 4, 5, 6}
funion_ab = set_a.union(set_b)
ounion_ab = set_a | set_b

print('union: ', funion_ab == ounion_ab)

# difference
fdiff_ab = a.difference(b)  # output-> {1, 2}
odiff_ab = a - b  # output-> {1, 2}

fdiff_ba = b.difference(a)  # output-> {4, 5}
odiff_ba = b - a  # output-> {4, 5}

# symmetric difference
symm_diff = a.symmetric_difference(b)
print(symm_diff)  # output-> {1, 2, 4, 5}

# upperset and subset
is_a_super_set = a.issuperset(b) == (a >= b)
print(is_a_super_set)  # False

is_a_subset_set = a.issubset(b) == (a <= b)
print(is_a_subset_set)  # False

# disjoint
a.disjoint(b)  # output-> False

# add
set_a.add(8)
print(set_a)  #output-> {1, 2, 3, 6, 8}

# update
update_ab = set_a.update(set_b)  # output-> {1, 2, 3, 4, 5, 6, 8}

# discard
set_a.discard(1)  # output-> {2, 3, 4, 5, 6, 8}
set_a.discard(10)  # output-> {2, 3, 4, 5, 6, 8}

# remove
set_a.remove(2)  # output-> {3, 4, 5, 6, 8}
set_a.remove(2)  # error-> KeyError
```

Practicals

Implement your own version of

```
 - Intersection
- Union
- difference
- symmetric_difference
- delete like discard or remove ( ignore the KeyError thing)
- update
- isset - it check is a sequence has no duplicate ( remember `count` )
```

> make use of all the we've learnt.

Summary

- A set is just more like a list but with no duplicates
- Sample set, `my_set = {1,3,4}`
- cast a sequence to a set, `set(sequence)`

Exercise 17 (Dictionary)

A dictionary just like a list, tuple and a set which are sequential and number indexed, is rather key-value paired. We reference with keys of the values.

Structure of a dictionary

```
# dict_var = { key : value }

# empty dictionary
my_dict = {}

# profile dictionary
profile = {
    'name': 'John Doe',
    'age' : 32,
    'job' : 'Software engineer'
}
```

Casting

Casting is done with `dict()`

```
# passing a key word and a value

# dict(key-word=value, ... )
my_dict = dict(
    name = 'John Doe',
    age =  32,
    job = 'Software engineer'
)

# convert a list of tuple to a dictionary
my_tupled_list = [('name', 'John Doe'), ('age', 32, ), ('job', 'Software engineer')]

my_dict = dict(my_tupled_list)

print(my_tupled_list)
# output-> {'name': 'John Doe', 'age': 32, 'job': 'Software engineer'}
```

Indexing and updating a dictionary

Indexing and updating is done just as we would do to a list.

```
 # consider this dictionary
profile = {
    'name': 'John Doe',
    'age' : 32,
    'job' : 'Software engineer'
}

# get the name and job
name = profile['name']
job = profile['job']

print(f"Candidates name is {name} and works, {job}")

# update the age
profile['age'] = 30

# add a new key
profile['lang'] = 'Python'

# delete a key to delete a value usind - del
del profile['age']
```

dictionary functions

Functions	description
clear	deletes all the items in the dictionary, similar to reassigning it to `dict()`
copy	returns a copy of the dictionary
get(`key` , `default value`)	returns a value of that key just like dict_object[key] but returns default value when key doesn't exist. This does not update the dictionary
items	returns the items in the dictionary
values	returns the values of the dictionary
keys	retuns the keys of the dictionary
pop(`key`)	deletes item with key just like del
popitem	deletes the last item in the dictionary
setdefault(`key` , `value`)	adds key value to dictionary if key doesn't exist, unlike `get`

Examples

```
 # empt dict
profile = dict()

# add an item - 3 ways use any
profile['name'] = 'John Doe'  # we use this more - simplier

profile.update(age=32)

profile.setdefault('job', 'Software engineer')

print(profile)

# get item from the dict

# get the keys from a dictionary
profile_keys = profile.keys()
print(profile_keys)
# output-> dict_keys(['name', 'age', 'job'])

# get the values from a dictionary
profile_values = profile.values()
print(profile_values)
# output-> dict_values(['John Doe', 32, 'Software engineer'])

# get the key and value as items
profile_content = profile.items()
print(profile_content)
# do the printing

# get element by key
username = profile['name']
print(f"user name: {username}")

# what if the key doesn't exist
# use get with default value
# profile['height'] -> KeyError
height = profile.get('height', 130)
print(height)  # -> 130

# but height won't be added to the dict
# use set default, update or dict[key] = value
if not 'height' in profile.keys():
    # any of this would work
    profile['height'] = 120
    # profile.update(height = 120)
    # profile.setdefault('height', 120)
else:
    print('Profile updated, height added')

print(profile)

# copy
new_profile = profile.copy()
print(new_profile)

# pop - remove height
profile.pop('height')  # or
# del profile['height']

# delete all items in the dict
profile.popitems()

print(len(new_profile) == len(profile))
```

Looping through a dictionary

Looping is the same every where in a list, set and tuple, in even a string, but for a dictionary we may loop using a key or and value.

```
 # consider this sample dictionary
profile = {
    'name': 'John Doe',
    'age' : 32,
    'job' : 'Software engineer'
}

# looping through keys
for key in profile.keys():
    print(f"key: {key}")

# looping through values
for value in profile.values():
    print(f"value: {value}")

# looping through the items in the dictionary
# we loop through the key and value at the same time
for key, value in profile.items():
    print(f"{key} has a value of {value}")
```

Practicals

- Write a program that creates a dynamic user profile. Make use of the `input()`
- Write a function that removes items with duplicate values
- Write a function that reverses a dictionary
- Write a function that sorts a dictionary by Key then value (2 functions)
- Write a function that takes a dictionary as an argument, return another dictionary that has the frequencey of the length of the value, if value is int or float, frequencey is number of digits. Keep the keys of the old as the new.

Summary

- is a `key-value` pair sequence
- is of structure, `my_dict = {key:value}`
- `dict(name='name')` casts to a dict
- dict_obj[`key`] returns the `value` at `key`
- `dict_obj[key] = value` to create new item in dictionary or update
- loop through dictionary by keys, values and items

Exercise 18 (Execptions)

An exception is an error generated when the code is executed. Sometimes, this is also know as runtime error because we only get such error when the code is running.

Some types of exceptions

There are a lot of Exceptions and We'd just list some of the mostly seen ones.

- ZeroDivisionError
- AttributeError
- EOFError
- ImportError
- IndexError
- KeyError
- NameError
- RuntimeError
- ValueError
- TypeError

Handling exceptions

It is import to handle exception in our code as it will prevent the abrupt halting of the software. We can handle these errors with a `try` and `except` clauses.

Try and catch structure

```
try:
    # code to check
except (exceptions to handle):
    # message
```

Example 1

```
# catch ZeroDivisionError
# error generated when dividing by zero
# we shall perform some simple division
#  where by we take 2 int inputs from the user
try:
    numerator = int(input('Enter the numerator: '))
    denominator = int(input('Enter the denominator: '))

    result = numerator / denominator

    print(f"The result is {result}")
except ZeroDivisionError as z:
    print(f'error: {z}')

# with this approach, our program would not crash badly.

# we can also catch this in another hack
numerator = int(input('Enter the numerator: '))
denominator = int(input('Enter the denominator: '))

if denominator == 0:
    print(f'ZeroDivisionError: can not divide by zero')
else:
    result = numerator / denominator
    print(f"The result is {result}")
```

Example 2

Lets try to catch any kind of execption using the `Exception` class.

```
# Lets catch an Exception without being specific

try:
    numerator = input('Enter the numerator: ')
    denominator = int(input('Enter the denominator: '))

    result = numerator / denominator + rate
    # note that the name, rate, is not defined

    print(f"The result is {result}")
except Exception as z:
    print(f'error: {z}')
```

Example 3

Lets catch multiple exceptions - as a tuple

```
try:
    numerator = int(input('Enter the numerator: '))
    denominator = int(input('Enter the denominator: '))

    result = numerator / denominator

    print(f"The result is {result}")
except (ZeroDivisionError, NameError, ValueError) as e:
    print(f'error: {e}')
```

Raising exceptions

We can raise our own exception using the `raise` keyword.

```
# let raise our own exception
# so we take an input from the user and we expect an even number else,
#  we raise the exception

try:
    user_input = int(input('Enter an even number: '))
    if user_input % 2 != 0:
        raise ValueError('Even number expected.')
        # if we don't know what kind of exception to raise
        # just raise Exception
    else:
        print(f"Cool, you entered, {user_input}")
except Exception as e:
    print(f"error: {e}")
```

Practicals

Fix this code base on the error message generated:

```
try:
    numerator = input('Enter the numerator: ')
    denominator = int(input('Enter the denominator: '))

    result = numerator / denominator + rate

    print(f"The result is {result}")
except Exception as z:
    print(f'error: {z}')
```

Summary

- Exceptions are error we get at runtime
- `Execption` class catches all exceptions in general
- Use `try` and `catch` to handle exceptions
- Raise exception using, `raise Exception_type(message)`

Exercise 19 (Files)

In this exercise we shall create, read and write into a file. We have actually been doing this with the `print()` function indirectly - recall `sys.stdout` .

File modes

A file mode is the privilege which we give to the file object when opening/accessing a file. This modes provides a read only, write only, append or read and wite privileges when handling files.

Mode	Parameter	Description

Mode	Parameter	Description
Write	`w` , `w+`	`w` - write mode. Write into file but create file when file does not exist or overwrite the content when file already exist. `w+` allows for write and read privileges.
Read	`r` , `r+`	`r` - read mode. Opens file for reading only. `r+` - allows reading and writing privileges.
Append	`a` , `a+`	`a` - append mode. Wite data to the end file. `a+` - create file if file does not exist.

Create file

Use the open function to create a file. `open(file_name, mode)`

```
# creating a file with name helloworld.py
open('helloworld.py', 'w+')

# helloworld.py file will be created in the current working file directory
# if helloworld.py will be overwritten if it already exist
```

Closing file

After opening, reading and writing to a file, we have to close it.

```
# close the file after opening to release resources
file_obj.close()
```

Open file

After creating the file, a file object is returned.

```
# creating a file with name helloworld.py
file_obj = open('helloworld.py', 'w+')
```

Alternative to open file

Use the `with ... as` clause to `open` and `close` the file. This is mostly peferred.

```
# creating a file with name helloworld.py
with open('helloworld.py', 'w+') as file_obj:
    print('file ceated successfully.')
```

Read from file

Read the content of the file using the file object. Create a sample file, `sample.py` and add the line `# hello world Python` to it. Save the file and close it. Use `read` , `readline` and `readlines` of the file object to read the content of the file.

File method	Function
`read(size)`	reads entire file content or just some part by passing an `int` argument as size.
`readline()`	reads file content line by line and returns one at a go. It moves the pointer to the next line afterwards.
`readlines()`	reads the entire file content, line by line as a list.

`read(size)`

```
 # read file content of sample.py
# sample.py is the file created from above
# else, create a file with name, sample.py
# add the line, # hello world Python

with open('sample.py', 'r') as file_obj:
    content = file_obj.read()
    print("File content")
    print("-------------")
    print(content)

# or go with the open and close
file_object = open('sample.py', 'r')
content = file_obj.read()
print("File content")
print("-------------")
print(content)
file_obj.close()
```

readline

```
 # read file content of sample.py
# a line at a time

print("File content")
print("-------------")

with open('sample.py', 'r') as file_obj:
    line = file_obj.readline()
    print(line)

# this returns a file pointer object
# this pointer is now on the second line after we called readline()
# when we call readline() again, it returns the second line
# then moves the pointer to the next line - the 3rd line
# and so on.. add print(file_obj.readline()) and see for yourself
```

readlines

```
 # read file content of sample.py
# line by line as a list object

# lets make this more fun
# prompt the user for the number of lines there is in the file
# prompt the user which line should be read
# then print the line

file_content = []

with open('sample.py', 'r') as file_object:
    file_content = file_object.readlines()

user_input = int(input("Enter line number: "))

line = file_content[user_input - 1]

print(user_input, line)
```

Write to file

When we print, we write to the screen but in this example we shall write to a file.

write method

Use the `write` method to write data into file.

```
# write into file with the write method
with open('samplewrite.txt', 'w+') as write_obj:
    write_obj.write("Hello world")

# add another line to the previous content
# using the append mode
with open('samplewrite.txt', 'a+') as write_obj:
    write_obj.write("I am a Python developer")
```

print function

Use the `print` function to write data into file.

```
# write into file with the print function
with open('samplewrite.txt', 'w+') as write_obj:
    content = "I remember foo and bar from my little years"
    print(content, file=write_obj)
```

Example

A program that counts the number of characters on the first line of file.

```
# a program that counts the number of characters on the first line of file.
with open('testsample.py', 'r') as read_obj:
    first_line = read_obj.readline()
    number_chars = "".join(first_line.split(" "))

    print('ws:', len(first_line))
    print('number_chars:', len(number_chars))
```

Practical

- Write a function that returns the number of lines and characters on each line in the entire file.
- Write a function that returns the document statistics of a give file. The document statistics are number of lines, number of words, number of characters with space and witout space.

  ```sample output

  ### file name

  Lines - 8 Words - 71 Char (ws) - 403 Char (wos) - 337 ```
- Write a program that allows creating, reading and updating of the content of a file.

## Summary

- open file in the `read` , `write` or `append` mode
- use `open(file_name, mode)` to open or create a file
- use `read` , `readline` or `readlines` to read the content of the file
- use `write` or `print(content, file=your_file_object)` to write into files

# Exercise 20 a (OOP)

OOP stands for Object Oriented Programming. So, with this, style of doing things ( coding), instead of using just variables or and functions, use them bundled together.

Everything in Python is an object. The integers, floats, strings, etc, are all object. Thus they may have some qualities ( properties/attributes) and some functionalities ( methods). It is that OOP, is used to mimic real life objects.

## Class
  ```

A class is like a blue-print of an object which defines the properties and functionalities of the object. For a real life example, lets think of a human being as an example of a class. A human being, surely has a name, age, address, maybe married thus marital status and so on. These are the properties (attributes) of the human being. They describe the 'object'. Again, a human is capable of doing certain things like talking, sleeping, drinking, jumping, thinking, etc. Humans have functionalities. An object does something.

> It is a conversion that, our *file name* for the class, matches the *class name* for our class object.

Structure of a class

Minimally, just like we define a function, we start with the `def` keyword followed by the `function_name` then a tuple of arguments or just empty parenthesis. Python has the `class` keyword for defining a class.

> Remember, that indentation matters in Python.

```
# a minimal class that does nothing
# with class_name, Human
# it is a conversion that one begins a class name with an
# uppercase letter
class Human:
    pass
```

> `pass` is a keyword, we use it when maybe we are not ready to implement a function or class yet

Constructor

When a class is created, one may like to initialize some data thus pass them to the constructor. The constructor is actually a function that is called when an `instance` of the class is created. It has a special name, `__init__(some_args)` .

```
# a class with a constructor that does nothig
class Human:
    def __init__():
        pass
```

Properties and Methods

A proerty is basically a variable. Yes, the variable that we have discussed in `Exercise 1 ( Creating a variable)` , just that here they belong to a class thus they are reffered to as properties of that class.

A method is a function of the class. There is only a slight difference. Lets create a class with an attribute, `name` , a `constructor` and a method `say_hello` , that prints `hello {name}` to the screen.

```
# a class with one attribute, name
# a method, say_hello, that prints hello
class Human:

    def __init__(self, name):
        self.name = name

    def say_hello():
        print(f"hello, {self.name}")
```

Code breakdown

- `class Human:` creates a class with name `Human`
- `def __init__(self, name)` creates a Constructor and passes a data, name to it.
- In the constructor block, the data, `name` , was assigned to an attribute of the class `self.name`
- The `self` here refers to the `class` object it self - `Human` .
- `self.name` is the same as `Human.name`
- `self` must always be used as it is a conversion in Python, which means we could use another keyword in place of `self`

Note

```
def say_hello(name):
        print(f"hello, {name}")
```

In `say_hello` , we could just pass the name data to it and then print `hello name` but we passed the name to the constructor instead. This will make the name attribute accessible everywhere in the class thus we reference it with the `self` keyword, `self.name` .

Creating an instance of a class

An instance of a class is an object of the class with all the attributes and methods of that class. So the `class Human` may have an instance of say, `John` . If we have a class of `Car` , `Ferrari` is an instance of the class `Car` .

```
# We created a class earlier
# we shall create an instance of that class
# class name is Human, with an attribute, name,
# passed to it's constructor
# and a method, say_hello() that prints hello with the data passed as name.

john = Human('John Doe')

# access/call the method
# class_object.method_name() or class_instance.method_name()
john.say_hello()
# output-> hello John Doe

# access the attribute
john.name = "Sandra Doe"
john.say_hello()
# output-> hello Sandra Doe
```

Note

- Use the dot, `.` operator to access methods and attributes of the class
- Python is case sensitve, thus `human` and `Human` are completely different things
- When a method is return a value, just like a normal function, get the returned value and print it.

Rectangle class

> create a file, with name `rectangle.py`

```
# this is a simple is class to model a rectangle
# it has a 2 methods that may return the area and perimeter.

class Rectangle:

    def __init__(self, length, breadth):
        self.length = length
        self.breadth = breadth

    def area(self):
        """ returns the area as, length * breadth """
        return self.length * self.breadth

    def perimeter(self):
        """ returns the perimeter as, 2 * (length + breadth) """
        return 2 * (self.length + self.breadth)

# instance of the rectangle class
rect_inst = Rectangle(2, 3)  # l = 2, b = 3

# get the area
area = rect_inst.area()

# get the perimeter
perrmeter = rect_inst.perimeter()

print(f"The area and perimeter of the rectangle are, {area} and {perrmeter} respectively")
```

> `""" this is a doc string """` , it is just like a comment.

Practicals

- Re-write the `Rectangle` class, and check for execptions.
- Write a class for basic mathematical operations, such as addition, subtraction, multiplication, division, floordivision (aka integer division), modulo (aka remainder), power, etc
- Write a class for a document statistics. Refer to the `exercise 19` to read about document statistics.
- Write a program that allows creating, reading and updating of the content of a file using OOP.

Summary

- OOP = Object Oriented Programming
- `class` is the blue print of an object
- classes have `attributes` and `methods`
- `attribute` is a data/variable
- `method` is a function
- constructor, `__init__(some_args)` , is used to initialize some attributes
- constructors to do not return any value
- say we have class, Human, `jojn = Human()` is an instance of the class
- `class_name().attribute_name` or `class_instance.attribute_name` to access a class attribute
- `class_name().method_name` or `class_instance.method_name` to access a class method

Exercise 20 b (OOP - Concepts)

There are certain important concepts when it comes to OOP that makes software engineering better and simpler.

> Read the code - relax, that is all there is to it - coding

OOP Concepts

There are some concepts that runs through OOP in all OO languages:

- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

> OOP in Python is basically creation of classes and inheritance.

Note

These OOP concepts cuts across all Object Oriented Programming languages but the implementation is quite different.

Inheritance

Inheritance allows us to replicate/extend/inherit/modify/use another class' (object's) attributes and methods without having to re-write the whole properties and methods again for the second class. So think about any item/product that has been produced and used over some time now. You realize that there are actually different versions of the original. So, say there is version 1, 2, 3, and so on. How do we think version 2, 3, and the others came about? The company developed the v1 and later started v2? or they rather improved on v1 to get v2 then on v2 to get v3? Well, these two are feasible actually, the latter approach is the solution. The latter approach which is improving upon v1 to get v2 and so on works best.

So v1 grows into v2, and thats inheritance or something like that is.

Structure of Inheritance

Just understand that we basically have two classes and one of these two classes would like to use the attributes and methods of the other class. Lets create an `Animal class` and a `Cat class` , where `Cat` inherits the `Animal` .

```
class Animal:
    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.age = age

class Cat:
    pass
```

For one class (the child class) to inherit from another class (the parent class), we pass the name of the parent class, just as we would pass an argument to a function, into a parenthesis before the colon, `:` .

```
class Parent:
    pass

class Child(Parent):
    pass
```

> It is a very good practice to have the classes in seperate files

Lets have a look at the code below,

```
# sharing of common functionanlity using classes

# Super (Parent) class
class Animal:
    def __init__(self, name, color, age):
        self.name = name
        self.color = color
        self.age = age

# Sub (Child) classes

class Cat(Animal):
    def purr(self):
        print("Purr..!!")

class Dog(Animal):
    def bark(self):
        print("Woof..!!")

# instances of the child classes
skull = Cat("Skull Crusher", "red", 5)
skull.purr()

granny = Dog("Granny Hairy", "green", 3)
granny.bark()
```

> Remember to pass in `self` and reference methods and attributes with `self`

Example with multiple inheritance

```
# there are three class
# Human, Robot and Hybrid
# Hybrid inherits froms

class Human:

    def__init__(self):
        print("I am a Human")

class Robot:

    def __init__(self):
        print("I am a Robot")

class Hybrid(Human, Robot):

    def __init__(self):
        print("I am a Hybrid")
```

`super()` function

Use the `super()` to access the original methods in the base class.

```
class Cat:
    def __init__(self, color, legs):
        self.color = color
        self.legs = legs

    def number_of_legs(self):
        return self.legs

    def color_of_obj(self):
        return self.color

    def run(self):
        print(f"I have {self.number_of_legs()} legs")
        print(f"I am {self.color_of_obj()} in complexion")

class Tiger (Cat):
    def __init__(self, color, legs, is_big):
        # to have the same functionality as the base class
        # pass the color and legs data to the constructor of
        # the super class. In this way we can modify the
        # constructor method of the child class to match
        # the parent class
        super().__init__(color, legs)
        # this saves us from doing
        # self.color = color
        # self.legs = legs
        self.is_big = is_big

    def run(self):
        # call the original run methods of the base class
        super().run()
        print("I have enormous claws for sprinting")

felix = Cat("yellow", 6)
felix.run()

tiger = Tiger("green", 5, True)
tiger.run()
```

> continuation in exercise 20 c

Exercise 20 c (continuation of OOP concepts)

Practical

> *We belive that at this stage, with patience, persistence and clear mind and an intension to actually solve a problem, you can do so many things, now. Surely some will be challenging but through this you become better*

Fixed this code. Do not panic, it is actually easier than you may sweat. Some little assistance.

- try to understand what every component (method) does.
    - copy the method somewhere and run it
    - or just comment the others out and run it
- run the code multiple times and actually write down something. Try to trace the execution of the code.
- break it and see what happens by stabbing it with various input types - jot something down - you are becomming a hacker.
    - stabbing means, pass some data to the method or attribute directly.
- try negative values too - just break it and fix it
- add comments where you think necessary - try commenting the code as you may come back in a week or more time.

```
import random
# we shall discuss the line above in the next exercise

# description: This is a simple text based console gambling game.
# Give the user a name and an initial cash (default, $1000)
# Ask the user for his name later in the game, when user is
# making more money - if the current cash is 5 time the initial
# Randomly generate a number in a range and prompt self to
```

```
# guess the number
# if user guess is right, double bet and add to cash
# else, double and subtract from cash
# bet are in folds of $100

# That's it.. Thanks

class Game:

    def __init__(self, username, cash=1000, bet=0):
        self.username = username
        self.cash = cash
        self.bet = bet

        # returns user details
    def game_info(self):
        strinfo = f"You have ${self.cash} in your account.. "
        strinfo += f"and you've placed a bet of ${self.bet}"
        strinfo += f".. Good luck ${self.name}"

        return strinfo

    # returns game details
    def player_info(self):
        strinfo = f"hello {self.username}, you have "
        strinfo += f"${self.cash} is your asset.."

        return strinfo

    # change the users name amids gaming
    # this function has not been used
    def change_username(self, newname):
        self.username = newname

    # makes sure that the bet is in a fold of 100
    # before allowing bet method
    def wager(self, bet):

        try:
            if bet != 0 and bet % 100 == 0:
                if int(bet) < self.cash + 1:
                    self.bet = int(bet)
                    self.cash -= self.bet
                else:
                    print(f"You only have ${self.cash} in your asset..")

                    if self.cash > 100:
                        self.bet = 100
                        print(
                            f"For the love of the game, by default your bet is ${self.bet}")

            else:
                # set bet to 0 since the user didn't chose
                # a bet other than a multiple of 100
                print("Your wager is as tinny as your broke bank..")
                print("Let's just have some fun and call it a day..")
                self.bet = 0

        except ValueError as e:
            print(f"You have to use something quite monery..\n{e}")
            # set a bet again
            bet = int(
                input("place your bet in a fold of hundreds [100, 200, 300, ...]: "))
            # call wager again and pass it, bet
            self.wager(bet)
```

```
    # the main function
    def start_game(self):
        # print user info
        print(self.player_info())

        end_game = "yes".lower()
        while(end_game != "no".lower()):

            # catch non numberic inputs
            try:
                # set a bet
                bet = int(
                    input("place your bet in a fold of hundreds [100, 200, 300, ...]: "))
                self.wager(bet)

                # print game info
                print(self.game_info())

                # checking if bet is greater than half of user player cash
                if self.bet >= 0.5 * self.cash:
                    print("You are a man who admires risks, we love you..")

                # this is where the random numbers are generated
                # alter this to suite for a different functionality
                start, end = 0, 2

                # on this like we use a component from the import
                random_number = random.randint(start, end + 1)

                guess = int(input(
                    f"{self.username}, be smart and pick your lucky number between {start} and {end}: "))

                if random_number == guess:
                    self.bet *= 2
                    self.cash += self.bet

                    # print game info
                    print(self.game_info())
                    print(
                        f"{self.username}, you won this time.. your cash is : {self.cash}")
                else:
                    self.bet = 0
                    # print game info
                    print("you loss..")
                    print(self.game_info())

                if self.cash < 100:
                    print(
                        "Man you have no cash.. You better walk away.. else i'd mob the dirty floor with your damn broke face..")

                    # print user info
                    print(self.game_info())
                    end_game = "no"
                else:
                    end_game = input("Do you want to keep playing? ").lower()

            except ValueError as e:
                print("Dikward.. i bet your broke fat beefy ass you are a cash crop.. start with a hundred and I will double it in a

        print(self.game_info())

# create an instance object of the Game class and call the start method
user = Game("f3erQ6$_", 1000)
user.start_game()
```

Summary

- Inheritance is an OOP Concept
- A child class inherits from a parent class
- pass the parent class as an argument to the child class
- pass multiple classes that way, for multiple inheritance
- use `super()` to access the original attributes and methods of the parent class

Exercise 21 (Modules)

A module is a file (script), containing definitions and statements. This can be any program that we write/wrote and hope to use that same code without copying and pasting the code into the new script. All we have to do is to `import` them.

Example

Consider the code below, this is taken from `exercise 12 a (Functions)` .

```
# file name: area.py
# description:
#   A program that calculates and prints the area of a triangle
#   taking the base and height as inputs

def calc_area():
    base = float(input("Enter base: "))
    height = float(input("Enter height: "))

    area = 0.5 * base * height

    print(f"The area of a triagle of base, {base} and height, {height} is {area}")
```

Import a module

Importing a module is very simple actually, all we need is the `import` keyword and the module name. The module name is the name of the file/script, without the suffix (extension), `.py`

```
import modele_name
```

Import `area.py`

To import the `area.py` , and make use of `calc_area` we have to import it.

```
import area

# think of this `import area` like a class
# with some properties and methods
# the dot notation will work, actually, that's how it would work

area.calc_area()

# refer to this same code in `exercise 12 a (Functions)`
# we called the function similar to how we've done here
# just without the module name and this is because the function is in the module
```

Import objects from a module

Lets assume that there is at least a function in the module and we want to make use of a particular one or all, using the `from module` annotation.

Add a function that calculates the perimeter of the triangle, `calc_peri` , to the `area.py` script and rename the module to `triangle.py` .

```
from module_name import object_name
```

Import a particular object

Our goal here is to import `calc_area` and `calc_peri` from the `triangle` module.

```
from triangle import calc_area, calc_peri

# call the calculate area function
calc_area()

# call the calculate perimeter function
calc_peri()
```

Import all objects

Our goal here is to import all the objects in the module. This can be achieved by using `*` to imply all. With these we have access to all the objects.

```
from triangle import *

# call the calculate area function
calc_area()

# call the calculate perimeter function
calc_peri()
```

Note

`from module import object` is almost similar to `import module` just that with the latter we have to do `module.object` to make use of the object.

> visit Python doc to read on the built-in modules and packages

Math module

The math module contains some mathetical functions algebra, logarithm, trignometry, some constants and others.

Lets make use of pi, e, gcd, exp, factorial, pow, sqrt, cos, sin.

> pi and e are constants and the rest are functions.

We are going to import a lot of things from the math module thus to keep it structured we shall group them as tuples though it will work fine anyways.

```
 # import objects from the math lib
# making use of some constants such as `pi` and `e` ,
# some trig function like the cosine and sine
# and other functions

from math import (
    pi, e,
    cos, sin, tan,
    sqrt, pow, exp,
    gcd, factorial
)

# pi and e

# area of a circle = pi times radius square
radius = 7  # cm
# area = pi * radius ** 2
# area = pi * radius * radius
area = pi * pow(radius, 2)

print(f"The area of the circle of radius, {radius} is {round(area,2)}cm^2")

print(end='\n\n')

# e(n) for some integer `n` , is the same as e ** n, pow(e, n) and e * e * ...n
# lets check if they are actually
if e * e == pow(e, 2) == e ** 2:
    print("Cool, e ** n, pow(e, n) and e * e * ...n are all the same")
else:
    print("Sorry, they are not the same")

print(end='\n\n')

# cos, sin and tan
# given that theta is 60 degree
theta = 60  # degree
print(f"Trig table for degree, {theta}")
print("-------------------------")
print(f"cos({theta}) {cos(theta)}")
print(f"sin({theta}) {sin(theta)}")
print(f"tan({theta}) {tan(theta)}")

print(end='\n\n')

# gcd and factorial
# we shall roughly make use of type hinting

# gcd of 81 and 72
first_int: int = 81
second_int: int = 72
gcd_int: int = gcd(first_int, second_int)

print(f"the gcd  of {first_int} and {second_int} is {gcd_int}")

# the factorial of gcd_int
fact_int: int = factorial(gcd_int)
print(f"The factorial, {gcd_int}!, is {fact_int}")
```

More example

```
# file name: employee.py
# calculating the gross pay

class Employee:

    def __init__(self, name, job, salary):
        self.name = name
        self.job = job
        self.salary = salary

        # some constants
        self.WORKING_DAYS_IN_A_MONTH = 30
        self.WORKING_HOURS_IN_A_DAY = 12
        self.PAY_RATE = 0.25

        # the total number of hours the employee should have done in a whole month
        self.WORKED_HOURS = self.WORKING_HOURS_IN_A_DAY * self.WORKING_DAYS_IN_A_MONTH

    # calculate the gross pay based on the time employee did for the whole month
    # the hours here is the total hours that the employee has overworked
    def return_gross_pay(self, hours):
        gross_pay = 0

        if hours > self.WORKED_HOURS:
            over_time = hours - self.WORKED_HOURS
            over_time_pay = over_time + (self.salary * self.PAY_RATE)

            gross_pay = self.salary + over_time_pay
        else:
            gross_pay = self.salary

        return gross_pay
```

Note

We can do, `from module import object as pseudoname` where the object is now pseudoname. This may allow the resolution of conflicting names

```
# import employee script
from Employee import Employee as App

print("Employee")

me = App("John Matthew Doe", "Python Developer", 1200.00)
print(f"My name is {me.name} and i am a {me.job}.")
print(f"My salary is {me.salary}.")
print(f"When i work overtime, My salary for the month is {me.return_gross_pay(400)}")
```

Or

```
from triangle import calc_area as area, calc_peri as perimeter

# call area
area()

# call perimeter
perimeter()
```

Practicals

- As practice project, write a script, `mathsmodule.py` that performs mathemathical operations such as addition, subtracion, etc. Use a class and create a `test.py` script then import it there. Our `module` should pass these test.
  Assume that the method names will include, `add, subtr, mult, div, floor_div, pow, mod` . Add comments and then create a new file, `mathsmodule.md` - this can by a `.txt` file too. The documentation, should have information about the developer, what the module is about, how to use the module, how and what to improve, how the methods work and how to use the methods, some errors we expect to occur, some constants that were used, etc.

We hope this is becoming clear.

- test add
  - add(1,2) = 10
  - add(1,-2) = -1
  - add(1,2,3,4) = 10
- test subtr
  - subtr(1,2) = -1
  - subtr(1,-2) = 3
- test mult
  - add(1,2) = 2
  - add(1,-2) = -2
  - add(1,2,4) = 8
- test div
  - div(1,2) = 0.5
  - div(1,-2) = -0.5
  - div(5, 0) = None
- test floor_div
  - floor_div(1,2) = 10
  - floor_div(1,-2) = -1
  - floor_div(1,2,3,4) = 10
- test pow
  - pow(1,2) = 1
  - pow(2,3) = 8
  - pow(2,-2) = 0.25
- test mod
  - mod(1,2) = 1
  - mod(22,7) = 1

- For a second version of this program, let a method take 3 args, operator, first_operand, second_operand. Instead of calling any of the methods such as add, just call, `evaluate` with arguments such as `evaluate('+', 2, 5)` to return 7. This should work even if the second and third arguments are in a form of strings such as `evaluate('+', '2', '5')` .
- For a third version, instead of the symbols, such as using, `+` , use `add` instead. Thus `evaluate(+, 2, 5)` , `evaluate('+', '2', '5')` and `evaluate('add', 2, 5)` or `evaluate('add', '2', '5')` works the same.

Summary

- A module is a script that we can reuse in another code - look at the advantages of using a function in `Excercise 12 a (Functions)`
- use the `import` keyword to bring in a module
- `from module import *` imports all the objects in the module
- `from module import some_objects` import these objects
- `from module import object as obj` imports object given a new name `obj` from module

#

Exercise 22 (Unit Testing)

Unit testing is an important part of software engineering. We do unit testing to check the correctness of our program. Usally in most firms, unit tests are written before the code is written - this is known as the Test Driven Development. We shall write the tests after we write the code. We shall use the built-in `unittest` module. There are many testing concepts but we shall basically look at `TestCase` .

Some testing methods

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`

Method	Checks that
`assertNotIs(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`isinstance(a, b)`	`type(a) == type(b)`
`isNotinstance(a, b)`	`type(a) != type(b)`

Example

Consider that we have a program that does some mathematical operations - addition, multiplication and division. We shall use the `assertEqual` and `assertIsNone` method to test if our methods actually are returning the same value as we expect.

```
# mathsy.py
class Mathsy:
    def __init__(self, operator, first_operand, second_operand):
        self.operator = operator
        self.first_operand = first_operand
        self.second_operand = second_operand

    def add(self):
        return self.first_operand + self.second_operand

    def mult(self):
        return self.first_operand * self.second_operand

    def div(self):
        if self.second_operand == 0:
            return None
        else:
            return self.first_operand / self.second_operand

    def evaluate(self):
        if self.operator == '+':
            return self.add()
        elif self.operator == '*':
            return self.mult()
        elif self.operator == '/':
            return self.div()
        else:
            raise Execption(f"{self.operator} not known")
```

```
# test.py
import unittest
from mathsy import Mathsy

# extend the unittest.TestCase
class MathsyTest(unittest.TestCase):

    def test_add(self):
        self.assertEqual(Mathsy('+', 2, 4).evaluate(), 6)
        self.assertEqual(Mathsy('+', 1000, 4).evaluate(), 1004)

    def test_mult(self):
        self.assertEqual(Mathsy('*', 2, 4).evaluate(), 8)
        self.assertEqual(Mathsy('*', 1000, 4).evaluate(), 4000)

    def test_div(self):
        self.assertEqual(Mathsy('/', 2, 4).evaluate(), 0.5)
        self.assertEqual(Mathsy('/', 1000, 4).evaluate(), 250)
        self.assertEqual(Mathsy('/', 0, 4).evaluate(), 0)
        self.assertEqual(Mathsy('/', 4, 0).evaluate(), None)

        # the expected value is None - so we check if it actually does return the None
        self.assertIsNone(Mathsy('/', 4, 0).evaluate())

if __name__ == "__main__":
    unittest.main()
```

Practicals

Write a unit test - TestCases - for the programs written since `exercise 12` .

Summary

- UnitTesting is very important in the world of software engineering
- It checks the correctness of our code
- to use pythons built-in unit testing package, import it, `import unittest`
- create a class, `ModuleTest` and subclass `unittest.TestCase`
- add `if __name__ == "__main__": unittest.main()` to actually run the test when called.

Exercise 23 (Git)

As developers, we may want to have different versions of our software and keep them too. We may want to revert to a previous state of a file or even the whole project and a Distributed Version Control System (DVCS) will do that job. One DVCS is Git. We shall look into how to use git minimally.

We would want to keep our projects on our local server, our disk could corrupt. Thus we would need a remote server for keeping our softwares or scripts thus we create a free Github account before we proceed any further, for our own good as developers who may wish other developers see and review or even share.

Install git

We shall install Git using this website, depending on the OS. Afterwards, we do `git --verion` on the commandline to check if git has been installed successfully.

Configure git

Open the commandline - power shell for windows.

- Set user name: `git config --global user.name your_user_name`
- Set user email: `git --global user.email your_user_email`

This actually can be changed for every software we want to create using git, else this configuration is default.

Create repository

A repository is basically a folder that contains our code and any other file we may need for the development of our software.

There may be two or more instances where we would need to create a git repository:

- when we have already created the folder on our PC, local server, but not under version control.

- when our project is already online (remote) under version control.

Local repository

When the project is created on our local server and we want it under version control, we do the following:

- change directory into the folder of interest, `cd ...`
- do, `git init` - to start git

Remote repository

When the project already exist online, say on Github, do the following:

- Navigate to where we want to create the project
- do, `git clone https://...` - to have the same version of the remote project
- or we can do, `git clone https://... repo_name` - where repo name is the name of the new folder we want to keep we files in instead of the original. This doesn't alter naything.
- then navigate into `repo_name` - this repository comes with the enire version history of the repository.

Adding an existing project to remote server

Here we already have the project or say we have finished developing it locally then we want to push it to the remote repository, do the following:

- initialize the git repo locally, `git init`
- add the files in the local repository, `git add .`
- then stage them, `git commit -m "some message"`
- now copy the remote repository url and do, `git remote add origin https://...` , where `https://../` is the remote url
- now send it to the remote repository, `$ git push origin https://...`

Git status

Every file in a VCS has a status which tells we what has been done to the file thus what state the file is in.

Table of file status

Status	Description
`tracked`	file git knows about
`untracked`	file git doesn't know about
`unmodified`	an unchanged tracked file
`modified`	a changed tracked file
`staged`	a saved tracked file

Check file status

we do, `git status` to see the status of the files in the current repository.

Track files

we do, `git add file_name` to track a single or `git add .` to track the entire files in this particular directory.

When we edit the file, the file status will become `modified` , then we do, `git add file_name` to track file by name, *file_name* or `git add .` to track all the changes made.

There are a lot of ways to see the content of a folder and one is using `ls or dir` for unix/linux or windows respectively, depending on our OS. Git has a special command for this, `git status` . Every file when we clone or initial git, all the files will be `untracked` and `unmodified` . We often check the status as we write to our project.

Stage files

After we track the files we have edited, we have to stage it - add the file to the VSC permanently, ready to be sent (pushed) to the remote server.

do `git commit -m "message"` , where `-m` flag means message and `message` is a text that describes what changes we made to the project or file.

do `git commit` - to open our default editor to give a more decriptive message of what really went down. Save the content and close the file to successfully stage it.

Note

- we can only stage (save) files we have tracked
- make sure that there is always a commit message

Update the repository

- to update the local repo with the remote repo do, `git pull` or sometimes `git pull https://...`
- to update the remote repo with the local repo do, `git push` or sometimes `git push https://...`
- to downloads all history from the repository from the remote repo do, `git fetch`

Check the commits

do `git log` to see the commits we have made. Thus provides we with some information we may use to reset/revert the repo when there is an error.

Reset

`git reset [commit]` where commit is the code we see when we make a commit. Use that code to reset the repository. It undoes all the commits but keeps the changes, `git reset --hard [commit]` does the same but discard the all histories upto the particular commit.

Branch

A branch is similar to a versioning. We can diverge from main code, work on the new version/branch without having to mess with the main code.

Operations with branching

- `git branch` to list all the branches
- `git branch branch_name` to create or divert to a new branch, `branch_name`
- `git checkout branch_name` to switch to another branch, `branch_name`
- `git merge branch_name` , combines the current branch with the content of `branch_name`
- `git branch -d branch_name` to delete `branch_name` from the branches

Practical

- push all we softwares to github
- using git, create a todo app and push the final result to github

Summary

- read about git here
- Every command in git, starts with `git`
- Table of commads we'd use more often
    - `git add file_name`
    - `git add --all` or `git add .`
    - `git commit -m "commit message"`
    - `git pull`
    - `git push`
    - `git log`
    - `git status`
    - `git fetch`
    - `git branch`
    - `git checkout branch`

Resource

- Learn git with Traversy Media
- w3school git
- Git tutorial from git-scm

#

Exercise 24 (SQL)

SQL = Structured Query Language.

This, we may say, is the language we shall use to talk to the database. Read more on about databases here and also learn SQL from Sololearn.

All we are interested in is `CRUD` . We want to learn how to create (insert), read (select), update and delete data. To continue any further, Download the SQLite Browser. It makes the work here easier.

Create a sample table

Lets create a database, `sample.db` , and save it into a folder of any choice, but we recommend the folder in which we have done the practicals in.

Copy and paste this SQL code into windows (text area) when we click on the `Execute SQL` tab.

```
CREATE TABLE `test_tb` (
    `id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    `name` TEXT
);
```

The code above

- This code creates a table with the name, `test_tb`
- There are two fields in the table, `id` and `name`
- The `id` field has some properties
    - `INTEGER` - type of data to store
    - `NOT NULL` - the column value must not be empty or null
    - `PRIMARY KEY` - makes every row unique
    - `AUTOINCREMENT` - increase the `PRIMARY KEY` sequentially. Thus we ignore the values for the `id` field because it is `PRIMARY KEY` and `AUTOINCREMENT`
- The `name` field has only one property, ie.the data type is a `TEXT`

We shall use this table in this discussion.

Insert

Add a row

We may add data (a row) to the table by inserting.

```
INSERT INTO `test_tb` ( `name` ) VALUES('John Doe')
```

Add multiple rows

```
INSERT INTO `test_tb` ( `name` ) VALUES ('Swift Python'), ('kirito'), ('kevin'), ('spit fire')
```

Read

Reading is done by selecting.

Read all rows and columns

This will read all the data and with all the field displaying.

```
SELECT * FROM `test_tb`
```

Read all rows and a particular column

This will read all the data but display only the `name` field.

```
SELECT `name` FROM `test_tb`
```

And this will read all the data but display only the `id` field.

```
SELECT `id` FROM `test_tb`
```

Read rows WHERE some column's value is given (condition)

This will read all the data where the `name` field is equal to `John Doe`

```
SELECT * FROM `test_tb` WHERE `name` = 'John Doe'
```

This will read a row whose column (id) value equals 3

```
SELECT * FROM `test_tb` WHERE `id` = 3
```

This will read a row whose column (id) value greater than 3

```
SELECT * FROM `test_tb` WHERE `id` > 3
```

Update

Let us update a row, with `id` = 1 and change the `name` value to `Terry`

```
UPDATE `test_tb` SET `name` = 'Terry' WHERE `id` = 1
```

Delete

Delete the row with `id` = 1

```
DELETE FROM `test_tb` WHERE `id` = 1
```

Delete the row with `name` = 'kirito'

```
DELETE FROM `test_tb` WHERE `name` = 'kirito'
```

Delete all data

Be careful when we do this.

```
DELETE FROM `test_tb
```

Note

SQL is case insensitive

Practical

use the DB Browser to create some tables and experiment with it

Summary

- SQL is the language of the databases
- Inserting, reading, updating and deleting data is feasible

Resources

- Download SQLite Browser
- Corey Schafer SQLite - Youtube
- wiki Databases
- SQLite
- w3schools SQL

#

Exercise 25 (Python SQLite)

In the previous exercise. `Exercise 24 ( SQL)` we discussed SQL and used it to write to and read from the database. In this exercise we shall make use of a built-in database know as `sqlite3` . Read more about sqlite3.

Create database and table with

We do believe SQLite Browser has been installed. We shall create a database, `sample.db` and save it into a folder, we shall use, `Sample` as the folder name.

Create create table using this script.

```
CREATE TABLE `profile` (
    `id` INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    `name` TEXT,
    `job` TEXT,
    `skill` TEXT,
    `salary` INTEGER
);
```

Connect to database

Before we use the `sqlite3` database, we must first `import` it, then connect to it.

```
import sqlite3

DATABASE_NAME = 'sample.db'

# create connection
connection = sqlite3.connect(DATABASE_NAME)
```

Cursor Object

After we create the connection to the database, we then make use of its cursor object to read and write to the database.

```
# cursor object
cursor = connection.cursor()
```

Execute

We pass an SQL query and some parameters to the `execute` method after we have created the `cursor` object.

SQL query

For the SQl query, it is recommended to use placeholders instead of passing the actual values directly into the qeury.

```
# consider some arbituary query

# don't do this
sql_query = "SELECT * some_tb WHERE `some_field` = 2"

# do this instead
sql_query = "SELECT * some_tb WHERE `some_field` = ?"

# the `?` is a placeholder
```

Execute method

```
# profile -> id:int, name:str, job:str, skill:str, salary:int
# `id` is a primary key and auto increments so we shall ignore it

name = "John Doe"
job = "Software Engineer"
skill = "Python Developer"
salary = 3000

# writing and updating has the same effect of
# affectting some rows, else rowcount is -1
# reading rather returns an iterable (a row - tuple)
sql_query = "INSERT INTO `profile` ( `name` , `job` , `skill` , `salary` ) VALUES(?, ?, ?, ?)"

# the second argument is of the form, *parameters - remember `*arg`
# there would be a change in the database, thus get the number of affected rows
# with `rowcount` attribute
num_affected_row = cursor.execute(sql_query, name, job, skill, salary).rowcount

# do something if num_affected_row > 0

# consider select query
# for more than one row return, we can use fetchone() to get one row
# and fetchall to return all
sql_query = "SELECT * FROM `profile` "
row_profiles = cursor.execute(sql_query).fetchall()

# do something with row_profiles
```

Commit

Sure commit here sound familiar, from `exercise 23 (Git)` . Commit mean save/write changes made to the database permanently. Thus after an `insert, update or delete` you have to commit.

```
connetion.commit()
```

Close cursor and connection

After every thing, we must close the cursor and close the database. This is done so that the database isn't blocked.

```
cursor.close()
connection.close()
```

Full code

```
import sqlite3

DATABASE_NAME = 'sample.db'

# create connection
connection = sqlite3.connect(DATABASE_NAME)

# cursor object
cursor = connection.cursor()

# profile -> id:int, name:str, job:str, skill:str, salary:int
# `id` is a primary key and auto increments so we shall ignore it

name = "John Doe"
job = "Software Engineer"
skill = "Python Developer"
salary = 3000

# insert/write to database
sql_query = "INSERT INTO `profile` ( `name` , `job` , `skill` , `salary` ) VALUES(?, ?, ?, ?)"

# check if there is a change in the database
num_affected_row = cursor.execute(sql_query, name, job, skill, salary).rowcount

if num_affected_row:
    print("profile written to database successful")
else:
    print("profile writing to database unsuccessful")

# save the changes
connetion.commit()

# close cursor and connection
cursor.close()
connection.close()
```

Reading

```
import sqlite3

DATABASE_NAME = 'sample.db'

# create connection
connection = sqlite3.connect(DATABASE_NAME)

# cursor object
cursor = connection.cursor()

# read data
sql_query = "SELECT * FROM `profile` "

# check if there is a change in the database
rows = cursor.execute(sql_query).fetchall()

# every row is like a tuple - integer indexed
if rows > 0:
    for row in rows:
        id = row[0]
        name = row[1]
        job = row[3]
        skill = row[4]
        salary = row[5]

        print(f"ID: {id} - {name} is a(n) {job} specialized in {skill} and earns {salary}")
else:
    print("profile writing to database unsuccessful")

# there is no need to commit here because no changes are made to the database

# close cursor and connection
cursor.close()
connection.close()
```

Practicals

> Use a class if possible

- Write a script that returns the number of characters in the entire file, and the number of characters on each line. Save these two into a database with the name of the file.
- Write a script that returns the document statistics of a give file. The document statistics are number of lines, number of words number of characters with space and witout space.

  ```
  file name
  ---------
  Lines      - 8
  Words      - 71
  Char (ws)  - 403
  Char (wos) - 337
  ```

  Write these into a database
- Write a scripts that backs the content of a file up. Save the back up in the database.

Summary

The concept or steps behind the use of `sqlite3` is quite simple.

- `sqlite3` is a built-in light weight database
- `connect` to the database
- create a `cursor` object
- `execute` some queries
- `commit` the changes
- `close` cursor and connection

#

Swift-Python

This is a swift presentation of the basics of programming, using the python programming language to the extent that we can take it.

This is meant for anyone who is interested in learning python given a limitted period or an experienced developer who want to pick up the python programming. Also students who have taken introduction to computer science or programming would find this helpful in their own frame.

The content and examples, is swift, quick and summarized with examples to try out after reading. We intend to make it better by improving upon it as time passes, add more practical examples for practice sake.

The content in here is very simple to follow, all you need is the latest or stable version of python available on your PC and your favourite text editor or simply install an IDE.

You basically read the content, take notes and try the exercises for your own good.

TO compile to pdf

Visit markdowntopdf, and select the exercise you want to turn to pdf.

To contribute

- write a content
- edit or update a content
- fix typos
- suggest updates or improvement
- create more practical examples
- add a documentation
- read the content and provide feedback
- you may compile a pdf version of all the files
- translate the content

To implement a project

One should break the project into basically four phases, i.e: * Analysis phase * Design phase * Implementation phase * Test phase

> The test phase should be ignored if a test is not needed.
>
> Add comments to make understanding the code better
>
> You can choose to add all phases in one file

Know that the file extension should be `.md`

#

Format used in preparing this content

We used a very simple structure that basically leads us to the brevity of these materials. In order,

- Content
- Examples and Notes (Warnings)
- Practicals (Exercise)
- Summary

Content

This is made up of the prose of the topic of interest with some basic illustrations.

Examples

This is made of examples from the prose, something that is quiet but speaks louder. We add some basic tips here as well.

Practicals

This is a simple exercise that follows after the reader has gone through the prose.

Summary

This is basically bulleting what we thought or think is really needed to carry way after reading the prose as such encouraged to remember.

Format for header text

Let all header text be Capitalized - that is , uppercase first character