

# The Winds Of Python

For the programming is dark and full of errors



Izan Majeed

ASIN: B083R86MFJ

First edition: January, 2020

Second Edition: July, 2021

Copyright © 2020 Izan Majeed

All rights reserved by the author. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system without the prior permission of the author.

Author: Izan Majeed

E-mail: [izan\\_28btech17@nitsri.net](mailto:izan_28btech17@nitsri.net)

GitHub: <https://github.com/izan-majeed>

LinkedIn: <https://www.linkedin.com/in/izan-majeed-886016233/>

Chapter image art:

[Naomi lord art](#)

Dedicated to my parents

Abu-G and Sister-G

## About the Author



The author is a man of many qualities, of which perseverance and deliberation have outshone others in the making of this wonderful text. Just having stepped into the third decade of his life, this book stands testament to the greatness and genius of his persona; to pen down a volume at such a young age and that too on a diverse and rapidly progressing subject. His intense enthusiasm for learning is commendable and evident from the fact that he himself has mastered this subject over a period of less than two years. He thoroughly utilizes his personal experiences in all walks of life and here as well he has cashed in on the difficulties and confusions he himself faced to deliver content that undoubtedly will be beneficial for the uninitiated.

The author is currently pursuing his bachelors in Computer Science and Engineering and resides in Srinagar along with his parents and two siblings (and a red hero sprint cycle).

*(Abdul Kashif Shah)*

## Table of Contents

### Introduction

1. The Zen of Python
2. Who is this book for? What's special about this book?
3. What is Python?
4. Why Python?
5. Downloading and Installing Python
6. IDLE
7. Preview of a Python program
8. Conventions used
9. PEP8 Conventions

### Chapter 1: Python Is Coming

1.	Ghosts are awesome .....	2
2.	Numb the Numbers .....	3
	2.1. Operations .....	5
	2.2. Build up with built-in functions .....	8
3.	Variables: Simple containers .....	12
	3.1. del statement .....	14
	3.2. Rules for variable names.....	15
	3.3. Assigning values .....	16
4.	Unraveling Strings .....	19
	4.1. Triple quoted strings.....	21
	4.2. String concatenation.....	22
	4.3. Escape Sequences .....	23
	4.4. Build up with built-in functions.....	26
	4.5. String Formatting .....	32
	4.6. Indexing and Slicing.....	36
	4.7. in and not operators .....	39
	4.8. String Methods .....	40
5.	What about Lists? .....	44
	5.1. Common Sequence Operations.....	45
	5.2. Strings and Lists .....	50

5.3.	List Methods	.....	52
6.	Tuples: Immutable Lists	.....	60
6.1.	Common Sequence Operations	.....	62
6.2.	Tuple Methods	.....	64
7.	Map the Mapping	.....	65
7.1.	Dictionaries are mutable	.....	66
7.2.	Dictionaries are unordered objects	.....	68
7.3.	dict() constructor	.....	68
7.4.	len() function	.....	69
7.5.	Dictionary Methods	.....	69
7.6.	Pretty Print	.....	73
8.	Ready, Set, Go!	.....	75
8.1.	Sets are not subscriptable	.....	76
8.2.	Sets are mutable	.....	76
8.3.	len() function	.....	77
8.4.	del statement	.....	78
8.5.	in and not operators	.....	78
8.6.	Set Operations	.....	79

## Chapter 2: Let Me Flow

1.	Comparisons	.....	82
2.	Control Flow	.....	85
2.1.	if, else statement	.....	85
2.2.	elif statement	.....	89
3.	Loops	.....	92
3.1.	while statement	.....	92
3.2.	for statement	.....	94
3.3.	range type	.....	99
3.4.	break, continue statement	.....	103
3.5.	else clause	.....	105
3.6.	zip() and enumerate() functions	.....	107
3.7.	Comprehensions	.....	109

## Chapter 3: We Do Not Repeat

1.	Functions .....	114
1.1.	return statement .....	116
1.2.	Documentation Strings.....	120
1.3.	map() and filter() functions.....	121
1.4.	lambda expression .....	122
1.5.	Default argument values.....	124
1.6.	*args and **kwargs .....	125
1.7.	pass statement .....	129
1.8.	Scope of a variable .....	130
1.9.	Recursion .....	138
1.10.	Decorators .....	140
1.11.	Generators and Iterators.....	141
2.	Modules .....	144
2.1.	Packages .....	149
2.2.	What is pip? .....	151
3.	Exception Handling .....	155
3.1.	Syntax Errors .....	155
3.2.	Exceptions .....	155
3.3.	try and except clause.....	157
3.4.	else and finally clause.....	160
3.5.	finally clause vs normal code.....	162
3.6.	raise statement .....	165

## Chapter 4: Better an OOPs than a what if

1.	Object Oriented Programming.....	167
1.1.	class variables vs instance variables.....	168
1.2.	self keyword .....	169
1.3.	<code>__init__()</code> method .....	169
1.4.	Private variables and private methods.....	172
1.5.	class methods, instance methods and static methods.....	175

2.	Special Methods .....	179
3.	The Fours Pillars of OOP .....	185
3.1.	Encapsulation (Information hiding).....	185
3.2.	Inheritance .....	186
3.3.	Polymorphism .....	191
3.4.	Abstraction (Implementation hiding).....	193

## Chapter 5: File and Regex

1.	File Handling .....	198
1.1.	Open () .....	198
1.2.	Mode .....	199
1.3.	with statement .....	201
2.	Regular Expressions .....	206
2.1.	<code>re.findall()</code> .....	206
2.2.	<code>re.search()</code> .....	216
2.3.	<code>re.split()</code> .....	218
2.4.	Contact Extractor .....	219

## Grow through what you go through

1.	Palindrome .....	221
2.	Armstrong .....	222
3.	Count Vowels .....	222
4.	Count the characters .....	223
5.	Classic FizzBuzz .....	224
6.	Largest Continuous Sum .....	224
7.	Tower of Hanoi .....	225
8.	Fibonacci Series .....	226
9.	String reversal .....	227
10.	Integer reversal .....	228
11.	Anagrams .....	228
12.	List Chunks .....	229

---

13. Collatz Conjecture .....	230
14. Title Case String .....	231
15. Unique Characters .....	231
16. Most Repeated Character .....	232
17. Linear Search .....	232
18. Binary Search .....	233
19. Calculator .....	234
20. Guess the Number .....	235
21. Tic Tac Toe .....	236
22. Assignment Project .....	242

## Data Structures and Algorithms

1. Data Structures .....	244
2. Algorithms .....	270
2.1. Sorting algorithms .....	270
2.2. Graph traversal algorithms.....	276

## Glossary

## What next?



## Introduction

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

*The Zen of Python, by Tim Peters*

---

## Who is this book for?

In this book, I've not compared Python with other programming languages out there; so if you don't know anything about programming, you're most welcome. Computer codes that seem nonsense to you, will make sense on the completion of this book.

If you know more or less about some other programming languages, Pythonic code will seem like pseudocode to you and you can extend your programming skills to the next level without any difficulty. Being a student myself, I've tried my best to make you understand various concepts with coding examples instead of writing geeky definitions, so you can actually learn something rather than end up cramming definitions.

Though this book does contain some official definitions, copied from official documentation, they are simplified to student friendly format. So if you're preparing for some university exams then this **book is not for you**. Everyone except this category can relish this book, even if you're an Arts and Music student. In short, this is the Hitchhiker's informal guide to Python.

## What's special about this book?

Written by a student; if you are a student too, then we're in the same boat.

## What is Python?

Computers are dumb machines. They may evaluate complex mathematical calculations within fractions of seconds but without any instructions, these machines are literally useless.

Python is one of the many programming languages with which you can give instructions to your computer to perform some kind of boring task which you don't want to do manually. Formally speaking, Python is an interpreted language, meaning the instructions are translated on the fly, one line at a time. The instructions given through any programming language to the dumb computer are referred to as a 'program'. Being a programmer simply means "**I know how to give instructions to my computer!**"

Python was created by **Guido van Rossum**, and first released on February 20, 1991. It is named after a British Comedy group **Monty Python** and the person who knows how to write a computer program in Python is known as **Pythonistas**.

## Why Python?

Cause you choose it, just kidding. There are many articles on the internet discussing why Python is better than other languages. I'll simply say Python's readability (clean syntax) makes it a great programming language. If you want to see a fair comparison of Python with other programming languages, I'll suggest this article from Python's official documentation <https://www.python.org/doc/essays/comparisons/>.

## Downloading and Installing Python

Now as you understand what Python is, or at least have some idea what we're doing here. Let's see how you can get Python into your machine (by machine I mean your computer). You came here to learn Python, right? Now my question to you is, who taught Python to your computer? The answer is no one. Then how can your computer understand a language that it was not taught? The answer is your computer only understands one language which is a language of 0's and 1's only and it is known as **binary language**. So there must be a translator installed on your computer that translates your Pythonic instructions to a form your dumb machine can understand. Python interpreter does this job for you.

You can download the Python Interpreter from <https://www.Python.org/> and make sure you download any version of Python 3, like Python 3.8.5, as this book strictly follows the syntax of Python 3. If you try to execute programs of this book on Python 2, they may or may not work properly.

Python is a free software and is available for Windows, Mac OS X and Linux; installation process may vary among different operating systems. Just follow the instructions; download Python 3 and install it on your machine.

Lucky Ubuntu users (GNU/Linux) just have to fire 2 commands from the terminal. Press Ctrl+Alt+T to quickly open the terminal, type the following commands and boom! Python is installed as if some magical Pythonic dust is sprinkled over your machine.

```
izan@20-04:~$ sudo apt-get install Python3
izan@20-04:~$ sudo apt-get install idle3
```

## IDLE

Once Python is installed successfully, you should be able to open IDLE, irrespective of the operating system running on your machine. GNU/Linux users can simply open the terminal type **python3** to launch IDLE in the terminal itself.

IDLE stands for Integrated Development and Learning Environment. It is a basic editor and interpreter environment which ships with the standard distribution of Python (formal definition). IDLE is an interactive shell, just like your terminal (or command prompt), which allows you to try fancy bits of Pythonic code without having to save and run an entire program. An IDLE window looks something like this:

```
Python 3.8.5 (default, May 27 2021, 13:30:53)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more
information.

>>>
```

The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt.

>>> represents the default Python prompt of the interactive shell.

## Preview of a Python program

Type the following line of code into IDLE

```
>>> print('I just wrote my first line of code in Python.')
I just wrote my first line of code in Python.
>>>
```

`print()` is Python's built-in function that yells out whatever is passed in parentheses. Inside IDLE, input and output is distinguished by the presence or absence of `>>>` respectively. If you quit from IDLE and open it again, all your work will have been lost. So how can you store your Python programs? Using Python's file editor, you can create files that contain Python programs and easily execute those files in IDLE; this is known as creating a script.

1. Open IDLE, press Ctrl+N (or select *File* ► *New Window*) to open the file editor. The code you write in the file editor is not executed immediately as you hit the *Enter* key on your keyboard.
2. Press Ctrl+S (or select *File* ► *Save*) to save the file with extension .py like `myscript.py`
3. Press F5 (or select *Run* ► *Run Module*) to run your script. As soon as you hit F5 your script should run in IDLE.

For example:

*myscript.py*

```
print('First line of my script.')
print('Second line of my script.')
```

*output*

```
First line of my script.
Second line of my script.
>>>
```

## Conventions Used

Most of the examples used in this book are directly written on IDLE which looks like:

```
>>>  
>>> if True:  
...  
...
```

... is the default Python prompt of the interactive shell when entering code for an indented code block or, within a pair of matching left and right brackets or triple quotes. This prompt may not appear in certain versions, so no need to worry.

Here's a little suggestion, as you open the interpreter, type:

```
>>> import this
```

Whatever is displayed, go through it and if possible memorize the magical words.

In this book, the source code of scripts written in the file editor are immediately followed by their respective outputs, separated by a horizontal line and the name of the scripts are written on the top. For example:

*scriptname.py*

```
source code  


---

  
output
```

---

## PEP8 Conventions

PEP 8, Python Enhancement Proposal, has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point. It may seem absolute nonsense if you're reading this for the first time.

1. Use four-space indentation, and no tabs. Four spaces are a good compromise between small indentation and large indentation.
2. Wrap lines so that they don't exceed 79 characters. This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
3. Use blank lines to separate functions and classes, and larger blocks of code inside functions.
4. When possible, put comments on a line of their own.
5. Use docstrings.
6. Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
7. Name your classes and functions consistently; the convention is to use CamelCase for classes and `lower_case_with_underscores` for functions and methods.
8. Always use `self` as the name for the first method argument.
9. Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
10. Likewise, don't use non-ASCII characters in identifiers if there is the slightest chance of people speaking a different language reading or maintaining the code.

*Source: pydocs*

As quoted by Tim Peters, Sparse is better than dense. So I've made this book sparse in terms of text, but abundant in coding examples which make up for the former. I've tried to cover all the concepts through examples and then at the end, projects are given. Try to solve these projects yourself. There is no hard and fast rule in programming; whatever makes sense, just write it down. Programming is all about practice, so try to solve as many problems as you can. It's like you know how to frame sentences in English but you can not write a novel just by learning grammar.



01. Python is Coming

## 1. Ghosts Are Awesome

Ghosts are the statements that are invisible to the interpreter. A hash (#) character transforms a normal statement into a ghost, and its effect is extended to the end of the physical line. These ghosts which are neglected by interpreter, are formally called as **Comments**.

So why do you need ghosts in your program? Can't you write a ghost-free script? Comments are added just to clarify what your code is trying to do. Adding comments doesn't affect your main program.

It's your personal choice whether to include them or not. If you want to make your programs readable, then you should add comments to your script, as mentioned in Zen of Python "*Readability counts*". It's a favour you're giving to your future self.

```
>>> # This is my first script
>>> print('Just playing with ghosts.')    # :(
Just playing with ghosts.

>>>
>>> # Print('Hehe, this comment is just for fun')
>>>
```

You can have multi-line comments using triple quotes (""""...""" or "'...'") but these are usually used for docstrings which you'll encounter in the subsequent sections.

*ghost.py*

```
'''Ghosts are everywhere.
I am a multiline comment.'''
print('Why am I scared?')
```

---

Why am I scared?

## 2. Numb the Numbers

The Python interpreter can be directly used as a calculator. Just type an expression on it and it will return the result. An expression is nothing but a piece of syntax which can be evaluated to some value.

```
>>> 3 + 7  
10  
>>>  
>>> 22 * 7  
154  
>>>  
>>> 13 / 7  
1.8571428571428572  
>>>
```

Python offers three distinct numeric types: *integers*, *floating point numbers*, and *complex numbers*. Among which integers and floating point numbers are commonly used.

1. **I**ntegers (`int`): Simple whole numbers which can be positive, negative or neutral (zero).

```
>>> type (-17)  
<class 'int'>  
>>>  
>>> type (0)  
<class 'int'>  
>>>  
>>> type (9)  
<class 'int'>  
>>>
```

- 
- 2. Floating point numbers** (float): Numbers with a decimal point; sometimes exponential notation is used to define a floating point number.  
e.g.  $3E2 = 3*10^2$ ,  $4E-1 = 4*10^{-1}$  or  $3e2 = 3*10^2$ ,  $4e-1 = 4*10^{-1}$ .

```
>>> type (3.17)
<class 'float'>
>>>
>>> 6E2
600.0
>>>
>>> 7e1
70.0
>>>
>>> 3142E-3
3.142
>>>
>>> type (2.12e3)
<class 'float'>
>>>
```

- 3. Complex numbers** (complex): Numbers having a real and an imaginary part.  
General form of complex numbers is  $a+bj$ ,  $a$  being real and  $b$  imaginary e.g  $3+4j$ ,  $3-2j$  etc.

```
>>> type (3+6j)
<class 'complex'>
>>>
>>> type (7-4j)
<class 'complex'>
>>>
>>> type (0j)
<class 'complex'>
>>>
```

Python's built-in constructors `int(x)`, `float(x)`, and `complex(re, img)` can be used to produce numbers of a specific type.

```
>>> int (3.754)
3
>>> float (6)
6.0
>>> complex (3, 9)
(3+9j)
>>> complex (8, -1)
(8-1j)
>>> complex (9)
(9+0j)
>>>
```

## 2.1 Operations

All numeric types support the basic arithmetic operations.

```
>>> 22 / 7
3.142857142857143
>>> 3 / 10
0.3
>>> 16 * 4
64
>>> 11 + 22 - 33
0
>>>
>>> (2-3j) + (6+2j)
(8-1j)
>>>
```

PEP 8 conventions recommend use of spaces around operators.

```
>>> 7 + 15 + 6 -3
25
>>> (5 + 2j) * (7 - 1j)
(37+9j)
>>>
>>> # without spaces
>>> 7+15+6-3
25
>>> (5+2j)*(7-1j)
(37+9j)
>>>
```

The order of operations (precedence) of operators is similar to that of mathematics; you may have heard of the BODMAS rule. If more than one operators of the same precedence are present, they are evaluated from left to right.

In the given table, operators are sorted by descending priority i.e. from highest precedence to the lowest.

Operator	Operation	Example
$x^{**} y$	$x$ to the power $y$	$4^{**} 2 = 16$
$x \% y$	Remainder of $x / y$	$8 \% 3 = 2$
$x // y$	Integer quotient of $x$ and $y$	$17 // 5 = 3$
$x / y$	Quotient of $x$ and $y$	$17 / 5 = 3.4$
$x * y$	Product of $x$ and $y$	$4 * 3 = 12$
$x - y$	Difference of $x$ and $y$	$7 - 1 = 6$
$x + y$	Sum of $x$ and $y$	$2 + 5 = 7$

// operator represents floor division that rounds down the quotient towards minus infinity.

```
>>> 29 / 10
2.9
>>> 29 // 10
2
>>> 10 // 3
3
>>>
```

Modulo operator (%) returns the remainder of its operands.

```
>>> 5 % 3
2
>>> 16 % 4
0
>>>
```

A set of parentheses () overrides the usual precedence. An expression that is enclosed within parentheses has the highest precedence i.e, that expression is evaluated first.

```
>>> 6 * 4 ** 2
96
>>> (6 * 4) ** 2
576
>>> 3 + 9 * 6 / 2** 4 - 1
5.375
>>> (3 + 9 * 6) / 2 ** 4 -1
2.5625
>>> (3 + 9 * 6) / ( 2 ** 4 -1 )
3.8
>>>
```

```
>>> 4 / 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
```

Wait! What was that? What just happened when you tried to divide 4 by 0? It's just an error, in this case it's ZeroDivisionError. The interpreter always throws an error if you try to evaluate an invalid expression. Don't panic if you get an error message, it's fun to have errors. Feel proud everytime the interpreter throws an error; the more errors you encounter, the more you learn.

## 2.2 Build up with Built-in Functions

Python has a number of functions built into it that are always available. Functions are identified as some name followed by parentheses (). Parameters (or arguments) are the values that are given to functions. Different functions take different number of parameters, some are optional and some compulsory. You'll learn more about the functions in chapter 3 of this book. For now, you can play around with some of the built-in functions listed below.

### 1. `abs(x)`

Returns the absolute value of a number; if a number is complex, its magnitude is returned.

```
>>> abs (7)
7
>>> abs (-5.14)
5.14
>>> abs (3+4j)
5.0
>>>
```

---

## 2. bin (x)

Returns the binary representation of an integer prefixed with “0b” (zero-b).

```
>>> bin (4)
'0b100'
>>>
>>> bin (-7)
'-0b111'
>>>
>>> bin (0)
'0b0'
>>>
>>> bin (3.6)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'float' object cannot be interpreted as an integer
>>>
```

## 3. divmod (x, y)

Returns a pair of numbers consisting of their quotient and remainder (q, r).

```
>>> divmod (16, 4)
(4, 0)
>>>
>>> divmod (9, 7)
(1, 2)
>>>
>>> divmod (10, 3)
(3, 1)
>>>
```

---

#### 4. hex (x)

Returns the hexadecimal representation of an integer prefixed with “0x”.

```
>>> hex (10)
'0xa'
>>>
>>> hex (9)
'0x9'
>>>
>>> hex (64)
'0x40'
>>>
>>> hex (255)
'0xff'
>>>
```

#### 5. oct (x)

Returns the octal representation of an integer prefixed with “0o”.

```
>>> oct (12)
'0o14'
>>>
>>> oct (8)
'0o10'
>>>
```

#### 6. pow (x, y)

Returns x to the power y, equivalent to  $x**y$ .

```
>>> pow (3, 4)
81
>>>
```

pow() can have third optional argument as `pow(x, y, z)`, which returns x to the power y, modulo z i.e.,  $((x^{**} y) \% z)$ .

```
>>> pow (2, 3, 4)
0
>>> pow (3, 2, 5)
4
>>>
```

## 7. round (n, ndigits)

Rounds a number to a given precision in decimal digits; if *ndigits* is omitted an integer is returned.

```
>>> round (6.6)
7
>>> round (3.123)
3
>>> round (9.1475, 2)
9.15
>>> round (-7.3428, 1)
-7.3
>>>
```

### 3. Variables: Simple containers

You've probably heard this term, 'variable' in your elementary classes of algebra. It's an alphabet which refers to an unknown mathematical value, right? The concept is a little bit different in programming. Here, a variable is a container in which you can store any value like numbers. An equal sign (=) followed by some value, assigns that value to the variable.

*Syntax:*

```
variable_name = value
```

You can even assign a variable which is assigned earlier, to another variable.

```
>>> num_1 = 32
>>> num_2 = 33
>>>
>>> result = num_1 + num_2
>>> print (result)
50.6
>>>
```

To check the value of a variable, either pass the variable name to `print()` function or just type the variable name to the interpreter and hit *Enter*. The Previous value of a variable is overwritten if it is reassigned.

```
>>> var = 123
>>> var
123
>>> var = 3.14
>>> var
3.14
>>>
```

Interpreter throws an error if a variable is declared without assigning any value.

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

Multiple variables can be initialized in a single line of code. This multiple assignment is also mentioned in the next section.

```
>>> x, y, z = 92, 3.14, 8+3j
>>>
>>> x
92
>>>
>>> y
3.14
>>>
>>> z
(8+3j)
>>>
>>> x = y = z = 911
>>> x
911
>>> y
911
>>> z
911
>>>
```

Python provides some shorthand methods to assign variables, a modified version of their previous value.

```
>>> a = 7
>>> a += 1      # a = a+1
>>> a
8
>>> a /= 4      # a = a/4
>>> a *= 2      # a = a*2
>>> a -= 3      # a = a-3
>>> a
1.0
>>>
```

### 3.1 *del* statement

You can delete a variable using Python's **del** keyword, which stands for delete, followed by variable name.

*Syntax:*

```
del variable_name
```

```
>>> a = 16
>>> a
16
>>>
>>> del a
>>> a
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
>>>
```

## 3.2 Rules for variable names

You can't define any fancy junk of symbols as a variable name. Python follows some rules for naming a variable, these are:

1. Only alphabets, numbers and underscores are to be used.

```
>>> my_1st_variable = 3.14  
>>>
```

2. Variable names can't start with a number.

```
>>> 1_item = 23  
File "<stdin>", line 1  
 1_item = 23  
^  
SyntaxError: invalid token  
>>>
```

3. Whitespaces not allowed.

```
>>> my age = 99  
File "<stdin>", line 1  
  my age = 99  
^  
SyntaxError: invalid syntax  
>>>
```

4. Can't use special symbols like ? ! @ # % ^ & \* ~ - + .

```
>>> abc@gmail.com = 199  
File "<stdin>", line 1  
  SyntaxError: can't assign to operator  
>>>
```

- 
5. Don't use built-in keywords like *int*, *float* as variable names.

```
>>> int (7.6)
7
>>> int = 36
>>> int (7.6)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>>
```

You can alternatively add an underscore at the end like `int_`, `float_` without affecting the functionality of that keyword.

```
>>> int_ = 33
>>> int (1.83)
1
>>>
```

### 3.3 Assigning values

Python is a dynamically typed language which means, the type of variable, whether it is *int*, *float* etc, is unknown until the code is executed. Unlike most programming languages, variable type is not specified when the variable is initialized. Python allows you to assign values of different types, to the same variable.

```
>>> x = 7
>>> type (x)
<class 'int'>
>>>
>>> x = 3.14
>>> type (x)
<class 'float'>
>>>
```

```
>>> x = 3 + 2j
>>> type(x)
<class 'int'>
>>> x
(3+2j)
>>>
```

Python is so flexible that it supports multiple assignments i.e., you can assign values to more than one variable in a single line.

*Syntax:*

```
var_1, var_2, ..., var_n = value_1, value_2, ..., value_n
```

Number of variables on the Left Hand Side must match the number of values on the Right Hand Side. However, if you assign multiple values to a single variable, a **tuple** is created, which is another data type that you'll encounter later.

```
>>> a, b = 11, 22
>>> a + b
33
>>> int_1, float_1, complex_1 = 7, 3.14, 5+6j
>>> int_1
7
>>> float_1
3.14
>>> complex_1
(5+6j)
>>>
>>> var_1, var_2 = 11, 22, 33
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
>>>
```

```
>>> x, y = 1.32
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot unpack non-iterable float object
>>>
>>> x = 11, 22, 33
>>> type (x)
<class 'tuple'>
>>>
```

With this concept of multiple assignments, you can easily swap the value of variables.

```
>>> pi = 3.14
>>> e = 2.71
>>>
>>> pi, e = e, pi
>>>
>>> pi
2.71
>>> e
3.14
>>>
```

## 4. Unraveling Strings

If you're familiar with String theory, you might be good at physics. That theory has absolutely nothing to do with Python Strings. A Python string is just a sequence of characters which allows you to use alphabets, words or even sentences in your program. In formal words, textual data in Python is handled with strings where data is stored as text sequences.

String objects are enclosed within **single** or **double quotes**. If you have no idea what an object means in programming, don't worry, that topic is explained in Chapter 4 of this book. For now, just remember everything in Python is an object which is why Python is so special.

```
>>> empty_string = ""
>>> type (empty_string)
<class 'str'>
>>>
>>> alphabet = 'a'
>>> print (alphabet)
a
>>> type (alphabet)
<class 'str'>
>>>
>>> note = 'Everything in Python is an object.'
>>> print (note)
Everything in Python is an object.
>>> type (note)
<class 'str'>
>>>
>>> where_are_you = "Stuck within double quotes!"
>>> print (where_are_you )
Stuck within double quotes!
>>> type (where_are_you )
<class 'str'>
>>>
```

Use a combination of double and single quotes for a string that contains single or double quotes inside its text, else *SyntaxError* will be thrown by the interpreter.

```
>>> who_am_i = "I'm a Pythonistas."  
>>>  
>>> who_am_i  
"I'm a Pythonistas."  
>>>  
>>> print (who_am_i)  
I'm a Pythonistas.  
>>>  
>>> who_am_i = 'I'm a Pythonistas.'  
File "<stdin>", line 1  
    who_am_i = 'I'm a Pythonistas.'  
          ^  
  
SyntaxError: invalid syntax  
>>>  
>>> old_saying = 'Someone once said, "No matter where you go,  
there you are!"'  
>>>  
>>> old_saying  
'Someone once said, "No matter where you go, there you are!"'  
>>>  
>>> print (old_saying)  
Someone once said, "No matter where you go, there you are!"  
>>>
```

## 4.1 Triple-quoted strings

A triple-quoted string is a string which is bounded by three instances of either a quotation mark ("") or an apostrophe (''). It allows you to include both single and double quotes within a string.

*"Three single quotes" or """"Three double quotes"""*

```
>>> x = '''I've just used "triple quoted string".'''
>>> print (x)
I've just used "triple quoted string".
>>>
```

Triple quoted strings are commonly used to span multiple lines unlike double or single quoted strings. All associated whitespace are included in the string literal.

```
>>> why_now = '''Let me do it now.
... Let me not defer nor neglect it;
... for I shall not pass this way again.'''
>>>
>>> print (why_now)
Let me do it now.
Let me not defer nor neglect it;
for I shall not pass this way again.
>>>
>>> why_now = "Let me do it now.
File "<stdin>", line 1
    why_now = "Let me do it now.
          ^
SyntaxError: EOL while scanning string literal
>>>
```

Instead of >>>, interpreter prompts ... , while entering a multi-line construct.

## 4.2 String Concatenation

String concatenation allows you to combine two or more strings together with the plus (+) operator. It is simply the addition of two or more strings.

```
>>> 'Awesome' + 'Python'  
'AwesomePython'  
>>>  
>>> 'Awesome ' + 'Python'  
'Awesome Python'  
>>>  
>>> 'Awesome' + ' ' + 'Python'  
'Awesome Python'  
>>>  
>>> x = "Everywhere!"  
>>> 'Grammar ' + 'Nazis ' + x  
'Grammar Nazis Everywhere!'  
>>>
```

Other arithmetic operations, like subtraction, are not supported by strings. However you can replicate a string using asterisk (\*) symbol.

```
>>> 'Women' - 'Wo'  
Traceback (most recent call last):  
File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for -: 'str' and 'str'  
>>>  
>>> 'She sells sea-shells. ' * 3  
'She sells sea-shells. She sells sea-shells. She sells  
sea-shells.'  
>>>
```

## 4.3 Escape Sequences

Escape sequences consist of a backslash (\) followed by a particular character, each having some special meaning.

Name	Escape Character
Alert	\a
Backspace	\b
Backslash	\\\
Carriage return	\r
Double quote	\”
Form feed	\f
Horizontal tab	\t
Single quote	\’
Vertical tab	\v

For example:

```
>>> arryn = "\"As high as honour\""
>>> print (arryn)
"As high as honour"
>>>
>>> tully = '\'Family, Duty, Honour\''
>>> print (tully)
'Family, Duty, Honour'
>>>
```

Besides \' and \", \\n and \\t are commonly used escape characters; you can try all other escape sequences by yourself.

If you put a variable name directly into the interpreter, it will spit the value stored in that variable including escape characters; use `print()` function instead.

```
>>> new_line = 'new\\nline'
>>> new_line
'new\\nline'
>>>
>>> print (new_line)
new
line
>>>
>>> hor_tab = "Horizontal\\ttab"
>>> print (hor_tab)
Horizontal tab
>>>
```

If backslash (\) is not followed by any character, the interpreter will throw an *EndOfLine* error. However, a whitespace after backslash can solve this error.

```
>>> only_backslash = '\\'
      File "<stdin>", line 1
          only_backslash = '\\'
                           ^
SyntaxError: EOL while scanning string literal
>>>
>>> backslash_with_space = '\\ '
>>> print(backslash_with_space)
\
>>>
```

Remember that Python uses the backslash as an escape sequence in string literals; if the escape sequence isn't recognized by the interpreter, the backslash and subsequent character are included in the resulting string.

```
>>> print ('\z is invalid escape seq.')
\z is invalid escape seq.
>>>
>>> print ('Another invalid escape seq. is \p')
Another invalid escape seq. is \p
>>>
>>> print ('before \c after')
before \c after
>>>
>>> print ('Vertical \v tab.')
Vertical
tab.
>>>
```

To print escape characters in normal strings, backslash should be repeated twice. This can be confusing, especially with regular expressions (yet to cover), so it's highly recommended that you use **raw strings** instead.

```
>>> print ('new lines \\n are no longer new.')
new lines \n are no longer new.
>>>
>>> print("horizontal tab \\t")
horizontal tab \t
>>> print("vertical tab \\v")
vertical tab \v
>>>
```

## Raw strings

In Python, you can completely ignore all escape characters just by placing **r** before the beginning quotation mark of a string. These special strings are called raw strings.

```
>>> normal_string = "can not ignore \n and other escape
characters."
>>> print (normal_string)
Can not ignore
and other escape characters.

>>>
>>> raw_string = r"Ignores \n and all escape characters."
>>> print (raw_string)
Ignores \n and all escape characters.

>>>
```

## 4.4 Build up with built-in functions

### 1. str(x) and repr(x)

Just like *int()*, *float()* and other constructors; Python has a built-in constructor for string objects as *str()*. There is a similar function *repr()* which also returns string representation of an object.

```
>>> str (159)
'159'
>>> str (3.14)
'3.14'
>>>
>>> repr (6.626)
'6.626'
>>> type (repr (6.626))
<class 'str'>
>>>
```

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter. For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`.

(directly copied from pydocs)

## 2. print (value, ...)

The value passed to `print()` function is displayed to standard output (monitor). You can pass variable names or simple strings to it. Multiple values are separated by a comma (,).

```
>>> print ('String to be printed.')
String to be printed.
>>> lang = 'Python'
>>> version = 3.8
>>> print (lang, version)
Python 3.8
>>>
>>> print ('Current version of', lang, 'is', version)
Current version of Python is 3.8
>>>
```

`print()` function can have various optional arguments, you'll mostly use `end` and `sep`.

`end`: string appended after the last value, default a newline (\n).

`sep`: string inserted between values, default a space (' ').

```
>>> a = 10
>>> b = 17
>>> print (a, b)
10 17
>>> print (a, b, sep="\t")
10 17
>>> print (a, b, end="\t")
10 17 >>>
```

### 3. input (prompt)

This function allows you to enter data externally into your program as it reads a string from standard input (keyboard). You can pass an optional message to be displayed before reading input.

```
>>> age = input()
55
>>> age
'55'
>>> age = input ('Enter your age: ')
Enter your age: 55
>>>
>>> input ("Enter a random string: ")
Enter a random string: Winter is coming
'Winter is coming'
>>>
```

Remember, the *input()* method always accepts data as string, even if you enter a value of any other type. You've to explicitly cast the *input()* function into the required type.

```
>>> a = input ("String value: ")
>>> String value: I'm a string
>>> b = input ("Integer value: ")
Integer value: 36
>>> c = input ("Float value: ")
Float value: 3.14
>>>
>>> print (type(a), type(b), type(c))
<class 'str'> <class 'str'> <class 'str'>
>>>
>>> b = int (input ("Integer value: "))
Integer value: 36
>>> c = float (input ("Float value: "))
Float value: 3.14
```

```
>>>  
>>> print (type(a), type(b), type(c))  
<class 'str'> <class 'int'> <class 'float'>  
>>>
```

#### 4. `help(object)`

This built-in function invokes Python's built-in help system; a help page is generated about the object passed. If no parameters are given, the interactive help system starts. If the object to be passed is a function, pass it without parentheses () .

```
>>> help(input)  
Help on built-in function input in module builtins:  
  
input(prompt=None, /)  
    Read a string from standard input.  The trailing newline is  
    stripped.  
  
    The prompt string, if given, is printed to standard output  
    without a  
        trailing newline before reading input.  
  
    If the user hits EOF (*nix: Ctrl-D, Windows: Ctrl-Z+Return),  
    raise EOFError.  
    On *nix systems, readline is used if available.  
  
>>>
```

## 5. len (x)

It returns the length of *sequence* type object like strings.

```
>>> len ('Cartoon')
7
>>> x = 'Ice and Fire'      #whitespace counts
>>> len (x)
12
>>> len (15467)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>>
```

## 6. ord (c) and chr (u)

*ord()* returns the Unicode code point of a character.

*chr()* is the inverse of *ord()*.

The valid range for the Unicode code is from 0 through 1,114,111.

```
>>> ord ('a')
97
>>> ord ('A')
65
>>> ord ('1')
49
>>> chr (97)
'a'
>>> chr (1114111)
'\U0010ffff'
>>> chr (1114112)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: chr() arg not in range(0x110000)
>>>
```

## 7. min() and max()

*max()* returns the largest item of an iterable, like strings, while *min()* returns the smallest item. Strings are evaluated as per ordinal value of characters.

```
>>> min (3)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'int' object is not iterable
>>>
>>> min ('Hobbit')
'H'
>>> ord ('H')
72
>>> max ('Aaron')
'r'
>>>
```

However, if two or more arguments are passed the largest one is returned. In case of strings, the ordinal value of the first character of each string is compared; if same, next characters are compared and so on.

```
>>> min (2,6)
2
>>>
>>> min (7, 3, 1, -5)
-5
>>>
>>> max (6, 56, 78, 0)
78
>>>
>>> min ('Aardvark', 'Aardwolf')
'Aardvark'
>>>
```

## 4.5 String Formatting

Instead of trying to chain items together using commas or string concatenation, string formatting allows us to inject values more efficiently. Python provides three ways of string formatting:

### 1. Placeholder formatting

Also known as old string formatting, uses modulo operator (%) to inject different types of variables.

*Syntax:*

```
print ("... %operator1 ... %operator2... " %(value1, value2...))
```

A particular % specifier is used for a particular data type. The three % specifiers that are used in Python are given in the following table:

Operator	Data type
%s	string
%d	integer
%f	floating point number

```
>>> who = "You"
>>> print ('Remember why %s started' %who)
Remember why You started
>>> pi = 3.141592
>>> print ('pi, %f is not equal to 22/7' %pi)
pi, 3.141592 is not equal to 22/7
>>>
>>> print ('Python %d is awesome' %3)
Python 3 is awesome
>>> print ('%s to %s' %('Allergic', 'Mornings'))
Allergic to Mornings
```

Floating point numbers use the format `%width.precisionf`, where `width` (fieldwidth) is whitespace padding and `precision` represents how to show numbers after the decimal point. Padding is usually used to style the output.

```
>>> pi = 3.141592653589793
>>> print ("value of pi: %1.3f" %pi)
value of pi: 3.142
>>> print ("value of pi: %10.4f" %pi)
value of pi:      3.1416
>>>
```

## 2. String `format()` method

This is a better way to inject objects into strings using a set of curly brackets `{}` and `str.format()` method.

*Syntax:*

```
print ("...{}...".format(value))
```

```
>>> who = "Me"
>>> print ('Hear {} Roar'.format(who))
Hear Me Roar
>>> print ("{} is the {}".format ("Ours", "Fury"))
Ours is the Fury
>>>
```

A number in curly braces can be used to refer to the position of a value passed to `str.format()` method; numbering starts from zero.

```
>>> print ("{} is composed of {}".format("Forever", "nows"))
Forever is composed of nows
>>> print ("{}1{} is composed of {}".format("Forever", "nows"))
nows is composed of Forever
>>>
```

You can assign your own positioning values and format a string accordingly.

```
>>> print ("{} wakes up the {} cells".format('Nonsense',
'Brain'))
Nonsense wakes up the Brain cells
>>> print ("{} wakes up the {} cells".format(n='Nonsense',
b='Brain'))
Brain wakes up the Nonsense cells
>>>
```

This method also takes care of *width* and *precision* of floating point numbers. Text padding and alignment can also be done using this method as *position: width.precisionf*.

```
>>> print ("{pi: 1.6f}".format (pi=3.1415926535))
3.141593
>>>
>>> print ("{:0:6}{:1:6}{:2:7}{:3:7}".format ("This", "text",
"looks", "great"))
This text looks great
>>>
>>> print ("{:0:6}{:1:6}{:1:7}{:3:7}".format ("This", "text",
"looks", "great"))
This text text great
>>>
```

### 3. f strings

This is the easiest and most flexible method that I personally prefer over all types of formattings. Normal strings are converted into *fstring* literals by placing **f** or *F* before the starting quotation mark of a string just like you put **r** to raw strings.

Syntax:

```
f"....{expression}...."
```

```

>>> x = 'Blessed'
>>> y = 'Obsessed'
>>> z = f'Stressed, {x} and Python {y}'
>>> print ('z')
Stressed, Blessed and Python Obsessed
>>> print (f'Stressed, {x} and Python {y}')
Stressed, Blessed and Python Obsessed
>>>
>>> # str.format() method
>>> print ('Stressed, {} and Python {}'.format(x, y))
Stressed, Blessed and Python Obsessed
>>>
>>> # old string formatting
>>> print ('Stressed, %s and Python %s' %(x, y))
Stressed, Blessed and Python Obsessed
>>>

```

*Precision* and *width* can be handled greatly by this type of formatting as  
**value: width.precisionf.**

```

>>> pi = 3.141592653589793
>>>
>>> # old string formatting
>>> print ("Value of pi: %1.7f" %pi)
Value of pi: 3.1415927
>>>
>>> # str.format() method
>>> print ("Value of pi: {0:1.7f}".format (pi))
Value of pi: 3.1415927
>>>
>>> # f string
>>> print (f"Value of pi: {pi:1.7f}")
Value of pi: 3.1415927

```

## 4.6 Indexing and Slicing

As strings are text sequences, you can access individual characters using the subscript operator `[]` after an object, to call its index. Strings can be indexed (subscripted) from 0 to  $length-1$ . Moreover, Python also supports negative indexing.

```
>>> 'Satan'[2]
't'
>>>
>>> a = "Indexing starts at zero"
>>> a[0]
'I'
>>>
>>> fourth_element = a[3]
>>> fourth_element
'e'
>>>
>>> a[8]      # whitespace counts
' '
>>> a[-1]     # grabs the last character
'o'
>>>
```

If you attempt to use an index that is too large, an *IndexError* will be thrown, as the string can't access an element which is out of its bounds.

```
>>> a[99]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

In addition to indexing, Python also supports slicing. While indexing is used to obtain individual characters, slicing allows you to obtain substrings. Sushi operator (`:`) is used to perform slicing which grabs text slices up to a designated point. e.g. `a[x:y]`, starting from index x upto y (including x but excluding y).

```
>>> m = 'Slice me!'
>>> m[2:7]
'ice m'
>>>
```

You can also specify the step size of a slice as `string[start:stop:step]`; the default step size is 1.

```
>>> a = 'Stay trippy little hippie'
>>> a[3:9]
'y trip'
>>>
>>> a[3:9:2]
'ysi'
>>>
>>> a[3:9:3]
'yr'
>>>
>>> message = "Lxoesit Umnoiscxozren"
>>> len (message)
23
>>> message [0:23:2]
'Lost Unicorn'
>>>
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
>>> text = 'Cruising the waves'  
>>>  
>>> text[:]      #grabs everything  
'Cruising the waves'  
>>>  
>>> text[::-1]  
'Cusn h ae'  
>>>  
>>> text[::-2]  
'sevaw eht gnisiurC'  
>>>
```

Out of range slice indexes are handled gracefully when used for slicing.

```
>>> hodor = 'Hold the door'  
>>>  
>>> hodor[2:7]  
'ld th'  
>>>  
>>> hodor[3:99]  
'd the door'  
>>>  
>>> hodor[99:999]  
'  
>>>
```

## Strings are immutable

Python strings have a special property of immutability i.e., Once a string is created, the characters within it can't be changed or replaced. Therefore, assigning values to an indexed position in a string throws an error.

```
>>> x = 'Cue the Confetti'
>>> x[5] = 'z'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

## 4.7 *in* and *not* operators

To check whether a character or substring is contained within a string, use Python's *in* keyword that returns *True* if substring is found, else *False*. *not* operator simply reverses the functionality of *in* operator.

```
>>> vowels = 'aeiou'
>>> 'a' in vowels
True
>>> 'z' in vowels
False
>>> 'x' not in vowels
True
>>>
>>> statement = 'Let go or be dragged!'
>>> 'Let go' in statement
True
>>> 'or' not in statement
False
>>> "Let's go" in statement
False
>>>
```

## 4.8 String Methods

Methods are special functions that are called on an object, this may seem an absolute gibberish to you, that's ok, you'll encounter objects later. For now just remember, strings are objects which have some built-in functions (methods) that make our life a lot easier.

Methods are called with a dot (.) operator followed by the name of the method. General form of a method is `object.method_name(arguments)`. You have already seen the `str.format()` method of string objects. Let's try some other methods.

### 1. `str.capitalize()`

It returns a copy of the string with its first character capitalized and the rest lowercased.

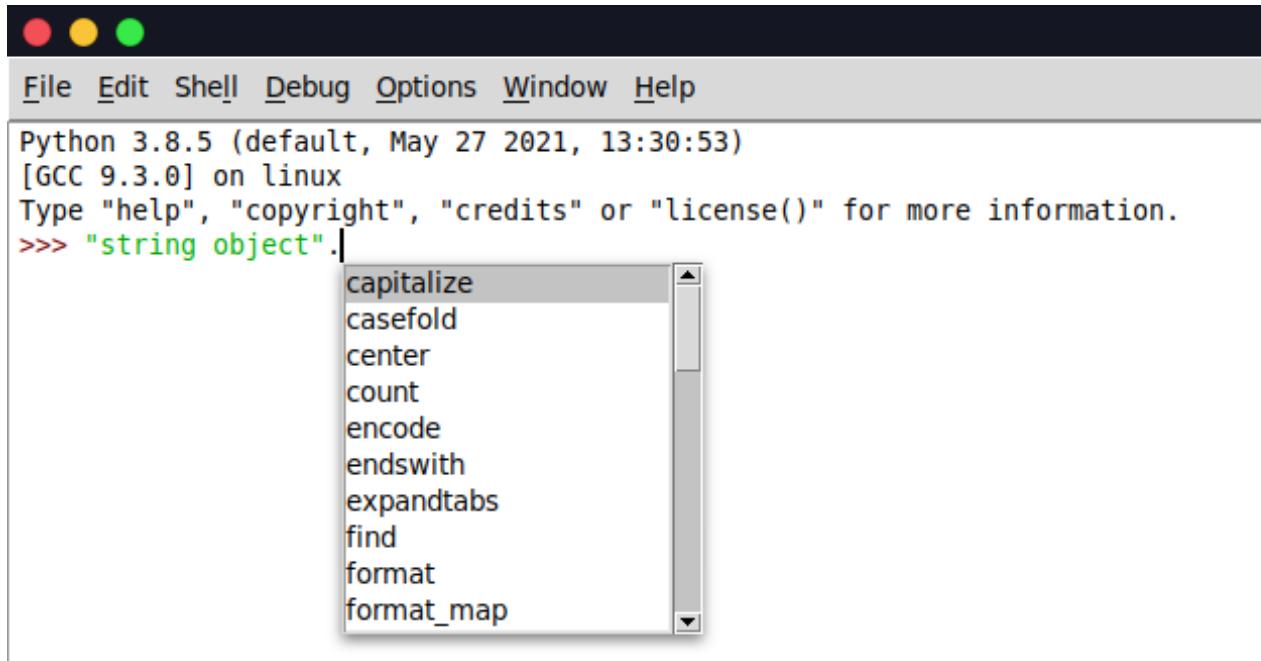
```
>>> 'hypermnesia'.capitalize()
'Hypermnesia'
>>>
>>> greyjoy = 'wE d0 n0t s0w'
>>> greyjoy.capitalize()
'We do not sow'
>>> greyjoy
'wE d0 n0t s0w'
>>>
```

### 2. `str.title()`

It returns a titlecased copy of the string i.e. the first character of each word is uppercased and the remaining characters are in lowercase.

```
>>> 'valar morghulis'.title()
'Valar Morghulis'
>>>
>>> bran = 'three-eyed raven'.title()
>>> print (bran)
Three-Eyed Raven
>>>
```

There are several other methods with which you can easily manipulate string objects. It's a boring task to memorize all of these. So what's the way out? Just try to understand how a particular method works and what can be achieved with that method, the rest of the job is done by the *Tab* key. The Tab key suggests all the methods that are available for a particular type of object. Hit the *Tab* key after the dot (.) operator to see a list of all available methods.



If you want to check the description of a particular method, you can pass that method to our helper, the *help()* function as *help(str.method\_name)*.

```
>>> help(str.casefold)
Help on method_descriptor:
casefold(self, /)
    Return a version of the string suitable for caseless
    comparisons.

>>>
```

### 3. str.count (substring)

It returns the number of non-overlapping occurrences of a substring.

```
>>> 'hippopotomonstrosesquippedaliophobia'.count('po')
2
>>> 'hippopotomonstrosesquippedaliophobia'.count('o')
7
>>>
>>> 'eeeeeeeeeeeeee'.count('e')
14
>>> 'eeeeeeeeeeeeee'.count('ee')
7
>>> 'eeeeeeeeeeeeee'.count('eee')
4
>>>
```

### 4. str.index (substring)

It returns the index of the first instance of substring or character, passed to it.

```
>>> x = 'Mess with North, leave in hearse!'
>>> x.index ('s')
2
>>> x.index ('orth')
11
>>>
>>> x.index ('z')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

---

## 5. str.lower()

It returns a lowercased copy of the string.

```
>>> 'sElcOUth'.lower()
'selcouth'
>>>
```

## 6. str.upper()

It returns an uppercased copy of the string.

```
>>> 'dell'.upper()
'DELL'
>>>
```

## 7. str.replace(old, new)

It returns a copy of the string with all occurrences of substring, old replaced by new.

```
>>> 'pet'.replace('e', 'u')
'put'
>>>
>>> 'North'.replace('th', 'way')
'Norway'
>>>
>>> x = 'Pele'
>>> x.replace('e', 'o')
'Polo'
>>> x
'Pele'
>>>
```

## 5. What about Lists?

List, as the name suggests, is a list of something. It is another *sequence* data type which stores similar or different object types as a list. A list is created within square brackets [] with comma-separated values (items) in between. You can also use `list()` constructor and pass a *sequence* type argument like strings to quickly create a list. Just as strings are ordered sequences of characters, lists are ordered sequences of objects. Moreover, Python lists neither have a fixed size nor fixed type.

**Array** is another fancy name for list data type. However, to handle complex mathematical array operations, Python has *numpy arrays* for that, which is a whole new topic in itself. For now, let's stick to the *list* data type only.

```
>>> empty_list = []
>>> type (empty_list)
<class 'list'>
>>>
>>> houses = ['Stark', 'Targaryen', 'Lannister', 'Baratheon']
>>> print (houses)
['Stark', 'Targaryen', 'Lannister', 'Baratheon']
>>>
>>> season = 1
>>> episode = 7
>>>
>>> name = 'You Win or You Die'
>>> info = [name, season, episode]
>>>
>>> print (info)
['You Win or You Die', 1, 7]
>>>
>>> list ('aeiou')
['a', 'e', 'i', 'o', 'u']
>>>
```

It's a good programming practice to spread out the items of a list i.e., one item per line and if you're consistently adding and removing items, it's useful to put a comma at the end (last item) of the list as well.

```
>>> cities = [
... 'White Harbor',
... 'Braavos',
... 'Tyrosh',
... ]
```

## 5.1 Common Sequence Operations

Strings and other *sequence* type objects, whether mutable or immutable, have certain similar operations which are listed in the following table:

Operation	Result
<code>x in s</code>	True if an item of <code>s</code> is in <code>x</code> , else False
<code>x not in s</code>	False if an item of <code>s</code> is in <code>x</code> , else True
<code>s + t</code>	Concatenation of <code>s</code> and <code>t</code>
<code>s * n</code>	Replication of <code>s</code> , <code>n</code> times
<code>s[i]</code>	Item at <b>i</b> th index of <code>s</code> ,
<code>s[i:j]</code>	Slice of <code>s</code> from <code>i</code> to <code>j</code>
<code>s[i:j:k]</code>	Slice of <code>s</code> from <code>i</code> to <code>j</code> with step <code>k</code>
<code>len(s)</code>	Length of <code>s</code>
<code>min(s)</code>	Smallest item of <code>s</code>
<code>max(s)</code>	Largest item of <code>s</code>

where `s` is a sequence type object

## 1. Concatenation and Replication

Lists support operations like *concatenation* (+) and *replication* (\*).

```
>>> ['a', 'e', 'i', 'o', 'u'] + [11, 22, 33]
['a', 'e', 'i', 'o', 'u', 11, 22, 33]
>>>
>>> a = ['Harry', 'Potter']
>>> a * 3
['Harry', 'Potter', 'Harry', 'Potter', 'Harry', 'Potter']
>>>
```

## 2. Indexing and Slicing

Lists can also be indexed and sliced. A slice operation returns a copy (*shallow*) of the provided list. This concept of *shallow* and *deep* copy is discussed in section 5.3.

```
>>> basket = ["apples", "grapes", "oranges", "mangoes",
"bananas"]
>>>
>>> basket [0]
'apples'
>>>
>>> basket [-1]
'bananas'
>>>
>>> basket [2:5]
['oranges', 'mangoes', 'bananas']
>>>
>>> basket [4]
'bananas'
>>>
```

## Nesting

Lists can be nested i.e., you can have a list within a list. Items of a nested list (list of lists) can be accessed by *multiple* indexing.

```
>>> characters = [ ['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Rob', 'Bran'] ]
>>> characters[0]
['Cersei', 'Jammie', 'Tyrion']
>>> characters[0][2]
'Tyrion'
>>> characters[1][0]
'Arya'
>>>
>>> matrix = [ [11,22], [33,44] ]
>>> matrix[0][1]
22
>>>
```

### 3. *del* statement

Just as a *del* statement deletes a variable, it can delete item(s) from a list as well.

```
>>> even_nums = [22, 44, 55, 66, 88]
>>> del even_nums[2]
>>> even_nums
[22, 44, 66, 88]
>>> del even_nums[1:3]
>>> even_nums
[22, 88]
>>> del even_nums
>>> even_nums
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'even_nums' is not defined
>>>
```

#### 4. *in* and *not* operators

These operators check if an item belongs to a given list or not and then returns a boolean accordingly (*True* or *False*).

```
>>> bag_of_words = ["include", "reverse", "response", "lazy"]
>>>
>>> 'include' in bag_of_words
True
>>>
>>> 'path' in bag_of_words
False
>>>
>>> bag_of_words = ["include", "reverse", "response", "lazy"]
>>> 'render' not in bag_of_words
True
>>>
```

#### 5. *len()* and *sum()*

*len()* function can be applied to return the number of items in a list.

*sum()* adds the items of a list from left to right and returns the total.

```
>>> prime_nums = [2, 3, 5, 7, 11]
>>> len (prime_nums)
5
>>>
>>> x = [951, 753]
>>> sum (x)
1704
>>>
>>> sum ([11, 22, 33])
66
>>>
```

## 6. `min()` and `max()`

Python's built-in `min()` and `max()` functions allow you to grab the minimum and maximum element from a list respectively.

```
>>> x = [11, 22, 33, 44, 55, 66]
>>>
>>> min (x)
11
>>> max (x)
66
>>>
>>> a = ['apple', 'Adam', 'grapes', 'Gutenberg']
>>>
>>> min (a)
'Adam'
>>> max (a)
'grapes'
>>>
>>> ord ('A')
65
>>>
>>> b = ['apple', 'apollo', 'Adapter', 'Addle']
>>>
>>> min (b)
'Adapter'
>>> max (b)
'apple'
>>>
>>> ord ('a')
97
>>>
```

## 5.2 Strings and Lists

### 1. Lists are mutable

Unlike strings, lists are mutable, i.e. items of a list can be changed.

```
>>> nums = [0, 11, 22, 33, 44, 55]
>>>
>>> nums[2] = 99
>>>
>>> nums[3:5] = [300, 400]
>>>
>>> nums
[0, 11, 99, 300, 400, 55]
>>>
>>> nums[1:3] = []      #removes items at index 1 and 2
>>> nums
[0, 300, 400, 55]
>>>
```

### 2. str.split()

Strings can be directly casted into lists, using `list()` constructor. A better option for this conversion is to use `str.split()` method which is more flexible than `list()` constructor.

```
>>> martell = 'Unbowed Unbent Unbroken'
>>>
>>> list(martell)
['U', 'n', 'b', 'o', 'w', 'e', 'd', ' ', 'U', 'n', 'b', 'e',
 'n', 't', ' ', 'U', 'n', 'b', 'r', 'o', 'k', 'e', 'n']
>>>
>>> martell.split()
['Unbowed', 'Unbent', 'Unbroken']
>>>
```

`str.split()` method returns a list of all words contained in the given string, with an optional separator. If separator is omitted, the string is separated on whitespaces; if provided, the string is separated on the passed character into a list of substrings.

```
>>> novel = "A song of Ice and Fire"
>>> novel.split()
['A', 'song', 'of', 'Ice', 'and', 'Fire']
>>>
>>> novel = "A      song    of Ice      and Fire   "
>>> novel.split()
['A', 'song', 'of', 'Ice', 'and', 'Fire']
>>>
>>> emails = 'abc@gmail.com xyz@yahoo.com'
>>> emails.split()
['abc@gmail.com', 'xyz@yahoo.com']
>>>
>>> emails.split('@')
['abc', 'gmail.com xyz', 'yahoo.com']
>>>
>>> local_host = '127.0.0.1'
>>> local_host.split()
['127.0.0.1']
>>>
>>> local_host.split('.')
['127', '0', '0', '1']
>>>
>>> message = 'Humanxafterxall'
>>> message.split()
['Humanxafterxall']
>>>
>>> message.split('x')
['Human', 'after', 'all']
>>>
```

### 3. str.join()

It returns a string that is created by joining a list of strings on a particular character.

```
>>> x = ['Make', 'Me', 'Crazy']
>>>
>>> ''.join (x)
'MakeMeCrazy'
>>>
>>> ' '.join (x)
'Make Me Crazy'
>>>
>>> '#'.join (x)
'Make#Me#Crazy'
>>>
>>> '_'.join (x)
'Make_Me_Crazy'
>>>
>>> s = 'too often the only escape is sleep'
>>>
>>> m = s.split()
>>> m
['too', 'often', 'the', 'only', 'escape', 'is', 'sleep']
>>>
>>> x = '_'.join (m)
>>> x
'too Often the Only escape is sleep'
>>>
```

## 5.3 List methods

Python lists have a number of methods, some of the commonly used methods are discussed in this section. You can play around with other methods as well, just hit the *Tab* key to get a complete list of methods and see what else can be done with Python lists.

## 1. list.append (x)

It adds an item at the end of the list.

```
>>> branches = ['CSE', 'ECE']
>>> branches.append('IT')
>>> branches
['CSE', 'ECE', 'IT']
>>>
```

## 2. list.copy ()

It returns a *shallow* copy of the list which is equivalent to `x[:]`, where `x` is a list. Let's first discuss what this *shallow* and *deep* copy means. According to the official Python documentation:

A **shallow** copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original. A shallow copy is only ***one level deep***. The copying process is not recursive and therefore won't create copies of the child objects themselves.

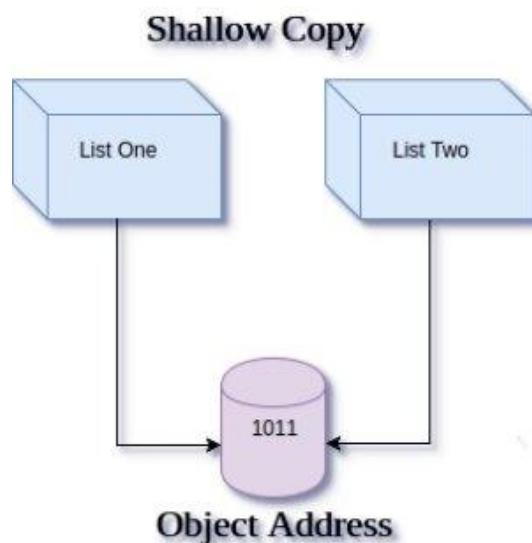
```
>>> house_tully = ['Hoster', 'Edmure']
>>> shallow_copy = house_tully.copy()
>>>
>>> house_tully.append('Catelyn')
>>>
>>> house_tully
['Hoster', 'Edmure', 'Catelyn']
>>>
>>> shallow_copy          #shallow copy is one level deep
['Hoster', 'Edmure']
>>>
>>> characters = [ ['Cersei', 'Jammie', 'Tyrion'], ['Arya',
'Rob', 'Bran'] ]
>>> shallow_copy = characters.copy()
>>>
>>> shallow_copy.append(['Daenerys'])
```

```

>>> shallow_copy
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Rob', 'Bran'],
['Daenerys']]
>>>
>>> characters
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Rob', 'Bran']]
>>>
>>> characters[1][1] = 'Jon Snow'
>>>
>>> characters
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Jon Snow', 'Bran']]
>>>
>>> shallow_copy
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Jon Snow', 'Bran'],
['Daenerys']]
>>>

```

The change ‘*Jon Snow*’, is reflected in both parent list (*characters*) and its shallow copy (*shallow\_copy*). As the shallow copy is only one level deep, it does not create the whole new list of its parent, it just stores the references.

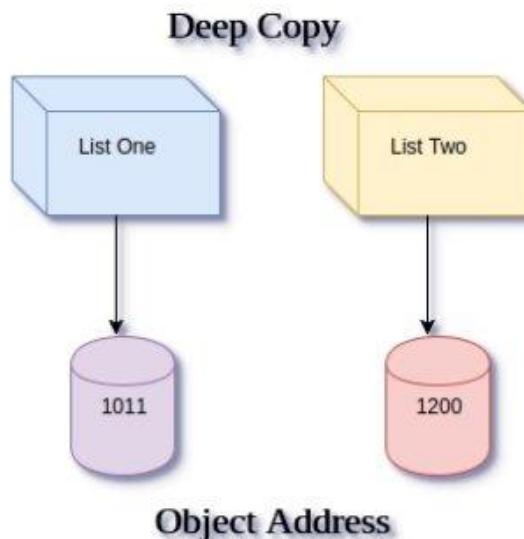


A **deep** copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original. Copying objects this way creates a **fully independent clone** of the original object and all of its children.

To create a deep copy of objects, Python has a special module for that, which is named as *copy*. We'll be discussing more about Python packages in chapter 3.

```
>>> import copy
>>> characters = [ ['Cersei', 'Jammie', 'Tyrion'], ['Arya',
'Rob', 'Bran'] ]
>>> deep_copy = copy.deepcopy(characters)
>>>
>>> characters[1][1] = 'Jon Snow'
>>> characters
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Jon Snow', 'Bran']]
>>> deep_copy
[['Cersei', 'Jammie', 'Tyrion'], ['Arya', 'Rob', 'Bran']]
>>>
```

As you can see in the above example, the change in the parent list (*characters*) is not reflected in its deep copy (*deep\_copy*) as deep copy creates a whole new object of its parent.



### 3. list.count (x)

It returns the number of times an item appears in the list.

```
>>> score_card = [14, 35, 4, 68, 14, 36, 88, 35]
>>> score_card.count(14)
2
>>> score_card.count(88)
1
>>>
```

### 4. list.extend (iterable)

It appends all the items from an *iterable* to the list one by one. *Iterable* is an object that is capable of returning its members one at a time, all *sequence* types like *lists*, *str* etc are *iterables*.

```
>>> houses = ['Stark']
>>> houses.extend('Targaryen')
>>> houses
['Stark', 'T', 'a', 'r', 'g', 'a', 'r', 'y', 'e', 'n']
>>> del houses[1:]
>>> houses.append('Targaryen')
>>> houses
['Stark', 'Targaryen']
>>> x = ['Lannister', 'Baratheon', 'Mormont']
>>> houses.append (x)
>>> houses
['Stark', 'Targaryen', ['Lannister', 'Baratheon', 'Mormont']]
>>> del houses [2]
>>> houses
['Stark', 'Targaryen']
>>> houses.extend (x)
>>> houses
['Stark', 'Targaryen', 'Lannister', 'Baratheon', 'Mormont']
>>>
```

## 5. list.index (item)

It returns the index of the first instance of an item in the list.

```
>>> house_arryn = ['Jon', 'Lysa', 'Robert']
>>> house_arryn.index ('Lysa')
1
>>>
```

## 6. list.insert (i, x)

It adds an item at a particular index to a list.

```
>>> house_stark = ['Eddard', 'Catelyn', 'Robb', 'Arya', 'Bran']
>>>
>>> house_stark.insert(3, 'Sansa')
>>> house_stark
['Eddard', 'Catelyn', 'Robb', 'Sansa', 'Arya', 'Bran']
>>>
```

## 7. list.pop (i)

It removes an item from a specific index. If no argument is passed, it removes the last item from the list and the removed item is returned.

```
>>> house_tyrell = ['Mace', 'Olenna', 'Margaery', 'Loras']
>>> house_tyrell.pop ()
'Loras'
>>> house_tyrell
['Mace', 'Olenna', 'Margaery']
>>> removed_name = house_tyrell.pop (1)
>>> removed_name
'Olenna'
>>> house_tyrell
['Mace', 'Margaery']
>>>
```

---

## 8. list.remove (x)

It removes the item passed as argument and returns nothing. In Python, `None` object refers to nothing.

```
>>> house_greyjoy = ['Balon', 'Theon', 'Yara']
>>>
>>> removed_name = house_greyjoy.remove ('Theon')
>>>
>>> removed_name
>>> type (removed_name)
<class 'NoneType'>
>>>
>>> house_greyjoy
['Balon', 'Yara']
>>>
```

## 9. list.reverse ()

It reverses the order of items in a list. This method is inplace i.e., its effect is reflected to the original list as soon as it is called. `None` object is returned by this method.

```
>>> type(list().reverse())
<class 'NoneType'>
>>>
>>> house_baratheon = ['Robert', 'Stannis', 'Renly']
>>> house_baratheon.reverse()
>>> house_baratheon
['Renly', 'Stannis', 'Robert']
>>>
>>> nums = [33, 11, 22, 99]
>>> nums
[99, 22, 11, 33]
>>>
```

## 10. list.sort (reverse)

It sorts a numeric list in ascending order and a list of strings in alphabetical order. This method is also inplace and returns nothing. If the *reverse* parameter is set to *True*, this method will sort the items of a list in reverse order.

```
>>> type(list().sort())
<class 'NoneType'>
>>>
>>> countries = ['Spain', 'Canada', 'France', 'Turkey', 'China']
>>>
>>> countries.sort()
>>>
>>> countries
['Canada', 'China', 'France', 'Spain', 'Turkey']
>>>
>>> countries.sort(reverse=True)
>>>
>>> countries
['Turkey', 'Spain', 'France', 'China', 'Canada']
>>>
>>> nums = [33, 11, 22, 99]
>>>
>>> nums.sort()
>>>
>>> nums
[11, 22, 33, 99]
>>>
>>> nums.sort(reverse=True)
>>>
>>> nums
[99, 33, 22, 11]
>>>
```

## 6. Tuples: Immutable lists

There is another standard *sequence* data type, the *tuple*, which is very similar to the *list* data type. Unlike lists, tuples are **immutable** i.e., you can't modify the value of an item, once you create a tuple of items. Tuples are not used as often as lists in programming.

Tuples are created using a pair of parentheses () and the items are separated by comma. A singleton tuple (only one item) can be created using a trailing comma as: *a*, or (*a*,) where *a* is an item. You can also create a tuple by passing an *iterable* to the *tuple()* constructor.

```
>>> empty_tuple = ()
>>> x = 'singleton',
>>> type(x)
<class 'tuple'>
>>>
>>> my_tuple = (11, 22, 33, 44, 55, 66)
>>> my_tuple[3] = 99
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
>>> list_ = [6, 9, 1]
>>> tuple_ = tuple(list_)
>>> tuple_
(6, 9, 1)
>>>
>>> # Tuple Nesting
>>> characters = ( ('Cersei', 'Jammie', 'Tyrion'), ('Arya',
'Rob', 'Bran') )
>>> characters [0][2]
'Tyrion'
>>> characters [1][0]
'Arya'
>>>
```

Sometimes items are not enclosed in (), which is referred to as *tuple packing* and the reverse process is referred as *tuple unpacking*. The multiple assignment discussed earlier, is just a combination of tuple *packing* and sequence *unpacking*.

```
>>> # tuple packing
>>> x = "one", 2, "three"
>>> x
('one', 2, 'three')
>>>
>>> # or simply
>>> x = ("one", 2, "three")
>>> x
('one', 2, 'three')
>>>
>>> # tuple unpacking
>>> a, b, c = x
>>> print(a, b, c)
one 2 three
>>>
```

## Tuples are immutable

Tuples are immutable sequences, i.e. you can't modify the items of a tuple.

```
>>> nums = (0, 11, 22, 33, 44, 55)
>>>
>>> nums [2] = 99
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>>
```

## 6.1 Common Sequence Operations

Tuples are *sequence* type objects. So, all the sequence operations, mentioned earlier, can be used with tuples as well.

### 1. Concatenation and Replication

```
>>> ('a', 'e', 'i', 'o', 'u') + (11, 22, 33)
('a', 'e', 'i', 'o', 'u', 11, 22, 33)
>>>
>>> a = ('Harry', 'Potter')
>>> a * 3
('Harry', 'Potter', 'Harry', 'Potter', 'Harry', 'Potter')
>>>
```

### 2. Indexing and Slicing

```
>>> basket = ("grapes", "oranges", "mangoes", "bananas")
>>>
>>> basket[0]
grapes
>>> basket[-1]
'bananas'
>>>
>>> basket[1:4]
('oranges', 'mangoes', 'bananas')
>>>
>>> basket[:]
('grapes', 'oranges', 'mangoes', 'bananas')
>>>
>>> basket[::-2]
('grapes', 'mangoes')
```

### 3. del statement

As tuples are immutable, you can't delete items from them. However, you can delete the tuple object.

```
>>> even_nums = (22, 44, 55, 66, 88)
>>>
>>> del even_nums[2]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item deletion
>>>
>>> del even_nums
>>>
>>> even
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'even' is not defined
>>>
```

### 4. *in* and *not* operators

```
>>> bag_of_words = ("include", "reverse", "response", "lazy")
>>>
>>> 'include' in bag_of_words
True
>>> 'path' in bag_of_words
False
>>>
>>> bag_of_words = ("include", "reverse", "response", "lazy")
>>> 'render' not in bag_of_words
True
>>>
```

---

## 5. len()

```
>>> len( (2, 3, 5, 7, 11) )  
5  
>>>
```

## 6. min() and max()

```
>>> x = (11, 22, 33, 44, 55, 66)  
>>> min (x)  
11  
>>> max (x)  
66  
>>> a = ('apple', 'Adam', 'grapes', 'Gutenberg')  
>>> min (a)  
'Adam'  
>>> max (a)  
'grapes'  
>>>
```

## 6.2 Tuple methods

Tuples have only two methods *tuple.count()* and *tuple.index()*.

*tuple.count()*: Returns number of occurrences of an item.

*tuple.index()*: Returns index of first instance of an item.

```
>>> house_targaryen = ('Daenerys', 'Viserys', 'Aerys')  
>>> house_targaryen.index('Aerys')  
2  
>>> house_targaryen.count('Daenerys')  
1  
>>>
```

## 7. Map the Mapping

There is another data type built into Python called **Dictionary**, which may look like a *sequence* type but it is a *mapping*. Mapping is a technique in which objects are indexed by keys, where a *sequence* type is indexed by a range of numbers.

Python currently offers four *sequence* types – *strings*, *lists*, *tuples* and *range*; and only one standard mapping type, the *Dictionary*.

A Dictionary is a set of *key-value* pairs, with the requirement that the keys are unique. These are also called as *associative arrays* or *Hash tables* in other programming languages. A dictionary is a comma-separated list of **key:value** pairs enclosed within curly braces `{}`. A value can be accessed by calling the key of that particular value like `my_dict[key]`; if the key is not present, *KeyError* is thrown.

```
>>> empty_dict = {}
>>> type (empty_dict)
<class 'dict'>
>>>
>>> ginger = {'name': 'Yigrette', 'allegiance': 'Free Folk',
'season': 2}
>>> ginger["name"]
'Ygritte'
>>> ginger[name]      #name is string not a variable
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
>>>
>>> ginger ['season']
2
>>> ginger ['culture']
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'culture'
>>>
```

## 7.1 Dictionaries are mutable

All mappings are **mutable** objects, as dictionaries are the only built-in *mapping* type in Python, they can be modified.

Value assigned to a key is updated to the dictionary if the given key exists; if the key is absent, the new key-value pair is added. Being mutable, dictionary items can also be deleted by *del* statement.

```
>>> tyrion = {'origin': 'Casterly Rock', 'title': 'Master of
Coin'}
>>>
>>> tyrion['age'] = 38
>>>
>>> tyrion
{'origin': 'Casterly Rock', 'title': 'Master of Coin', 'age':
38}
>>>
>>> tyrion['title'] = 'Hand of the Queen'
>>>
>>> tyrion
{'origin': 'Casterly Rock', 'title': 'Hand of the Queen', 'age':
38}
>>>
>>> del tyrion['origin']
>>>
>>> tyrion
{'title': 'Hand of the Queen', 'age': 38}
>>>
```

**Note:** Among *sequence* and *mapping* types, only *immutable* types (strings and tuples) can be used as dictionary *keys*. However, dictionary values can have any type which means that the dictionaries can be nested (dictionary within dictionary) only on values. Moreover, Numeric types used for keys obey the normal rules.

```
>>> factors = {6: [1,2,3], 8: [1,2,4]}
>>> factors[6]
[1, 2, 3]
>>>
>>> factors = {'6': [1,2,3], '8': [1,2,4]}
>>> factors['8']
[1, 2, 4]
>>>
>>> lcm = {[1,2,3]: 6, [1,2,4]: 8}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>>
>>> lcm = {(1,2,3): 6, (1,2,4): 8}
>>> lcm [(1,2,3)]
6
>>> pen = {'parker': {'color': 'black'}}
>>> pen ['parker']
{'color': 'black'}
>>>
>>> pen = {{'color': 'black'}: 'parker'}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
>>>
>>> stationery = {'pencil': {'shade': '2H'}, 'ruler': {'length': '30 cm'}}
>>> stationery ['ruler']
{'length': '30 cm'}
>>>
>>> ordinal = {'a': ord('a'), 'b': ord('b'), 'c': ord('c')}
>>> ordinal
{'a': 97, 'b': 98, 'c': 99}
>>>
```

## 7.2 Dictionaries are unordered objects

Unlike *sequence* types, dictionaries don't preserve the order of their items. A dictionary is an *associative* array, where keys are mapped to their values. '`==`' is a *comparison* operator which returns *True*, if objects are equal. You'll encounter *comparison* operators in the later sections.

```
>>> [11, 22, 33] == [22, 33, 11]
False
>>> (11, 22, 33) == (33, 11, 22)
False
>>> 'abc' == 'bca'
False
>>>
>>> a = {'one': 1, 'two': 2, 'three': 3}
>>> b = {'two': 2, 'three': 3, 'one': 1}
>>> a == b
True
>>>
```

## 7.3 dict() constructor

Like other data types, dictionary objects can also be created by its built-in constructor, *dict()*.

```
>>> grandh = dict(akh=1, zi=2, trai=3)
>>> grandh
{'akh': 1, 'zi': 2, 'trai': 3}
>>>
>>> # Passing a list of tuples
>>> dict([('akh', 1), ('zi', 2), ('trai', 3)])
{'akh': 1, 'zi': 2, 'trai': 3}
>>>
```

## 7.4 len (x)

*len()* function returns the number of key-value pairs present in a dictionary.

```
>>> ordinal = {'a': 97, 'b': 98, 'c': 99}
>>> ordinal
{'a': 97, 'b': 98, 'c': 99}
>>> len(ordinal)
3
>>>
```

## 7.5 Dictionary methods

You've already seen that there are certain methods that are associated with each data type. Likewise, *dictionary* object has its own set of methods which include:

### 1. dict.clear ()

It removes all the key-value pairs that are present in the dictionary.

```
>>> olenna = {'house': 'Tyrell', 'title': 'Queen of thorns'}
>>> olenna
{'house': 'Tyrell', 'title': 'Queen of thorns'}
>>>
>>> olenna.clear()
>>> olenna
{}
>>>
```

### 2. dict.copy ()

It returns a *shallow* copy of the dictionary.

```
>>> jammie = {'house': 'Lannister', 'title': 'Ser'}
>>> shallow_copy = jammie.copy()
>>> deep_copy = jammie
>>>
```

```
>>> deep_copy ['culture'] = 'Andal'
>>> jammie
{'house': 'Lannister', 'title': 'Ser', 'culture': 'Andal'}
>>> deep_copy
{'house': 'Lannister', 'title': 'Ser', 'culture': 'Andal'}
>>> shallow_copy
{'house': 'Lannister', 'title': 'Ser'}
>>>
```

### 3. dict.get (key)

If the *key* is in the dictionary, it returns its corresponding value. If the key is not present, instead of raising *KeyError*, it returns nothing.

```
>>> my_device = {'RAM': '3 GB', 'model name': 'MC13x'}
>>> my_device.get ('model name')
'MC13x'
>>> my_device.get('baseband version')
>>>
```

### 4. dict.items ()

It provides a view on dictionary object's key-value pairs as tuples.

```
>>> grocery = {'potato': '500g', 'tomato': '1Kg',
'onion':'750g'}
>>>
>>> grocery.items()
dict_items([('potato', '500g'), ('tomato', '1Kg'), ('onion',
'750g')])
>>> type(grocery.items())
<class 'dict_items'>
>>>
```

## 5. dict.keys ()

It provides a view of keys on the dictionary object.

```
>>> grocery.keys ()
dict_keys(['potato', 'tomato', 'onion'])
>>>
```

## 6. dict.values ()

It provides a view of values on the dictionary object.

```
>>> grocery.values ()
dict_values(['500g', '1Kg', '750g'])
>>>
```

## View Object

The objects returned by *dict.items()*, *dict.keys()* and *dict.values()* are *view* objects. They provide a dynamic view on the dictionary's entries, which means that if a key-value pair is modified, the view object associated with that dictionary object will be changed accordingly.

The above mentioned methods are productive when used with looping statements (yet to cover), *in* operator and *not in* operator. The default value of the dictionary object is *dict.keys()*, when used with these statements.

```
>>> regent = {'name': 'Daario', 'allegiance': 'Second Sons',
  'culture': 'Tyroshi'}
>>> regent
{'name': 'Daario', 'allegiance': 'Second Sons', 'culture':
 'Tyroshi'}
>>>
>>> 'name' in regent
True
>>>
```

```
>>> 'Tyroshi' in regent
False
>>> 'Tyroshi' in regent.values()
True
>>>
>>> # Equivalent to: 'allegiance' in regent
>>> 'allegiance' in regent.keys()
True
>>>
>>> 'culture' in regent.items()
False
>>> {'name':'Daario'} in regent.items()
False
>>>
>>> # tuple check
>>> ('name', 'Daario') in regent.items()
True
>>>
```

## 7. dict.pop (key)

It removes a specified key and returns the corresponding value.

```
>>> nums = {'one': 1, 'two': 2, 'three': 3}
>>> nums.pop('one')
1
>>> nums
{'two': 2, 'three': 3}
>>> nums.pop('four')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'four'
>>>
```

## 8. dict.setdefault (key, default)

If the key is in the dictionary, it returns its value; if not, it adds the key with a value provided as *default*. In other words, this method only adds a new *key-value* pair without updating the key, if it already exists. The default value of parameter *default* is *None*.

```
>>> white_walkers = {'ruler': 'Night King', 'lang': 'Skroth'}
>>>
>>> white_walkers.setdefault('ruler', 'Bran')
'Night King'
>>>
>>> white_walkers
{'ruler': 'Night King', 'lang': 'Skroth'}
>>>
>>> white_walkers.setdefault('eye-color', 'blue')
'blue'
>>>
>>> white_walkers
{'ruler': 'Night King', 'lang': 'Skroth', 'eye-color': 'blue'}
>>>
```

## 7.6 Pretty Print

If you have a giant dictionary, like all the countries of the world as *keys* and the corresponding list of states as *values*, the normal *print()* function will display an ugly view of that dictionary.

Remember, what Tim Peters said in his Zen of Python, “*Beautiful is better than ugly*”.

Luckily, Python comes with a module, named *pprint*, which provides a function, named after the module itself as *pprint()* which solves this problem of pretty printing. This function works not only with dictionaries but with other *sequential* objects, say *lists*, as well. This is the second Python package used till now, the first one was *copy* module. You’ll learn more about Python packages (or libraries) in the coming chapter.

```
>>> from pprint import pprint
>>>
>>> us_capitals_10 = {'Alabama': 'Montgomery', 'Alaska':
'Juneau', 'Arizona': 'Phoenix', 'Arkansas': 'Little Rock',
'California': 'Sacramento', 'Colorado': 'Denver', 'Connecticut':
'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee',
'Georgia': 'Atlanta'}
>>>
>>>
>>> print(us_capitals_10)
{'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona':
'Phoenix', 'Arkansas': 'Little Rock', 'California':
'Sacramento', 'Colorado': 'Denver', 'Connecticut': 'Hartford',
'Delaware': 'Dover', 'Florida': 'Tallahassee', 'Georgia':
'Atlanta'}
>>>
>>>
>>> pprint(us_capitals_10)
{'Alabama': 'Montgomery',
 'Alaska': 'Juneau',
 'Arizona': 'Phoenix',
 'Arkansas': 'Little Rock',
 'California': 'Sacramento',
 'Colorado': 'Denver',
 'Connecticut': 'Hartford',
 'Delaware': 'Dover',
 'Florida': 'Tallahassee',
 'Georgia': 'Atlanta'}
>>>
```

## 8. Ready, Set, Go!

You might have studied set theory in mathematics, not kidding this time, that concept is actually applied here. According to the mathematical definition, “A set is an unordered collection of unique elements”.

Python has a separate built-in data type that offers the functionality of sets. *Set* objects support similar mathematical operations used in set theory, like *union*, *intersection* etc. *Set* objects are also created by curly braces {}, like dictionaries, with commas separating each element. You can also use the built-in *set()* constructor.

**Remember:** An empty pair of curly braces {} creates a *dictionary* not a *set*. If you want to instantiate an empty *set* object, use *set()* constructor.

```
>>> a = {11, 22, 33, 44}
>>> a
# sets do not preserve order
{33, 11, 44, 22}
>>>
>>> age_set = {14, 16, 12, 16, 13, 14, 15, 13}
>>> age_set
{12, 13, 14, 15, 16}
>>>
>>> # set(iterable)
>>> eye_color_list = ['black', 'brown', 'black', 'blue',
'green', 'brown']
>>> distinct_eye_colors = set(eye_colors)
>>> distinct_eye_colors
{'brown', 'blue', 'black', 'green'}
>>>
>>> set('Floccinaucinihilipilification')
{'l', 'n', 'f', 't', 'h', 'F', 'i', 'o', 'u', 'p', 'a', 'c'}
>>>
>>> set(('Red', 'Woman'))
{'Woman', 'Red'}
>>>
```

## 8.1 Sets are not subscriptable

Being an unordered collection, *sets* don't retain element position or order of insertion, so they can't support indexing, slicing, or other *sequence-like* operations.

```
>>> x = set([11, 22, 33])
>>> x[2]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
>>>
```

## 8.2 Sets are mutable

Although you can't directly access elements by indexing, you can add or delete elements from a given set with the help of following methods:

### 1. `set.add (e)`

It adds an element to the *set* and has no effect if the element already exists.

```
>>> prime_nums = {2, 3, 5, 7, 11, 13}
>>> prime_nums.add(17)
>>> prime_nums
{2, 3, 5, 7, 11, 13, 17}
>>>
```

### 2. `set.discard (e)`

It removes element *e* from the *set*, if it is present and returns *None* object.

```
>>> prime_nums.discard (11)
>>> prime_nums
{2, 3, 5, 7, 13, 17}
>>> prime_nums.discard (19)
>>>
```

### 3. set.remove (e)

It removes element *e* from the set and throws *KeyError* if *e* is not contained in the *set*.

```
>>> prime_nums = {2, 3, 5, 7, 11, 13}
>>> prime_nums.remove(5)
>>> prime_nums.remove(23)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 23
>>>
```

### 4. set.pop ()

It removes and returns an arbitrary element from the *set* and throws *KeyError* if the *set* is empty.

```
>>> nums = {11, 22, 33}
>>> nums.pop()
33
>>> nums.pop()
11
>>> nums.pop()
22
>>> nums.pop()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
>>>
```

## 8.3 *len()* function

In set theory, the number of elements in a *set* is called its *cardinal* number. The *len()* function is used to achieve the same functionality, it returns the number of elements present in the *set* (cardinality of set).

```
>>> s = {5, 4, 3, 2, 1}
>>> len(s)
5
>>>
```

## 8.4 *del* statement

*del* statement deletes the whole set object.

```
>>> x = {97, 98, 99}
>>> del x
>>> x
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
>>>
```

## 8.5 *in* and *not* operators

```
>>> word = set ('quidnunc')
>>> word
{'q', 'c', 'u', 'i', 'd', 'n'}
>>>
>>> 'i' in word
True
>>> 'x' in word
False
>>> 'z' not in word
True
>>>
```

## 8.6 Sets Operations

Python provides almost all operations that are performed on *sets* which include the following:

### 1. Union of sets ( A.union(B) )

If  $A$  and  $B$  are two sets, then  $A$  union  $B$  is the set of all elements that are either in  $A$  **or** in  $B$  or in both.  $A.\text{union}(B)$  returns the union of sets as a new set i.e., all elements that are in either set. You can use pipe operator (`|`) as a shortcut for union.

```
>>> A = {11, 22, 33, 44, 55}
>>>
>>> B = {44, 55, 66, 77, 88}
>>>
>>> A.union (B)
{33, 66, 11, 44, 77, 22, 55, 88}
>>>
>>> A | B
{33, 66, 11, 44, 77, 22, 55, 88}
>>>
```

### 2. Intersection of sets ( A.intersection(B) )

$A$  intersection  $B$  is the set of all elements common to both  $A$  **and**  $B$ .  $A.\text{intersection}(B)$  returns the intersection of two sets as a new set i.e., all elements that are in both sets. Ampersand (`&`) can be used as a shortcut for intersection.

```
>>> A.intersection (B)
{44, 55}
>>>
>>> A & B
{44, 55}
>>>
```

---

### 3. Difference of sets ( A.difference(B) )

$A$  minus  $B$  is the set of all those elements that are in  $A$  but not in  $B$ .  $A.difference(B)$  returns a new set of elements that are in  $A$  but not in the  $B$ . Minus (-) can be used as a shortcut.

```
>>> A.difference (B)
{33, 11, 22}
>>>
>>> A - B
{33, 11, 22}
>>>
```

You can explore other methods by yourself, just press the *Tab* key after the dot operator, *set()*.*Tab*, to get a list of all available *set* methods.



02. *Let me Flow*

## 1. Comparisons

Comparison operators are also called Relational operators. Before going directly into the control flow statements, let's have a look at how comparisons are done in Python. Comparison operators simply compare two or more elements and return a *boolean* data type, either *True* (1) or *False* (0).

The *boolean* data type is a subtype of integers that are finally evaluated to either *True* or *False*.

Operator	Meaning
<	strictly less than
<=	less than or equal
>	strictly greater than
>=	greater than or equal
==	equal to
!=	not equal to
is	object identity
is not	negated object identity

**Note:** Single equal to operator (=) is an *assignment* operator which assigns values to variables. While double equal to operator (==) is *comparison* operator which checks the condition of equality. A condition is simply a *boolean* expression that is evaluated to either *True* or *False*.

Also remember that the dictionaries only support equality comparisons. If you try to use other comparison operators with dictionaries, *TypeError* will be thrown.

```
>>> 14 == 3
False
>>> "Phobophobia" == "Phobophobia"
True
>>> "Acumen" == "acumen"
False
>>> "12" == 12 #str object is not equal to int object
False
>>>
>>> ['11', '22', '33'] != [11, 22, 33]
True
>>> [11, 22, 33] == [11, 22, 33]
True
>>> [11, 22, 33] == [22, 33, 11] #list is ordered sequence
False
>>> (11, 22, 33) != (33, 11, 22)
True
>>>
>>> (37>5) is True
True
>>> 3 is 3
True
>>> not True
False
>>> {'one': 1} == {'one': 1}
True
>>> {'one': 1} != {'one': 1}
False
>>> {'one': 1} > {'one': 1}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: '>' not supported between instances of 'dict' and
'dict'
>>>
```

Multiple comparisons are chained together using Python's *boolean* operators, *and* and *or*. *and* returns *True* only if both the conditions are *True*; *or* returns *True* if either of the conditions or both the conditions are *True*.

Keyword	Description
and	True, if both the conditions are True, else False.
or	False, if both the conditions are False, else True.

```
>>> 23 < 4 and 28 > 6
False
>>> 6 == 6 or 15 != 17
True
>>>
>>> "string" == "string" and -14 < 0
True
>>>
>>> "Covid".lower() == "coVid".lower() and float(1) == 1
True
>>>
>>> #(I or II) or III
>>> 11 == 22 or 33 == 33 or 44 == 55
True
>>> #(I and II) and III
>>> 11 == 22 and 33 == 33 and 44 == 55
False
>>> #(I or II) and III
>>> 11 == 22 or 33 == 33 and 44 == 55
False
>>>
```

## 2. Control Flow

According to wikipedia, control flow (or flow of control) is the order in which individual statements, instructions or function calls of an imperative program are executed or evaluated. I know, this is a bit of a mouthful, let's try to understand what it actually means.

The code you write is usually executed from top to bottom, left to right, line by line, unless the computer runs across a control flow statement which changes the flow of the current order of execution.

The conditions are provided within the control flow statements, if the given set of conditions are met, only then the following block of code is executed. Let's code these control flow statements and see how the program flow is altered.

### 2.1 *if, else* statement

An *if* statement is the most commonly used control flow statement. After the *if* statement, a condition is given, followed by an **indented** block of code, which is executed only if the condition is *True*.

In Python, various statements can be grouped together as a single block of code, by **indentation**.

The *if* statement is optionally followed by an *else* statement. You don't have to specify the condition for an *else* statement. If the *if* statement's condition is *False* then the block of code indented under the *else* statement is executed.

*Syntax:*

```
if condition1:  
    execute block1  
  
else:  
    execute block2
```

As mentioned, after the *if* statement, a condition (or multiple conditions) is provided, followed by a colon (:). The block of code to be executed under *if* statement, is provided in the next line with proper **indentation**.

The *else* statement is indented to the same level as that of *if* statement, followed directly by a colon (:), no condition provided. Again, the block of code to be executed under the *else* statement, is provided in the next line with proper **indentation**.

A block of code indented with a conditional statement is usually called a *clause*, like *if* clause. Python clauses heavily rely on **indentation**. You've to put a *tab* or space(s) for each indented line in your Python scripts. *PEP 8* conventions recommend to use 4-space indentation, and no tabs as four spaces are a good compromise between small and large indentation but I personally prefer *tab* indentation.

From now on, most of the examples are written in the file editor. The control flow statements must be perfectly indented else your program will crash even if all the statements are written correctly.

*valar\_morghulis.py*

```
greetings = 'Valar Morghulis'

# if (greetings == 'Valar Morghulis')
# or

if greetings == 'Valar Morghulis':
    print ("Valar Dohaeris")

else:
    print ("Nothing")
```

---

Valar Dohaeris

Each line within a block must be indented with the same amount of spaces (or tabs) else the Python interpreter will throw an *IndentationError*.

*valar\_morghulis.py*

```
greetings = 'Valar Morghulis'

if greetings == 'Valar Morghulis':
    print ("Valar Dohaeris")

else:
    print ("Nothing")
```

---

```
File "valar_morghulis.py", line 4
print ("Valar Dohaeris")
^
IndentationError: expected an indented block
```

The same Python script just crashed because of the wrong indentation, even if there is no logical error. It is important to have a good understanding of how indentation works in Python in order to maintain the structure and the order of your code. Let's explore more examples:

*winterfell.py*

```
words = input ("Words: ")
if words == "Winter Is Coming":
    print ("Welcome to Winterfell!")
else:
    print ("I don't know you.")
```

---

```
Words: winter is coming
I don't know you.
```

The interpreter just skipped the *if* clause and executed the *else* block. You may be wondering why this happened, it was supposed to execute the *if* block as the input string matches exactly with the string given in *if* condition. Observe carefully, “Winter Is Coming” is not equal to “winter is coming”. To overcome this case-sensitive behaviour of strings, you can use default string methods.

*winterfell.py*

```
words = input("Words: ")

if words.title() == "Winter Is Coming":
    print("Welcome to Winterfell!")

else:
    print("I don't know you.")
```

---

```
Words: wIntEr Is cOmIng
Welcome to Winterfell!
```

Enclosing condition in parentheses () is a good programming practice as it makes your program more readable; *readability counts*. I personally prefer this convention as other programming languages also follow the same.

*even\_odd.py*

```
num = int (input('Integer: '))
if (num%2 == 0):
    print("Even number")

else:
    print("Odd number")
```

---

```
Integer: 73
Odd Number
```

## 2.2 *elif* statement

Most of the time you have more than 2 condition checks in your program, for that Python provides the *elif* statement (which means else if). You can have as many *elif* statements as you need in your program.

*Syntax:*

```
if condition1:  
    execute block 1  
  
elif condition2:  
    execute block 2  
  
elif condition3:  
    execute block 3  
  
. . .  
else:  
    execute block n
```

You can use nested *if else* clauses instead of *elif* statements but it is recommended to use *elif* statements as quoted by Tim Peters ‘*Flat is better than nested*’.

*even\_odd.py*

```
num = int (input ("Enter an integer: "))  
  
if (num%2 == 0 and num >= 0):  
    print (f"{num} is a positive even number.")  
  
elif (num%2 == 0 and num < 0):  
    print (f"{num} is a negative even number.")  
  
elif (num%2 != 0 and num >= 0):  
    print (f"{num} is a positive odd number.")
```

```
else:  
    print (f"{num} is a negative odd number.")
```

---

```
Enter an integer: 19  
19 is a positive odd number.
```

*even\_odd\_nested.py*

```
num = int (input ("Enter an integer: "))  
  
if (num%2 == 0):  
    if (num >= 0):  
        print (f"{num} is a positive even number.")  
    else:  
        print (f"{num} is a negative even number.")  
  
else:  
    if (num >= 0):  
        print (f"{num} is a positive odd number.")  
    else:  
        print (f"{num} is a negative odd number.")
```

---

```
Enter an integer: -6  
-6 is a negative even number.
```

Just like the *copy* and *pprint* module, Python has one more beautiful library named *math*. According to pydocs (official Python documentation), this module provides access to the mathematical functions defined by the C standard but cannot be used with complex numbers, use *cmath* instead. In the following example I've imported the value of *pi*, from *math* package with upto 15 digit precision. Be patient, *modules* will be covered soon.

---

*pi.py*

```
from math import pi

user_input = int (input ("Digit precision for pi: "))

if (user_input <= 15):
    print (round(pi, user_input))

else:
    print ("Only 15 digit precision is available.")
```

---

```
Digit precision for pi: 6
3.141593
```

## 3. Loops

Looping statements allow you to run a particular block of code over and over again. In Python, and in most of the programming languages, two famous keywords are used for this purpose, **for** (for loop) and **while** (while loop).

### 3.1 **while** statement

Like *if* statements, *while* statements also have a condition (or multiple conditions) followed by a colon (:) and an indented block of code. Unlike *if* statement, the indented block of code is repeated over and over again as long as the condition remains *True*.

*Syntax:*

```
while condition:  
    execute these statements repeatedly
```

*current\_value.py*

```
x = 4  
  
while (x < 9):  
    print(f"Current value of x: {x}")  
    x += 1      #x = x+1  
  
print (f"Final value of x: {x}")
```

---

```
Current value of x: 4  
Current value of x: 5  
Current value of x: 6  
Current value of x: 7  
Current value of x: 8  
Final value of x: 9
```

In the above example, if you accidentally omit to increment the value of variable *x* (*x+=1*), the loop will run forever.

Sometimes the condition within the *while* loop is always *True*, resulting in an infinitely running loop. To exit an infinite loop press *Ctrl+C* which kills the process.

*ned.py*

```
ned = "Lord of Winterfell"
while(ned):
    print("Peace")
```

---

```
Peace
Peace
Peace
^CPeace
Traceback (most recent call last):
File "ned.py", line 4, in <module>
print ("Peace")
KeyboardInterrupt
```

A declared variable always exists, so it is never *False*. *True* value is numerically represented as 1 and *False* as 0.

*fibonacci.py*

```
n = int (input ('Enter the number of terms: '))
first, second = 0, 1      #multiple assignment
count = 0

while(count < abs(n)):
    print(first)
    next_term = first + second
    first, second = second, next_term      #swapping
    count += 1      #increment
```

```
Enter the number of terms: 7
```

```
0  
1  
1  
2  
3  
5  
8
```

### 3.2 *for* statement

A *for* loop acts as an *iterator* that iterates over the items of an *iterable* maintaining the order. Though the definition sounds like a tongue twister, let's see what this *iterable* thing means. *Iterables* include all *sequence* types like *str*, *list*, *tuple*; and some non-sequence types like *dict*, *set*.

*Syntax:*

```
for i in x:  
    execute statements repeatedly
```

where *i* is a variable, you can choose any variable name, representing a single item of iterable *x*.

*loop.py*

```
s = "Loop"  
for alphabet in s:  
    print(alphabet)
```

```
L  
o  
o  
p
```

---

*stark.py*

```
family = ['Eddard', 'Catelyn', 'Robb', 'Sansa', 'Arya', 'Bran',  
'Rickon']  
  
for memb in family:  
    print(f'{memb} Stark')
```

---

```
Eddard Stark  
Catelyn Stark  
Robb Stark  
Sansa Stark  
Arya Stark  
Bran Stark  
Rickon Stark
```

*couples.py*

```
couples = [('Jon', 'Ygritte'), ('Jammie', 'Cerci'), ('Tyrion',  
'Shae'), ('Grey Worm', 'Missandei')]  
  
for i in couples:  
    print(i)
```

---

```
('Jon', 'Ygritte')  
('Jammie', 'Cerci')  
('Tyrion', 'Shae')  
('Grey Worm', 'Missandei')
```

---

*sum.py*

```
list_ = [11, 22, 33, 44, 55]
total = 0

for number in list_:
    total += number
    print (f"Current number: {number}\tCurrent Sum: {total}")

print (f"Final sum: {total}")
```

---

```
Current number: 11      Current Sum: 11
Current number: 22      Current Sum: 33
Current number: 33      Current Sum: 66
Current number: 44      Current Sum: 110
Current number: 55      Current Sum: 165
Final sum: 165
```

*couples.py*

```
couples = [('Jon', 'Ygritte'), ('Jammie', 'Cerci'), ('Tyrion',
'Shae'), ('Grey Worm', 'Missandei')]

for i in couples:
    print (f'{i[0]} loves {i[1]}')
```

---

```
Jon loves Ygritte
Jammie loves Cerci
Tyrion loves Shae
Grey Worm loves Missandei
```

---

*Couples.py* (Tuple unpacking)

```
couples = [('Jon', 'Ygritte'), ('Jammie', 'Cerci'), ('Tyrion',  
'Shae')]  
  
for male, female in couples:  
    print(male, female, sep='\t')
```

---

```
Jon  Ygritte  
Jammie  Cerci  
Tyrion  Shae
```

As discussed earlier, a Python *dictionary* is a *mapping* (not *sequence*) and you can iterate through the *view* objects returned by *dict.items()*, *dict.keys()* and *dict.values()*. By default the *dictionary* iterates over its *keys*.

```
>>> synonyms = {'lawless': 'criminal', 'dignity': 'honour',  
'fetter': 'shackle'}  
>>>  
>>> for word in synonyms:  
...     print(word)  
...  
lawless  
dignity  
fetter  
  
>>>
```

---

*anonymous.py*

```
my_info = {'name': 'Anonymous', 'age': 19, 'height': 180}

print('my_info:', end=' ')
for i in my_info:
    print(i, end=' ')

print('\nmy_info.keys ():', end=' ')
for key in my_info.keys():
    print(key, end=' ')

print('\nmy_info.values ():', end=' ')
for value in my_info.values():
    print(value, end=' ')

print('\nmy_info.items ():', end=' ')
for k,v in my_info.items():
    print(f'{k}: {v}', end=' ')
```

---

```
my_info: name age height
my_info.keys (): name age height
my_info.values (): Anonymous 19 180
my_info.items (): name: Anonymous, age: 19, height: 180
```

If you don't need a variable in the *for* statement, use an underscore (*\_*). It's just a convention followed in Python indicating that the variable is not significant.

```
>>> for _ in range (7):
...     print ('Make the onions cry.')
...
Make the onions cry.
>>>
```

### 3.3 *range* type

It is a built-in *sequence* type object that generates an *immutable* sequence of numbers (arithmetic progressions) and is commonly used for looping in a *for* loop for a specific number of times. The range function, *range()*, accepts three arguments: *start*, *stop*, *step*, which must be integers.

The value of *start* argument is included while *stop* is excluded. *stop* is a required parameter, i.e. it can not have a *None* (null) value. *start* and *step* are optional parameters; *start* defaults to zero and *step* size to one. If *step* size is zero, *ValueError* is thrown by the interpreter.

*Syntax:*

```
for i in range(stop):
    execute some statements

for i in range(start, stop):
    execute some statements

for i in range(start, stop, step):
    execute some statements
```

The *range* function is a *generator*, it just produces (or generates) the values instead of storing the list of numbers in memory.

The advantage of the *range* type over a regular *list* or *tuple* is that a *range* object always takes the same (small) amount of memory, no matter how long the size of the range it represents. If you want a list of numbers upto a certain value, cast the *range* function into *list* constructor.

```
>>> range(3)
range(0, 3)
>>> type (range(99))
<class 'range'>
>>> list(range (7))
[0, 1, 2, 3, 4, 5, 6]
>>> list(range (5,15))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> list(range (1,10,2))
[1, 3, 5, 7, 9]
>>> list(range (10, 1, -1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
>>>
```

Being *sequence* type, *range* objects implement all of the common *sequence* operations except *concatenation* and *replication*.

```
>>> range(1,99)[55]
56
>>> range (12)[2:6]
range(2, 6)
>>>
>>> 3 in range(10)
True
>>> len(range(1,7))
6
>>>
```

```
>>> for i in range(3):
...     print('What is Dead May Never Die')
...
What is Dead May Never Die
What is Dead May Never Die
What is Dead May Never Die
>>>
```

*even\_odd.py*

```
even = odd = []

for num in range(10):
    if (num%2 == 0):
        even.append(num)
    else:
        odd.append(num)

print(f"Even numbers: {even}")
print(f"Odd numbers: {odd}")
```

---

```
Even numbers: [0, 2, 4, 6, 8]
Odd numbers: [1, 3, 5, 7, 9]
```

To iterate over the indices of a *sequence*, you can combine *range()* and *len()* as follows:

```
>>> random_words = ['Hegemon', 'Pokemon', 'Doraemon',
'Cacodemon']
>>> for word in random_words:
...     print(word)
...
```

```
Hegemon
Pokemon
Doraemon
Cacodemon
>>>
>>> for i in range (len(random_words)):
...     print (f'{random_words[i]} is at index {i}')
...
Hegemon is at index 0
Pokemon is at index 1
Doraemon is at index 2
Cacodemon is at index 3
>>>
```

Looping statements, *while* and *for*, can be easily nested i.e., you can have a loop within another loop.

*orphyic.py*

```
word_1 = 'Antidisestablishmentarianism'
word_2 = 'pneumonoultramicroscopicsilicovolcanoconiosis'
collection = [[word_1, len(word_1)], [word_2, len(word_2)]]


for item in collection:
    print(f'Item: {item}')
    for i in item:
        print(i, end="\t")
    print("\n")
```

---

```
Item: ['Antidisestablishmentarianism', 28]
Antidisestablishmentarianism      28
```

```
Item: ['pneumonoultramicroscopicsilicovolcanoconiosis', 45]
pneumonoultramicroscopicsilicovolcanoconiosis      45
```

## 3.4 *break, continue* statement

### 1. *break* statement

The *break* statement, breaks out of the current closest enclosing for or while loop.

*current\_value.py*

```
i = 0
while(True):
    i += 1
    print (f"Current Value: {i}")

    if (i == 4):
        break
```

---

```
Current Value: 1
Current Value: 2
Current Value: 3
Current Value: 4
```

*current\_value.py*

```
for i in range(1, 1000):
    print (f"Current Value: {i}")

    if (i == 4):
        break
```

---

```
Current Value: 1
Current Value: 2
Current Value: 3
Current Value: 4
```

---

## 2. *continue* statement

The *continue* statement, skips the current iteration and continues with the next iteration of the loop.

*no\_name.py*

```
num = 0

while (num < 6):
    num += 1
    if (num == 3):
        continue
    print (num)
```

---

```
1
2
4
5
6
```

*season\_1.py*

```
season_1 = ['The Kingsroad', 'Lord Snow', 'The Pointy End',
'Baelor']

for episode in season_1:
    if (episode == 'Lord Snow'):
        continue
    print(episode)
```

---

```
The Kingsroad
The Pointy End
Baelor
```

### 3.5 *else* clause

The looping statements may have an optional *else* clause. If the *break* statement breaks a loop, whether *for* or *while*, the *else* clause is skipped (not executed). The coding block under *else* clause is executed if and only if:

- *for* statement: the loop terminates through exhaustion of the list.
- *while* statement: the condition becomes false.

*no\_name.py*

```
num = 1
while (num < 3):
    print (num)
    num += 1
else:
    print ("Display this block, if num >= 3")
```

---

```
1
2
Display this block, if num >= 3
```

*no\_name.py*

```
i = 2
while (True):
    if (i == 3):
        break
    print(i)
    i += 1
else:
    print ("Execute this, if condition is False.")
```

---

```
2
```

---

*season\_2.py*

```
season_2 = ['The North Remembers', 'Blackwater', 'Valar
Morghulis']

for i in range( len(season_2) ):
    print(season_2[i])

else:
    print("Whole list is scanned.")
```

---

```
The North Remembers
Blackwater
Valar Morghulis
Whole list is scanned.
```

*season\_2.py*

```
season_2 = ['The North Remembers', 'Blackwater', 'Valar
Morghulis']

for episode in season_2:
    if (episode == 'Blackwater'):
        break
    print (episode)

else:
    print ("Whole list is scanned.")
```

---

```
The North Remembers
```

## 3.6 `zip()` and `enumerate()`

### 1. `zip()` function

`zip()` is a *generator* function that generates a list of *tuples* by zipping up two lists together.

```
>>> name = ['Bran', 'Tyrion', 'Daenerys']
>>> house = ['Stark', 'Lannister', 'Targaryen']
>>> zip(name, house)
<zip object at 0x7f21fce84c8>      # memory address
>>>
>>> list(zip(name, house))
[('Bran', 'Stark'), ('Tyrion', 'Lannister'), ('Daenerys',
'Targaryen')]
>>> dict(zip(['akh', 'zi', 'trai'], [1, 2, 3]))
{'akh': 1, 'zi': 2, 'trai': 3}
>>>
```

With `zip()` you can iterate over two or more sequences at the same time.

```
>>> name = ['Bran', 'Tyrion', 'Daenerys']
>>> title = ['The 3-eyed Raven', 'The Imp', 'Khaleesi']
>>> for person in zip(name, title):
...     print(person)
...
('Bran', 'The 3-eyed Raven')
('Tyrion', 'The Imp')
('Daenerys', 'Khaleesi')
>>> for n,t in zip(name, title):
...     print(f'{n}: {t}')
...
Bran: The 3-eyed Raven
Tyrion: The Imp
Daenerys: Khaleesi
>>>
```

## 2. *enumerate(iterable, start)*

When looping through a *sequence*, the count (which defaults to zero) and the corresponding values can be retrieved at the same time using Python's built-in *enumerate()* function.

This method takes two parameters: *iterable* and *start* (optional) and returns a *tuple* containing the item count and with its corresponding value. The default value of *start* is 0 i.e. the items are counted from 0.

```
>>> # enumerate(iterable, start)
>>>
>>> north = ['Winterfell', 'Deepwood Motte', 'Karhold', 'The
Dreadfort']
>>>
>>> for i, castle in enumerate(north):
...     print(f'{i}. {castle}')
...
0. Winterfell
1. Deepwood Motte
2. Karhold
3. The Dreadfort
>>>
>>> for count,castle in enumerate(north, 1):
...     print(f'{count}. {castle}')
...
1. Winterfell
2. Deepwood Motte
3. Karhold
4. The Dreadfort
>>>
```

## 3.7 Comprehension

Python provides some advanced methods for creating objects like *lists*, in a concise way.

### 1. List Comprehension

It's a compact way to process all (or some parts) of the elements in a sequence, returning a list of these elements.

```
>>> # Normal way
>>> name = 'Westeros'
>>>
>>> alphabets = []
>>> for i in name:
...     alphabets.append(i)
...
>>> print(alphabets)
['W', 'e', 's', 't', 'e', 'r', 'o', 's']
>>>
>>> # List comprehension
>>> alphabets_2 = [i for i in "Westeros"]
>>>
>>> print(alphabets_2)
['W', 'e', 's', 't', 'e', 'r', 'o', 's']
>>>
```

You can also use *list()* constructor to get the list of alphabets from a given string but list comprehensions are more flexible and provide better control.

A list comprehension consists of square brackets [] containing an expression followed by a *for* statement and then any number of *for* or *if* clauses.

As they say “Talk is cheap, show me your code”, let’s cover all the possible cases you may think of, while using list comprehension, with coding examples.

```

>>> integers = [x for x in range(10)]
>>> integers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
>>> squares = [n**2 for n in range(10)]
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>>
>>> temp_C = [10, 27, -40, 3]
>>> temp_F = [(9/5*i)+32 for i in temp_C]
>>> temp_F
[50.0, 80.6, -40.0, 37.4]
>>>
>>> absolute = [abs(x) for x in [11, -17, -13, 6]]
>>> absolute
[11, 17, 13, 6]
>>>
>>> [(x, x**2) for x in range(1, 5)]
[(1, 1), (2, 4), (3, 9), (4, 16)]
>>>

```

List comprehensions having *if* statements.

```

>>> even = [num for num in range(10) if (num%2==0) ]
>>> even
[0, 2, 4, 6, 8]
>>>
>>> coordinates = [(x,y) for x in range(1,4) for y in [1,2,3]
if x!=y]
>>>
>>> coordinates
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>>

```

Equivalent program to create the same list using normal *for* loops.

```
coordinates = []

for x in range(1, 4):
    for y in [1, 2, 3]:
        if x != y:
            coordinates.append ((x,y))

print(coordinates)
```

The only case where *if* statement comes before the *for* loop is when *if* statement is followed by an *else* clause.

```
>>> x = [num if num%2==0 else num**2 for num in range (10)]
>>> x
[0, 1, 2, 9, 4, 25, 6, 49, 8, 81]
>>>
```

## 2. Dictionary Comprehension

You can create dictionaries the same way, you created lists with list comprehensions. However, dictionary comprehensions are not so common.

```
>>> squares = {x:x**2 for x in range(1,5)}
>>> squares
{1: 1, 2: 4, 3: 9, 4: 16}
>>>
>>> a = ['akh', 'zi', 'trai']
>>> b = [1, 2, 3]
>>> {k:v for k,v in zip(a, b)}
{'akh': 1, 'zi': 2, 'trai': 3}
>>>
```

---

### 3. Set Comprehension

Like lists and dictionaries, *set* objects can be created with *set* comprehensions.

```
>>> unique_letters = {x for x in 'Essos'}
>>>
>>> unique_letters
{'s', 'E', 'o'}
>>>
>>> place = 'Plains of the Jogos Nhai'
>>>
>>> vowels = 'a e i o u'.split()
>>>
>>> vowel_free_place = {x for x in place if x not in vowels}
>>>
>>> vowel_free_place
{'h', 'l', 's', 'n', 'J', ' ', 'f', 'g', 'P', 't', 'N'}
```



03. *We do not Repeat*

## 1. Functions

Functions are the important piece of the puzzle, which follows a famous principle of coding- ***DRY*** (Don't Repeat Yourself). A function is simply a block of code which executes only if it is called. A function can be called multiple times, provided that the function is defined within your script. You've already seen some of Python's built-in functions. Now the question is, how can you create your own custom functions within Python? With a *def* statement.

The keyword *def* defines a function, followed by the function name and the parenthesized list of parameters (or arguments). An argument is the value passed to a function. Body of a function is the block of code indented within the *def* statement. The naming convention for functions as per PEP 8 is to use *snake\_case*.

*Syntax:*

```
def function_name(p1, p2, ...):
    function body
```

*test.py*

```
def who_are_you():
    print("I'm a function")
```

The output of the above script is blank, as function is only defined, not called. To execute a function's body, it must be called at least once.

*greet\_me.py*

```
def greet_me(name):
    print(f'Welcome {name}.')
name = "Guido van Rossum"
greet_me(name)
```

---

Welcome Guido van Rossum.

---

*even\_odd.py*

```
def even_odd(num):
    if (num%2 == 0):
        print(f"{num} is even")
    else:
        print(f"{num} is odd")

even_odd(4)
even_odd(73)
```

---

4 is even

73 is odd

In Python, there are two types of arguments that can be passed to a function.

- **Keyword Arguments**

The arguments which are preceded by an identifier in a function call, like *some\_func(age=99)*, or passed as values in a dictionary preceded by \*\*. I guess the definition is confusing, let's code an example:

```
>>> round(number=1.2154785, ndigits=3)
1.215
>>> round (**{'number': 1.2154785, 'ndigits': 3})
1.215
```

- **Positional Arguments**

The positional arguments can be passed directly without any identifier, or as elements of an iterable preceded by \*.

```
>>> round(2.718281828, 4)
2.7183
>>> round(*(2.718281828, 4))
2.7183
```

## 1.1 *return* statement

The *return* statement returns a value (or an expression) from a function. Function without a return statement returns *None*. Recall *None* type, it represents the absence of an object.

*cube\_root.py*

```
def cube_root(num):
    return num**(1/3)

x = cube_root(27)
print(x)
```

---

3.0

*goblin.py*

```
def goblin():
    print("I won't return anything.")

x = goblin()
print(f"goblin() returns {x}")
#goblin is not called in the above print function
```

---

I won't return anything.  
goblin() returns None

A *return* statement breaks out all the loops within that function and returns its value to the main program i.e. the execution of a function is shut down as it hits a *return* statement.

---

*is\_prime.py*

```
def is_prime(num):
    for i in range(2, num):
        if (num%i == 0):
            return False
    else:
        return True

print(is_prime(17))
print(is_prime(9))
```

---

True

False

In a function call, arguments passed must match the order of arguments defined in function definition and no argument should receive a value more than once.

*info.py*

```
def display(name, height, age):
    print(f"{name}, who is {height} cm tall, is {age} years old.")

display('Tormund', 185, 29)
display(160, 'Ryan', 15)
display(age=19, height=175, name='Mark')
```

---

Tormund, who is 185 cm tall, is 29 years old.

160, who is Ryan cm tall, is 15 years old.

Mark, who is 175 cm tall, is 19 years old.

---

*swapping.py*

```
def swap(x, y):
    temp = x
    x = y
    y = temp
    return(x, y)

a = 11
b = 22

print(f'Values before swapping \na: {a}, b:{b}\n')
a, b = swap(a, b)
print(f'Values after swapping \na: {a}, b:{b}'')
```

---

Values before swapping

a: 11, b:22

Values after swapping

a: 22, b:11

You can swap the values of two variables without using a temporary variable by implementing the following logic.

*swapping.py*

```
def swap (x, y):
    x = x+y
    y = x-y
    x = x-y
    return(x, y)
```

---

*sum\_of\_first\_n\_natural\_numbers.py*

```
def cumulative_sum(n):
    if (n == 0):
        return 0
    return (n * (n+1)) / 2

print(cumulative_sum(10))
print(cumulative_sum(27))
print(cumulative_sum(100))
```

---

```
55.0
378.0
5050.0
```

*max\_element.py*

```
def max_element(arr):
    n = len(arr)
    ele = arr[0]

    for i in range(1, n-1):
        if (arr[i] > ele):
            ele = arr[i]
    return ele

my_max = max_element([33, 11, 99, 55, 77])
print(my_max)
```

---

```
99
```

## 1.2 Documentation Strings

A docstring is a *triple quoted*, optional string that appears as the first expression in the function's body. It is a kind of multiline comment describing the purpose of the function. Docstrings are ignored by the interpreter but it is a good programming practice to include docstrings in your functions.

The following example is a famous sorting algorithm- bubble sort. It is the simplest sorting algorithm which repeatedly swaps the wrongly placed adjacent items of the given list.

*bubblesort.py*

```
def bubble_sort (arr):
    """
    This Function sorts a numeric list as it
    repeatedly steps through the list to be sorted;
    compares each pair of adjacent items and
    swaps them if they are in the wrong order
    """

    n = len (arr)
    for pass_ in range(n-1):
        for j in range((n-pass_-)-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

sorted_list = bubble_sort([33, 11, 99, 55, 77])
print(sorted_list)
```

---

[11, 33, 55, 77, 99]

## 1.3 *map()* and *filter()*

### 1. *map()* function

The built-in *map()* function maps a function to an *iterable* i.e., it allows you to apply a function to an *iterable* by iterating through every item. *map()* function returns a *generator* like that of *range()*.

*Syntax:*

```
map(function, iterable)
```

It is important to note that the function within *map()* is **not called**, it is passed as an argument (without parentheses).

*greet\_me.py*

```
def greet_me(name):
    return f"Hello! {name} Lannister"

name_list = ['Tywin', 'Tyrion', 'Jaime', 'Cersei']
print(map(greet_me, name_list))
print(list(map(greet_me, name_list)))

<map object at 0x7f33a740f438>
['Hello! Tywin Lannister', 'Hello! Tyrion Lannister', 'Hello!
Jaime Lannister', 'Hello! Cersei Lannister']
```

### 2. *filter()* function

If a function filters the input, based on some conditions and returns either *True* or *False*, use the *filter()* instead of *map()*. The *filter* function works the same as *map*, the only difference is, *filter* returns only those values of an *iterable*, for which the function is *True*. Just as *map()*, this function is also a *generator* function.

*Syntax:*

```
filter(function, iterable)
```

---

*is\_even.py*

```
def is_even(num):
    if (num%2 == 0):
        return True
# def is_even(num):
#     return (num%2 == 0)

num_list = [3,47,6,52,19]
print(filter (is_even, num_list))
print(list (filter (is_even, num_list)))
```

---

```
<filter object at 0x7f11e6474390>
[6, 52]
```

## 1.4 *lambda* expression

Small functions can be created anonymously using the *lambda* keyword, without defining a function properly. Lambda expressions can have multiple arguments but syntactically it is restricted to a *single expression*. A lambda function has no *def* keyword, no *return* value(s); just arguments and an expression.

**Note:** Every function can not be converted directly into lambda expression.

*Syntax:*

```
lambda arg1, arg2, ... : expression
```

```
>>> lambda x: x**(1/3)
<function <lambda> at 0x7ff0bb793e18>
>>>
>>> cube_root = lambda x: x**(1/3)
>>> cube_root (27)
3.0
>>>
```

The above example's equivalent function would be something like this:

```
def cube_root (x):
    return x**(1/3)
```

Lambda expressions are commonly used within *map()* and *filter()* functions.

```
>>> nums = [1, 4, 9, 16, 25]
>>>
>>> list (map (lambda num: num**(1/2), nums))
[1.0, 2.0, 3.0, 4.0, 5.0]
>>>
>>> list (filter (lambda num: num%2 == 0, nums))
[4, 16]
>>>
```

Let's now explore a more complex lambda expression.

```
>>> even_squares = list(map(lambda x: x**2, filter(lambda num:
num%2==0, range(10))))
>>>
>>> even_squares
[0, 4, 16, 36, 64]
>>>
```

The equivalent function of the above example would be:

```
def even_squares():
    list_ = []
    for num in range(10):
        if (num%2 == 0):
            list_.append (num**2)
    return list_
```

## 1.5 Default argument values

While defining a function, you can set its arguments to have default values. Now, if you call that function, without passing values of defined arguments, the function automatically assigns the default values to its argument list. However, if a value is passed, then the default value is overwritten with the value passed during the function call.

*choice.py*

```
def choice(name, language='Python'):
    print(f"{name} likes {language}.")  
  
choice ('Tommen')
choice ('Joffrey', 'Java')
```

---

Tommen likes Python.  
Joffrey likes Java.

**Note:** The default value is evaluated only once by the Python interpreter. It can be observed when the default argument is set to a mutable object such as a *list*, *dictionary*.

*word\_to\_letter.py*

```
def word_to_letter (word, alphabets=[]):
    for i in word:
        alphabets.append(i)
    print (alphabets)  
  
word_to_letter('Palestine')
word_to_letter('Kashmir')
```

---

['P', 'a', 'l', 'e', 's', 't', 'i', 'n', 'e']
['P', 'a', 'l', 'e', 's', 't', 'i', 'n', 'e', 'K', 'a', 's', 'h', 'm', 'i', 'r']

You can restrict this default behaviour between subsequent function calls as:

*word\_to\_letter.py*

```
def word_to_letter(word, alphabets=None):
    #if (not alphabets):
    if (alphabets == None):
        alphabets = []

    for i in word:
        alphabets.append(i)
    print(alphabets)

word_to_letter('Palestine')
word_to_letter('Kashmir')
```

---

```
['P', 'a', 'l', 'e', 's', 't', 'i', 'n', 'e']
['K', 'a', 's', 'h', 'm', 'i', 'r']
```

## 1.6 \*args and \*\*kwargs

A function can have multiple arguments, or no argument at all. If you're not sure about the number of positional arguments while defining a function, you can simply pass `*args`, to have any arbitrary number of arguments, in the form of a tuple. It's not necessary to name it as `args`, you can choose any name for it like `*haput`. Since it's a Pythonic convention to name it as `*args`, you should stick to that.

*is\_even.py*

```
def is_even(*args):
    even_list = [num for num in args if num%2==0]
    return even_list
print (is_even(11,22,33,44,55,66))
```

---

```
[22, 44, 66]
```

---

*third\_square.py*

```

def squared_third_item(*args):
    if (len (args) > 2):
        return args[2]**2
    else:
        return 'Insufficient length.'

a = squared_third_item (7, 13, 2, 8)
b = squared_third_item (11, 22, 33, 44, 55)

print(a, b, sep='\t')
print(squared_third_item( ))

```

---

```

4      1089
Insufficient length.

```

`**kwargs` (Keyword arguments) is frequently used to handle the cases where you have an arbitrary number of named arguments. Instead of tuples, it receives a dictionary containing all keyword arguments.

*info.py*

```

def student_info(**kwargs):
    print(f"Name: {kwargs['name']}, Enroll: {kwargs['enroll']}")

student_info(name='Jose', age=19, enroll=213)
student_info(name='Mark', enroll=240, hair_color='black')

```

---

```

Name: Jose, Enroll: 213
Name: Mark, Enroll: 240

```

---

*info.py*

```
def student_info(**kwargs):
    print(f"Name: {kwargs['name']}, Enroll: {kwargs['enroll']}")

student_info(**{'name': 'Jose', 'age': 19, 'enroll': 213})

student_info(**{'name': 'Mark', 'enroll': 240, 'hair_color':
'black'})
```

---

```
Name: Jose, Enroll: 213
Name: Mark, Enroll: 240
```

*info.py*

```
def student_info(**kwargs):
    print(f"Name: {kwargs['name']}, Enroll: {kwargs['enroll']}")

student_info('Joshua', age=23, enroll=619)
```

---

```
Traceback (most recent call last):
  File "info.py", line 4, in <module>
    student_info ('Joshua', age=23, enroll=619)
TypeError: student_info() takes 0 positional arguments but 1 was
given
```

The *TypeError* is thrown by the interpreter because the function is expecting only keyword arguments but you're providing a positional argument as well.

## Combining `*args` and `**kwargs`

In a function call, *positional* arguments must appear first which are then followed by *keyword* arguments.

*sales.py*

```
def total_sales(*args, **kwargs):
    total = sum(args)
    print(f"{kwargs['day']} sales: {total}")

total_sales (73,15, day='Wednesday')
total_sales (14,65,33, day='Friday')
```

---

```
Wednesday sales: 88
Friday sales: 112
```

*sales.py*

```
def total_sales (brand_name, *args, **kwargs):
    total = sum(args)
    print(f"{kwargs['day']} sales: {total} of {brand_name}")

total_sales ('Puma', 73, 15, day='Wednesday')
total_sales ('Nike', 14, 65, 33, day='Friday')
```

---

```
Wednesday sales: 88 of Puma
Friday sales: 112 of Nike
```

```
sales.py

def total_sales(brand_name, *args, **kwargs):
    total = sum(args)
    print(f"{kwargs['day']} sales: {total} of {brand_name}")

total_sales (day='Wednesday', 'Puma', 73,15)
total_sales (day='Friday', 'Nike', 14,65,33)
```

---

```
File "sales.py", line 5
    total_sales (day='Wednesday', 'Puma', 73,15)
               ^
SyntaxError: positional argument follows keyword argument
```

## 1.7 *pass* statement

The *pass* keyword is usually used when a statement is required syntactically but the program requires no action. Sometimes you just want a placeholder function whose body is initially kept empty and is defined later in your code, *pass* statement is made for that. It can also be used with loops and even with classes (yet to cover).

```
>>> def dummy():
...     pass
...
>>> for x in range(10):
...     pass
...
>>> class demo:
...     pass
...
>>>
```

---

*placeholder.py*

```
def placeholder_func():

# print is not within function indentation
print("I'll define placeholder function later.")
```

---

```
File "placeholder.py", line 3
    print("I'll define placeholder function later.")
    ^
IndentationError: expected an indented block
```

*placeholder.py*

```
def placeholder_func():
    pass

print("I'll define placeholder function later.")
```

---

I'll define placeholder function later.

## 1.8 Scope of a variable

The scope of a variable determines its visibility in the various parts of your program i.e. variables defined within a certain block can only be used in specific parts of that program, depending on their scope.

### 1. Local Scope

Variables and function arguments that are defined inside a function are said to be in the local scope of that function. Local scope extends from its point of declaration to the end of that particular block. Variables in local scope are called local variables.

---

*addition.py*

```
def addition(arg1, arg2):
    result = arg1 + arg2
    return result

print(result)
```

---

```
Traceback (most recent call last):
  File "addition .py", line 5, in <module>
    print (result)
NameError: name 'result' is not defined
```

In the above example, variable *result* and parameters *arg1* and *arg2* are in local scope to the *addition* function and can only be accessed within that function. *NameError* is thrown as I tried to access a local variable outside the function's scope. *locals()* is a built-in function that returns a dictionary containing the local variables of current scope.

## 2. Global Scope

A variable which is defined outside the functions is called a *global* variable. The scope of a *global* variable extends from the point of declaration to the end of that program.

*no\_name.py*

```
def func():
    print(f"{num} is also accessible to local scope.")
num = 17
print(f"{num} is a global variable.")
func()
```

---

```
17 is a global variable.
17 is also accessible to local scope.
```

The *global* variables are available in *zero-level* indentation and can be accessed within all functions. `globals()` is another built-in function that returns the dictionary containing the global variables of current scope.

## LEGB Rule (*local enclosing global built-in*)

A variable can either be *local* (or nonlocal) or *global* but can't be both at the same time. So, what if a *local* and a *global* variable share the same name, how can you access these variables?

There are two important things that you must take into consideration if you have same variable names in different scopes:

- a. A function defined inside another function (nested function) can easily refer to the variables of the outer function.
- b. Local variables can read and write to the innermost scope. Likewise, global variables can read and write to the global scope.

Python follows a set of rules to determine the accessibility of variables that are in different scopes. At any point of time during execution, there are at least three nested scopes which are directly accessible:

1. **Local** scope (the innermost scope), which is searched first, contains the *local* variables.
2. Scope of any enclosing function, that is searched starting with the nearest enclosing scope, which contains the **non-local** variables.
3. **Global** scope (the next-to-last scope) which contains the *global* variables.
4. The scope which is searched last is **built-in** scope (the top level scope), which contains built-in names like `len`, `range`, `list`. You should never overwrite the values of these reserved keywords.

The scope of variables is a very important concept of programming, you should refer to other reading material on this topic as well. I'll try to clear the basics of this concept with the following coding examples.

---

*nested\_func.py*

```
def enclosing_func():
    name = 'Enclosing'

    def nested_func():
        name = 'Local'
        print(name)

    nested_func()

name = 'Global'
enclosing_func()
```

---

Local

*nested\_func.py*

```
def enclosing_func():
    name = 'Enclosing'

    def nested_func():
        print(name)

    nested_func()

name = 'Global'
enclosing_func()
```

---

Enclosing

---

*nested\_func.py*

```
def enclosing_func():

    def nested_func():
        print(name)

    nested_func()

name = 'Global'
enclosing_func()
```

---

```
Global
```

*nested\_function.py*

```
def house_martell():

    words = 'Unbowed, Unbent, Unbroken'

    def prince_of_dorne():
        prince = 'Doran Martell'
        print(f'House: Martell \nWords: {words} \nPrince: {prince}')

    prince_of_dorne()

house_martell()
```

---

```
House: Martell
Words: Unbowed, Unbent, Unbroken
Prince: Doran Martell
```

```
nested_function.py

def house_martell():
    words = 'Unbowed, Unbent, Unbroken'

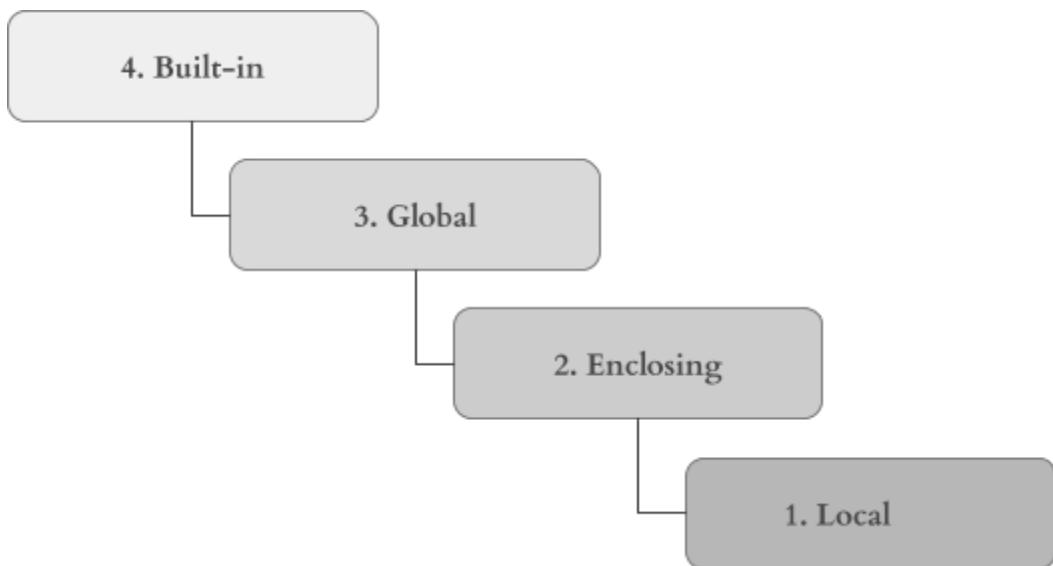
def prince_of_dorne():
    prince = 'Doran Martell'
    print (f'House: Martell \nWords: {words} \nPrince:
{prince}')

prince_of_dorne()

prince_of_dorne()

-----
Traceback (most recent call last):
  File "nested_function.py", line 10, in <module>
    prince_of_dorne()
NameError: name 'prince_of_dorne' is not defined
```

The following image summarizes the LEGB concept:



---

## global statement

If you try to modify the value of a global variable directly, within the local scope, the program may give you incorrect results. When you assign a new value to the global variable within local scope, a local variable gets created, having the same name as that of global variable and the original value of the global variable is retained. This problem can be easily solved with the *global* statement.

Syntax:

```
global variable_name
```

*scope\_test.py*

```
def scope_test():
    x = 32
    print(f"Local scope: {x}")
x = 7
scope_test()
print(f"Global scope: {x}")
```

---

```
Local scope: 32
Global scope: 7
```

*scope\_test.py*

```
def scope_test():
    global x
    x = 32
    print(f"Local scope: {x}")
x = 7
scope_test()
print(f"Global scope: {x}")
```

---

```
Local scope: 32
Global scope: 32
```

## nonlocal statement

In simple words, a *nonlocal* variable is a variable which is neither *local* nor *global*. As you can access the *global* variables with *global* keyword likewise with the *nonlocal* statement you can reference a nonlocal variable in an enclosing scope.

*pydocs\_example.py*

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

---

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

## 1.9 Recursion

Recursion is a general mathematical and programming concept that occurs when something is defined in terms of itself. In programming, it occurs when a function is called within the function itself. Apparently recursion defines an infinite loop but it is done in such a way that no infinite loop can occur, we call it a *base case*. A base case simply sets some condition that terminates the subsequent function calls. Without a base case the function recursively makes infinite calls to itself.

*infinite\_recursion.py*

```
def infinite_recursion():
    print('Walk of Punishment')
    return infinite_recursion()

infinite_recursion()
```

---

```
Walk of Punishment
Walk of Punishment
Walk of Punishment
Traceback (most recent call last):
  File "infinite_recursion.py", line 4, in <module>
    infinite_recursion ()
  File "infinite_recursion.py", line 3, in infinite_recursion
    return infinite_recursion ()
  File "infinite_recursion.py", line 3, in infinite_recursion
    return infinite_recursion ()
  File "infinite_recursion.py", line 3, in infinite_recursion
    return infinite_recursion ()
[Previous line repeated 992 more times]
  File "infinite_recursion.py", line 2, in infinite_recursion
    print ('Walk of Punishment')
RecursionError: maximum recursion depth exceeded while calling a
Python object
```

---

*sum\_of\_digits.py*

```
def sum_of_digits(n):
    #base case
    if (n == 0):
        return 0

    last_digit = n%10
    n //= 10
    return (last_digit + sum_of_digits (n))

total = sum_of_digits(12457)
print(f'Total: {total}')
```

---

Total: 19

*fibonacci.py*

```
def fib(n):
    #base case
    if (n == 1 or n == 0):
        return n

    return ( fib (n-1) + fib (n-2) )

print(fib (7))
print(fib (9))
```

---

13  
34

## 1.10 Decorators

A decorator is a function which modifies the functionality of another function. It basically returns a function that is defined (nested) in it. A decorator is usually applied as a function transformation, using the *At* symbol ‘@’. The concept of decorators is an advanced topic so don’t worry if you don’t understand it completely. At this level, you won’t use decorators that often in your scripts.

*decorator.py*

```
def decorator_func(any_normal_func):
    def modified_func():
        print('Modification above the function.')
        any_normal_func()
        print('Modification below the function.')
    return modified_func

def normal_func_1():
    print('Please do not decorate me\n')

@decorator_func
def normal_func_2():
    print('Decorate me.')

normal_func_1()
normal_func_2()
```

---

Please do not decorate me

Modification above the function.  
Decorate me.  
Modification below the function.

The above example may seem absolute nonsense to you but I could not think of any example simpler than that. Let's understand this example function wise. The first function is a decorator function that accepts a function as its argument, wraps a modified function and then returns it. The modified function encloses the passed function and its job is to add the functionality to the passed function. Then the normal functions are defined, which execute normally. The defined decorator can then be used to modify other normal functions by adding an *At (@)* symbol above the definition of that normal function.

Although you can create your own decorators, as shown in the above example but mostly you'll be using readymade decorators for your advanced scripts like if you're creating a login system in *Django* (Python's web framework), you'll use *@login\_required* decorator which is build into *Django* that automatically checks whether the user is logged in or not.

```
@login_required  
def some_function ():  
    do something
```

## 1.11 Generators and Iterators

A generator is a function which returns a *generator iterator*. It looks like a normal function except that it contains *yield* expressions for producing a series of values, usable in a *for* loop. Now the question is when to use generator functions instead of normal functions? If you're dealing with a large range of values and by large I mean millions of data points within a single data structure, use *generator* functions as these are generated on the fly, hence memory efficient. Instead of allocating the memory to all items, a generator function only stores the result of the previous item to generate the next one.

*generator.py*

```
def generator_function():  
    for i in range(6):  
        yield i**32
```

```
x = generator_function()
print(x)
print(".....")
for element in x:
    print(element)
```

```
<generator object generator_function at 0x7fb7105b6740>
.....
0
32
64
96
128
160
```

Items of a generator function can also be retrieved one at a time with the built-in *next()* function. This function simply allows you to access the next item of the sequence returned by the generator function.

```
>>> def generator_function():
...     for i in range(3):
...         yield i*32
...
>>> x = generator_function()
>>> next(x)
0
>>> next(x)
32
>>> next(x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

A *StopIteration* exception is thrown as all the items are already retrieved from the sequence and you try to access an item which is beyond the limit.

**Note:** *Strings, lists, tuples, dictionaries* and *range* objects are *iterables* i.e. you can iterate through them with a *for* loop; but these are not *iterators*. If you try to apply the *next* function to them, the interpreter will throw a *TypeError*.

```
>>> x = [11,22,33]
>>> next(x)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'list' object is not an iterator
>>>
```

However, you can use *next()* function with the above-mentioned iterables if you convert these into *iterator* objects using Python's built-in *iter()* function.

```
>>> a = 'The Rains of Castamere'
>>> b = iter(a)
>>> b
<str_iterator object at 0x7f82e3b77940>
>>> next(b)
'T'
>>> for i in range(4):
...     print(next(b))
...
h
e

R
>>>
```

Phew! That's a lot to take in. These *iterators*, *iterables* and *generators* may seem confusing at first but as you keep implementing these, you'll have a better understanding.

## 2. Modules

According to an article published in Wired, an engineer at Google estimated that the software which runs all of Google's Internet services, from Google Search to Gmail to Google Maps, spans some *2 billion* lines of code. By comparison, Microsoft's Windows operating system, one of the most complex software tools ever built for a single computer, is likely in the realm of *50 million* lines of code. Imagine if all these lines of code were stored as a single file, it could have been impossible for these corporations to manage such huge code within a single file.

Now coming back to the individual level. Your projects may contain just a few hundred lines of code, or even thousands. If you're a stubborn kind of programmer, you may write the whole code within a single script but eventually at a certain point of time, you'll end up with the modular approach.

Wikipedia says, Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

You can achieve this modularity, using the concept of *modules*. A module is just a Python file (script) containing executable statements as well as function definitions. Modules can import other modules, using the *import* statement. It is customary but not required to place all the import statements at the beginning of a module. Imported modules are executed only the first time, as soon as the module name is encountered in an *import* statement.

Python comes with a library of standard modules, some modules are built into the interpreter like *copy*, *pprint*, *math* etc. There is a built-in function, *dir()* that returns a list of attributes and functions that are defined in that module.

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

*NameError* is thrown by the interpreter because the *math* module is not imported. First, you need to import a module, only then you can access various methods and attributes defined within that module. The concept of methods and attributes is discussed in detail in the next chapter of this book. For now just remember the following statement

*function:method :: variables:attributes.*

```
>>> import math
>>>
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh',
'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees', 'dist',
'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt',
'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf',
'nan', 'perm', 'pi', 'pow', 'prod', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
>>>
>>> math.pi
3.141592653589793
>>>
>>> math.log10(100)
2.0
>>>
>>> math.log2(512)
9.0
>>>
>>> math.ceil(3.14)
4
>>>
>>> math.floor(3.14)
3
>>>
```

Let's create our own custom module that can be imported to other Python scripts, just like the *math* module.

*module.py*

```
def print_something():
    print("I'm from module.py")

def string_reversal(s):
    return s[::-1]
```

Now create another Python script in the same directory as *main.py*. Start the new script by importing *module.py* within it. Use dot (.) operator to access the functions that are defined in *module.py*.

*main.py*

```
import module

module.print_something()
rev_string = module.string_reversal("Hello")
print(rev_string)
```

---

```
I'm from module.py
olleH
```

When you import a Python module, along with the functions and classes (yet to cover), the code in the main body of module is also imported. This behaviour of modules can create problems as the unnecessary code within the body gets executed automatically. Let's write some code within the body of *module.py*, without modifying *main.py*.

---

*module.py*

```
def string_reversal(s):
    return s[::-1]

print("Starting body of module.py")
x = string_reversal("Python")
print(x)
print("Ending body of module.py")
```

*main.py*

```
import module

rev_string = module.string_reversal ("Jon Snow")
print(rev_string)
```

---

```
Starting body of module.py
nohtyP
Ending body of module.py
wonS noJ
```

The whole code of the *module.py*, including body, got executed as soon as the *main.py* was executed. You definitely don't want this to happen with your programs. Luckily Python has a built-in variable `__name__`, which if set to string "`__main__`", restricts the code after it from execution, if the script is imported as a module.

However, if the module is executed directly, then the above mentioned condition will have no effect and the program executes normally. With this flexibility, you can make your files usable as scripts as well as importable modules.

---

*module.py*

```
def print_something():
    print("I'm from module.py")

def string_reversal(s):
    return s[::-1]

if __name__ == "__main__":
    print("Hehe!")
    x = string_reversal("Python")
    print(x)
```

*main.py*

```
import module

module.print_something()
rev_string = module.string_reversal("Nazi")
print(rev_string)
```

---

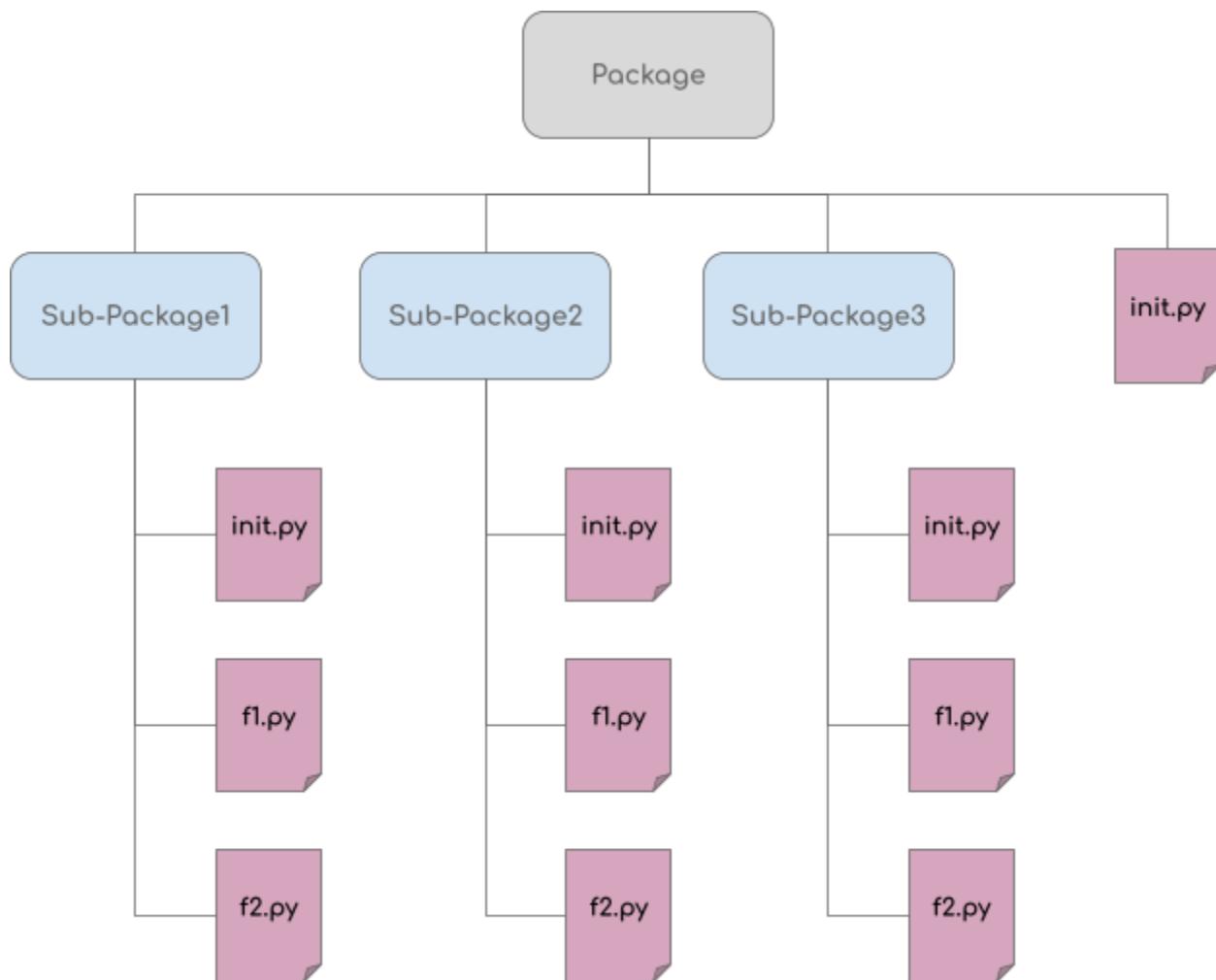
```
I'm from module.py
izaN
```

Behind the scenes, Python interpreter handles this importing of modules efficiently. For this reason, each module is only imported once per interpreter session. Therefore, if you modify your modules, you must restart the interpreter, else you'll get the same results even after modifications.

## 2.1 Packages

A package is simply a collection of modules and subpackages (package within package). You can say that a package is a folder (or a directory) which contains multiple modules. There is a special file which is named as `__init__.py` that goes into each package indicating that the current directory is not a common directory but a Python package.

In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package. The package hierarchy is shown in the following structure:



You can import individual modules from the packages (or subpackages) using `from` statement in conjunction with `import` statement.

```
from package import item
```

The `item` can be either a subpackage or a function, class or variable defined in that particular package. The `import` statement first searches the item in the package; if the item is not found in the package, it assumes that it is a module and attempts to load it, if it fails to find the module, an `ImportError` is raised.

```
>>> from math import nothing
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'nothing' from 'math' (unknown
location)
>>>
>>> import unknown
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'unknown'
>>>
```

You can use asterisk (\*) to import everything from a package (or module) and if you import everything from a module, you don't need to use a dot (.) operator.

```
>>> from math import *
>>> pi
3.141592653589793
>>>
>>> floor(3.124)
3
>>>
```

In general, the practice of importing everything from a module (or package) often makes your code poorly readable. However, for the sake of typing, you can use it in interactive sessions. If the name of a function, to be imported, is too long or conflicts with an existing function name, you can use a nickname for that function, using *as* keyword.

```
>>> import collections as c
>>>
>>> c.OrderedDict({1: 'one', 2:'two'})
OrderedDict([(1, 'one'), (2, 'two')])
```

## 2.2 What is pip?

*pip* is the standard package manager for Python with which you can install and manage Python packages. It connects to an online repository of public packages, called the Python Package Index, PyPI for short. You can visit the official website of PyPI at <https://pypi.org/>.

With Python installation (3.4 or above), pip gets installed automatically. However, it is also possible that pip is not installed by default. You can ensure pip installation with one of the following commands:

```
izan@20-04:~$ python3 -m pip
izan@20-04:~$ python3 -m pip --default-pip
```

After ensuring pip installation, if it is not installed by default, it can be easily downloaded from <https://pip.pypa.io/en/stable/installing/>. Linux users can install pip more easily by running the following command into the terminal:

```
izan@20-04:~$ sudo apt-get install python3-pip
[sudo] password for izan:
Building dependency tree
Reading state information... Done
python3-pip is already the newest version (20.0.2-5ubuntu).
```

You can now download the latest versions of the Python packages from your terminal with just a single command. You can also specify the version number of the package to be installed.

```
izan@20-04:~$ pip install packagename  
or  
izan@20-04:~$ pip install packagename==version
```

Let's download our first package from PyPI with the help of pip.

```
izan@20-04:~$ pip install kashmiri  
Collecting kashmiri  
  Downloading kashmiri-0.0.2.tar.gz (233 kB)  
   |██████████| 233 kB 552 kB/s  
Building wheels for collected packages: kashmiri  
  Building wheel for kashmiri (setup.py) ... done  
    Created wheel for kashmiri:  
      filename=kashmiri-0.0.2-py3-none-any.whl size=237908  
      sha256=8021e511eb94189c3016  
    Stored in directory:  
      /home/izan/.cache/pip/wheels/52/5f/90/d0031f1e55ec97011  
Successfully built kashmiri  
Installing collected packages: kashmiri  
Successfully installed kashmiri-0.0.2
```

If you try to install a Python package that is already installed, pip will inform you that the package is already installed by displaying the following message:

```
izan@20-04:~$ pip install kashmiri  
Requirement already satisfied: kashmiri in  
./.local/lib/python3.8/site-packages (0.0.2)
```

You can check the list of all packages that are installed on your machine by typing the following command into the terminal:

```
izan@20-04:~$ pip list
Package           Version
-----
appdirs           1.4.4
apt-clone         0.2.1
apturl            0.5.2
attrs              19.3.0
.
.
.
xkit              0.0.0
zeroconf           0.24.4
zipp              1.0.0
```

If you want a detailed view of a particular package, use the following command:

```
izan@20-04:~$ pip show kashmiri
Name: kashmiri
Version: 0.0.2
Home-page: https://github.com/izan-majeed/Kaeshir-Database
Author: Izan Majeed
Author-email: izanmajeed03@gmail.com
License: Apache License 2.0
Location: /home/izan/.local/lib/python3.8/site-packages
```

You can also remove the installed packages with the following command:

```
izan@20-04:~$ pip uninstall kashmiri
Found existing installation: kashmiri 0.0.2
Uninstalling kashmiri-0.0.2:
Successfully uninstalled kashmiri-0.0.2
```

The package, ***kashmiri*** which is used in the above examples, is a collection of 4.5k English words with the corresponding Kashmiri meaning. I'm the author of this package and all these words were manually collected by me. It's an open-source project and also has Android support with the name [Kaeshir Dictionary](#).

```
>>> from kashmiri import find
>>> from pprint import pprint
>>> find("admire")
{'title': 'Admire', 'pos': '/ əd-\'mī(-ə)r/, verb',
'englishMeaning': 'Khosh karun ', 'kashmiriMeaning': 'خوش کران ', 'englishExample': 'I admire Manika.', 'kashmiriExample': 'Winni
chai kaem chalan'}
>>>
>>> x = find('about')
>>> pprint(x)
{'englishExample': 'What do you think about me?',
'englishMeaning': 'Mutalik',
'kashmiriExample': 'Tsai kya ba:sa:n mai mutalik?',
'kashmiriMeaning': 'متعلق',
'pos': '/ ə-\'baʊt/, adverb',
'title': 'About'}
>>>
>>> find("osama bin laden")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File
  "/home/izan/Documents/Kaeshir-Database/kashmiri/kashmiri.py",
line 5, in find
    assert word.isalpha(), "You might be a Haput, else you could
have entered a correct word."
AssertionError: You might be a Haput, else you could have
entered a correct word.
>>>
```

## 3. Exception handling

You've already seen errors chasing you continuously, crashing your programs. With the concept of exception handling, you can ask the Python interpreter to handle the errors instead of crashing the entire script. Python provides a neat way to get rid of these crashes, by handling the various types of errors. There can be two types of errors: Syntax errors and Exceptions.

### 3.1 Syntax Errors

Syntax errors are the most common and unintentional errors which are caused by the syntactic mistakes. These errors are sometimes referred as *parsing* errors. Syntax errors can be fixed by looking at the error messages which suggest what the interpreter is looking for and writing the correct syntax.

```
>>> print "This should throw an error"
      File "<stdin>", line 1
          print "This should through an error"
                      ^
SyntaxError: Missing parentheses in call to 'print'. Did you
mean print("This should throw an error")?
>>>
```

### 3.2 Exceptions

Even if your script is syntactically correct, there is no guarantee that it is going to execute without crashing. This is because your script may have logical errors, which are formally called as exceptions. One of the famous exception is the *ZeroDivisionError*, which is thrown by the interpreter whenever you try to divide a number by zero.

Other common exceptions are: *TypeError*, *FloatingPointError*, *ImportError*, *IndexError*, *KeyError*, *ModuleNotFoundError*, *KeyboardInterrupt*, *NameError* and *ValueError*.

```
>>> 32/0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> import nothing
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'nothing'
>>>
>>> int('abc')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'abc'
>>>
>>> '46' ** 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for ** or pow(): 'str'
and 'int'
>>>
>>> a = [11, 22]
>>> a[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
>>> d = {'exception': True}
>>> d['syntax']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'syntax'
>>>
```

### 3.3 *try* and *except* clauses

In Python, exceptions are handled with *try-except* clauses. The code indented within the *try* clause is executed first, if an exception is caught, the rest of the *try* clause is skipped and the *except* clause is executed immediately. However, if there is no exception, the *except* clause is skipped. A *try* block is always followed by an *except* block. It is important to note that *SyntaxErrors* are not handled by *try-except* clauses.

*Syntax:*

```
try:  
    block of code where exceptions can occur  
  
except:  
    executes if exception is caught
```

The *except* clause catches all exceptions caught in the *try* block. If you mention a particular exception after the *except* clause, the interpreter will handle that exception only. You can also have multiple *except* clauses for the same *try* clause handling different exceptions.

```
try:  
    block of code where exceptions can occur  
  
except Exception1:  
    executes only if exception of ExceptionName1 is caught  
  
except Exception2:  
    executes only if exception of ExceptionName2 is caught  
  
except Exception3:  
    executes only if exception of ExceptionName3 is caught
```

The following example is executed three times:

1. In the first case, *the except clause* is skipped as no exception is caught.
2. In the second case, *ZeroDivisionError* is handled by the *except clause*.
3. Third case throws a *ValueError*, as input expects an integer but a floating point number is given. This exception is also handled by the same *except clause* as it is not followed by any specific ExceptionName.

*pilot.py*

```
import math

try:
    num = int(input('Number: '))
    result = math.pi / num
    print(f"Result: {result}")

except:
    print("Something is wrong.")

print('Outside try-except')
```

---

```
Number: 17
Result: 0.18479956785822313
Outside try-except
```

---

```
Number: 0
Something is wrong.
Outside try-except
```

---

```
Number: 3.14
Something is wrong.
Outside try-except
```

---

*pilot\_modified.py*

```
import math

try:
    num = int(input('Number: '))
    result = math.pi/ num
    print(f"Result: {result}")

except ZeroDivisionError:
    print("You are attempting to divide a number by zero.")

except ValueError:
    print("Only integers are allowed.")

except:
    print("Other exceptions.")

print('Outside try and except clauses.')
```

---

```
Number: 5
Result: 0.6283185307179586
Outside try and except clauses.
```

---

```
Number: 0
You are attempting to divide a number by zero.
Outside try and except clauses.
```

---

```
Number: 1.26
Only integers are allowed.
Outside try and except clauses.
```

### 3.4 *else* and *finally* clauses

A try-except clause can be followed by an *else* statement, which is executed only if no exception is caught in the *try* block. In other words, if there are no exceptions and the *except* clause is skipped, the *else* clause is executed.

*import\_error.py*

```
try:  
    import math  
except ImportError:  
    print("Module is currently unavailable.")  
else:  
    print("Module was imported successfully.")  
  
print('Normal code...')
```

---

```
Module was imported successfully.  
Normal code...
```

*import\_error.py*

```
try:  
    import ghost  
except ImportError:  
    print("Module is currently unavailable.")  
else:  
    print("Module was imported successfully.")  
  
print('Normal code...')
```

---

```
Module is currently unavailable.  
Normal code...
```

The try-except clause can have another optional clause named ***finally***, which is always executed. The block of code indented under the *finally* clause is always executed whether an exception is caught or not.

*index\_error.py*

```
try:
    x = input("Value of x: ")
    print(x[6])

except IndexError:
    print("Out of bounds.")

else:
    print("Program has no IndexError.")

finally:
    print("I don't care.")

print('Normal code...')
```

---

```
Value of x: finally
y
Program has no IndexError.
I don't care.
Normal code...
```

---

```
Value of x: main
Out of bounds.
I don't care.
Normal code...
```

### 3.5 *finally* clause vs normal code

You might be wondering, if the *finally* block is always executed like the normal code outside *try-except* block, then why does the *finally* clause even exist. Observe the following example programs carefully.

*is\_vowel.py*

```
def is_vowel(alphabet):
    try:
        if alphabet in ['a', 'e', 'i', 'o', 'u']:
            print(True)
        else:
            print(False)

    except:
        print("You're not passing a valid argument.")

    else:
        print("No Exception caught.")

    finally:
        print("Block of the finally clause.")

    print("Normal text outside try-except within function.")

is_vowel('a')
```

---

```
True
No Exception caught.
Block of the final clause.
Normal text outside try-except within function.
```

---

*is\_vowel.py*

```
def is_vowel(alphabet):
    try:
        if alphabet in ['a', 'e', 'i', 'o', 'u']:
            return True
        else:
            return False

    except:
        print("You're not passing a valid argument.")

    else:
        print("No Exception caught.")

    finally:
        print("Block of the finally clause.")

    print("Normal text outside try-except within function.")

print(is_vowel('a'))
```

---

Block of the finally clause.

True

Note how the *return* statement in the *try* block affects the output of *is\_vowel()* function. It is evident that if a *return* statement exists in a *try-except* block, the program flow is returned back to main. The remaining code of the function, including all clauses and statements, is skipped except the *finally* clause. The *finally* clause is always executed before the program control is returned back to main by the *return* statement. The *break* and *continue* statements also affect the program like that of *return* statement, if the *try-except* block is within a loop.

---

*is\_pi.py*

```
import math

def is_pi():
    while True:
        try:
            num = float (input ("Your Number: "))
            if (num == original_pi):
                print ("Awesome")
                break
            print ("after break within try.")

        except:
            print ("Exception!!!")

        finally:
            print ('Finally')

    print ('Outside try-except within loop.\n')

original_pi = round(math.pi, 2)
is_pi()
print("Back to main.")
```

---

```
Your Number: 8.3
Finally
Outside try-except within loop.
```

```
Your Number: 3.14
Awesome
Finally
Back to main.
```

I know there is a lot to digest from the above example. Don't worry you'll rarely encounter such situations. The thing is you should know how exceptions are raised and how you can handle them to prevent your programs from crashing.

### 3.6 *raise* statement

The *raise* statement allows you to raise a specific exception forcefully.

Syntax:

```
raise ErrorName("message to be displayed")
```

```
>>> raise NameError("I can forcefully crash any program")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: I can forcefully crash any program
>>>
```

An exception is re-raised, if you use the *raise* statement within the *try-except* block.

*reraise.py*

```
try:
    print("Before raise statement.")
    raise IndexError("I manually raised this Exception")
    print("After raise statement.")

except IndexError:
    print("I caught your Exception.")
```

---

```
Before raise statement.
I caught your Exception.
```



04. Better an OOPs than a what if

# 1. Object Oriented Programming

Object! Object! Object! I've been talking a lot about objects, even in the beginning I quoted a phrase *Everything in Python is an Object*. But the question is, what actually is an Object in programming? The formal definition of an object is, any data with state (attributes or values) and defined behavior (methods). It means that an object is a data type with which variables and functions are associated.

In Python, classes are created using **class** keyword and objects are nothing, but the instances of a class. You can say that a class is a template or blue-print for creating user-defined objects. PEP 8 conventions suggest using *CamelCase* for class names.

Let's understand this concept with an analogy, if *Human* is a class, then every person is an instance of it. A person can have various features, like two hands, one four-chambered heart etc. These features are formally called **attributes**, associated with *Human* object. Also, there are various functionalities that are associated with a person, like walking, talking etc, these are the **methods** that are associated with it.

*pilot.py*

```
class BasicClass():
    attr = 14

inst = BasicClass()
print(inst.attr)
```

---

14

In Python, it's optional to have parentheses () after the class name. In the above mentioned example, a class named *BasicClass* is created with only one *class variable* or *class object attribute*. An instance of *BasicClass* is instantiated with name *inst* and dot (.) operator is used to access its class variables.

## 1.1 class variables vs instance variables

A class variable is defined directly within a class (not enclosed in any of its methods) while an attribute (or instance variable) is a value associated with an object defined within a special method and is commonly known as *constructor*.

Attributes are unique to each instance while class variables are shared by all the instances of the same class. Unlike instance variables, you can directly access class variables without creating an instance as *ClassName.class\_variable*.

Both attributes and methods are accessed using dot (.) operator. As methods are functions, their name must be followed by a set of parentheses e.g. *ClassName.func(args)* while attributes are just values and can be accessed like *ClassName.attr*. If you press a *Tab* key after the dot (.) operator, the interpreter will provide a list of all attributes and methods associated with that object.

*useless.py*

```
class Useless():
    def what_to_do():
        print('Just print this.')

a = Useless()
a.what_to_do()
```

```
Traceback (most recent call last):
  File "useless.py", line 6, in <module>
    a.what_to_do()
TypeError: what_to_do() takes 0 positional arguments but 1 was given
```

What? I didn't pass any positional argument, right? Why is it throwing a *TypeError*? The answer is simple, when you access a method through an instance, it automatically adds an extra argument- *self*, to its argument list.

## 1.2 *self* keyword

It is passed as the first argument of an instance method and refers to the object for which the method is called. This variable is famous by the name *this* in most of the programming languages.

In Python, *self* is just a convention, you can use any name for this argument. But, the problem occurs when you import some third-party Python packages and use any other name instead of *self*, your scripts start behaving abnormally, as the first argument of an instance method is named as *self* almost everywhere in Python.

*useless.py*

```
class Useless():
    def what_to_do(self):
        print('Just print this.')

a = Useless()
a.what_to_do()
```

---

Just print this.

## 1.3 *\_\_init\_\_()* method

Python has some special methods, which are discussed in the next section of this book, that are called implicitly i.e. by the interpreter itself, to execute some specific operations. The name of these special methods start with double underscore (*dunder*) and end with the same which may look like *\_\_method\_\_*.

The most common special method is *\_\_init\_\_()*, known as constructor or initializer which is invoked automatically as an instance of some class is instantiated. All the attributes (instance variables) are initialized within this special method as:

*Syntax*

```
def __init__(self):
    self.attr1 = attribute_value
```

---

*student.py*

```
class Student():
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.school = 'Unregistered International School'

stu_1 = Student('Joshua', 14)
stu_2 = Student('Mark', 13)

print(stu_1.name, stu_1.school)
print(stu_2.age, stu_2.school)
```

---

```
Joshua Unregistered International School
13 Unregistered International School
```

It is not necessary to match the argument name with its corresponding attribute name but it is a good programming practice to use this convention in order to avoid confusions.

*fruit.py*

```
class Fruit():
    def __init__(self, x):
        self.taste = x

lemon = Fruit('sour')
print(lemon.taste)
```

---

```
sour
```

An instance of a class can be destroyed with a ***del*** statement.

*fruit.py*

```
class Fruit():
    def __init__(self, taste):
        self.taste = taste

lemon = Fruit('sour')
apple = Fruit('sweet')
print(f"Apple is {apple.taste} and Lemon is {lemon.taste}")

del lemon
print(lemon.taste)
```

---

```
Apple is sweet and Lemon is sour
Traceback (most recent call last):
  File "fruit.py", line 10, in <module>
    print(lemon.taste)
NameError: name 'lemon' is not defined
```

As you've seen an attribute within an instance method can be accessed as *self.attribute\_name*. The class object attributes (or class variables) are accessed as *ClassName.class\_variable\_name*. Although class attributes can also be accessed with the *self* keyword as *self.class\_variable\_name* but the convention is to use class name only.

*flower.py*

```
class Flower():
    kingdom = 'Plantae'      # class variable
    def __init__(self, name):
        self.name = name      # instance variable
    def display(self):
        # print(self.kingdom, self.name, sep='\t')
        print(f"Name: {self.name}, Kingdom: {Flower.kingdom}")
```

```
print(f"Class var is accessed directly: {Flower.kingdom}")
a = Flower('Rose')
a.display()
```

```
Class variable is accessed directly: Plantae
Name: Rose, Kingdom: Plantae
```

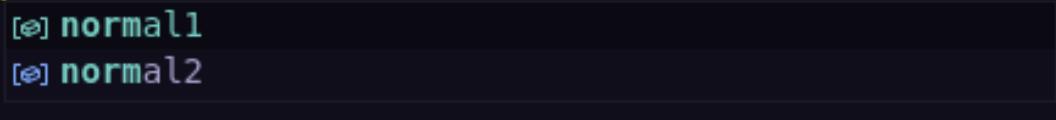
## 1.4 Private variables and private methods

As you've already seen, all the attributes (and class variables) of a given class can be accessed with a dot (.) operator. Assume, you're creating a class and you define some variables within it which are meant for the internal use only. These hidden variables are called *Private variables*. Although other programming languages support this concept of variable abstraction, there is no existence of Private variables in Python.

However, if you put an underscore (\_) before the name of a variable, that variable is partially hidden as it won't appear in the attribute list that is provided by the interpreter using the tab key after the dot (.) operator. But the variable can be easily accessed as *ClassName.\_variable*. The same logic is applied to methods also.

```
class HiddenDemo():
    def __init__(self):
        self._hidden = "A hidden variable"
        self.normal1 = "A normal variable"
        self.normal2 = "Another normal variable"

x = HiddenDemo()
x.norm|
```



The screenshot shows a code editor with the following code:

```
class HiddenDemo():
    def __init__(self):
        self._hidden = "A hidden variable"
        self.normal1 = "A normal variable"
        self.normal2 = "Another normal variable"

x = HiddenDemo()
x.norm|
```

The cursor is at the end of "norm". A completion dropdown is open, showing two options:

- [e] normal1
- [e] normal2

There is a more robust way of hiding variables and methods within a class by prefixing a double underscore (`_`) before the name of the variable (or method). If you try to access these variables (or methods) normally, the interpreter will throw an *AttributeError*.

```
class HiddenDemo():
    def __init__(self):
        self._x = "Partially hidden"
        self.__y = "Protected with double underscore"
        self.z = "Normal instance variable"

inst = HiddenDemo()
print("Variable _x: ", inst._x)
print("Variable z: ", inst.z)
print("Variable __y: ", inst.__y)
```

---

```
Variable _x: Partially hidden
Variable z: Normal instance variable
Traceback (most recent call last):
  File "main.py", line 10, in <module>
    print("Variable __y: ", inst.__y)
AttributeError: 'HiddenDemo' object has no attribute '__y'
```

Although it seems by prefixing double underscore (`_`), you protected the variable completely from outside access but that's not true. You can still access this pseudo-hidden variable as: `instance._ClassName__attr`

```
print("Variable __y: ", inst._HiddenDemo__y)
```

---

```
Variable __y: Protected with double underscore
```

---

*basket.py*

```
class Basket():
    def __init__(self):
        self._basket = []

    def fill(self, fruit):
        self._basket.append(fruit)

    def delete(self, fruit):
        if fruit in self._basket:
            self._basket.remove(fruit)

    def show(self):
        for fruit in self._basket:
            print(fruit)

obj = Basket()
obj.fill('Banana')
obj.fill('Grapes')
obj.fill('Mango')
obj.show()

print('\n')
obj.delete('Grapes')
obj.show()
```

---

Banana

Grapes

Mango

Banana

Mango

## 1.5 *class methods, instance methods and static methods*

You've already seen the concept of class variables and instance variables, likewise there are class methods and instance methods. Python can have a third type of method- static method.

All the methods that are used in the above examples are instance methods. The first argument passed to instance methods is *self* and you know this very well that *self* keyword is associated with the instance of a class. You can't directly invoke these methods, you need an instance for that.

```
>>> class ToyClass:  
...     def instance_method(self):  
...         print('called')  
...  
>>> ToyClass.instance_method()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: instance_method() missing 1 required positional  
argument: 'self'  
>>>  
>>> x = ToyClass()  
>>> x.instance_method()  
called  
>>>
```

A *class method* is a method which is bound to the class and not to an instance. In a class method, the class itself (*cls*) is passed as the first argument. Just like *self* represents an instance, *cls* represents a class; this is just a naming convention, you can choose any name for these representatives .To define a class method you need a special decorator that is built into the standard library- `@classmethod`. These methods can be directly invoked without creating an instance.

```
class ToyClass:  
...     def __init__(self):  
...         self.name = "Example"  
  
...     @classmethod  
...     def class_method(cls):  
...         print("called")  
...  
>>> ToyClass.class_method()  
called  
>>>
```

Static methods on the other hand do not receive any implicit first argument. These methods can neither modify object state nor class state. These methods are defined within a class with `@staticmethod` decorator and can be invoked directly without an instance.

```
>>> class ToyClass:  
...     def __init__(self):  
...         self.name = "Example"  
  
...     @staticmethod  
...     def static_method():  
...         print('called')  
...  
>>> ToyClass.static_method()  
called  
>>>
```

With all these concepts, let's now generalize the basic syntax for the instantiation of new classes:

*Syntax:*

```
class ClassName():

    # class variables
    class_object_attribute = value

    # constructor
    def __init__(self, *args):
        # attributes      (instance variables)
        self.attr = value

    # instance methods
    def regular_instance_method(self, *args):
        pass

    # class methods
    @classmethod
    def a_class_method(cls, *args):
        pass

    # static methods
    @staticmethod
    def a_static_method(*args):
        pass
```

---

*student.py*

```
class Student():
    semester = 'III'

    def __init__(self, name, enroll):
        self.name = name
        self.enroll = enroll
        self.sub = {'physics': None, 'maths': None}

    def set_marks(self, physics, maths):
        self.sub['physics'] = physics
        self.sub['maths'] = maths

    def display(self):
        print(f"Name: {self.name}")
        print(f"Semester: {Student.semester}")
        print(f"Physics: {self.sub['physics']}, Maths: {self.sub['maths']}")

stu1 = Student('Joshua', 213)
stu1.set_marks(physics=87, maths=81)
stu1.display()
print(stu1)
```

---

```
Name: Joshua
Semester: III
Physics: 87, Maths: 81
<__main__.Student object at 0x7fdb65b4f670>
```

Everything just worked fine, but when I try to print an instance of *Student*, instead of printing the name of student, the interpreter is showing the memory address of that instance. This issue can be resolved by applying other special methods.

## 2. Special methods

Like `__int__()`, Python has other special class methods that make your life a lot easier. These are sometimes called *magic methods* (I hate this name). All special method names start with dunder (double underscore) followed by name and terminate with dunder again as `__magic__()`. Let's explore some of these magical spells.

### 1. `__str__()`

If an object is passed to a function and that function returns a string, like `print(object)`, this method is called automatically. In other words, `__str__()` returns the string representation of an object.

*student.py*

```
class Student():
    def __init__(self, name, enroll):
        self.name = name
        self.enroll = enroll

    def display(self):
        print(f"{self.name}'s enroll is {self.enroll}.")

    def __str__(self):
        return (self.name)

x = Student('Joshua', 213)
x.display()
print(x)
```

---

```
Joshua's enroll is 213.
Joshua
```

## 2. `__len__()`

If the built-in `len()` function is called on an object, this method is invoked by the interpreter.

*student.py*

```
class Student():
    def __init__(self, name, subjects):
        self.name = name
        self.subjects = subjects

    def __str__(self):
        return (self.name)

    def __len__(self):
        return len(self.subjects)

s1 = Student ('Joshua', ['Physics', 'Chemistry', 'Maths'])
print (f'{s1} has {len(s1)} subjects.')
```

---

Joshua has 3 subjects.

## 3. `__contains__()`

This method allows you to use `in` keyword with your objects.

*basket.py*

```
class Basket():
    def __init__(self, owner, items):
        self.owner = owner
        self.items = items

    def __str__(self):
        return (self.owner)
```

```
def __contains__(self, fruit):
    return (fruit in self.items)

my_basket = Basket('Izan', ['Mango', 'Strawberry'])

if ('Mango' in my_basket):
    print(my_basket)

else:
    print ('Nothing')
```

Izan

#### 4. *\_\_iter\_\_()*

Make your objects *iterable* with this method.

*basket.py*

```
class Basket():
    def __init__(self, owner, items):
        self.owner = owner
        self.items = items

    def __iter__(self):
        return iter(self.items)

my_basket = Basket('Izan', ['Mango', 'Strawberry'])
for fruit in my_basket:
    print(fruit)
```

Mango  
Strawberry

## 5. *\_\_del\_\_()*

The Python interpreter invokes this method implicitly when an instance of a certain class is destroyed with a *del* statement.

*house.py*

```
class House():
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return (self.name)

    def __del__(self):
        print ('House deleted!')

arryn = House('Arryn')
del arryn
```

---

House deleted!

In the following example, a few more Python decorators are demonstrated that are used within classes. This is an advanced topic, feel free to skip this example if you're not comfortable with it at this level.

```
class_decorators.py

class People():
    category = "Elite"

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

    # fullname is treated as an attribute
    @property
    def fullname(self):
        return f"{self.name} {self.color}")

    # sets the value directly
    @fullname.setter
    def fullname(self, x):
        name, color = x.split()
        self.name = name
        self.color = color

    @classmethod
    def change_category(cls, x):
        cls.category = x

    # acts as a constructor
    @classmethod
    def from_given_string(cls, string):
        name, color = string.split()
        return cls(name, color)
```

```
def __str__(self):
    return (self.name)

person1 = People("Sirius", "Black")
print(person1)

person1.fullname = "Jennifer Grey"
print(person1)

person2 = People.from_given_string("Rachael Green")
print(person2)

print(f"\nClass status: {People.category}")
person1.change_category("poor")
print(f'Class status: {People.category}')
```

---

```
Sirius
Jennifer
Rachael
```

```
Class status: Elite
Class status: poor
```

## 3. The Fours Pillars of OOP

There are four main pillars in object-oriented programming: *Abstraction*, *Encapsulation*, *Inheritance* and *Polymorphism*. This concept is general and is applied across all object-oriented programming languages.

### 3.1 Encapsulation (Information hiding)

Encapsulation simply means to encapsulate or protect your objects from accidental modifications. As you already know that Python doesn't support the concept of Private variables (or methods) completely, encapsulation can still be achieved with the same convention of double underscore (`__`). The private attributes are protected from accidental modifications by restricting the direct access of these attributes. This functionality can be achieved by using the concept of getters (gets an attribute) and setters (sets an attribute).

```
class Person:
    def __init__(self, name):
        self.name = name
        self.__id = None

    def set_id(self, id):      # setter
        self.__id = id

    def get_id(self):          # getter
        return self.__id

x = Person("Izan")
x.set_id(24)
x.__id = 0
print(x.get_id())
```

## 3.2 Inheritance

Inheritance simply means inheriting properties from ancestors. This second pillar of object-oriented programming is fully supported by Python classes. With this concept you can derive new classes from an existing class which is already defined (base class). If you have multiple classes that are correlated, inheritance can greatly reduce the length of your script by re-using the code of already defined classes.

For example, a certain college may have different departments. Each department may have different academic activities but at the same time they have several common attributes like the name of college. Here, you can apply the concept of inheritance by creating a base class as *College* having all of the common attributes and methods. Various departments can then be derived from the *College* base class.

Similarly, if you create an *Employee* class you can derive *Director*, *Manager*, *Assistant* and several other groups, each represented by its own class derived from *Employee*.

*Syntax:*

```
class BaseClass():
    common attributes and methods

class DerivedClass1(BaseClass):
    unique attributes and methods of this class only

class DerivedClass2(BaseClass):
    unique attributes and methods of this class only
```

All attributes and methods of the Base class are inherited by the derived class. The constructor of Base can be called within the constructor of derived class as:

```
class DerivedClass (BaseClass):
    def __init__(self, args):
        BaseClass.__init__(self, args)
```

Let's create an example of single inheritance where the derived class is inherited from a single base class.

*college.py*

```
class College():
    def __init__(self, name):
        self.name = name

    def display(self):
        print(f"College: {self.name}")

    def __str__(self):
        return self.name

class CompScience(College):
    def __init__(self, name, vacancies):
        College.__init__(self, name)
        self.vacancies = vacancies

    def __str__(self):
        return "Computer Science"

dept = CompScience("NIT Srinagar", 7)
dept.display()

print(f"There are {dept.vacancies} vacancies in {dept},
{dept.name}")
```

---

```
College: NIT Srinagar
There are 7 vacancies in Computer Science, NIT Srinagar
```

---

## ***super()* method**

In single inheritance, *super()* function can be used to refer to the parent classes, without naming them explicitly and without passing a *self* argument to the constructor of base class.

*college.py*

```
class College():
    def __init__(self, name):
        self.name = name

class CompScience(College):
    def __init__(self, name, vacancies):
        super().__init__(name)
        self.vacancies = vacancies

dept = CompScience("NIT Srinagar", 12)
print(f"There are {dept.vacancies} vacancies.")
```

---

There are 12 vacancies.

Apart from single inheritance, there are four more types. All these five types are listed below:

1. Single inheritance
2. Multiple inheritance
3. Multi-level inheritance
4. Hierarchical inheritance
5. Hybrid inheritance.

Multiple inheritance is discussed in the next section of this book. You can explore other types by yourselves.

## Multiple Inheritance

Multiple inheritance means a single class is derived from two or more classes. All methods and attributes of these base classes are inherited by derived classes.

*multiple.py*

```
class Base1():
    def __init__(self, x):
        self.x = x

    def base1_method(self):
        print('method of Base1')

class Base2():
    def __init__(self, y):
        self.y = y

    def base2_method(self):
        print('method of Base2')

class Derived(Base1, Base2):
    def __init__(self, x, y, z):
        Base1.__init__(self, x)
        Base2.__init__(self, y)
        self.z = z

    def own_method(self):
        print ('My own method')

obj = Derived(11, 22, 33)
obj.base1_method()
obj.base2_method()
obj.own_method()
print(obj.x, obj.y, obj.z)
```

```
method of Base1
method of Base2
My own method
11 22 33
```

There is a famous problem, the **Diamond Problem** which may occur due to multiple inheritance. This problem occurs when two classes have a common ancestor and there is another class which is deriving both these classes, for example:

```
class Ancestor():
    def same(self):
        print('From Ancestor')

class B(Ancestor):
    def same(self):
        print('From B')

class C(Ancestor):
    def same(self):
        print('From C')

class Derived(B, C):
    pass
```

Since *Derived* class is deriving both *B* and *C*, and both these classes have *same()* method, which they derive from *Ancestor*, what should *Derived* class derive? Python handles it beautifully, the *same()* method of first class, here class *B*, takes the precedence.

```
Derived().same()
```

```
From B
```

### 3.3 Polymorphism

Poly is an ancient Greek word which means many and morph means form (or shape). In programming, it refers to the methods of different classes sharing the same name.

*valar\_morghulis.py*

```
class Student():
    def __init__(self, name, enroll):
        self.name = name
        self.enroll = enroll

    def display(self):
        print(f'{self.name} has enroll {self.enroll}')

class Fruit():
    def __init__(self, name, taste):
        self.name = name
        self.taste = taste

    def display(self):
        print(f'{self.name} is {self.taste} in taste')

stu1 = Student('Joshua', 213)
fruit1 = Fruit('Lemon', 'Sour')
stu1.display()
fruit1.display()
```

---

```
Joshua has enroll 213
Lemon is Sour in taste
```

---

## Method overloading (compile-time polymorphism)

Method overloading allows you to define different methods with the same name within the same scope where the number of parameters vary. Python does not support method overloading by default and if you attempt overloading, you may overload the methods but can only use the latest defined method.

*overloading.py*

```
class Arithmetic():
    def add(self, x, y):
        return (x+y)

    def add(self, x, y, z):
        return (x+y+z)

    def prod(self, x, y):
        return (x*y)

dummy = Arithmetic()
print(dummy.add(1, 3, 5))
print(dummy.add(1, 3))
```

---

9

```
Traceback (most recent call last):
  File "overloading.py", line 10, in <module>
    print(dummy.add(1, 3))
TypeError: add() missing 1 required positional argument: 'z'
```

## Method overriding (runtime polymorphism)

In inheritance, it is possible to override a derived method i.e. you can modify a method of the parent class in the derived class which is called method overriding.

*overriding.py*

```
class Parent():
    def display(self):
        print("I'm parent class")

class Child(Parent):
    def display(self):
        print("I modified this method")

x = Child()
x.display()
```

---

```
I modified this method
```

## 3.4 Abstraction (Implementation hiding)

Abstraction is the process of hiding the unnecessary details from the users and emphasizing only on usage. For example, if you define a *swap()* function, that swaps values between two variables. The user of that script is not concerned with the implementation part, whether the function is using two or three variables for swapping, rather the user is concerned with its usage. This concept of abstraction can be extended to classes too.

The idea of abstraction, in inheritance, is applied by the concept of *Abstract Base Classes, ABCs* for short. Just as class is a blueprint for its instances, an *ABC* is blueprint for other classes. In Python, you cannot directly create an abstract class, you have to import the *abc* module for that.

---

*abstraction.py*

```
from abc import ABC

class BaseClass(ABC):
    def base_method(self):
        print("Base Method")

class DerivedClass(BaseClass):
    def derived_method(self):
        print("Derived Method")

x = BaseClass()
x.base_method()

y = DerivedClass()
y.base_method()
y.derived_method()
```

---

```
Base Method
Base Method
Derived Method
```

The code in the above example executes normally as if it is single inheritance, then what's the role of an Abstract Base Class? Abstract classes contain abstract methods which have declaration but no implementation. The abstract methods can be declared with `@abstractmethod` decorator which comes with the `abc` module. The abstract methods in `ABC` force its derived classes to override these methods. Moreover, you cannot create instances of `ABC` after declaring abstract methods with it.

---

*abstraction.py*

```
from abc import ABC, abstractmethod

class BaseClass(ABC):
    def base_method(self):
        print("Base Method")

    @abstractmethod
    def some_method(self):
        pass

class DerivedClass(BaseClass):
    def derived_method(self):
        print("Derived Method")

    def some_method(self):
        print("Some method")

y = DerivedClass()
y.base_method()
y.some_method()

x = BaseClass()
```

---

```
Base Method
Some method
Traceback (most recent call last):
  File "abstraction.py", line 21, in <module>
    x = BaseClass()
TypeError: Can't instantiate abstract class BaseClass with
abstract methods some_method
```

There is another way to achieve abstraction in Python by raising *NotImplementedError* exception in the methods of a normal base class.

*not\_implemented.py*

```
class Base():
    def __init__(self, x):
        self.x = x

    def display(self):
        raise NotImplementedError

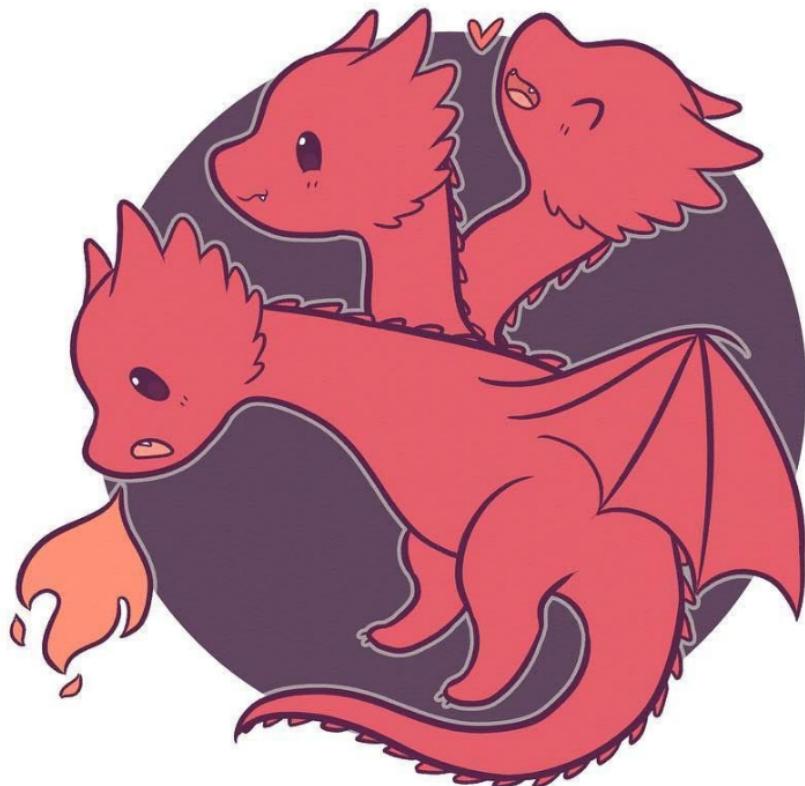
class Derived(Base):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def display(self):
        print('Derived Class')

derived = Derived('X', 'Y')
derived.display()
base = Base('X')
base.display()
```

---

```
Derived Class
Traceback (most recent call last):
  File "not_implemented.py", line 19, in <module>
    base.display()
  File "main.py", line 6, in display
    raise NotImplementedError
NotImplementedError
```



## 05. File and Regex

## 1. File Handling

With Python, you can handle almost all types of files including text files, pdf, docx, excel, html, json, archives (zip, tar etc) and much more. For each file type, Python modules are available which can be downloaded from PyPI using pip. In this section, you'll work only with text files which don't need any additional module. You can manipulate text files with the built-in methods of Python.

### 1.1 *open()*

It is one of Python's built-in functions that accepts a string containing the filename followed by another string, describing the way in which the file is going to be used and returns an ***IO*** object.

*Syntax:*

```
io_object = open('filename.txt', 'r')
```

If the file is not in the current working directory, you can provide the file path (relative or absolute) as the first argument to the *open()* function. However, if the file is not found, exception *FileNotFoundException* is thrown by the interpreter.

You've to be cautious with the file path as different operating systems follow different conventions:

- Windows: *open("C:\\\\Users\\\\username\\\\Home\\\\MyFolder\\\\xyz.txt")*
- MacOS and GNU/Linux: *open("/home/username/MyFolder/abc.txt")*

You can type *OS* independent path using Python's **os** module as:

```
>>> import os
>>> os.path.join('home', 'username', 'folder', 'file.txt')
'home/username/folder/file.txt'
>>>
```

## 1.2 Mode

The second argument passed to `open()` function is mode. It is an optional argument that defaults to 'r' (read only). You can specify the mode of a file with the following options:

Mode	Description
'r'	read only (default)
'w'	write only; overwrites the existing file.
'x'	open for exclusive creation, fails if the file already exists
'a'	append only i.e., add to the end of file
'b'	binary mode
't'	text mode
'+'	extends the functionality of given mode (reading and writing)
'U'	universal newlines mode (deprecated)

### Mode r

To read the content of a file, open the file in 'r' mode (default mode). You can use the `.read()` method of the `IO` object which returns the entire content of the file as string. `.read()` method accepts an optional argument- `size`.

Create a new text file in the notepad (or any other text editor) and save it to a specific directory and then open that text file in the reading mode (default) with the above-mentioned `open()` function.

*xyz.txt*

This is the first sentence of my text file.  
The quick brown fox jumped over a lazy dog.

```
>>> f = open('/home/izan/Documents/Notes/xyz.txt')
>>> f.read()
'This is the first sentence of my text file.\nThe quick brown
fox jumped over a lazy dog.'
>>>
>>> f.read()
''
```

Why an empty string is returned when I call *f.read()* for the second time? This is because once you read a file the control (or cursor) reaches to the end of the text within that file, so there is nothing to read. This issue can be resolved with the *.seek()* method. This method takes an argument which accepts three values:

- 0: resets the position at the beginning of the file.
- 1: sets the cursor at the current file position.
- 2: sets the cursor at the end of the file.

```
>>> f.seek(0)
0
>>> f.read()
'This is the first sentence of my text file.\nThe quick brown
fox jumped over a lazy dog.'
>>> f.close()
>>>
```

Once you're done with your job, you should always call the *.close()* method on the *IO* object in order to close the file and immediately free the system resources used by it. Files may behave abnormally, if you don't close them manually.

*.readlines()* is another method that returns a list of lines within the text file.

```
>>> f.readlines()
['This is the first sentence of my text file.\n', 'The quick
brown fox jumped over a lazy dog.']
```

An efficient way to read lines from a file is by looping over the *IO* object.

```
>>> for line in f:  
...     print (line)  
...  
This is the first sentence of my text file.  
The quick brown fox jumped over a lazy dog.  
>>>
```

### 1.3 *with* statement

Instead of manually closing the files, you can use *with* keyword that automatically closes a file, even if an exception is raised at some point.

```
>>> with open('xyz.txt') as f:  
...     print(f.read())  
...  
This is the first sentence of my text file.  
The quick brown fox jumped over a lazy dog.  
>>>
```

The text file *xyz.txt* is opened as an *IO* object, with the name *f* (alias) which lives under the indentation of *with* block only. This IO object is only accessible within the *with* block and if you attempt to access it outside, the interpreter will throw a *ValueError*.

```
>>> with open('xyz.txt') as f:  
...     print(f.read())  
...  
This is the first sentence of my text file.  
The quick brown fox jumped over a lazy dog.  
>>> f.read()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: I/O operation on closed file.
```

## Mode *w*, *w+* and *r+*

Mode ‘*w*’ opens a file in write only mode i.e., you can only write to a file. If you write to a file that doesn’t exist, a new file gets created with the mentioned name. However, if a file already exists then the text is overwritten i.e. previous data of the file is destroyed.

.*write()* method inserts a specified text and returns the length of characters which are written to the IO object.

```
>>> with open('xyz.txt', 'w') as f:  
...     f.write('The Night Lands.\nThe Dragon and the Wolf.')  
...  
41  
>>> with open('xyz.txt') as f:  
...     print(f.read())  
...  
The Night Lands.  
The Dragon and the Wolf.  
>>>
```

Attempting to write to a file in read only (‘*r*’) mode throws the following exception:

```
>>> with open('xyz.txt', 'r') as f:  
...     f.write('Erased everything')  
...  
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
    io.UnsupportedOperation: not writable  
>>>
```

Instead of ‘*w*’, if you open a file in ‘*w+*’ mode, you can perform both read as well as write operations on the *IO* object. Since ‘*w+*’ is a modified version of ‘*w*’, this mode also overwrites the existing file or creates a new one in case the file doesn’t exist.

```
>>> with open('/home/izan/Desktop/MyNewFile', 'w+') as f:  
...     f.write('This is a new text file.\nThis is opened in w+  
mode.')  
...     f.seek(0)  
...     print(f.read())  
...  
51  
0  
This is a new text file.  
This is opened in w+ mode.  
>>>
```

*f.write()* writes to file and returns the length of the string passed.

*f.seek()* resets the position and returns 0 indicating the start of file.

*f.read()* returns the content of the file as a string.

The ‘r+’ mode provides the same functionality of reading and writing to a file. The only difference is that it can’t create a new file i.e. if a file doesn’t exist, the interpreter will throw a *FileNotFoundException*, same as ‘r’ mode.

```
>>> with open('/home/izan/fsociety.txt', 'r+') as f:  
...     f.write('White rose')  
...     print(f.read())  
...  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
FileNotFoundException: [Errno 2] No such file or directory:  
 '/home/izan/fsociety.txt'  
>>>
```

## Mode a and a+

Opening a file in ‘w’ mode destroys the existing data. So, it’s not a good option to open a file in ‘w’ (or ‘w+’) mode if you want to add something to the existing file. For this reason only, there is a third mode that supports append operation. This append mode also creates a new file if the file doesn’t exist.

```
>>> with open('xyz.txt', 'a') as f:  
...     f.write('\nAppend me.')  
...  
11  
>>> with open('xyz.txt', 'r') as f:  
...     print(f.read())  
...  
The Night Lands.  
The Dragon and the Wolf.  
Append me.  
>>>
```

The ‘a’ mode only supports the `.write()` method. In order to have both read and write operations use ‘a+’ instead.

```
>>> with open ('xyz.txt', 'a+') as f:  
...     f.write ('\n Added in a+ mode')  
...     f.seek(0)  
...     print(f.read())  
...  
18  
0  
The Night Lands.  
The Dragon and the Wolf. Append me.  
Added in a+ mode  
>>>
```

Here is the summary of file methods which are associated with IO objects:

Method	Description
.read()	returns a giant string of whole text from the file.
.seek()	changes the position of the file object.
.readlines()	returns a list of lines from the file.
.write()	overwrites or appends text to a file, depending on mode.
.close()	closes the IO object

**Note:** For the binary read-write access, mode '**w+b**' opens and truncates the file to 0 bytes while '**r+b**' opens the file without truncation.

## 2. Regular Expressions

Regular expressions or simply regexes are essentially a tiny, highly specialized programming language embedded inside Python which is made available through the regular expression- ***re*** module. These expressions are used in pattern matching like finding email addresses in a given text. With this module you've to define your own patterns and set specific rules, the rest of the job is done by the module. This book explains three commonly used methods of ***re*** module:

Method	Description
<code>re.findall()</code>	returns a list of matches.
<code>re.search()</code>	returns a match object.
<code>re.split()</code>	returns a list, splitted on a given pattern.

### 2.1 ***re.findall(pattern, text)***

This method finds all substrings where the regex matches, and returns them as a list.

```
>>> import re
>>> text = 'Seated in Winterfell, Stark is the principal house
of North.'
>>> pattern = 'Winterfell'
>>> re.findall(pattern, text)
['Winterfell']
>>>
>>> t = 'The fear of suffering is worse than suffering itself.'
>>>
>>> re.findall('suffering', t)
['suffering', 'suffering']
>>>
```

You can pass an optional third argument, ***re.I*** which restricts the case sensitive behaviour

of `.findall()` method

```
>>> text = 'BeyOnd thE wAll'
>>> pattern = 'e'
>>>
>>> re.findall(pattern, text)
['e']
>>>
>>> re.findall(pattern, text, re.I)
['e', 'E']
>>>
```

```
>>> p = '.'
>>>
>>> re.findall(p, '3.14')
['3', '.', '1', '4']
>>>
>>> re.findall('?', 'what?')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "/usr/lib/python3.8/re.py", line 241, in findall
        return _compile(pattern, flags).findall(string)
    File "/usr/lib/python3.8/sre_parse.py", line 668, in _parse
        raise source.error("nothing to repeat",
re.error: nothing to repeat at position 0
>>>
```

What just happened with the last two patterns? This is not an abnormal behavior of `re` module rather these are metacharacters which have special meanings while matching a pattern with regexes, these special characters include:

. ^ \$ \* + ? { } [ ] \ | ()

## 1. Period (.)

It matches any character except a newline character ('\n').

```
>>> pattern = 'The Watchers on The Wall'
>>> pattern = '.'
>>> re.findall(pattern, text)
[['T', 'h', 'e', ' ', 'W', 'a', 't', 'c', 'h', 'e', 'r', 's', ' ', 'o', 'n', ' ', 'T', 'h', 'e', ' ', 'W', 'a', 'l', 'l']]
```

## 2. Square brackets []

The square brackets are used for specifying a *character class*. A character class is a set of characters that you want to match with the text. Characters in a character set can either be listed individually or a range of characters represented by two characters with a hyphen (-) in between.

```
>>> text = 'ThE DrAgOn And ThE WOLF'
>>> pattern = '[a-z]'
>>> re.findall(pattern, text)
['h', 'r', 'g', 'n', 'n', 'd', 'h', 'l', 'f']
>>>
>>> pattern = '[A-Z]'
>>> re.findall(pattern, text)
[['T', 'E', 'D', 'A', 'O', 'A', 'T', 'E', 'W', 'O']]
>>>
>>> text = 'The series premiered on HBO in the US on April 17, 2011'
>>> pattern = '[0-9]'
>>> re.findall(pattern, text)
['1', '7', '2', '0', '1', '1']
```

**Note:** All the metacharacters are *not active* inside these character classes.

```
>>> text = 'http://127.0.0.1:8888/?token=d38b98bb497b'
>>> pattern = '[0-7]'
>>> re.findall(pattern, text)
['1', '2', '7', '0', '0', '1', '3', '4', '7']
>>>
>>> # period is active outside the character class
>>> pattern = '[0-7].'
>>> re.findall(pattern, text)
['12', '7.', '0.', '0.', '1:', '38', '49', '7b']
>>>
>>> # period is not active inside the character class
>>> pattern = '[0-7.]'
>>> re.findall(pattern, text)
['1', '2', '7', '.', '0', '.', '0', '.', '1', '3', '4', '7']
>>>
```

### 3. Caret (^)

This metacharacter simply matches the pattern that *starts with* character(s) followed after the caret '^'. It matches at the beginning of the line.

```
>>> text = 'The Army Of Dead'
>>> pattern = '^T'
>>> re.findall(pattern, text)
['T']
>>> pattern = '^A'
>>> re.findall(pattern, text)
[]
>>>
>>> text = 'In the southern principality of Dorne.'
>>> pattern = '^In....'
>>> re.findall(pattern, text)
['In the']
```

```
>>> text = 'A Song of Ice and Fire!'
>>> pattern = '[a-zA-Z]'
>>> re.findall(pattern, text)
['A', 'S', 'o', 'n', 'g', 'o', 'f', 'I', 'c', 'e', 'a', 'n',
'd', 'F', 'i', 'r', 'e']
>>>
```

Inside a character class, it matches the characters that are not listed within the class by complementing the set which is indicated by including a '^' as the first character of the class.

```
>>> text = "Few punctuation marks are: the period., question
mark?, exclamation!, colon:, comma, and semicolon;"
>>> pattern = '[^!?.;,"]'
>>> re.findall (pattern, text)
['F', 'e', 'w', ' ', 'p', 'u', 'n', 'c', 't', 'u', 'a', 't',
'i', 'o', 'n', ' ', 'm', 'a', 'r', 'k', 's', ' ', 'a', 'r', 'e',
':', ' ', 't', 'h', 'e', ' ', 'p', 'e', 'r', 'i', 'o', 'd', ' ',
'q', 'u', 'e', 's', 't', 'i', 'o', 'n', ' ', 'm', 'a', 'r', 'k',
' ', 'e', 'x', 'c', 'l', 'a', 'm', 'a', 't', 'i', 'o', 'n', ' ',
'c', 'o', 'l', 'o', 'n', ':', ' ', 'c', 'o', 'm', 'm', 'a', ' ',
'a', 'n', 'd', ' ', 's', 'e', 'm', 'i', 'c', 'o', 'l', 'o', 'n']
```

#### 4. Dollar (\$)

It matches at the end of a line, which is defined as either the end of a string, or any location followed by a newline character.

Just as '^' means `startswith`, '\$' means `endswith`.

```
>>> text = 'The Army Of Dead'
>>> pattern = 'd$'
>>> re.findall(pattern, text)
['d']
>>>
>>> text = 'In the southern principality of Dorne, Ellaria Sand
(Indira Varma) seeks vengeance against the Lannisters.'
>>> pattern = '.....s.$'
>>> re.findall(pattern, text)
['Lannisters.']
>>>
```

## 5. Asterisk (\*)

It specifies that the previous character is matched zero or more times.

```
>>> text = 'There are trench coats and balmacaans in silver'
>>>
>>> # 'a' followed by zero or more 'a'
>>> pattern = 'aa*'
>>> re.findall(pattern, text)
['a', 'a', 'a', 'a', 'a', 'aa']
>>>
>>> # any lower case alphabet followed by zero or more 'a'
>>> pattern = '[a-z]a*'
>>> re.findall(pattern, text)
['h', 'e', 'r', 'e', 'a', 'r', 'e', 't', 'r', 'e', 'n', 'c',
'h', 'c', 'o', 't', 's', 'a', 'n', 'd', 'b', 'a', 'l', 'm', 'c', 'a', 'a',
'n', 's', 'i', 'n', 's',
'i', 'l', 'v', 'e', 'r']
>>>
>>> text = 'He suffers from
hippopotomonstrosesquippedaliophobia.'
>>>
```

```
>>> # 'o' followed by zero or more 'p'  
>>> pattern = 'op*'  
>>> re.findall(pattern, text)  
['o', 'op', 'o', 'o', 'o', 'o', 'op', 'o']  
>>>
```

## 6. Plus (+)

It specifies that the previous character is matched one or more times.

```
>>> text = 'There are trench coats and balmacaans in silver'  
>>>  
>>> # 'a' followed by one or more 'a'  
>>> pattern = 'aa+'  
>>> re.findall(pattern, text)  
['aa']  
>>>  
>>> # any lower case alphabet followed by one or more 'a'  
>>> pattern = '[a-z]a+'  
>>> re.findall(pattern, text)  
['oa', 'ba', 'ma', 'caa']  
>>>  
>>> text = 'He suffers from  
hippopotomonstrosesquippedaliophobia.'  
>>>  
>>> #'o' followed by one or more 'p'  
>>> pattern = 'op+'  
>>> re.findall(pattern, text)  
['op', 'op']  
>>>
```

## 7. Question mark (?)

It specifies that the previous character is matched one or zero times.

```
>>> text = 'There are trench coats and balmacaans in silver'  
>>>  
>>> # 'a' followed by zero or one 'a'  
>>> pattern = 'aa?'  
>>> re.findall(pattern, text)  
['a', 'a', 'a', 'a', 'a', 'aa']  
>>>  
>>> # any lower case alphabet followed by one or zero 'a'  
>>> pattern = '[a-z]a?'  
>>> re.findall(pattern, text)  
['h', 'e', 'r', 'e', 'a', 'r', 'e', 't', 'r', 'e', 'n', 'c',  
'h', 'c', 'o', 't', 's', 'a', 'n', 'd', 'b', 'a', 'l', 'm', 'a', 'c',  
'a', 'n', 's', 'i', 'n',  
's', 'i', 'l', 'v', 'e', 'r']  
>>>  
>>> text = 'He suffers from  
hippopotomonstrosesquippedaliophobia.'  
>>>  
>>> #'o' followed by one or zero 'p'  
>>> pattern = 'op?'  
>>> re.findall(pattern, text)  
['o', 'op', 'o', 'o', 'o', 'o', 'op', 'o']  
>>>
```

## 8. Curly braces {x, y}

This qualifier means there must be at least x repetitions and at most y i.e., characters are repeated x to y times.

```

>>> text = 'Zxzxzzxzzzxxxxx'
>>> pattern = 'xz{1,2}'
>>> re.findall (pattern, text)
['xz', 'xzz', 'xzz', 'xzz', 'xz']
>>>
>>> pattern = 'xz{3,4}'
>>> re.findall (pattern, text)
['xzzz', 'xzzzz']
>>>
>>> pattern = 'xz{2}'
>>> re.findall (pattern, text)
['xzz', 'xzz', 'xzz']
>>>

```

## 9. Backslash (\)

Besides escape sequences, a backslash is also used to escape the metacharacters. As a backslash itself must be escaped in normal Python strings, you've to use two backslashes. It's not a good idea to use double backslashes with pattern matches as it reduces the code readability. The raw strings (r'') eliminate this problem and maintain the code readability.

Character	Meaning
\d	any decimal digit; equivalent to the class [0-9]
\D	any non-digit character; equivalent to the class [^0-9]
\s	whitespace; equivalent to the class [\t\n\v]
\S	non-whitespace; equivalent to the class [^\t\n\v]
\w	alphanumeric; equivalent to the class [a-zA-Z0-9_]
\W	non-alphanumeric; equivalent to the class [^a-zA-Z0-9_]

```
>>> text = 'The Gift - s5 e7'
>>> pattern = r'\d'
>>> re.findall(pattern, text)
['5', '7']
>>>
>>> pattern = r'\D'
>>> re.findall(pattern, text)
['T', 'h', 'e', ' ', 'G', 'i', 'f', 't', ' ', '-', ' ', 's', ' ', 'e']
>>>
>>> pattern = r'\s'
>>> re.findall(pattern, text)
[' ', ' ', ' ', ' ']
>>>
>>> pattern = r'\S'
>>> re.findall(pattern, text)
['T', 'h', 'e', 'G', 'i', 'f', 't', '-', 's', '5', 'e', '7']
>>>
>>> pattern = r'\w'
>>> re.findall(pattern, text)
['T', 'h', 'e', 'G', 'i', 'f', 't', 's', '5', 'e', '7']
>>>
>>> pattern = r'\W'
>>> re.findall(pattern, text)
[' ', ' ', ' ', ' ', ' ']
>>>
```

## 10. Pipe operator (|)

If A and B are two regular expressions then A|B matches any string that either matches with A or B. In order to match a pipe operator itself, use a backslash followed by pipe operator as \|. You can also enclose it inside a character class like [], as metacharacters are inactive inside character classes.

```
>>> text = 'Nobs | Noble | Noise | Noisome | North | Norway'
>>> pattern = 'Nob..|Noi..'
>>> re.findall(pattern, text)
['Nobs ', 'Noble', 'Noise', 'Noiso']
>>>
>>> pattern = 'North | Norway'
>>> re.findall(pattern, text)
['North ', ' Norway']
>>>
>>> pattern = 'North \| Norway'
>>> re.findall(pattern, text)
['North | Norway']
>>>
```

## 2.2 `re.search(pattern, text)`

If the provided match is found, this method returns a *match object* containing information about the match, else returns a *None* object. The information provided by this method includes the starting and ending point of a match, the matched substring and much more.

```
>>> text = 'Dark Wings Dark Words'
>>> pattern = 'Dark'
>>> re.search(pattern, text)
<re.Match object; span=(0, 4), match='Dark'>
>>>
```

The information from the match object can be extracted with the following methods:

- `.group()` returns the first substring that was matched by the regex.
- `.start()` and `.end()` returns the starting and ending index of the match respectively.
- `.span()` returns both start and end indexes in a single tuple.

```
>>> text = 'hannahbaker13@reasons.com'
>>> pattern = 'n'
>>> mo = re.search(pattern, text)
>>> mo.group()
'n'
>>>
>>> mo.span()
(2, 3)
>>>
>>> mo.start()
2
>>> mo.end()
3
>>>
>>> pattern = r'\d\d'
>>> mo = re.search(pattern, text)
>>> mo.group()
'13'
>>>
```

## Grouping

Groups are marked by a set of parentheses, ( ). These parentheses group together the expressions contained inside them, and can repeat the contents of a group with a repeating qualifier, such as \*, +, ?, or {x,y}.

```
>>> text = "My enrollment number is 2017BCSE028"
>>> pattern = r'\d{4}B[a-zA-Z]{3}\d{3}'
>>> mo = re.search(pattern, text)
>>> mo.group()
'2017BCSE028'
>>>
```

If you want to separate the batch (2017), branch (CSE) and id (028) from the above example, you can use grouping in the regex pattern. In order to identify each group separately, groups are numbered starting with 0 and group 0 is always present.

```
>>> pattern = r'(\d{4})B([a-zA-Z]{3})(\d{3})'
>>> mo = re.search(pattern, text)
>>> mo.group()
'2017BCSE028'
>>> mo.group(0)
'2017BCSE028'
>>>
>>> mo.group(1)
'2017'
>>> mo.group(2)
'CSE'
>>> mo.group(3)
'028'
>>>
```

## 2.3 *re.split(split\_pattern, text)*

The *re.split()* method splits the provided text based on the pattern passed. It returns a list of substrings wherever the regular expression is matched.

```
>>> text = 'Floccinaucinihilipilification'
>>> pattern = 'i'
>>> re.split(pattern, text)
['Flocc', 'nauc', 'n', 'h', 'l', 'p', 'l', 'f', 'cat', 'on']
>>>
>>> text = 'Dd38b98bb497ba'
>>> pattern = r'\d'
>>> re.split(pattern, text)
['Dd', '', 'b', '', 'bb', '', '', 'ba']
>>>
```

## 2.4 Contact Extractor

With the powers of regular expressions, try to write a script by yourself that extracts all the emails and contact numbers from a given text file. Do not look at the solution without giving it a shot. This is just an example script, you can create a more robust contact extractor.

*contact\_extractor.py*

```
import re

with open('your_text_file_path') as file:
    text_doc = file.read()

phone_pattern = r"[+91]?\[0-9\]{10}"
phone_list = re.findall(phone_pattern, text_doc)

email_pattern = r"\w+\@[a-z]+\.\w+"
email_list = re.findall(email_pattern, text_doc)
```



## Grow through what you go through

Congratulations! You should be proud of yourself that you made it this far. Unless and until you do not practice your skills, you're equivalent to those who have the slightest idea of coding or do not know coding at all. Finishing this book or some other tutorials and printing "Hello World" is not what makes you a programmer. You must be consistent with the things you've learnt and practice coding problems as much as you can, that's the only way out. There are a lot, a lot of non-technical guys out there who can do the same but there must be some difference between you and them and that difference comes with practice.

Given below is a list of some basic programs along with their solutions. The solutions given are not optimized, that is a whole subject in itself. These examples are only focused on logic and not on space and time complexities.

### 1. Palindrome

Write a function that takes a string as an argument and returns *True* if the given string is palindrome. A palindromic string is a string that is equal to its reversed version. For example: eye, racecar, pip, madam etc.

*palindrome.py*

```
def palindrome(string):
    string = string.lower()
    for i in range(len(string) // 2):
        if (string[i] != string[len(string) - 1]):
            return False
        string = string [:len(string) - 1]
    return True

def palindrome(string):
    return (string.lower() == string [::-1].lower ())

# The downside of the second approach is the code readability as
# the logic of that one liner is not clear.
```

## 2. Armstrong Number

Write a function that takes an integer as an argument and returns *True* if the given integer is an Armstrong number. An Armstrong number is a positive n-digit number that is equal to the sum of the *n<sup>th</sup>* powers of its digits. For example:  $371 = 3^3 + 7^3 + 1^3$

*armstrong.py*

```
def armstrong(n):
    number = n
    total_digits = len(str(n))
    sum_ = 0

    for i in range(total_digits):
        last_digit = n%10
        last_digit **= total_digits
        sum_ += last_digit
        n //= 10

    if (sum_ == number):
        return True

    return False
```

## 3. Count Vowels

Write a function that takes a string as an argument and returns the count of each vowel in the given string.

*count\_vowel.py*

```
def count_vowel(string):
    vowels = 'a,e,i,o,u'.split(',')
    vowel_count = {}
```

```
for char in string:  
    if (char.lower() in vowels):  
        vowel_count.setdefault (char, 0)  
        vowel_count[char] += 1  
  
return vowel_count
```

## 4. Count the Characters

Write a function that takes a string as an argument and returns a dictionary of its character count.

*character\_count.py*

```
def count_the_characters(string):  
    seen = dict()  
  
    for alphabet in string:  
        if alphabet in seen.keys():  
            seen[alphabet] += 1  
        else:  
            seen[alphabet] = 1  
  
    return seen  
  
# using built-in dict.setdefault() method  
def character_count(s):  
    count = {}  
    for i in s:  
        count.setdefault (i, 0)  
        count[i] += 1  
  
    return count
```

## 5. Classic FizzBuzz

Write a function that prints all numbers from 1 to 100 with the given conditions:

- for multiples of 3 print *Fizz*
- for multiples of 5 print *Buzz*
- for multiples of both print *FizzBuzz*

*fizz\_buzz.py*

```
def fizz_buzz():
    for i in range (1,101):
        if (i%3 == 0) and (i%5 == 0):
            print ("FizzBuzz")

        elif (i%3 == 0):
            print("Fizz")

        elif (i%5 == 0):
            print("Buzz")

        else:
            print(i)
```

## 6. Largest Continuous Sum

Write a function that takes a list as an argument and returns the largest continuous sum of its elements.

*large\_sum.py*

```
def large_sum(arr):
    current_sum = max_sum = arr[0]

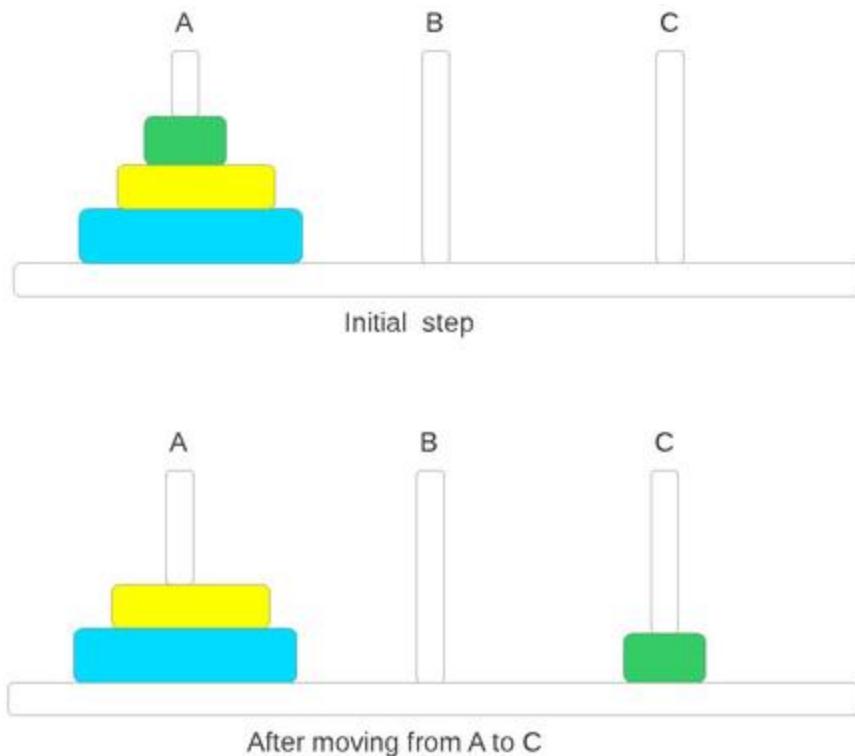
    for num in arr[1:]:
        current_sum = max(num, current_sum+num)
        max_sum = max(current_sum, max_sum)

    return max_sum
```

## 7. Tower of hanoi

Tower of hanoi is the famous mathematical puzzle in which three pegs A, B, C are given with the different sized disks.

At the beginning, the disks are stacked on peg A, such that the largest sized disk is on the bottom and the smallest sized disk on top. You have to transfer all the disks from source peg A to the destination peg C by using an intermediate peg B.



Following are the rules that are to be followed during the transfer:

- Transferring the disks from the source peg to the destination peg such that at any point of transformation no large size disk is placed on the smaller one.
- Only one disk can be moved at a time.
- Each disk must be stacked on any one of the pegs.

---

*tower\_of\_hanoi.py*

```
def toh(n, A='A', B='B', C='C'):
    if (n > 0):
        toh(n-1, A, C, B)
        print(f"Move the disc {n} from {A} to {C}")
        toh(n-1, B, A, C)

toh(3)
```

---

```
Move the disc 1 from A to C
Move the disc 2 from A to B
Move the disc 1 from C to B
Move the disc 3 from A to C
Move the disc 1 from B to A
Move the disc 2 from B to C
Move the disc 1 from A to C
```

## 8. Fibonacci Series

Write a function that takes an integer n as argument and returns the first n terms of fibonacci sequence. You can do it either by recursion or simply by iteration.

*fibonacci.py*

```
def fib (n):
    first, second = 0,1
    print (first, second, sep='\n')

    for i in range(n):
        print (first+second)
        first, second = second, first+second
```

## 9. String reversal

Write a function that takes a string as an argument and returns a reversed copy of it.

*string\_rev.py*

```
# first approach
def reverse(string):
    s = ''
    for _ in range(len(string)):
        s += string [len (string) - 1]
        string = string [:(len (string) - 1)]
    return s

# second approach
def reverse(s, output=None):
    if (len(s) == 1):
        output += s
        return output

    if not output:
        output = ''

    last = s[len(s) - 1]
    s = s[:len(s) - 1]
    output += last
    return reverse(s, output)

# third approach
def reverse(s):
    return s[::-1]
```

## 10. Integer reversal

Write a function that takes an integer as an argument and returns the reverse order of digits of that integer.

*int\_reversal.py*

```
def rev_int(num):
    string = ""
    if (num == 0):
        return 0
    elif (num < 0):
        num = abs(num)
        rev_num = rev_int(num)
        return -int(rev_num)
    while num > 0:
        last = num%10
        string += str(last)
        num //= 10
    return int(string)
```

## 11. Anagrams

Write a function that takes two strings as arguments and returns *True* if those two strings are anagrams of each other. A pair of strings are said to be anagrams of each other if the second string can be created from the first string by rearranging its letters. For example: *public relations* is an anagram of *crap built on lies*.

*anagram.py*

```
def anagram(s1, s2):
    s2 = s2.replace(' ', '').lower()
    s1 = s1.replace(' ', '').lower()
    if len(s1) == len(s2):
        return (set(s1) == set(s2))
    return False
```

```
# without using the concept of sets
def anagram(s1, s2):
    s2 = s2.replace(' ', '').lower()
    s1 = s1.replace(' ', '').lower()
    if len(s1) == len(s2):
        for i in s1:
            if i not in s2:
                return False
        return True
    return False
```

## 12. List Chunks

Write a function that takes two arguments, list and chunk size, and returns a list of grouped items based on chunk size. Try to implement it without slicing.

For example: *chunk\_list([1,2,3,4,5,7,8,9], 3)* →→ [[1, 2, 3], [4, 5, 7], [8, 9]]

*list\_chunks.py*

```
def chunk_list (arr, size):
    chunk = []
    arr2 = []

    for ele in arr:
        if (len(chunk) == size):
            arr2.append(chunk)
            chunk = []
        chunk.append(ele)

    arr2.append(chunk)
    return arr2
```

## 13. Collatz Conjecture

Start with a number  $n$  with  $n > 1$ . Find the number of steps it takes to reach unity (1) by the following process:

- If  $n$  is even, divide it by 2
- if  $n$  is odd, multiply it by 3 and add 1.

*collatz\_conjecture.py*

```
def collatz_conjecture(num, count=0):  
    if (num < 1):  
        print("Number must be greater than 1")  
        return  
  
    if (num == 1):  
        return count  
  
    if (num%2 == 0):  
        count += 1  
        return collatz_conjecture(num/2, count)  
  
    if (num%2 != 0):  
        count += 1  
        return collatz_conjecture( (num*3)+1, count)  
  
if __name__ == "__main__":  
    while "infinity":  
        try:  
            n = int(input("Enter the Number: "))  
            print (collatz_conjecture(n))  
  
        except ValueError:  
            print ('Invalid Input.')
```

## 14. Title Case String

Write a function that takes a string as an argument and returns the titlecased copy of it without using built-in `str.title()` method.

*titlecased\_string.py*

```
def capitalized_string(string):
    str_list = string.split(' ')
    string = ''

    for i in str_list:
        string = string + i.replace(i[0], i[0].upper()) + " "

    return string.strip()
```

## 15. Unique Characters

Write a function that takes a string as an argument and returns *True* if it doesn't have duplicate characters.

*unique\_char.py*

```
def unique_char(string):
    seen = []
    for a in string:
        if a in seen:
            return False
        else:
            seen.append(a)
    return True

# or

def uni_char(s):
    return (len (set (s)) == len (s))
```

## 16. Most Repeated Character

Write a function that takes a string as an argument and returns the most repeated character in that string.

*max\_char.py*

```
def max_char (string):
    string = string.replace(' ', '')
    count = {}

    for alphabet in string:
        count.setdefault(alphabet, 0)
        count[alphabet] += 1

    for k,v in count.items():
        if v == max(count.values()):
            return k
```

## 17. Linear Search

Write a function to search an item in a sequence object (sorted or unsorted). The basic approach is to use a loop to iterate through every element and check whether a certain element is there or not. This is known as the *sequential search algorithm* or *Linear search algorithm*.

*linear\_search.py*

```
def seq_search(arr, ele):
    for i in range(len (arr)):
        if (ele == arr[i]):
            print("Element found at index: ", i)
            break
    else:
        print("Oops! Element is not in the list.")
```

## 18. Binary Search

Binary search efficiently searches an element in a given list, provided that the list is already sorted.

*binary\_search.py*

```
#Recursion
def bin_search(arr, ele):
    if len(arr):
        mid = len(arr) // 2
        if (ele == arr[mid]):
            print("Element found.")
            return
        elif (ele < arr[mid]):
            bin_search(arr[:mid], ele)
        elif (ele > arr[mid]):
            bin_search(arr[mid+1:], ele)
    else:
        print("Element not found.")
        return

#Iteration
def bin_search (arr, ele):
    first = 0
    last = len (arr) - 1

    while (first <= last):
        mid = (first + last) // 2
        if (ele == arr[mid]):
            return True
        elif (ele < arr[mid]):
            last = mid-1
        elif (ele > arr[mid]):
            first = mid+1
    return False
```

## 19. Calculator

Simulate the functionality of a calculator for two operators and a single operand.

*calculator.py*

```
def calculate(num1, op, num2):
    if op == "+":
        return num1+num2

    elif op == "-":
        return num1-num2

    elif op == "*":
        return num1*num2

    elif op == "/":
        return num1/num2

    elif op == "//":
        return num1//num2

    elif op == "**":
        return num1**num2

if __name__ == "__main__":
    print ('Hit "Ctrl+C" to exit. \n')
    num1 = int (input ('Number: '))
    op = None

    while 1:
        op = input('Operator: ')
        num2 = int (input ('Number: '))
        num1 = calculate (num1, op, num2)
        print(f"Ans: {num1}")
```

## 20. Guess the Number

Write a script that randomly generates a number from 1 to 10 and allows you to guess that number within a certain number of tries. You need to import **random** module for this game which will randomly generate the secret number.

*guess.py*

```
import random

def play():
    secretNumber = random.randint(1,10)

    #Loop for three tries
    for i in range(4,1,-1):
        print("[Enter a number between 1 to 10] \n")
        playerNumber = int(input("\nTry your luck: "))

        if (playerNumber == secretNumber):
            print("\nYou were lucky this time \n")
            break

        elif (playerNumber > secretNumber):
            print("\nHard Luck! [ Hint: Try a smaller number ]")
            print("Tries left: ", i-2)

        elif (playerNumber < secretNumber):
            print("\nHard Luck! [ Hint: Try a higher value ]")
            print("Tries left: ", i-2)

    else:
        print("You're not entering a valid value")

    #if the number of tries are exhausted
    else:
        print("\nThe number secret number was: ", secretNumber)
```

```
playerName = input("If you're lame, then don't Enter your Name:\n")

print(f"\nWelcome {playerName}! to 'Guess the Number Game' \n")

play()

while True:
    a = input("\nWanna try again?(y/n)\n ")

    if (a.lower() == "yes" or a.lower() == 'y'):
        play()

    else:
        print("Such a loser!")
        break
```

## 21. Tic Tac Toe

This is your first milestone project. You know how tic tac toe is played, right? Before directly going for the given source code try to solve this by yourself. Remember errors are your best friends, so don't feel bad whenever they greet you. You can break this large problem into smaller chunks and try to convert those chunks into functions or classes, whatever you're comfortable with.

As I mentioned before, there are no hard and fast rules in programming. Your priority should be the solution, no matter how long your logic is. Once you have built a working model, you can then revisit your script and optimize your code.

**Note:** The code given below may not get properly indented, if you directly copy and paste it. Although the logic is perfectly fine, you may get indentation errors .

*TicTacToe.py*

```
from random import randint

#----- main starts -----#

print('Welcome to Tic Tac Toe!\n')

print(""" These are the locations on the board
+-----+-----+-----+
|     |     |     |
| 1   | 2   | 3   |
+-----+-----+-----+
|     |     |     |
| 4   | 5   | 6   |
+-----+-----+-----+
|     |     |     |
| 7   | 8   | 9   |
+-----+-----+-----+
""")

name1 = input('P1 Name: ')
name2 = input('P2 Name: ')

print(f"\n{name1}! You're 'X'. {name2}! You're 'O'.\n")
print("The luckier one goes first.")

B = [ ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']

combinations = [(0,1,2), (3,4,5), (6,7,8), (0,3,6), (1,4,7),
(2,5,8), (0,4,8), (2,4,6)]

#----- function definitions -----#
```

```
#Board
def display_board():
    print("""
|-----|-----|-----|
|   {}   |   {}   |   {}   |
|-----|-----|-----|
|   {}   |   {}   |   {}   |
|-----|-----|-----|
|   {}   |   {}   |   {}   |
|-----|-----|-----|
\n""".format(B[0], B[1], B[2], B[3], B[4], B[5], B[6], B[7],
B[8],))

#player one
def player_one():
    print (f"\n{name1}! Select your location: ")

    location1 = int (input () )

    B[location1-1] = "X"

    display_board()

#player two
```

```
def player_two():
    print(f"\n{name2}! Select your location: ")
    location2 = int(input())
    B[location2-1] = "O"
    display_board()

#Toss
def random_player(name1, name2):
    toss = randint(1,2)

    if (toss == 1):
        print(f"{name1} is luckier, this time!")
        return "X"

    elif (toss == 2):
        print(f"{name2} is luckier, this time!")
        return "O"

#winner
def check_winner():
    for (a,b,c) in combinations:
        if (B[a] == "X" and B[b] == "X" and B[c]== "X"):
            return 1

        elif (B[a] == "O" and B[b] == "O" and B[c]== "O"):
            return 2

    else:
        continue

#-----End of functions-----#
#----- main continues -----#
who_goes_first = random_player(name1, name2)
```

```
for i in range (5):

    if (who_goes_first == "X"):
        #if player 1 wins the toss

            player_one()
            result = check_winner()

        if (result == 1):
            print (f"Congratulations {name1}! You won the game.")
            break

    elif (" " in B):
        player_two()
        result = check_winner()

        if (result == 2):
            print(f"Congratulations {name2}! You won the game")
            break

    else:
        print ("Draw")
        break

elif (who_goes_first == "O"):
    #if player 2 wins the toss

        player_two ()
        result = check_winner()

    if (result == 2):
        print (f"Congratulations {name2}! You won the game")
        break
```

```
elif (" " in B):
    player_one()
    result = check_winner()

    if (result == 1):
        print(f"Congratulations {name1}! You won the game")
        break

    else:
        print("Draw! ")
        break

#It executes only when the for loop runs completely!
else:
    print("Draw!")
```

---

## 22. BrainFuck Interpreter

BrainFuck is an Esolang, a kind of pseudo programming language you can say. It recognizes only 8 operators [ ] < > , . + - everything else is ignored by its interpreter i.e. all other characters except these eight are nothing but comments.

- , is used for input
- . is used for output (input and output is only ASCII characters)
- < is used to move the pointer left
- > is used to move the pointer right
- + is used to increment the value the pointer is pointing to
- - is used to decrement the value the pointer is pointing to
- [] is used for looping

You can explore some of the BrainFuck visualizers which are available online. Here is a link to one of the BrainFuck visualizers: <https://fatiherikli.github.io/brainfuck-visualizer/>. The interpreter of BrainFuck beautifully interprets everything based on ASCII values.

The challenge for you is to make an interpreter that can interpret the code written in BrainFuck and display the output accordingly.



# 1. Data Structures

You've already seen some of the data structures that are built into Python such as *dictionaries*. There are a lot of data structures, each data structure is meant for a particular problem. This book is not a data structure and algorithm book, so the theory of different data structures is not discussed here. However, you can find the implementation of some important data structures in this book.

## 1. Stacks

*stacks.py*

```
class stack():
    def __init__(self):
        self._A = []

    def __len__(self):
        return len(self._A)

    def push(self, ele):
        self._A.append(ele)

    def pop(self):
        if self.isEmpty():
            return ("Stack is empty!")
        return self._A.pop()

    def isEmpty(self):
        return self._A == []

    def top(self):
        if self.isEmpty():
            return ("Stack is empty!")
        return self._A[-1]
```

```
a = stack()
print (f'a.isEmpty(): {a.isEmpty()}'')
print (f'a.pop(): {a.pop()}'')
print (f'a.top(): {a.top()}'')
a.push(32)
print (f'a.top(): {a.top()}'')
print (f'a.pop(): {a.pop()}'')
print (f'a.isEmpty(): {a.isEmpty()}'')
```

## 2. Queues

*queues.py*

```
class Queue():
    def __init__(self):
        self._A = []

    def __len__(self):
        return len(self._A)

    def enqueue(self, ele):
        self._A.append(ele)

    def dequeue(self):
        if self.isEmpty():
            return ("Empty!")
        return self._A.pop(0)

    def isEmpty(self):
        return self._A == []

    def front(self):
        if self.isEmpty():
            return ("Empty!")
        return self._A[0]
```

```

def rear (self):
    if self.isEmpty ():
        return ("Empty!")
    return self._A [-1]

a = Queue()
print (f'a.isEmpty(): {a.isEmpty()}' )
a.enqueue(10)
a.enqueue(20)
a.enqueue(30)
a.enqueue(40)
a.enqueue(50)
a.enqueue(60)
print (f'a.rear(): {a.rear()}' )
print (f'a.isEmpty(): {a.isEmpty()}' )
print (f'len(a): {len(a)}' )
print (f'a.dequeue(): {a.dequeue()}' )
print (f'a.front(): {a.front()}' )
print (f'len(a): {len(a)}' )

```

### 3. Deques

*deques.py*

```

class Deque ():
    def __init__ (self):
        self._A = []

    def __len__ (self):
        return len (self._A)

    def add_front (self,ele):
        self._A.insert (0, ele)

```

```
def add_rear (self,ele):
    self._A.append (ele)

def remove_front (self):
    if self.is_empty () :
        return ("Queue is Empty!")
    return self._A.pop (0)

def remove_rare (self):
    if self.is_empty () :
        return ("Queue is Empty!")
    return self._A.pop ()

def is_empty (self):
    return len (self._A) == 0

a = Deque()
print (f'a.is_empty(): {a.is_empty()}' )
print (f'len(a): {len(a)}')
a.add_front(10)
a.add_front(20)
a.add_front(30)
a.add_front(40)
print (f'a.remove_front(): {a.remove_front()}' )
a.add_rear(80)
print (f'a.remove_rare(): {a.remove_rare()}' )
print (f'a.remove_front(): {a.remove_front()}' )
print (f'a.remove_rare(): {a.remove_rare()}' )
print (f'a.remove_front(): {a.remove_front()}' )
print (f'a.remove_rare(): {a.remove_rare()}' )
print (f'a.remove_front(): {a.remove_front()}' )
```

## 4. Queue Using Stacks

*queue\_using\_stacks.py*

```
class stack():
    def __init__(self):
        self._A = []

    def __len__(self):
        return len(self._A)

    def push(self, ele):
        self._A.append(ele)

    def pop(self):
        if self.is_empty():
            return ("Empty!")
        return self._A.pop()

    def is_empty(self):
        return self._A == []

    def top(self):
        if self.is_empty():
            return ("Empty!")
        return self._A[-1]

    def display(self):
        for i in self._A[::-1]:
            print(i)

    def r_display(self):
        for i in self._A:
            print(i)
```

```
class Queue():
    push_stack = stack()
    pop_stack = stack()
    def enqueue(self, ele):
        if not self.pop_stack.is_empty():
            for i in range(len(self.pop_stack)):
                self.push_stack.push(self.pop_stack.pop())
        self.push_stack.push(ele)

    def dequeue(self):
        if not self.push_stack.is_empty():
            for i in range(len(self.push_stack)):
                self.pop_stack.push(self.push_stack.pop())
        return self.pop_stack.pop()

    def is_empty(self):
        return (self.push_stack.is_empty() and
self.pop_stack.is_empty())

    def __len__(self):
        return max(len(self.pop_stack), len(self.push_stack))

    def display(self):
        if not self.push_stack.is_empty():
            self.push_stack.display()
            return

        if not self.pop_stack.is_empty():
            self.pop_stack.r_display()
            return

    else:
        return "empty"
```

```
a = Queue()
a.enqueue(11)
a.enqueue(22)
a.enqueue(33)
a.enqueue(44)
a.enqueue(55)
a.display()
print(f'a.dequeue(): {a.dequeue()}')
a.enqueue(99)
a.display()
print(f'a.dequeue(): {a.dequeue()}')
print(f'a.dequeue(): {a.dequeue()}')
print(f'a.dequeue(): {a.dequeue()}')
a.enqueue(111)
a.display()
print(f'a.dequeue(): {a.dequeue()}')
print(f'a.dequeue(): {a.dequeue()}')
print(f'a.dequeue(): {a.dequeue()}')
print(f'a.dequeue(): {a.dequeue()}')
```

## 5. Linked Lists

*linked\_lists.py*

```
class Node():
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class SLL():
    def __init__(self, n):
        self.root = None
        self._count = n

    for i in range(n):
        new_node = Node (int (input (f"Node {i+1}: ")))

        if not self.root:
            self.root = new_node

        else:
            temp = self.root
            while (temp.next):
                temp = temp.next
            temp.next = new_node
            self.leaf = temp.next

    def __len__(self):
        return self._count

    def insert (self, data, pos):
        temp = self.root

        if (pos == 1):
            new_node = Node(data)
            new_node.next = self.root
            self.root = new_node
            self._count += 1

        elif (1 < pos <= self._count):
            # pos-2:
            # first, you have to stop before that node
            # second, temp is already at root node
```

```
for _ in range(pos-2):
    temp = temp.next
new_node = Node(data)
new_node.next = temp.next
temp.next = new_node
self._count += 1

elif (pos == (self._count + 1)):
    temp = self.root
    for _ in range (pos-2):
        temp = temp.next
    new_node = Node(data)
    temp.next = new_node
    self.leaf = new_node
    self._count += 1

else:
    print ("Invalid position")

def delete (self, pos):
    temp = self.root

    if (pos == 1):
        temp = self.root
        self.root = self.root.next
        del (temp)
        self._count -= 1

    elif (1 < pos < self._count):
        temp = self.root
        for _ in range(pos-2):
            temp = temp.next
        temp2 = temp.next
        temp.next = temp.next.next
```

```
del (temp2)
self._count -= 1

elif (pos == self._count):
    temp = self.root
    for _ in range (pos-2):
        temp = temp.next
    temp2 = temp.next
    temp.next = None
    del (temp2)
    self.leaf = temp
    self._count -= 1

else:
    print ("Invalid position")

def display(self):
    temp = self.root

    while temp:
        print (f"|{temp.data}|", end = " ")
        temp = temp.next


n = int(input("Enter the no. of nodes: "))
a = SLL(n)
a.display()
print(f"\nRoot: {a.root.data} \nLeaf: {a.leaf.data}")
print("You can also perform insert and delete operations")
```

## 6. Hash Tables

The actual structure of a hash table is very complicated. However, the basic idea of implementation remains the same. The following program is the simplest implementation of hash tables where only integers can act as keys.

```
hash_table.py

class HashTable():
    def __init__(self, size):
        self.size = size
        self.data = [None] * self.size
        self.key = [None] * self.size

    def put(self, key, data):
        index = self.hash_function(key)

        #insert
        if not self.key[index]:
            self.key[index] = key
            self.data[index] = data
            print(f"{data} added!")
            return

        #update
        if self.key[index] == key:
            self.data[index] = data
            print(f"{key} updated!")
            return

        #rehashing (handles collision)
        next_index = self.rehash(index)

        while (next_index != index):
            #insert
            if not self.key[next_index]:
```

```
        self.key[next_index] = key
        self.data[next_index] = data
        print (f"{data} added!")
        return

#update
if self.key[next_index] == key:
    self.data[next_index] = data
    print (f"{key} updated!")
    return

#rehashing
next_index = self.rehash (next_index)

#list_is_full
print ("list is full")
return

def get (self, key):
    index = self.hash_function (key)
    if self.key[index] == key:
        return self.data[index]

    next_index = self.rehash (index)
    while (next_index != index):
        if self.key[next_index] == key:
            return self.data [next_index]
        next_index = self.rehash (next_index)

    return False

def hash_function (self, key):
    size = len (self.key)
    return key % size
```

```
def rehash (self, old_hash):
    size = len (self.key)
    return (old_hash+1) % size

def __getitem__ (self, key):
    return self.get (key)

def __setitem__ (self, key, data):
    self.put (key, data)

def __contains__ (self, n):
    return n in self.data

def __iter__ (self):
    return iter (self.key)

h = HashTable(10)
print ('\n')
h[1] = "Apple"
h[2] = "Banana"
h[4] = "Cherry"
h[6] = "Grapes"
h[8] = "Lemon"
print ('\n')
h[8] = "Orange"
print ('\n')
h[258] = 8
print ('\n')

for i in h:
    if i:
        print (h[i])
```

## 7. Binary Search Trees

*bst.py*

```
class Node():
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST():
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)
        print(f"{key} inserted!")

    def _insert(self, node, key):
        if not node:
            node = Node(key)

        if (key < node.key):
            node.left = self._insert(node.left, key)

        if (key > node.key):
            node.right = self._insert(node.right, key)

        return node

    def inorder(self):
        if self.root:
            self.root = self._inorder(self.root)
```

```
def _inorder (self, node):
    if node.left:
        node.left = self._inorder (node.left)
    print (node.key)

    if node.right:
        node.right = self._inorder (node.right)
    return node

def min_value (self):
    if self.root:
        return self._min (self.root)

def _min(self, node):
    if node.left:
        return self._min (node.left)
    return node.key

def max_value (self):
    if self.root:
        return self._max (self.root)

def _max (self, node):
    if node.right:
        return self._max( node.right)
    return node.key

def find (self, val):
    if self.root:
        self._find (val, self.root)
```

```
def _find (self, val, node):
    if node:
        if (val < node.key):
            self._find (val, node.left)

        elif (val > node.key):
            self._find (val, node.right)

        elif (val == node.key):
            print (f"Yes {val} is there!")

    else:
        print (f"{val} is not present in this tree")

def remove (self, key):
    self.root = self._remove (key, self.root)

def _remove (self, key, node):
    if not node:
        print ("The key you entered is not in the tree!")
        return node

    if (key < node.key):
        node.left = self._remove (key, node.left)

    elif (key > node.key):
        node.right = self._remove (key, node.right)

    elif (key == node.key):
        if (not node.left) and (not node.right):
            del (node)
            return None
```

```
if not node.right:
    temp = node.left
    del (node)
    return temp

elif not node.left:
    temp = node.right
    del (node)
    return temp
temp = self._pred (node.left)
node.key = temp.key
node.left = self._remove (temp.key, node.left)

return node

def _pred (self, node):
    if node.right:
        return self._pred (node.right)
    else:
        return node

a = BST ()

a.insert (56)
a.insert (40)
a.insert (60)

a.remove (56)
a.remove (40)

a.inorder()
```

## 8. AVL Trees

*avl.py*

```
class Node():
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 0

    def __del__(self):
        print(f"{self.data} is removed from tree")

class AVL():
    def __init__(self):
        self.root = None

    # None ---> -1
    def get_height(self, node):
        if not node:
            return -1
        return node.height

    # if BF > 1 ---> left heavy ---> right rotation
    # if BF < -1 ---> right heavy ---> left rotation

    def calc_balance_factor(self, node):
        if not node:
            return 0
        return (self.get_height(node.left) - self.get_height(
            node.right))
```

```
def rotate_right (self, node):
    print (f"Rotating to right on node {node.data}")

    tempL = node.left
    t = tempL.right
    tempL.right = node
    node.left = t

    node.height = max (self.get_height (node.left),
self.get_height(node.right) ) + 1
    tempL.height = max (self.get_height (tempL.left),
self.get_height(tempL.right) ) + 1

    return tempL

def rotate_left (self, node):
    print (f"rotating to left on node {node.data}")

    tempR = node.right
    t = tempR.left
    tempR.left = node
    node.right = t

    node.height = max (self.get_height(node.left),
self.get_height(node.right) ) + 1
    tempR.height = max (self.get_height(tempR.left),
self.get_height(tempR.right) ) + 1

    return tempR

def insert (self, data):
    self.root = self._insert (data, self.root)
```

```
def _insert (self, data, node):
    if not node:
        return Node (data)

    if (data < node.data):
        node.left = self._insert (data, node.left)

    if (data > node.data):
        node.right = self._insert (data, node.right)

    node.height = max (self.get_height(node.left),
self.get_height(node.right) ) + 1
    return self.check_AVL_property (data, node)

def check_AVL_property (self,data, node):
    BF = self.calc_balance_factor (node)

    if ( BF > 1 and data < node.left.data ):
        print ("left-left heavy situation")
        return self.rotate_right (node)

    if ( BF < -1 and data > node.right.data ):
        print ("right-right heavy situation")
        return self.rotate_left (node)

    if (BF > 1 and data > node.left.data):
        print ("left-right heavy situation")
        node.left = self.rotate_left (node.left)
        return self.rotate_right (node)

    if (BF < -1 and data < node.right.data):
        print ("right-left heavy situation")
        node.right = self.rotate_right (node.right)
        return self.rotate_left (node)
```

```
    return node

def delete (self, data):
    if self.root:
        self.root = self._delete (self.root, data)

def _delete (self, node, data):
    if not node:
        return node

    if (data < node.data):
        node.left = self._delete (node.left, data)

    if (data > node.data):
        node.right = self._delete (node.right, data)

    if (data == node.data):
        if not node.left and not node.right:
            del (node)
            return None

        if not node.right:
            temp = node.left
            del (node)
            return temp

        if not node.left:
            temp = node.right
            del (node)
            return temp
```

```
if (node.left and node.right):
    pred = self.maximum (node.left)
    pred.data, node.data = node.data, pred.data
    node.left = self._delete (node.left, pred.data)

    node.height = max (self.get_height (node.left),
self.get_height(node.right)) + 1
    bf = self.calc_balance_factor (node)
    bf_l = self.calc_balance_factor (node.left)
    bf_r = self.calc_balance_factor (node.right)

    if (bf > 1) and (bf_l >= 0):
        return self.rotate_right (node)

    if (bf > 1) and (bf_l < 0):
        node.left = self.rotate_left (node.left)
        return self.rotate_right (node)

    if (bf < -1) and (bf_r <= 0):
        return self.rotate_left (node)

    if (bf < -1) and (bf_r > 0):
        node.right = self.rotate_right (node.right)
        return self.rotate_left (node)

    node.height = max (self.get_height(node.left),
self.get_height(node.right)) + 1
    return node

def maximum (self, node):
    if node.right:
        return self.maximum (node.right)
    return node
```

```
def inorder (self):
    if self.root:
        self._inorder (self.root)

def _inorder (self, node):
    if node.left:
        self._inorder (node.left)
    print (node.data)
    if node.right:
        self._inorder (node.right)

avl = AVL()
avl.insert(44)
avl.insert(17)
avl.insert(70)
avl.insert(8)
avl.insert(30)
avl.insert(78)
avl.insert(5)
avl.insert(20)
avl.insert(40)
avl.insert(75)
avl.insert(35)

print ('\n')
avl.inorder()
print ('\n')

avl.delete(5)
avl.delete(8)
avl.delete(20)
print ('\n')
avl.inorder()
print ('\n')
```

## 9. Graphs

*graph.py*

```
class Vertex():
    def __init__(self, key):
        self.key = key
        self.adjacent_vertices = {} # {Vertex(key): weight}

    def add_neighbor(self, v, w):
        self.adjacent_vertices[v] = w

    def get_adjacent_vertices(self):
        l = list(self.adjacent_vertices.keys())
        adj_vert_list = [x.key for x in l]
        return adj_vert_list

    def __str__(self):
        return self.key


class Graph():
    def __init__(self):
        self.vertlist = {} # {key: Vertex(key), ...}

    def add_vertex(self, key):
        self.vertlist[key] = Vertex(key)
        print(f"{key} added to graph!")
        return

    def get_vertex(self, key):
        return self.vertlist[key]
```

```
def add_edge(self, S, E, w=0):
    start = self.vertlist[S]
    end = self.vertlist[E]
    start.add_neighbor(end, w)
    print(f"{S} is connected to {E}")
    return

def get_vertices(self):
    return list(self.vertlist.keys())

def __iter__(self):
    return iter(self.vertlist.values())

def __contains__(self, n):
    return n in self.vertlist.keys()

g = Graph()
g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")
g.add_vertex("D")
g.add_vertex("E")
g.add_vertex("F")
g.add_vertex("G")
g.add_vertex("H")

print ('\n')
g.add_edge("A", "E", 3)
print ('\n')

g.add_edge("B", "C", 5)
g.add_edge("B", "E", 2)
g.add_edge("B", "F", 6)
```

```
print ('\n')
g.add_edge("C", "B", 3)
g.add_edge("C", "D", 5)
g.add_edge("C", "F", 2)
g.add_edge("C", "G", 5)
print ('\n')

g.add_edge("D", "G", 6)
g.add_edge("D", "H", 4)
g.add_edge("D", "F", 6)
g.add_edge("D", "C", 4)
print ('\n')

g.add_edge("E", "F", 4)
g.add_edge("E", "A", 4)
g.add_edge("E", "B", 4)
print ('\n')

g.add_edge("F", "B", 6)
g.add_edge("F", "E", 4)
g.add_edge("F", "D", 6)
g.add_edge("F", "C", 4)
print ('\n')

g.add_edge("G", "D", 6)
g.add_edge("G", "C", 4)
print ('\n')
g.add_edge("H", "D", 6)

d = g.get_vertex("D")
print ('\n')
print (d)
```

## 2. Algorithms

This book covers only the implementation of sorting algorithms and some famous graph traversal algorithms.

### 2.1 Sorting algorithms

#### 1. Bubble Sort

*bubble\_sort.py*

```
def bubble_sort (arr):
    n = len (arr)
    for pass_ in range (n-1):
        for j in range ((n-pass_) - 1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

def bubble_sort_2 (arr):
    n = len (arr)
    for i in range (n-1, 0, -1):
        for j in range (i):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr

def bubble_sort_optimized (arr):      #yen optimization
    n,swapping = len(arr), 1
    for i in range (n-1, 0, -1):
        if swapping:
            swapping = 0
            for j in range (i):
                if arr[j] > arr[j+1]:
                    swapping = 1
                    arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
```

---

## 2. Insertion Sort

*insertion\_sort.py*

```
def insertion_sort (arr):
    n = len (arr)

    for i in range (1, n):
        current_item = arr[i]
        position = i

        while (position > 0 and arr[position-1] > current_item):
            arr[position] = arr[position-1]
            position -= 1

        arr[position] = current_item
    return arr
```

## 3. Selection Sort

*selection\_sort.py*

```
def selection_sort (arr):
    n = len(arr)

    for i in range(n-1):
        i_max = 0

        for j in range(1,(n-i)):
            if arr[j] > arr[i_max]:
                i_max = j
        arr[i_max], arr[n-i-1] = arr[n-i-1], arr[i_max]

    return arr
```

```
def selection_sort (arr):
    n = len(arr)
    for i in range (n-1):
        min_ele = arr[i]
        for j in range(i+1, n):
            if arr[j] < arr[i]:
                min_ele = arr[j]
        temp = arr[j]
        arr[j] = arr[i]
        arr[i] = temp
    return arr
```

#### 4. Merge Sort

*merge\_sort.py*

```
def merge_sort (arr):
    n = len (arr)
    if (n > 1):
        mid = n//2
        l = arr[:mid]
        r = arr[mid:]

        print (f"array: {arr}")
        print (f"left subarray: {l}")
        print (f"right subarray: {r}")
        print (f"mid: {mid}")
        print ("\n")

        merge_sort (l)
        merge_sort (r)
        merge (l,r,arr)

    return arr
```

```

def merge (l,r,arr):
    i = j = k = 0
    while (i < len (l) and j < len (r)):
        if (l[i] < r[j]):
            arr[k] = l[i]
            i += 1
            k += 1
        else:
            arr[k] = r[j]
            j += 1
            k += 1

    while (i < len (l)):
        arr[k] = l[i]
        i += 1
        k += 1

    while (j < len(r)):
        arr[k] = r[j]
        j += 1
        k += 1

    print(f"Merger...\\n {l} {r}: {arr}")
    print("\\n")

```

## 5. Quick Sort

*quick\_sort.py*

```

def quick_sort (arr):
    n = len (arr)
    _quick_sort (arr, 0, n-1)
    return arr

```

```
#helper function
def _quick_sort(arr, first, last):
    if first < last:
        pivot_index = partition (arr, first, last)
        _quick_sort (arr, first, pivot_index-1)
        _quick_sort (arr, pivot_index+1, last)

def partition (arr, first, last):
    pivot = arr[first]
    l = first + 1
    r = last
    done = False

    while not done:
        while l <= r and arr[l] <= pivot:
            l += 1

        while l <= r and arr[r] >= pivot:
            r -= 1

        if l > r:
            done = True
        else:
            arr[l], arr[r] = arr[r], arr[l]

    arr[first], arr[r] = arr[r], arr[first]
    return r
```

---

## 6. Shell Sort

*shell\_sort.py*

```
def shell_sort (arr):
    n = len (arr)
    gap = n // 2

    while gap > 0:

        for start in range (gap):
            for i in range (start+gap, n, gap):
                current_item = arr[i]
                position = i
                while (position >= gap and arr[position-gap] >
current_item):
                    arr[position] = arr[position-gap]
                    position = position-gap
                arr[position] = current_item
            gap ///= 2

    return arr
```

## 2.2 Graph traversal algorithms

### 1. Breadth First Search and Depth First Search

*bfs\_dfs.py*

```
class Node():
    def __init__(self, name):
        self.name = name
        self._adjacent_vertices = []
        self.visited = False
        self.predecessor = None

    def add_vertex(self, node):
        self._adjacent_vertices.append(node)

def bfs(starting_vertex):
    queue = [starting_vertex]
    starting_vertex.visited = True

    while queue:
        node = queue.pop(0)
        print(node.name)
        for i in node._adjacent_vertices:
            if not i.visited:
                i.visited = True
                queue.append(i)

def dfs(starting_vertex):
    starting_vertex.visited = True
    print(starting_vertex.name)

    for i in starting_vertex._adjacent_vertices:
        if not i.visited:
            dfs(i)
```

```
node_1 = Node("A")
node_2 = Node("B")
node_3 = Node("C")
node_4 = Node("D")
node_5 = Node("E")

node_1.add_vertex(node_2)
node_1.add_vertex(node_5)
node_2.add_vertex(node_3)
node_3.add_vertex(node_4)
node_4.add_vertex(node_5)

print('Breadth First Search:')
bfs(node_1)

node_1 = Node("A")
node_2 = Node("B")
node_3 = Node("C")
node_4 = Node("D")
node_5 = Node("E")

node_1.add_vertex(node_2)
node_1.add_vertex(node_3)
node_2.add_vertex(node_4)
node_2.add_vertex(node_5)

print('\nDepth First Search:')
dfs(node_1)
```

## 2. Dijkstra's Shortest Path Algorithm

*dijkstra.py*

```
import math

class Vertex():
    def __init__(self, key):
        self.key = key
        self.adjacent_vertices = {} # {Vertex(key): weight}
        self.visited = False
        self.dist = math.inf
        self.parent = None

    def add_neighbor(self, v, w):
        self.adjacent_vertices[v] = w

    def get_weight(self, v):
        return self.adjacent_vertices[v]

    def get_adjacent_vertices(self):
        l = list(self.adjacent_vertices.keys())
        adj_vert_list = [x.key for x in l]
        return adj_vert_list

class Graph():
    def __init__(self):
        self.vertlist = {} # {key: Vertex(key), ...}

    def add_vertex(self, key):
        self.vertlist[key] = Vertex(key)
        print(f"{key} added to graph!")
        return

    def get_vertex(self, key):
        return self.vertlist[key]
```

```
def add_edge(self, S, E, w=0):
    start = self.vertlist[S]
    end = self.vertlist[E]
    start.add_neighbor(end, w)
    print(f"{S} is connected to {E}")
    return

def get_vertices(self):
    return list(self.vertlist.keys())

def __iter__(self):
    return iter(self.vertlist.values())

def __contains__(self, n):
    return n in self.vertlist.keys()

def dijkstra(graph, source, target):
    source.dist = 0
    q = []      # h = Heap()
    for v in graph.vertlist.values():
        q.append(v)    # h.insert(v)

    while q:
        u = min_node(q)# u = h.del_min()
        q = dequeue(q, u)

        for v in u.adjacent_vertices:
            if (u.dist + u.get_weight(v)) :
                v.dist = u.dist + u.get_weight(v)
                v.parent = u
getShortestPathTo(target)
```

```
def getShortestPathTo(target):
    print(f"Shortest path cost: {target.dist} units")
    while target:
        print(target.key)
        target = target.parent

def min_node(q):
    vertices = [i.key for i in q]
    min_ele = min(vertices)
    for i in q:
        if i.key == min_ele:
            return i

def dequeue(q, u):
    for i in q:
        if i.key == u.key:
            q.remove(i)
    return q

g = Graph()
g.add_vertex("A")
g.add_vertex("B")
g.add_vertex("C")
g.add_vertex("D")
g.add_vertex("E")
g.add_vertex("F")
print ('\n')
g.add_edge("A", "B", 2)
g.add_edge("A", "C", 4)
g.add_edge("B", "E", 7)
g.add_edge("B", "C", 1)
g.add_edge("C", "D", 3)
g.add_edge("D", "E", 2)
```

```

g.add_edge("D", "F", 5)
g.add_edge("E", "F", 1)
source = g.get_vertex("A")
target = g.get_vertex("F")
print('\n')
dijkstra(g, source, target)

```

### 3. Bellman-Ford Algorithm

*bellman.py*

```

import math

class Node():
    def __init__(self, name):
        self.name = name
        self.adj_list = []
        self.visited = False
        self.parent = None
        self.min_dist = math.inf

    class Edge():
        def __init__(self, S, E, w):
            self.start = S
            self.end = E
            self.weight = w

def bellman_ford(vert_list, edge_list, source, destination):
    source.min_dist = 0

    for i in range (len(vert_list)-1):
        for edge in edge_list:
            u = edge.start
            v = edge.end
            dist = u.min_dist + edge.weight

```

```
if dist < v.min_dist:
    v.min_dist = dist
    v.parent = u

for edge in edge_list:
    if (edge.start.min_dist + edge.weight) < edge.end.min_dist
    :
        print("Negative weighted cycle detected")
        return

node = destination
while node:
    print(node.name)
    node = node.parent
print(f"\nShortest path cost: {destination.min_dist} units")

node_1 = Node("A")
node_2 = Node("B")
node_3 = Node("C")
node_4 = Node("D")
node_5 = Node("E")
node_6 = Node("F")
node_7 = Node("G")
A_B = Edge(node_1, node_2, 6)
A_C = Edge(node_1, node_3, 5)
A_D = Edge(node_1, node_4, 5)
B_E = Edge(node_2, node_5, -1)
C_B = Edge(node_3, node_2, -2)
C_E = Edge(node_3, node_5, 1)
D_C = Edge(node_4, node_3, -2)
D_G = Edge(node_4, node_7, -1)
E_F = Edge(node_5, node_6, 3)
G_F = Edge(node_7, node_6, 3)
node_1.adj_list.append(A_B)
```

```
node_1.adj_list.append(A_C)
node_1.adj_list.append(A_D)
node_2.adj_list.append(B_E)
node_3.adj_list.append(C_B)
node_3.adj_list.append(C_E)
node_4.adj_list.append(D_C)
node_4.adj_list.append(D_G)
node_5.adj_list.append(E_F)
node_7.adj_list.append(G_F)

vert_list = [node_1, node_2, node_3, node_4, node_5, node_6,
node_7]

edge_list = [A_B, A_C, A_D, B_E, C_B, C_E, D_C, D_G, E_F, G_F]

bellman_ford(vert_list, edge_list, node_1, node_6)
print('\n\nXYZ graph:')
node11 = Node("X")
node22 = Node("Y")
node33 = Node("Z")

E1 = Edge(node11, node22, 5)
E2 = Edge(node22, node33, 3)
E3 = Edge(node33, node11, -10)

node11.adj_list.append(E1)
node22.adj_list.append(E2)
node33.adj_list.append(E3)

v = [node11, node22, node33]
e = [E1,E2,E3]
bellman_ford(v, e, node11, node33)
```

## Glossary

**Annotation:** A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a type hint.

**Argument:** A value passed to a function (or method) when calling the function.

**Attribute:** A value associated with an object which is referenced by name using dotted expressions.

**Binary file:** A file object able to read and write bytes-like objects. Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), sys.stdin.buffer, sys.stdout.buffer, and instances of io.BytesIO and gzip.GzipFile.

**class:** A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**class variable:** A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

**Complex number:** An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part.

**Decorator:** A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

**Dictionary:** An associative array, where arbitrary keys are mapped to values.

**Dictionary view:** The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list, use `list(dictview)`.

**Docstring:** A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module.

**Duck-typing:** A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used “If it

looks like a duck and quacks like a duck.” By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution.

**EAFP:** Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many try and except statements.

**Expression:** A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as if. Assignments are also statements, not expressions.

**f-string:** String literals prefixed with 'f' or 'F' are commonly called “f strings” which is short for formatted string literals.

**File object:** An object exposing a file-oriented API (with methods such as read() or write()) to an underlying resource.

**Floor division:** Mathematical division that rounds down to the nearest integer. The floor division operator is //.

**Function:** A series of statements which returns some value to a caller. It can also be passed zero or more arguments which may be used in the execution of the body.

**Generator:** A function which returns a generator iterator. It looks like a normal function except that it contains yield expressions for producing a series of values usable in a *for-loop* or that can be retrieved one at a time with the *next()* function. Usually refers to a generator function, but may refer to a generator iterator in some contexts. In cases where the intended meaning isn’t clear, using the full terms avoids ambiguity.

**Generator Iterator:** An object created by a generator function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator iterator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

---

**Generator Expression:** An expression that returns an iterator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function.

**IDLE:** An Integrated Development Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

**Immutable:** An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

**Interpreted:** Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly.

**Iterable:** An object capable of returning its members one at a time. Examples of iterables include all sequence types such as list, str, and tuple and some non-sequence types like dict, file objects, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements Sequence semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop.

**Iterator:** An object representing a stream of data. Repeated calls to the iterator's `__next__()` method or passing it to the built-in function `next()` return successive items in the stream. When no more data is available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted.

---

**lambda:** An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is *lambda [parameters]: expression*

**LBYL:** Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the EAFP approach and is characterized by the presence of many if statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between “the looking” and “the leaping”. For example, the code, if key in mapping: return mapping[key] can fail if another thread removes key from mapping after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

**list:** A built-in Python sequence.

**list comprehension:** A compact way to process all or part of the elements in a sequence and return a list with the results.

**Mapping:** A container object that supports arbitrary key lookups and implements the methods specified in the Mapping or MutableMapping abstract base classes. Examples include *dict*, *collections.defaultdict*, *collections.OrderedDict* and *collections.Counter*.

**Method:** A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called *self*).

**Module:** An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of importing.

**Mutable:** Mutable objects can change their value but keep their id.

**Namespace:** The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods).

**Nested scope:** The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local

variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The nonlocal allows writing to outer scopes.

**Object:** Any data with state (attributes or value) and defined behavior (methods).

**Package:** A Python module which can contain submodules or recursively, subpackages.

**Parameter:** A named entity in a function (or method) definition that specifies an argument (or in some cases, arguments) that the function can accept.

**PEP:** Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features. PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

**Pythonic:** An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement.

**Sequence:** An iterable which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `ranges`.

**Slice:** An object usually containing a portion of a sequence. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses slice objects internally.

**Special method:** A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in special names.

**Statement:** A statement is part of a suite (a “block” of code). A statement is either an expression or one of several constructs with a keyword, such as `if`, `while` or `for`.

**Text file:** A file object able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the text encoding automatically.

---

Examples of text files are files opened in text *mode('r' or 'w')*, *sys.stdin*, *sys.stdout*, and instances of *io.StringIO*.

**Triple-quoted string:** A string which is bound by three instances of either a quotation mark ("") or an apostrophe (''). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

**type:** The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its *\_\_class\_\_* attribute or can be retrieved with *type(obj)*.

**Zen of Python:** Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing “*import this*” at the interactive prompt.

[source: pydocs]

## What next?

Now that you've successfully completed this book, you should be proud of yourselves. You still have a lot to learn in programming but at least you can write basic coding scripts. I just want to give you one piece of advice, do not get stuck in tutorial hell.

Well, a tutorial hell is when you keep on watching tutorials, reading articles one after another and you feel like you're some computer wizard but when you start working on your own projects, you realize that you actually don't know anything. The only way to learn programming is to practice as much as you can.

If you're preparing for placements, you should try online coding platforms like Hackerrank, codechef, geeksforgeeks, solelearn, hackerearth and a lot more. My personal favorite is leetcode, though I've not solved more than 30 questions on this platform, still I consider it the best online coding platform.

If you don't like competitive coding, then you're like me. You should practice your skills by doing some real life projects. By projects, I didn't mean huge industry level projects, a simple email extractor can be one of your projects, renaming all your files recursively with a single script, converting .jpg files to .png files and so on. Just be consistent.

If you ever get stuck at some point, take references from stack overflow or some other online coding communities. You should join GitHub where you can explore other people's projects. You can also join various coding discord servers or other coding clubs.

You can join my Python coding club, named Pyrates, which is functional across the valley under NIT Srinagar.