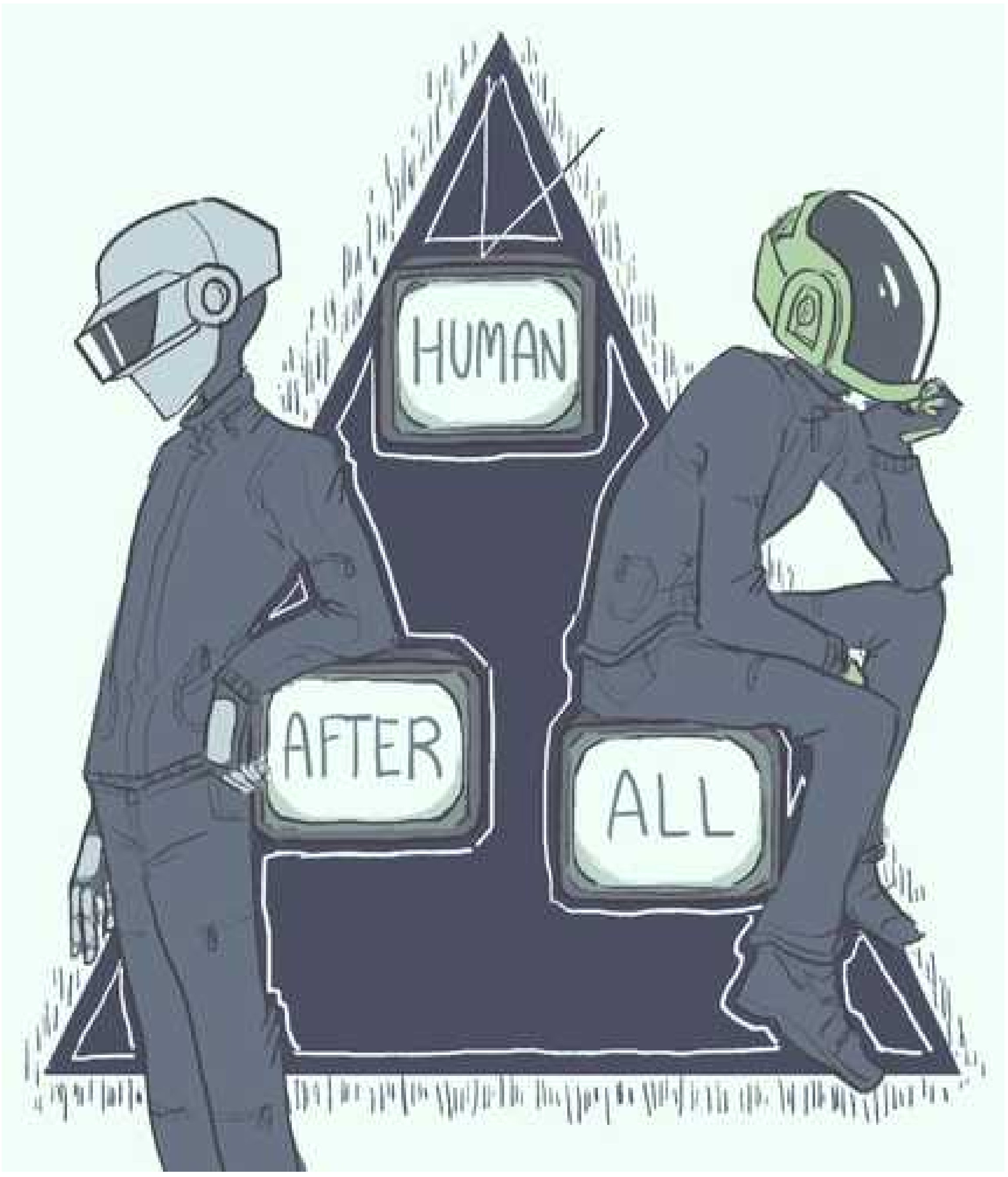
THE WINDS OF PYTHON

For the programming is dark and full of errors

IZAN MAJEED

The Winds of Python

For programming is dark and full of errors



Izan Majeed

Dedicated to my Parents, of course

Abu-G and Sister-G

About the author

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

Table of Contents

Chapter 2: Let Me Flow

Chapter 3: We Do Not Repeat

Chapter 4: Better an OOPs than a what if

Chapter 5: File and Regex

Grow through what you go through

IMPORT OS
CUR_PATH = OS.GETCWD()
IGNORE_SET = SET(["__INIT__.PY", "COUNT_SOURCELINES.PY"])
YEAH, HE'S A PARSER-TONGUE.
I DIDN'T KNOW HARRY SPOKE PYTHON.

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 completion of this book.

If you know more or less of some other programming language, Pythonic code will seem like pseudocode and you can extend your programming skills to next the 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 end up cramming definitions.

Though this book does contain some official definitions, copied from official documentation, but 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 Hitchhikers informal guide to Python.*

What’s special about this book?

Nothing

What is Python?

Computers are dumb machines. They may evaluate complex mathematical calculations within fractions of seconds but without any instructions, these machines are literary 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, it 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 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 (I personally hate this name).

Why Python?

Cause you choose it.

Downloading and Installing Python

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 3.7.3, 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 these commands in terminal; press Ctrl+Alt+T to quickly open the terminal:

```
sudo apt-get install python3
sudo apt-get install idle3
```

and boom! Python is installed as if some magical Pythonic dust is sprinkled over your machine.

IDLE

Once Python is installed successfully, you should be able to open **IDLE**, irrespective of operating system running on your machine.

IDLE stands for *Integrated Development 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 3.7.3 (v3.7.3:2c5fed8, Mar 27 2019, 22:11:17)  [MSC v.1900 64 bit (Intel)] on win64
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 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.')
```

IDLE

```
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:
...
```

... This 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 on 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.

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 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 gramma

Python
Is Coming!

1. Ghosts Are Awesome

Ghosts are the statements that are invisible to the interpreter. A *hash* (#) character transforms a 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 effect 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*".

ghost.py

```
# This is my first script
print ('Just playing with ghosts.')          # Hehe, this comment's just for fun
# Script just ended :(
```

```
Just playing with ghosts.
```

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

ghost.py

```
''' Ghosts are everywhere
Close triple quotes
And the magical spell will work'''
print ('Why am I scared?')
```

```
Why am I scared?
```

2. Numb the Numbers

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.

```
>>> 22 * 7
154
>>>
```

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

You can easily check the type of a number using Python's built-in ***type(object)*** function, which returns the type of object passed.

1. Integers (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 (7-4j)
<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.

```
>>> 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 BODMAS rule. If more than one operators of 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 to 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 )
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 I 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 every-time interpreter throws an error;
more the 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 can play around with these built-in functions.

1. abs (x)
Returns the absolute value of a number; if 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 algebra.
It’s a container in which you store a value. 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 see 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.

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 having same value can be created in a single line of code.

```
>>> 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 i*nt*, *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 allows you to assign values of different types to a same variable.

```
>>> x = 7
>>> type (x)
<class 'int'>
>>> x = 3.14
>>> type (x)
<class 'float'>
>>> x = 3 + 2j
>>> type (x)
<class 'complex'>
>>> x
(3+2j)
>>>
```

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

Syntax:

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

Number of variables on Left Hand Side must match the number of values on 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 multiple assignment, 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 in 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 C*hapter IV* 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 combination of double and single quotes for a string that contains single or double quotes inside its text, else SyntaxError will be thrown by 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 string which is bound by three instances of either a quotation mark (“) or an apostrophe (‘) 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 will be 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 **...** when entering a multi-line construct.

4.2 String Concatenation

String concatenation allows you to combine two or more strings together with 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

```
>>> 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 one by one for better understanding.

If you put the 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 = "Horizonta\ttab"
>>> print (hor_tab)
Horizonta     tab
>>>
```

If backslash (\) is not followed by any character, interpreter will through 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
>>>
>>> print ('\ ')
\
>>>
```

Remember that Python uses the backslash as an escape sequence in string literals; if the escape sequence isn’t recognized by 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.
>>>
```

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 as *raw strings*.

```
>>> normal_string = "cant ignore \n and other escape characters."
>>> print (normal_string)
cant ignore
 and other escape characters.
>>>
>>> raw_string = r"Ignores \n and all escape characters."
>>> print (raw_string)
Ignores \n and all escape characters.
>>>
>>> print (r'not a new \n line.')
not a new \n line.
>>>
```

To print escape characters in normal strings, backslash should be repeated *twice*. This can be confusing, 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.
>>>
```

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()*.

```
>>> str (159)
'159'
>>> str (3.14)
'3.14'
>>>
```

There is a similar function *repr()* which also return string representation of an object.

```
>>> 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 name or simple strings to it. Multiple values are separated by a comma (,).

```
>>> print ('String to be printed.')
String to be printed.
>>> lang = 'Python'
>>> version = 3.7
>>> print (lang, version)
Python 3.7
>>>
>>> print ('Current version of', lang, 'is', version)
Current version of Python is 3.7
>>>
```

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 *input()* method always accepts data as string even if you enter any other type. You’ve to cast *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 the Python’s built-in help system; 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 (print)
Help on built-in function print in module builtins:

print (...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
>>>
```

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 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...))
```

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 {0}" .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 ("{b} wakes up the {n} 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 f string 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'
>>>
```

Attempting to use an index that is too large will result in an error.

```
>>> 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 substring.

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 and 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]
```

```
'yti'
>>> 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]
'Cruising the waves'
>>> text [::2]
'Cusn h ae'
>>>
>>> text [::-1]                               #trick to reverse a string
'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 the string results in 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 recondite information 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 makes our life a lot easier.

Methods are called with a dot (.) operator followed by the name of method. General form of a method is *object.method_name (arguments).*

You have already seen *str.format()* method of string objects. Let’s try some other methods.

1. str.capitalize ()
It returns a copy of string with its first character capitalized and the rest lowercased.

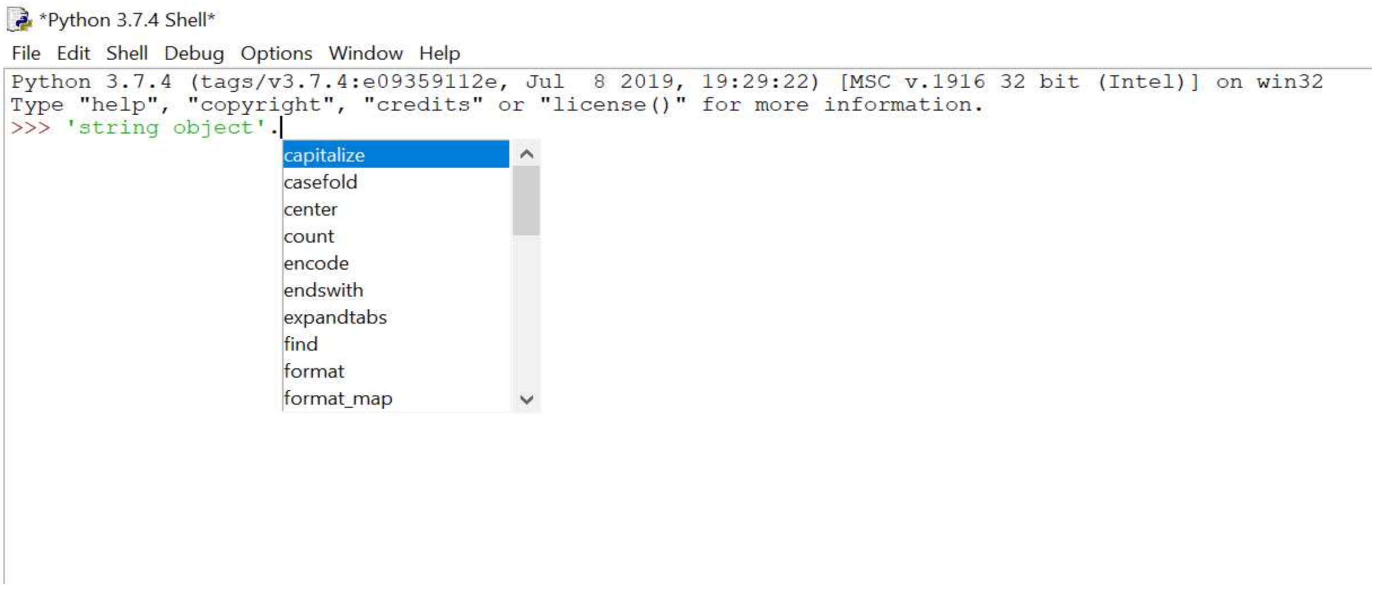
```
>>> 'hypermnesia'.capitalize ()
'Hypermnesia'
>>>
>>> greyjoy = 'wE dO nOt sOw'
>>> greyjoy.capitalize ()
'We do not sow'
>>> greyjoy
'wE dO nOt sOw'
>>>
```

2. str.title ()
It returns a titlecased copy of string i.e., first character of each word is uppercased and the remaining characters are lowercased.

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

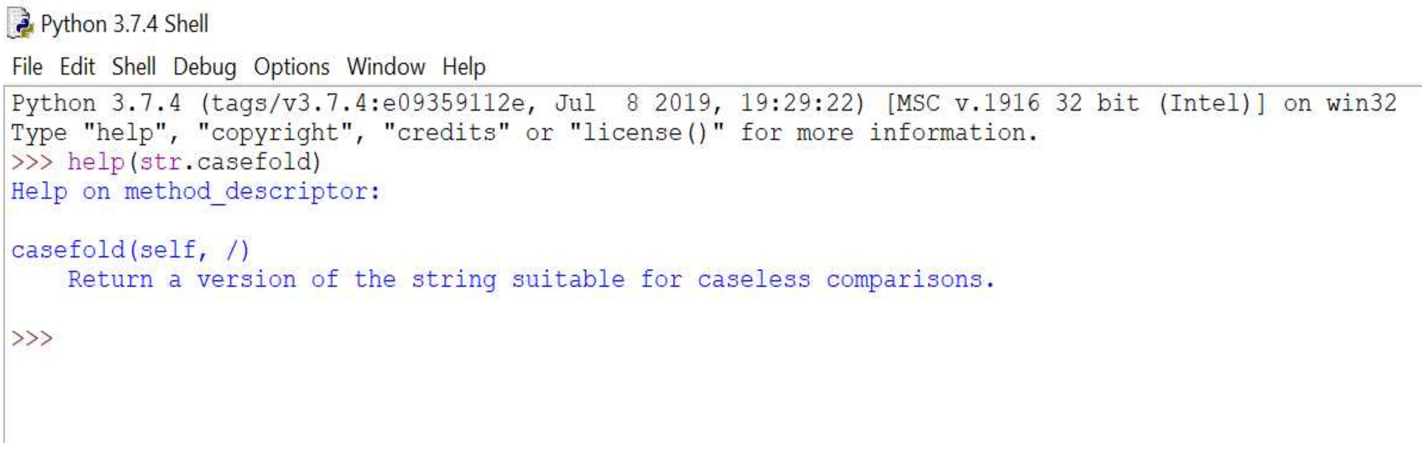
There are several other methods with which you can easily manipulate string objects. It’ll be a boring task to memorize all of these, try to understand how a particular method works and what can be achieved, rest of the job is done by *Tab* key.

The *Tab* key suggests all the methods that are available for a particular type of object. Hit the *Tab* key after dot (.) operator to see a list of all available methods.



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 'string object'.
```

You can then pass string methods to *help()* function as ***help (str.method_name)*** to see the description of that particular method.



```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> help(str.casefold)
Help on method_descriptor:

casefold(self, /)
    Return a version of the string suitable for caseless comparisons.

>>>
```

4. str.count (substring)

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

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

3. str.index (substring)

It returns the index of 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
>>>
```

4. str.lower ()

It returns a lowercased copy of string.

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

6. 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'
>>>
```

7. str.upper()

It returns an uppercased copy of string.

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

5. What about Lists?

List, as 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 and nor fixed type.

Array is another fancy name for list data type. However, for complex mathematical array operations, Python has *numpy arrays* to handle those operations easily.

```
>>> 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']
>>>
```

You can create a good looking list if you spread out the items i.e., one item per line. If you’re consistently adding and removing items it’s useful to put a comma at the end of the list as well.

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

5.1 Common Sequence Operations

Like strings and all other sequence types whether mutable or immutable, share certain common operations.

Table CSO

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

*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 new (shallow) copy of the list; will discuss *shallow* and *deep* copy later.

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

```
>>> basket [2:5]
['oranges', 'mangoes', 'bananas']
>>> basket [4]
'bananas'
>>>
```

2.1 Nesting

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

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

2.2 Lists are mutable

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

```
>>> nums = [0, 11, 22, 33, 44, 55]
>>> nums [2] = 99
>>> nums
[0, 11, 99, 33, 44, 55]
>>>
>>> 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]
```

2.3 *del* statement

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

```
>>> 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
>>>
```

3. *in* and *not* operators

These check if an item belongs to a given list and return either True or False (*boolean type*).

```
>>> 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
>>>
```

4. *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
>>>
```

5. *min()* and *max()*

Python’s built-in *min*() and *max*() functions allows us 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'
>>>
```

5.2 Strings and Lists

1. str.split()

Strings can be directly casted into a list using *list()* constructor. A better option 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 the words in the string, with an optional argument; if provided, string is split on the passed value and if omitted, string is split on whitespaces.

```
>>> local_host = '127.0.0.1'
>>> local_host.split()
['127.0.0.1']
>>> local_host.split('.')
['127', '0', '0', '1']
>>>
>>> emails = 'abc@gmail.com xyz@yahoo.com'
>>> emails.split()
['abc@gmail.com', 'xyz@yahoo.com']
>>> emails.split('@')
['abc', 'gmail.com xyz', 'yahoo.com']
>>>
>>> message = 'Humanxafterxall'
>>> message.split()
```

```
['Humanxafterxall']
>>> message.split('x')
['Human', 'after', 'all']
>>>
```

2. 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, I’ll discuss some of the commonly used methods. You can play around with other methods, 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 (new) copy of the list. It is equivalent to x[:]

```
>>> house_tully = ['Hoster', 'Edmure']
>>> deep_copy = house_tully
>>> shallow_copy = house_tully.copy()
>>>
>>> deep_copy.append ('Catelyn')
>>>
>>> house_tully
['Hoster', 'Edmure', 'Catelyn']
>>> shallow_copy
['Hoster', 'Edmure']
>>> deep_copy
['Hoster', 'Edmure', 'Catelyn']
>>>
>>> shallow_copy.append ('Lysa')
>>>
>>> shallow_copy
['Hoster', 'Edmure', 'Lysa']
>>> house_tully
['Hoster', 'Edmure', 'Catelyn']
>>> deep_copy
['Hoster', 'Edmure', 'Catelyn']
>>>
```

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* are iterable.

```
>>> houses = ['Stark']
>>> houses.extend('Targaryen')
>>> houses
['Stark', 'T', 'a', 'r', 'g', 'a', 'r', 'y', 'e', 'n']
>>> del houses[1:]
>>> houses
['Stark']
>>> 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 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 item from a specific index. If no argument is passed, removes the last item from 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.

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

```
[99, 22, 11, 33]
>>>
```

10. list.sort ()

It sorts a numeric list in ascending order and a string list in alphabetical order. This method is also *inplace* and returns nothing.

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

6. Tuples: Immutable lists

There is another standard *sequence* data type, the ***tuple***, which is very similar to *list* data type. Unlike lists, tuples are **immutable** i.e., you can’t modify the value of an item, once you’ve created a tuple of items.

Tuples are created using a pair of parentheses () with items separated by comma or using a trailing comma for a *singleton* tuple as ***a,*** or ***(a,)***.

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)
>>>
```

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

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

Tuples are not used as often as lists in programming.

6.1 Common Sequence Operations

As tuples are *sequence* type objects, they also follow all the operations mentioned earlier in *Table CSO*

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

Tuple items can be accessed by indexing or unpacking.

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

2.1 Nesting

```
>>> characters = ( ('Cercie', 'Jammie', 'Tyrion'), ('Arya', 'Rob', 'Bran') )
>>> characters [0][2]
'Tyrion'
>>> characters [1][0]
'Arya'
```

2.2 Tuples are immutable

Tuples are immutable sequences, i.e., item of a list can’t be changed.

```
>>> 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
>>>
```

2.3 del statement

As tuples are *immutable*, you can’t delete items from them.

```
>>> 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
>>>
```

3. *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
>>>
```

4. *len ()*

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

5. *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 key is not present, *KeyError* is thrown.

```
>>> empty_dict = {}
>>> type (empty_dict)
<class 'dict'>
>>>
>>> ginger = {'name': 'Yigrette', 'allegiance': 'Free Folk', 'season': 2}
>>> ginger
{'name': 'Yigrette', 'allegiance': 'Free Folk', 'season': 2}
>>> ginger ["name"]
'Yigrette'
>>> ginger [name]                    #name is string not 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 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}
>>>
```

Among sequence and mapping type, only *immutable* types (strings and tuples) can be used as dictionary *keys; values* can be of any type which means dictionaries can be nested only on values (dictionary within dictionary). 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 order of its 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 later.

```
>>> [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 object 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}
>>>
>>> dict ([('akh', 1), ('zi', 2), ('trai', 3)])
{'akh': 1, 'zi': 2, 'trai': 3}
>>>
```

7.4 *len (x)*

len() function returns the number of items in a dictionary.

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

7.5 Dictionary methods

Methods that are associated with dictionaries are:

1. dict.clear ()
Empty your dictionary with this method.

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

2. dict.copy ()
It returns a shallow (new) 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 key is in the dictionary, it returns its corresponding value. If 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')])
>>>
```

5. dict.keys ()

It provides a view on dictionary object’s keys.

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

6. dict.values ()

It provides a view on dictionary object’s values.

```
>>> 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 when the dictionary changes, the view reflects these changes.

These three methods are productive when used with *in, not in* operators and in looping statements (not covered yet). By default, dictionary object acts as *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
>>> 'allegiance' in regent.keys()                  # 'allegiance' in regent
True
>>>
>>> 'culture' in regent.items()
False
>>> {'name':'Daario'} in regent.items()
False
>>> ('name', 'Daario') in regent.items()      # tuple
True
>>>
```

7. dict.*pop (key)*
It remove 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 *key* is in the dictionary, it returns its value; if not, it adds the *key* with a value of *default*. Default value of *default* is *None*.

In other words, this method only adds new *key-value* pair and won’t update the key, if it already exits.

```
>>> 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 corresponding states as values, normal *print()* will display an ugly view of that dictionary.

Beautiful is better than ugly.

Luckily, Python comes with a module, named *pprint* which provides a *pprint()* function that can solve this problem of pretty printing. Not only dictionaries, gaint objects of other data structures like lists can also be pretty printed with this module.

```
>>> from pprint import pprint
>>>
>>> us_capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix', 'Arkansas':
'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver', 'Connecticut': 'Hartford', 'Delaware':
'Dover', 'Florida': 'Tallahassee', 'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois':
'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas': 'Topeka', 'Kentucky':
'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine': 'Augusta', 'Maryland': 'Annapolis', 'Massachusetts':
'Boston', 'Michigan': 'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri':
'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada': 'Carson City', 'New Hampshire':
'Concord', 'New Jersey': 'Trenton', 'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina':
'Raleigh', 'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City', 'Oregon':
'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence', 'South Carolina': 'Columbia',
'South Dakota': 'Pierre', 'Tennessee': 'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West Virginia': 'Charleston',
'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}
>>>
```

```
>>> print (us_capitals)
{'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona': 'Phoenix', 'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado': 'Denver', 'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida': 'Tallahassee', 'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise', 'Illinois': 'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines', 'Kansas': 'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge', 'Maine': 'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston', 'Michigan': 'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson', 'Missouri': 'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln', 'Nevada': 'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton', 'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh', 'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City', 'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence', 'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee': 'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont': 'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West Virginia': 'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}
>>>
>>>
>>> pprint (us_capitals)
{'Alabama': 'Montgomery',
 'Alaska': 'Juneau',
 'Arizona': 'Phoenix',
 'Arkansas': 'Little Rock',
 'California': 'Sacramento',
 'Colorado': 'Denver',
 'Connecticut': 'Hartford',
 'Delaware': 'Dover',
 'Florida': 'Tallahassee',
 'Georgia': 'Atlanta',
 'Hawaii': 'Honolulu',
 'Idaho': 'Boise',
 'Illinois': 'Springfield',
 'Indiana': 'Indianapolis',
 'Iowa': 'Des Moines',
 'Kansas': 'Topeka',
 'Kentucky': 'Frankfort',
 'Louisiana': 'Baton Rouge',
 'Maine': 'Augusta',
 'Maryland': 'Annapolis',
 'Massachusetts': 'Boston',
 'Michigan': 'Lansing',
 'Minnesota': 'Saint Paul',
 'Mississippi': 'Jackson',
 'Missouri': 'Jefferson City',
 'Montana': 'Helena',
 'Nebraska': 'Lincoln',
 'Nevada': 'Carson City',
 'New Hampshire': 'Concord',
 'New Jersey': 'Trenton',
 'New Mexico': 'Santa Fe',
 'New York': 'Albany',
 'North Carolina': 'Raleigh',
 'North Dakota': 'Bismarck',
 'Ohio': 'Columbus',
 'Oklahoma': 'Oklahoma City',
 'Oregon': 'Salem',
 'Pennsylvania': 'Harrisburg',
 'Rhode Island': 'Providence',
 'South Carolina': 'Columbia',
 'South Dakota': 'Pierre',
 'Tennessee': 'Nashville',
 'Texas': 'Austin',
 'Utah': 'Salt Lake City',
 'Vermont': 'Montpelier',
 'Virginia': 'Richmond',
 'Washington': 'Olympia',
 'West Virginia': 'Charleston',
 'Wisconsin': 'Madison',
 'Wyoming': 'Cheyenne'}
>>>
```

The first line of this code may seem absolute nonsense, just type it for now, you’ll learn about *modules* and *packages* later.

8. Ready, Set, Go!

You might have studied *set theory* in mathematics. A *set* is an **unordered** collection of **unique** elements.

Python has a separate built-in data type to handle *Sets*. Set objects support various mathematical operations like *union*, *intersection*. Set objects are created by curly braces {} with comma separating each element or you can simply use 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 dont 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 behavior.

```
>>> 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, but you can add and delete elements from a given set.

1. set.add (e)

It adds an element to a 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 a 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 (s)*

In set theory, number of elements in a set is called its *cardinal number*.
len() function returns the number of elements in set *s* (cardinality of s).

```
>>> 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 Operations of Sets

Python provides almost all operations that are performed on sets.
The main *set* operations are:

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.union(B) returns the union of sets as a new set i.e., all elements that are in either set.
Pipe operator (|) can be used as shortcut.

```
>>> 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.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 shortcut.

```
>>> 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 with elements that are in *A* but not in the *B*. Minus (-) can be used as shortcut.

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

As usual, press the T*ab* button after dot operator (*set.Tab*) and explore other set methods.

LET ME
FLOW

1. Comparisons

Before directly going into control flow statements, let’s have a look how comparisons are done in Python.

Comparison operators simply compare two or more elements and return a *boolean* data type. A boolean type is either *True* (1) or *False* (0). *Booleans* are a subtype of integers which are simple mathematical comparisons that are finally evaluated to either *True* or *False*.

Operation	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: “=” is assignment operator which assigns values to variables.
“==” checks the condition of equality. A condition is simply a boolean expression that is evaluated to either True or False.

```
>>> 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]             #ordered sequence
False
>>> (11, 22, 33) != (33, 11, 22)
True
>>>
```

```
>>> (37>5) is True
True
>>> 3 is 3
True
>>> 7 is 4
False
>>> not True
False
>>> 5 is not 6
True
>>>
```

Dictionaries only support equality comparisons.

```
>>> {'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 conditions are True.

Keyword	**Description**
and	True, if both the condition 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
>>>
```

2. Control Flow

Control flow statements allow you to write decision-making programs. If a given set of conditions are followed, only then a particular portion of program is executed.

2.1 *if, else* statement

An *if* statement is the most commonly used control flow statement. *if* statement is followed by a condition, then a block of code indented under *if* statement, which executes only if the condition is True. In Python, various statements can be grouped together as a single block of code, by indentation.

if statement is optionally followed by an *else* statement which doesn’t have any condition. If the *if* statement’s condition is False, block of code indented under else statement is executed.

Syntax:

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

After *if* statement, condition is provided which is followed by a colon (:) and an indented block of code starts on next line.
else statement is indented to the same level of *if*, followed directly by a colon (:).
The block of code indented with a conditional statement is usually called as *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. From now on, most of the examples are written in the file editor to understand indentation properly as control flow statements must be perfectly indented or your program will crash even if all the statements are correct.

PEP 8 conventions recommend to use 4-space indentation, and no tabs as four spaces are a good compromise between small and large indentation.

valar_morghulis.py

```
greeting = 'Valar Morghulis'

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

else:
    print ("Nothing")
```

```
Valar Dohaeris
```

Each line within a block must be indented by the same amount else interpreter will throw *IndentationError.*

valar_morghulis.py

```
greeting = 'Valar Morghulis'

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

else:
    print ("Nothing")
```

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

Same program just crashed because it was not indented properly even if the logic is correct. It is important to keep a good understanding of how indentation works in Python to maintain the structure and order of your code.

winterfell.py

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

if words == "Winter Is Coming":
    print ("Welcome to Winterfell!")
else:
    print ("I dont know you.")
```

```
Words: It is what it is
I dont know you.
```

winterfell.py

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

if words == "Winter Is Coming":
    print ("Welcome to Winterfell!")
else:
    print ("I dont know you.")
```

```
Words: winter is coming
I dont know you.
```

This is because string “*Winter Is Coming*” is not equal to “*winter is coming*”. You can use string methods to solves this issue.

winterfell.py

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

if words.title () == "Winter Is Coming":
    print ("Welcome to Winterfell!")
else:
    print ("I dont 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; r*eadability counts*. I personally prefer this method as other programming languages also follow this methodology.

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 multiple condition checks, and you want to execute only a particular block of code, *elif* statement (which means else if) provides this flexibility. 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* statement but remember '*Flat is better than nested*'. Rest is your personal choice.

even_odd.py

```
num = int (input ("Enter an integer: "))

if (num%2 == 0 and num >= 0):
    print (f"{num} is positive even number.")

elif (num%2 == 0 and num < 0):
    print (f"{num} is negative even number.")

elif (num%2 != 0 and num >= 0):
    print (f"{num} is positive odd number.")

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

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

even_odd.py

```
num = int (input ("Enter an integer: "))

if (num%2 == 0):
    if (num >= 0):
        print (f"{num} is positive even number.")

    else:
        print (f"{num} is negative even number.")

else:
    if (num >= 0):
        print (f"{num} is positive odd number.")

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

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

Just like *pprint*, *math* is another Python module from which you can import the value of *pi* upto 15 digit precision. Be patient, modules will be covered soon.

pi.py

```
from math import pi
user_input = int (input ("Digit precision to get the value of pi: "))

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

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

```
Digit precision to get the value of pi: 6
3.141593
```

3. Loops

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

3.1 *while* statement

Like *if* statement *while* statement has a condition followed with a colon and an indented block of code. Unlike *if* statement, indented block of code is repeated over and over again as long as the condition remains True.

Syntax:

```
while condition:
    execute some statements
```

current_value.py

```
x = 4
while (x < 10):
    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
Current value of x: 9
Final value of x: 10
>>>
```

If you accidentally omit to increment the value of variable (x+=1), the loop will run forever.

Sometimes a condition is always True which also creates 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
Peace
Peace
Peace
Peace
Peace
Peace
Peace
Peace
Peace
Peace
^CPeace
Traceback (most recent call last):
  File "test.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.

Examples of iterables include all sequence types like *str, list, tuple*; and some non-sequence types like *dict, set.*

Syntax:

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

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

Let's see some examples, open your file editor and start typing.

youmatter.py

```
s = "You matter"

for alphabet in s:
    print (alphabet)
```

```
Y
o
u

m
a
t
t
e
r
>>>
```

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
>>>
```

sum.py

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

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

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', 'Yigrette'), ('Jammie', 'Cerci'), ('Tyrion', 'Shae')]

for i in couples:
    print (i)
```

```
('Jon', 'Yigrette')
('Jammie', 'Cerci')
('Tyrion', 'Shae')
>>>
```

couples.py

```
couples = [('Jon', 'Yigrette'), ('Jammie', 'Cerci'), ('Tyrion', 'Shae')]

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

```
Jon loves Yigrette
Jammie loves Cerci
Tyrion loves Shae
>>>
```

Tuple unpacking

couples.py

```
couples = [('Jon', 'Yigrette'), ('Jammie', 'Cerci'), ('Tyrion', 'Shae')]

for male, female in couples:
    print (male, female, sep='\t')
```

```
Jon	Yigrette
Jammie	Cerci
Tyrion	Shae
>>>
```

Remember, Python dictionary is a mapping (not sequence) you can iterate the view objects returned by *dict.items(), dict.keys()* and *dict.values()*.

By default 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:')
for i in my_info:
    print (i)

print('\n\nmy_info.keys ():')
for key in my_info.keys ():
    print (key)

print('\n\nmy_info.values ():')
for value in my_info.values ():
    print (value)

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

```
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 *for* statement, use an underscore (_) which is a Python convention indicating that the variable is not significant.

```
>>> for _ in range (7):
...     print ('Make the onions cry.')
...
Make the onions cry.
Make the onions cry.
Make the onions cry.
Make the onions cry.
Make the onions cry.
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 a specific number of times in *for* loops.

Range function, *range(),* accepts three arguments *start*, *stop*, *step*, which must be integers. *start* value is included while *stop* is excluded.

start and *step* are optional; *start* defaults to zero and *step* size to one.
If *step* size is zero, *ValueError* is thrown.

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
```

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 will always take 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
>>>
```

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

orphic.py

```
word_1 = 'Antidisestablishmentarianism'
word_2 = 'pneumonoultramicroscopicsilicovolcanoconiosis'

collection = [[word_1, len (word_1)], [word_2, len (word_2)]]

for item in collection:
    print(f'Current item: {item}')

    for i in item:
        print (i)

    print("\n")
```

```
Current item: ['Antidisestablishmentarianism', 28]
Antidisestablishmentarianism
28


Current item: ['pneumonoultramicroscopicsilicovolcanoconiosis', 45]
pneumonoultramicroscopicsilicovolcanoconiosis
45
>>>
```

Simillarly, you can nest *while* loop within *while* , *for* within *while* or *while* within *for.*

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

Loop statements may have an optional *else* clause; which executes differently with *for* and *while* statements.

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

If *break* statement breaks a loop whether *for* or *while, else* clause will be skipped.

no_name.py

```
num = 1

while (num < 4):
	print (num)
	num += 1

else:
	print ("Executes only if num >= 4")
```

```
1
2
3
Executes only if num >= 6
>>>
```

no_name.py

```
i = 2
while (True):
	if i == 4:
		break
	print (i)
	i += 1

else:
	print ("Only executes when the condition is False.")
```

```
2
3
>>>
```

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 0x7f21fceb84c8>                #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()* function

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.

It returns a tuple containing *count* and the *values*.

```
>>> #enumerate (sequence, count)
>>>
>>> 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.

```
>>> name = 'Westeros'
>>> alphabets = []
>>> for i in name:
...     alphabets.append (i)
...
>>> print (alphabets)
['W', 'e', 's', 't', 'e', 'r', 'o', 's']
>>>
```

The above program is usual way of creating a list from a sequence type.

This whole program can be reduced to just one line if you use list comprehension.

```
>>> alphabets = [i for i in "Westeros"]
>>> print (alphabets)
['W', 'e', 's', 't', 'e', 'r', 'o', 's']
>>>
```

You can also use *list()* constructor to solve the above issue 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, then zero or more *for* or *if* clauses.

Talk is cheap, show me your code

```
>>> 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 same list of coordinates.

```
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 *for* is when *if* statement is followed by *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. Relatively 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

Similarly *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'}
>>>
```

WE DO NOT
REPEAT

1. Functions

Functions are the important piece of puzzle which violates the DRY principle (*Dont Repeat Yourself*). A function is simply a block of code that only executes when it is called. You can call a function, defined within your script, multiple times.

You've already seen some of the Python's built-in functions. You can define your own functions with *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.

PEP 8 conventions, recommend to use *snake_case* for function names.

Syntax:

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

test.py

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

Output of the above script will be 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, you can have two kinds of arguments that are passed to the functions.

1. Keyword Argument

An argument preceded by an identifier in a function call, like *some_func(age=99)*, or passed as a value in a dictionary preceded by **.

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

2. Positional Argument

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*. Remember, *None* type 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 wont return anything.")

x = goblin ()
print (f"goblin() returns {x}")
```

```
I wont 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., a function shuts 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 may receive a value more than once.

info.py

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

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

```
Tormund, who has 185 cm height, is 29 years old.
160, who has 15 cm height, is Ryan years old.
Mark., who has 175 cm height, 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 afrer swapping \na: {a}, b:{b}')
```

```
Values before swapping
a: 11, b:22

Values afrer swapping
a: 22, b:11
>>>
```

You can also swap two values without using a temporary variable.

swapping.py

```
def swap (x, y):
    x = x+y
    y = x-y
    x = x-y
    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 afrer swapping \na: {a}, b:{b}')
```

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))
```

```
55.0
>>>
```

1.2 Documentation Strings

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

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

It 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 *range()*.

Syntax:

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

It is important to note that the function within *map()* is not called, it’s 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), end='\n\n')
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 condition and returns either True or False, use *filter* function. The *filter* function works same as *map*, the only difference is, *filter* only returns those values of an iterable for which function is True. Like *map(),* it 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), end='\n\n')
print (list (filter (is_even, num_list)))
```

```
<filter object at 0x7f11e6474390>

[6, 52]
>>>
```

1.4 *lambda* expression

Small functions can be created anonymously using *lambda* keyword, without properly defining a function.

Lambda expressions can have multiple arguments but syntactically it is restricted to a ***single expression***.

A lambda function has, no *def*, no *return* keyword; just arguments and an expression. Not every function can directly be converted 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
>>>
```

Its equivalent function would be:

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

Lambda expressions are more useful when used with Python's built-in *map()* and f*ilter()* 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]
>>>
```

You can have more complex expressions, take your time to see what’s happening in the following program; try to split it into multiple steps to fully understand it.

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

Its equivalent function 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 a default value; if the function is called, without passing a value for that particular argument, default value for argument is set automatically. If value is passed then default value is overwritten with the value passed during the call.

choice.py

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

choice ('Tommen')
choice ('Joffrey', 'Java')
```

```
Tommen likes Python.
Joffrey likes Java.
>>>
```

The default value is evaluated only once. This makes a difference when the default is 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.

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 arguments at all. If you're not sure about the number of parameters while defining a function, you can simply pass **args* to have an arbitrary number of arguments in the form of a tuple.

**args* can be replaced with any other name of your choice like **chocolate,* but it's a Pythonic convention to use **args*.

third_square.py

```
def squared_third_item (*args):
    if (len (args) > 2):
        return args[2]**2

    else:
        print ('Insufficient length.')

a = squared_third_item (7, 13, 2, 8)
b = squared_third_item (11, 22, 33, 44, 55)
print (a, b, sep='\t')

c = squared_third_item ()
print (c)
```

```
4    1089
Insufficient length.
None
>>>
```

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]
>>>
```

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

info.py

```
def student_info (**kwargs):
	print (f"Name: {kwargs['name']} \tEnroll: {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']} \tEnroll: {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: 24
>>>
```

info.py

```
def student_info (**kwargs):
	print(f"Name: {kwargs['name']} \tEnroll: {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
>>>
```

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* statement does nothing. It is usually used when a statement is required syntactically but the program requires no action.

useless.py

```
def useless ():

# outside useless() function’s indentation
print ("useless function will be defined later.")
```

```
  File "useless.py", line 4
    print("useless function will be defined later.")
        ^
IndentationError: expected an indented block
>>>
```

useless.py

```
def useless ():
        pass

print ("useless function will be defined later.")
```

```
useless function will be defined later.
>>>
```

1.8 Scope of a variable

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

1. Local Scope

Variables and parameters (function arguments) which are defined inside a function are said to be in 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 as 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
>>>
```

Parameters *arg1, arg2* and *result,* are in local scope to *addition* function and can only be accessed within that function. *NameError* is thrown as I tried to access its parameters outside function’s scope.

***locals*()** is a built-in function that returns a dictionary containing the current scope's local variables.

2. Global Scope

A variable which is defined outside all function 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.
>>>
```

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 current scope's global variables.

LEGB Rule

A variable can either be *global* or *local*, cant 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?

Here 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 variables in the outer function.

b. Local variables both read and write in the innermost scope. Likewise, global variables 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 time during execution, there are at least three nested scopes which are directly accessible:

1. ***Local*** (the innermost scope), which is **searched first**, contains the local variables.

2. Scope of any ***Enclosing*** function, which is searched starting with the nearest enclosing scope, contains non-local, but also non-global variables.

3. ***Global*** (the next-to-last scope) contains the global variables.

And finally, there is ***Built-in*** (the top level scope), **last searched**, containing built-in names like *len, range, list.* never overwrite these names.

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 11, in <module>
    prince_of_dorne()
NameError: name 'prince_of_dorne' is not defined
>>>
```

global and *nonlocal* statement

If you want to change the value of a global variable within local scope, use *global* statement followed by the 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
>>>
```

The *nonlocal* statement indicates that particular variables live in an enclosing scope and should be rebound there.

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

If you call a function within itself, it is called as recursion. Be careful, whenever you use recursion, cause without having a base case you will make infinite calls to that function.

A base case simply sets some condition that terminates the subsequent function calls.

infinite_recursion.py

```
def infinite_recursion ():

    print ('Walk of Punishment')
    return infinite_recursion ()

infinite_recursion ()
```

```
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Walk of Punishment
Traceback (most recent call last):
  File "infinite_recursion.py", line 7, in <module>
    infinite_recursion ()
  File "infinite_recursion .py", line 4, in infinite_recursion
    return infinite_recursion ()
  File "infinite_recursion .py", line 4, in infinite_recursion
    return infinite_recursion ()
  File "infinite_recursion .py", line 4, in infinite_recursion
    return infinite_recursion ()
  [Previous line repeated 992 more times]
  File "infinite_recursion .py", line 3, 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))
```

```
13
>>>
```

1.10 Decorators

A decorator is a function which modifies the functionality of another function.

Actually, it’s a function that returns a function defined within it and it is usually applied as a function transformation using the symbol ‘@’.

This is an advanced topic so don’t worry if you dont 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')


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


normal_func_1 ()
print ('\n')
normal_func_2 ()
```

```
Please do not decorate me

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

Let’s break down ,what just happened in the above code example

First, a decorator function is defined, which accepts a function as its argument. Its job is to wrap the modified function and return it.

Second, a modified function is defined within the decorator function whose job is to add functionality to the function which is passed to decorator function, by calling it.

Then normal functions are defined, which execute normally.

If you want to add the functionality to any normal function, add *@decorator_function_name* above the function definition and the functionality is added automatically.

Although you can create your own decorators but mostly you'll be using Python's built-in decorators for your advanced scripts like *user authentication*, while creating a login system, which may look like this:

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

This *login_required* is built into one of the Python's web framework, *Django*, that automatically checks whether the user is logged in or not.

1.11 Generators and Iterators

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.

You can simply write a normal function and use *return* statement, instead of using a generator function with *yield* statement; but in case of large data sets generators are very useful as these are memory efficient.

A generator function only stores the previous result to generate the next one, instead of allocating the memory for all of the results at the same time.

generator.py

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

x = generator_function()
print (x)
```

```
<generator object generator_function at 0x7f2e2c49d840>
>>>
```

generator.py

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

x = generator_function ()

for element in x:
    print (element)
```

```
0
32
64
96
128
160
>>>
```

Items of a generator function can also be retrieved one at a time with the ***next()*** function.

This function simply allows you to access next item of the sequence returned by the generator function.

```
>>> def generator_function ():
...     for i in range (6):
...             yield i*32
...
>>> x = generator_function()
>>> next (x)
0
>>> next (x)
32
>>> next (x)
64
>>> next (x)
96
>>> next (x)
128
>>> next (x)
160
>>> next (x)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

As all the items of the sequences are accessed , a *StopIteration* exception is thrown, if you try to go beyond the limit.

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 *next* function to them interpreter will through 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
>>>
```

First, you've to convert these objects into iterator objects using Python's built-in ***iter()*** function, only then you can use *next*() function.

```
>>> a = 'The Rains of Castemere'
>>> b = iter (a)
>>> b
<str_iterator object at 0x7fa58fe9c9b0>
>>> next (b)
'T'
>>> for i in range(7):
...     next (b)
...
'h'
'e'
' '
'R'
'a'
'i'
'n'
>>>
```

At first, this concept of *iterators, iterables* and *generators* may seem confusing, but as you keep using these functions in your scripts, you'll understand them better.

2. Modules

If a program has several thousand lines of code (LoC), you may want to split it into multiple scripts for easier maintenance.

You may also want to use some handy functions, that you’ve written in several programs, without copying their definitions into each script. You can achieve this functionality, 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 *import* statement. It is customary but not required to place all *import* statements at the beginning of a module.

Imported modules are executed only first time, 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 *pprint*, *math*.

The built-in function, *dir*() returns a list of attributes and functions, a module defines, which can be accessed by dot (.) operator.

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

First you’ve to import *math* module, only then you can use various functions and attributes contained in that module.

Just like *method* is a *function* that is associated with an object;
an *attribute* is a *value* associated with object.

You’ll learn more about methods and attributes in Object Oriented Programming (*Better an OOPs than a what if*).

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

```
3.141592653589793
>>> math.log10 (100)
2.0
>>>
```

You can create your own modules by simply creating a Python script, which then can be imported using *import* statement.

MyModule.py

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

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

Now create another script as main.py and import MyModule.
Use dot (.) operator to access functions of *MyModule.py* in *main.py*

main.py

```
import MyModule

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

```
I'm from my_module.py
olleH
>>>
```

When you import a Python module, the code in the main body is also executed, which can create problems as you just want to use the functions of module without running its main body directly e.g.

MyModule.py

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

print ("Starting MyModule's main body.")
x =  string_reversal ("Python")
print (x)
print ("Ending MyModule's main body. \n")
```

main.py

```
import MyModule

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

```
Starting MyModule's main body.
nohtyP
Ending MyModule's main body.

wonS noJ
>>>
```

The whole code of the module is executed, just as it is imported, 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 executing if the script is imported as module and not executed directly.
With this flexibility, you can make your files usable as scripts as well as importable modules.

MyModule.py

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

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

if __name__ == "__main__":
      print ("Starting MyModule's main body.")
      x =  string_reversal("Python")
      print (x)
      print ("Ending MyModule's main body.")
```

main.py

```
import MyModule
MyModule.print_something ()
rev_string = MyModule.string_reversal ("Nazi")
print (rev_string)
```

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

Remember that for efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter.

2.1 Packages

A package is a collection of modules and subpackages (package within package). You can say, that a package is a folder (or a directory) which contains multiple Python scripts.

__init__.py is a special Python script which 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.

```
MainPackage/                    Top-level package
   __init__.py
   module1.py
   module2.py
   .
   .
   module999.py

   Subpackage1/
      __init__.py
      submodule1.py
      submodule2.py
      .
      .
      submodule999.py
```

You can import individual modules from the main package using *from* and *import* statements

from package *import* item

The item can be either a submodule (or subpackage) of the main package, or some other name defined in the package, like a function or variable.

The import statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an ***ImportError*** is raised.

```
from MainPackage import module2
from MainPackage.Subpackage1 import submodule2

module2.some_function()
submodule2.some_other_function()
```

You can use ***** to import everything from a package (or module).
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
>>> ceil (3.124)
4
>>>
```

Note that in general, the practice of importing * from a module or package often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

You can use a nickname for a function, if it is too long or the function name conflicts with the other existing function, using ***as*** keyword.

main.py

```
import MyModule as m

print (m.string_reversal ("Atlas"))
```

```
saltA
```

main.py

```
from MyModule import string_reversal as rev

print (rev ("Floccinaucinihilipilification"'))
```

```
noitacifilipilihinicuaniccolF
```

2.2 What is pip?

pip is the preferred installer program, with which you can manage all the packages, including third-party packages, easily.

You can visit *https://pypi.org* to see all the available packages.

It is possible that pip does not get installed by default (usually below Python 3.4), you can ensure pip installation by firing the following command into your terminal:

```
python -m ensurepip
                                        or
python -m ensurepip --default-pip
```

In case pip is not installed, you can easily download it.
Ubuntu (GNU/Linux) user simply have to fire following command in terminal:

```
sudo apt-get install python3-pip
```

With *pip* you can either install the latest, or a specified version of a package, by typing these commands into the terminal.

```
pip install packagename                    #for latest version

pip install packagename==version           #specific version
```

pydocs example

```
$ pip install novas
Collecting novas
Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
Running setup.py install for novas
Successfully installed novas-3.1.1.3


$ pip install requests==2.6.0
Collecting requests==2.6.0
Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

If you try to install a particular version of package that is already installed, pip will display *Requirement already satisfied*.

```
$ pip install numpy
Requirement already satisfied: numpy in ./lib/python3.7/site-packages (1.16.4)
```

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

```
$ pip list

Package                          Version
------------------------------  ---------
absl-py                          0.7.1
alabaster                        0.7.12
numba                            0.44.1
numexpr                          2.6.9
numpy                            1.16.4
Pillow                           6.1.0
pip                              19.1.1
pylint                           2.3.1
pyodbc                           4.0.26
PySocks                          1.7.0
pytest                           5.0.1
pytest-arraydiff                 0.3
pytest-astropy                   0.5.0
requests                         2.22.0
rope                             0.14.0
scikit-image                     0.15.0
scikit-learn                     0.21.2
scipy                            1.3.0
seaborn                          0.9.0
Sphinx                           2.1.2
spyder                           3.3.6
tensorflow                       1.14.0
terminado                        0.8.2
toolz                            0.10.0
tornado                          6.0.3
wrapt                            1.11.2
wurlitzer                        1.0.2
xlrd                             1.2.0
XlsxWriter                       1.1.8
xlwt                             1.3.0
zict                             1.0.0
zipp                             0.5.1
```

If you want a detailed view of a particular package, fire this command:

```
$ pip show packagename

pip show pandas
Name: pandas
Version: 0.24.2
Summary: Powerful data structures for data analysis, time series, and statistics
Home-page: http://pandas.pydata.org
Author: None
Author-email: None
License: BSD
Location: /home/izan/lib/python3.7/site-packages
Requires: pytz, python-dateutil, numpy
Required-by: statsmodels, seaborn
```

3. Exception handling

You’ve already seen errors chasing you continuously, crashing your programs. With Python, you can easily 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 type of errors that occur due to the mistakes in syntax, while writing a program.

These errors are sometimes referred as *parsing errors* and can easily be handled by looking at the error message, suggesting what interpreter is looking for.

```
>>> print "This should through 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 through an error")?
>>>
>>> {'key'='value1', 'key2'= 'value2'}
  File "<stdin>", line 1
    {'key'='value1', 'key2'= 'value2'}
          ^
SyntaxError: invalid syntax
>>>
>>> for i in [1,2,3]
  File "<stdin>", line 1
    for i in [1,2,3]
                   ^
SyntaxError: invalid syntax
>>>
```

3.2 Exceptions

Exceptions are the errors that are detected during execution, even if statements within the script are syntactically correct, resulting in crashing your entire program.

Some of common exceptions are:
AssertionError, AttributeError, TypeError,
FloatingPointError, ImportError, ModuleNotFoundError,
IndexError, KeyError, KeyboardInterrupt,
MemoryError, NameError, NotImplementedError,
ValueError, ZeroDivisionError.

```
>>> 32/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>>
>>> import NoModule
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'NoModule'
>>>
>>> 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'
>>>
>>> 'Hello'[56]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

3.3 *try* and *except* clause

In Python, *try-except* clauses handle the exceptions. *try* clause is executed first, if an exception occurs the rest of *try* clause is skipped and *except* clause is executed; if no exception occurs, the *except* clause is skipped.

It is important to note that SyntaxErrors are not handled by these clauses.

Syntax:

```
try:
    block of code where exceptions may be raised

except:
    executes if exception is caught
```

It is also possible to handle selected exceptions only, if you type the name of that exception after *except* clause.

If no particular exception is mentioned then *except* clause is executed for all exceptions caught in try block.

```
try:
    block of code where exceptions may be raised

except ExceptionName:
    executes only if exception ExceptionName is caught
```

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
>>>
```

In first case, *execpt* clause is skipped as no exception is caught.
In second case, *ZeroDivisionError* is handled by *except* clause.
Third case throws a *ValueError* as input expects an interger but floating point number is given. This exception is also handled by same *except* clause as it is not followed by any specific *ExceptionName*.

You can have multiple *except* clauses for same *try* clause handling different exceptions. This makes your scripts more readable.

pilot.py

```
import math

try:
    num = int (input ('Number: '))
    result = math.pi/ num
    print (f"Result: {result}")

except ZeroDivisionError:
    print ("You are attempting to divide by zero.")

except ValueError:
    print ("Only integers are accepted.")

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 by zero.
Outside try and except clauses.
>>>
```

```
Number: 1.26
Only integers are accepted.
Outside try and except clauses.
>>>
```

3.4 *else* and *finally* clause

A *try-except* clause can be followed by *else* statement, which is executed only if no exception is caught in try block. In other words, if *except* clause is skipped, *else* 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. Block of *finally* clause don’t care whether exception is caught or not, it just executes.

index_error.py

```
try:
    x = "Finally"
    print (x[3])

except IndexError:
    print ("Out of bounds.")

else:
    print ("Program has no IndexError.")

finally:
    print("I dont care.")

print ('Normally code goes here.')
```

```
a
Program has no IndexError.
I dont care.
Normally code goes here.
>>>
```

index_error.py

```
try:
    x = "Finally"
    print (x[7])

except IndexError:
    print ("Out of bounds.")

else:
    print ("Program has no IndexError.")

finally:
    print("I dont care.")

print ('Normally code goes here.')
```

```
Out of bounds.
I dont care.
Normally code goes here.
>>>
```

3.5 *finally* clause vs normal code

You might be wondering, if *finally* block is always executed like the normal code outside *try-except* block, then why *finally* clause even exits. Carefully observe the following example programs.

is_vowel.py

```
def is_vowel (alphabet):
    try:
        if alphabet in ['a', 'e', 'i', 'o', 'u']:
            print (True)

        else:
            pritn(False)

    except:
        print ("You're not passing a valid argument.")
    else:
        print ("No Exception caught.")

    finally:
        print ("Block of finally clause.")

    print ("Normal text outside try-except.")

is_vowel ('a')
```

```
True
No Exception caught.
Block of finally clause.
Normal text outside try-except.
>>>
```

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 finally clause.")

    print ("Normal text outside try-except.")

print (is_vowel('a'))
```

```
Block of finally clause.
True
>>>
```

Note how *return* statement in try block affect the output of *is_vowel()* function.

If *return* statements exists in *try* or *except* block, program flow is returned back to main and the rest of the function body, including all clauses and statements, is skipped except *finally* clause.

finally clause is always executed before the program control is returned back to main by *return* statement.

break and *continue* statements can also affect the program like *return* statement, if 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 but within function.')

original_pi = round(math.pi, 2)
is_pi()
print("Back to main.")
```

```
Your Number: 8.3
Finally
Outside try-except but within function.
Your Number: 3.14
Awesome
Finally
Back to main.
>>>
```

This might seem confusing but don’t worry, usually you won’t encounter these kind of situations. The thing you should know is, how exceptions are raised and how you can handle these, with t*ry-except* clauses to prevent your programs from crashing.

3.6 *raise* statement

The *raise* statement allows you to force a specified error to occur.

Syntax:

```
raise ErrorName ("message to be displayed")
```

```
>>> raise SyntaxError("Nothing really happened")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
SyntaxError: Nothing really happened
>>>
>>> raise NameError("I can forefully crash any program")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: I can forefully crash any program
>>>
```

If you use *raise* statement within *try-except*, an exception is re-raised

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
>>>
```

Better an **OOPs**

than a what if

1. Object Oriented Programming

"*Object*" you've now heard this word a lot of times and the phrase *Everything in Python is an Object.* Although, I've briefly introduced objects, but the question is, what actually is an Object in programming?

Formally, an object is any data with state (attributes or values) and defined behavior (methods) i.e., an object has some variables and functions associated with it. You'll better understand this important concept of objects once you'll start creating your own objects.

In Python, classes are created, using *class* keyword and objects are nothing, but the instances of a *class.* You can think of class as a template for creating user-defined objects.

Try to understand this 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 are the *attributes* associated with Human object.
Various functionalities are also associated with a person like walking, talking etc, these are the *methods* that are associated with it.

pilot.py

```
class BasicClass ():
    x = 14

i = BasicClass ()
print (i.x)
```

```
14
>>>
```

This is the most basic and simplest class that can exist. It's optional to have parentheses () after the class name.

In this example, *BasicClass* is created which has only one *class variabe* (or *class object attribute*).

Variable *i* is instantiated as an object of *BasicClass* and dot (.) operator is used to access class variable.

A class variable is defined directly (not enclosed in any method) within a class.

In addition to *class variable*s, classes also have *attributes* (or instance variables) and *methods*.
An *attribute* is a value associated with an object defined within a special method and a *method* is just a function which is defined inside a class body.

Attributes are unique to each instance and *class variables* are shared by all instances of the class.

Attributes and methods both are accessed using dot (.) operator.
As methods are functions, method names must be followed by a set of parentheses like ClassName.func(args) while attributes are values and can be aceessed like ClassName.attr
If you hit *Tab* key after dot (.) operator, the interpreter will provide a list of all attributes and methods associated with that object.

PEP 8 conventions suggest to use *CamelCase* for class names and
snake_case for methods.

useless.py

```
class Useless ():
    def what_to_do ():
        print ('Just print this.')

a = Useless ()
a.what_to_do()
```

```
Traceback (most recent call last):
  File "pilot.py", line 7, 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, then why is it displaying that one was given?

When you access a method through an instance, it will add the *self* argument to its argument list. The keyword *self* refers to the object for which a method is called.

self is just a convention, you can use any name for this argument. But if you import some third-party module and use any other name instead of *self* , your script may work abnormally, as almost everywhere in Python the first argument of a method is named as *self*.

useless.py

```
class Useless ():
    def what_to_do (self):
        print ('Just print this.')

a = Useless ()
a.what_to_do()
```

```
Just print this.
>>>
```

Python has some special methods that are called implicitly to execute a specific operation. 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 a*ttributes* are initialized within this special method as *self.attribute* = *some_value.*

student.py

```
class Student ():
    def __init__ (self):
        self.school = 'Unregistered International School'

x = Student ()
print (x.school)
```

```
Unregistered International School
>>>
```

__*init*__ method can have multiple arguments which can be passed to its instance as it is instantiated.

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)
print (stu_2.age)
```

```
Joshua
13
>>>
```

It is not necessary to match the argument name with its corresponding attribute but it is a good idea to use this convention 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 *del* statement.

fruit.py

```
class Fruit ():
    def __init__ (self, taste):
        self.taste = taste

lemon = Fruit('sour')
apple = Fruit('sweet')
print(f"Apple: {apple.taste}\t Lemon: {lemon.taste}")
del lemon
print(lemon.taste)
```

```
Apple: sweet        Lemon: sour
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print(lemon.taste)
NameError: name 'lemon' is not define
>>>
```

An attribute within a method can be accessed as *self.attribute_name.*
A class variable can also be accessed by *self.class_variable_name* but by convention class object attributes (or class variables) are accessed by *ClassName.class_variable_name*.

flower.py

```
class Flower ():
    kingdom = 'Plantea'

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

    def display(self):
        #print(self.kingdom, self.name, sep='\t')
        print(Flower.kingdom, self.name, sep='\t')

a = Flower('Rose')
a.display()
```

```
Plantea
Rose
>>>
```

If you have some attributes for internal use and you want to hide these attributes from user, start the attribute name with an underscore (_).

These attributes, whose name starts with an underscore, won’t appear in the list that is provided by interpreter as you press *Tab* key after dot operator of an instance.

However, a users can access that attribute, if they manually type the attribute name without using *Tab* key.

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
>>>
```

With all these examples, you can generalize the basic syntax to create a new class like this:

Syntax:

```
class ClassName():

    #class variables
    class_object_attribute_1 = value_1
    .
    .
    class_object_attribute_n = value_n

    #constructor
    def __init__(self, args):

        #attributes
        self.attr_1 = value_1
        .
        .
        self.attr_n = value_n

    #methods
    def method_1(self, args):
        pass
    .
    .
    def method_n(self, args):
        pass
```

Let’s re-create previous Student class and use all these concepts.

student.py

```
class Student():

    semester = 'III'

    def __init__ (self, name, enroll):
        self.name = name
        self.enroll = enroll
        self.sub = {'physics': None, 'maths': None}

    def put_marks (self, physics, maths):
        self.sub['physics'] = physics
        self.sub['maths'] = maths

    def display (self):
        print (f'''{self.name} got {self.sub['physics']} in Physics and {self.sub['maths']} in Maths (semester {Student.semester}).''')

stu1 = Student ('Joshua', 213)
stu1.put_marks (physics=87, maths=81)
stu1.display ()
print (stu1)
```

```
Joshua got 87 in Physics and 81 in Maths (semester III).
<__main__.Student object at 0x7f336db0e5f8>
>>>
```

Everything just worked fine, but when I try to print an instance of Student instead of printing who that student is, the interpreter is showing me object’s memory address where that instance is stored in the memory. This issue can be solved by adding other special methods to your script.

2. Special methods

Like ***__int__***, Python has other special methods that make our life lot easier. These are sometimes called as *magic methods* (which I hate to call these special methods).

All special methods' name start with dunder (double underscore) followed by name and terminate with dunder again: ***__magic__***.

Python has a lot of special methods, I'll cover commonly used ones.

1. __str__()

If an object is passed to a function that 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} has enroll {self.enroll}.")

    def __str__ (self):
        return (self.name)

x = Student ('Joshua', 213)
x.display ()
print (x)
```

```
Joshua has enroll 213.
Joshua
>>>
```

2. __len__()

If len() fuction is called on an object, this method executes automatically.

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', 'Grapes', 'Strawberry'])

if ('Mango' in my_basket):
    print (my_basket)

else:
    print ('Nothing')
```

```
Izan
```

4.__iter__()

This method makes your objects *iterable*.

basket.py

```
class Basket ():
    def __init__ (self, owner, items):
        self.owner = owner
        self.items = items

    def __str__(self):
        return (self.owner)

    def __iter__(self):
        return iter(self.items)



my_basket = Basket ('Izan', ['Mango', 'Grapes', 'Strawberry'])

for fruit in my_basket:
    print (fruit)
```

```
Mango
Grapes
Strawberry
>>>
```

5. __del__()

If you want to execute some code on destroying an instance of a certain class, type the code to be executed within *__del__()* method.

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, some of the Python’s built-in *decorator functions* that are defined for classes are used. Again this is an advanced topic, feel free to skip if you’re not comfortable with it at this level.

class_decorators.py

```
#Regular methods -> self (instance as first argument)
#Class methods -> cls (class as first argument)
#Static methods -> no self, no cls


class People():

    category = "Elite"

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


     # fullname is treated as an attribute and updated accordingly
     # if name or color is changed

    @property
    def fullname (self):
        return (f"{self.name} {self.color}")


#you can assign value to fullname attribute directly using @attrname.setter

    @fullname.setter
    def fullname (self, x):
        color, name = x.split ()
        self.name = name
        self.color = color

    @classmethod
    def change_category(cls, x):
        cls.category = x

    @classmethod
    def from_string_given (cls, string):          #acts as constructor
        name, color = string.split()
        return cls (name, color)

    @staticmethod
    def is_elite ():
        return (People.category == "Elite")

    def __str__(self):
      return (self.name)
```

```
person1 = People("Sirius", "Black")
print (person1.fullname)
print ('\n')

#This changes the value of class object attribute
person1.change_category("poor")

person2 = People("Rachael", "Green")
print(f'The category of {person2} is now poor as it was changed.')
print (f'{person2}.is_elite(): {person2.is_elite()}')

print ('\n')

person2.change_category("Elite")
print(f"The category of {person1} is not poor as it was again changed.")
print (f'{person2}.is_elite(): {person2.is_elite()}')
```

```
Sirius Black

The category of Rachael is now poor as it was changed.
Rachael.is_elite(): False


The category of Sirius is not poor as it was again changed.
Rachael.is_elite(): True
>>>
```

---

3. Inheritance

Inheritance means inheriting properties from ancestors.
Python classes support inheritance, which means you can derive new classes from an existing class which is already defined (base class).

Inheritance greatly reduces the length of your script, if you have multiple classes that are co-related. Instead of defining each class you simply inherit common thing from some base class.

For example, a student in certain college can be an arts, commerce or a science student, so different students may have different academic activities and at the same time they have several common attributes like student’s name and other methods.

A Student class can be created as a base class having all the common attributes and methods and then Arts, Commerce and Science classes are derived from Student 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

class DerivedClass2 (BaseClass):
    unique attributes and methods of this class
```

All the attributes and methods of the Base class are inherited by derived class. Base class constructor is called within derived class’ constructor as:

```
class DerivedClass (BaseClass):
    def __init__(self, args):
        BaseClass.__init__(self, args)
```

Let’s see an example of *single inheritance* where the derived class is inherited from a single base class.

student.py

```
class Student ():
    def __init__(self, student_id, name):
        self.student_id = student_id
        self.name = name

    def get_id (self):
        return self.student_id

    def __str__(self):
        return self.name

class CompScience (Student):
    def __init__(self, student_id, name, prog_lang):
        Student.__init__(self, student_id, name)
        self.prog_lang = prog_lang


stu1 = CompScience(213, 'Mark', 'Python')
x = stu1.get_id()
print (x, stu1.name, stu1.prog_lang, sep='\t')
```

```
213 Mark    Python
```

3.1 *super()*

In single inheritance, *super()* function can be used to refer parent classes, without naming them explicitly and without passing *self* argument to constructor of base class.

student.py

```
class Student ():
    def __init__(self, name):
        self.name = name

class CompScience (Student):
    def __init__(self, name, prog_lang):
        super().__init__(name)
        self.prog_lang = prog_lang

stu1 = CompScience('Joshua', 'Java')
print (f"{stu1.name} knows {stu1.prog_lang}.")
```

```
Joshua knows Java.
>>>
```

3.2 Method Overriding

All methods of Python objects are virtual i.e., if the base class and derived class share a same method name, then the derived class’ method overrides the base class’ method.

overriding.py

```
class Base ():
    def __init__(self, x):
        self.x = x

    def display (self):
        print ('Base class constructor')


class Derived (Base):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def display (self):
        print ('Derived Class constructor')


derived = Derived('X', 'Y')
derived.display()
```

```
Derived Class constructor
>>>
```

3.3 Abstract Base Class

If you want to restrict users from directly accessing the base class methods raise *NotImplementedError* exception to that method in base class.

abc.py

```
class Base ():
    def __init__(self, x):
        self.x = x

    def display (self):
        print ('Base class constructor')
```

```
class Derived (Base):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def display (self):
        print ('Derived Class constructor')


base = Base('X')
base.display()

derived = Derived('X', 'Y')
derived.display()
```

```
Base class constructor
Derived Class constructor
```

abc.py

```
class Base ():
    def __init__(self, x):
        self.x = x

    def display (self):
        print ('Base class constructor')
        raise NotImplementedError

class Derived (Base):
    def __init__(self, x, y):
        super().__init__(x)
        self.y = y

    def display (self):
        print ('Derived Class constructor')


base = Base('X')
base.display()

derived = Derived('X', 'Y')
derived.display()
```

```
Base class constructor
Traceback (most recent call last):
  File "abc.py", line 20, in <module>
    base.display()
  File "abc.py", line 7, in display
    raise NotImplementedError
NotImplementedError
>>>
```

3.4 Multiple Inheritance

A single class can be derived from two or more classes. This is known as *multiple inheritance*. All methods and attributes of base classes are inherited by derived class.

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
>>>
```

4. Polymorphism

Polymorphism is nothing, just a fancy term for methods of different classes sharing same name. In other words, it is possible to have same method names across different classes.

student.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
>>>
```

For Student object, *display()* method displays student’s info and for Fruit object, this method gives fruit’s name and taste, this means that any method defined inside a class works as per its definition irrespective of the fact that its name is shared by other classes.

FILE AND
REGEX

1. File Handling

With Python, you can handle almost all types of files including text files, pdf, docx, excel documents, html, json, archives (zip/rar/...).

For each file type, Python modules are available which can be easily downloaded with *pip*. In this section, you’ll work only with text files which dont need any additional module.

1.1 *open()*

open() function is Python’s built-in function which accepts a string containing the filename followed by another string, describing the way in which the file will be used and returns an *IO* object.

Syntax:

```
file_object = open (‘filename.txt’, ‘r’)
```

File path (relative or absolute) is provided, if the file isn't in the current working directory.

Windows users:
open("C:\\Users\\username\\Home\\MyFolder\\xyz.txt")
Recall escape sequences, double backslashes just print a single backslash.

MacOS and **GNU/Linux** users:
open("/home/username/MyFolder/abc.txt").

1.2 Mode

The second argument is *mode*, which is optional and defaults to 'r' (read only). Available mode options are given in following table:

Character	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 a mode (reading and writing)
'U'	universal newlines mode (deprecated)

1. Mode `r`

To read a file's contents open the file in 'r' mode (default mode).
You can use *.read()* method of *IO* object which returns the entire content of the file as string.
.read() method accepts an optional *size* argument.

Create a new text file in notepad (or any other text editor) and save it to a specific directory.

xyz.txt

```
This is the first sentence of my text file.
The quick brown fox jumped over a lazy dog.
```

Now open this text file in Python interpreter using open() function as:

```
>>> 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 called f.read() second time?

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 using *.seek()* method and passing ‘0’ (zero) as argument, that resets the position to the beginning of text.

```
>>> f.read ()
''
>>> 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 your job is done, you should always call *.close()* method on the *IO* object to close the file and immediately free up system resources used by it. Files may behave abnormally, if you don’t close these manually.

.readlines() is another method that returns a list of lines within the text file. This method also reaches to the end of file, use *seek(0)* to reset the position.

```
>>> 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 file 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.
>>>
```

If you try to open a file that didn’t exist, interpreter will throw *FileNotFoundError* exception.

```
>>> f = open('nofile.txt', 'r')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'nofile.txt'
>>>
```

with statement

Instead of manually closing a file object use *with* keyword that automatically closes a file, even if an exception is raised at some point.

Create a new Python script and save it to the same directory where *xyz.txt* is stored.

file.py

```
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.
>>>
```

This means that the text file *xyz.txt* is opened as an *IO* object, named ***f*** *(alias)* and is closed automatically as you come out of the block indented under *with* statement.

If you try to use *IO* object outside the *with* statement, the interpreter will throw *ValueError*.

file.py

```
with open('xyz.txt') as f:
    print ("Inside with statement")

f.read()
```

```
Inside with statement
Traceback (most recent call last):
  File "file.py", line 4, in <module>
    f.read()
ValueError: I/O operation on closed file.
>>>
```

2. Mode `w`, `w+` and `r+`

Mode ‘w’ opens a file in write only mode i.e., you can only write to a file.

If a file doesn’t exit, a new file of that name will be created and if file already exists then the text is overwritten i.e., previous data in the file is destroyed.

```
>>> 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 Dance of Dragons.
>>>
```

Attempting to read a file in write only (‘w’) mode throws an exception.

```
>>> with open ('xyz.txt', 'w') as f:
...     f.write ('The Night Lands.\nThe Dragon and the Wolf.')
...     f.read ()
...
41
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
io.UnsupportedOperation: not readable
>>>
```

You can pass ‘w+’ mode to open a file both in reading and writing mode.
Like ‘w’ mode, this mode overwrites the existing file or creates a new one in case 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.
>>> with open ('/home/izan/Desktop/MyNewFile', 'w+') as f:
...     f.write ('Previous data is destroyed.')
...     f.seek (0)
...     f.read ()
...
27
0
'Previous data is destroyed.'
>>>
```

f.write() writes to file and returns the length of string passed.
f.seek() resets the position and returns 0 indicating the start of file.
f.read() returns the content of file as a string.

‘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 *FileNotFoundError* like in ‘r’ mode.

```
>>> with open('/home/izan/Desktop/MyNewFile2.txt', 'r+') as f:
...     f.write('Some text')
...     f.read()
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'MyNewFile2.txt'
>>>
```

3. Mode `a` and `a+`

Opening a file in ‘w’ mode destroys the existing data, so it’s always better to open a file in append mode, if you want to add something to a file.

Like write mode, append mode also creates a new file if file doesn’t exist.

```
>>> with open ('xyz.txt', 'a') as f:
...     f.write (' Append me.')
...
11
>>> with open ('xyz.txt') as f:
...     print (f.read())
...
The Night Lands.
The Dance of Dragons. Append me.
>>>
```

‘a’ supports only *.write()* method; to add the functionality of reading use ‘a+’ mode.

```
>>> with open ('/home/izan/Desktop/BrandNewFile', 'a+') as f:
...     f.write('This is my brand new file.')
...     f.seek(0)
...     f.read()
...
26
0
>>> with open ('/home/izan/Desktop/BrandNewFile', 'a+') as f:
...     f.write(' This line will be added at the end.')
...     f.seek(0)
...     f.read()
...
36
0
'This is my brand new file. This line will be added at the end.
>>>
```

For binary read-write access, the mode 'w+b' opens and truncates the file to 0 bytes. 'r+b' opens the file without truncation.

Here is the summary of methods, which are associated with *IO* objects:

Method	**Description**
.read()	returns a giant string of whole text from file.
.seek()	changes the file object’s position.
.readlines()	returns a list of lines from file.
.write()	overwrites or appends text to a file, depending on mode.
.close()	closes the IO object

2. Regular Expressions

(source: pydocs)

Regular expressions or simply regexes are essentially a tiny, highly specialized programming language embedded inside Python which is made available through ***re*** module.

These expressions are used in pattern matching like finding email addresses in a given text. You define your own patterns and set specific rules, rest of the job is done by re module.

I’ll explain three commonly used methods of re module:

Method	Description
re.findall ()	returns a list of matches.
re.search ()	returns a match object.
re.split ()	returns a list, splitted on a given pattern.

2.1 *re.findall* (pattern, text)

re.findall finds all substrings where the regex matches, and returns them as a list.

```
>>> import re
>>> text = 'Seated in Winterfell, Stark is the principle 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 ignores the case sensitivity.

```
>>> 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 "/home/izan/lib/python3.7/re.py", line 223, in findall
    return _compile(pattern, flags).findall(string)
  File "/home/izan/lib/python3.7/re.py", line 286, in _compile
    p = sre_compile.compile(pattern, flags)
  File "/home/izan/lib/python3.7/sre_compile.py", line 764, in compile
    p = sre_parse.parse(p, flags)
  File "/home/izan/lib/python3.7/sre_parse.py", line 930, in parse
    p = _parse_sub(source, pattern, flags & SRE_FLAG_VERBOSE, 0)
  File "/home/izan/lib/python3.7/sre_parse.py", line 426, in _parse_sub
    not nested and not items))
  File "/home/izan/lib/python3.7/sre_parse.py", line 651, in _parse
    source.tell() - here + len(this))
re.error: nothing to repeat at position 0
>>>
```

What just happened with last two patterns?
There are some metacharacters, which have special meaning while matching a pattern with regexes, these are:

. ^ $ * + ? { } [ ] \ | ( )

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 []

They’re used for specifying a ***character class***, which is a set of characters that you wish to match.

Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a hyphen (-).

```
>>> 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']
>>>
>>> re.findall (pattern, text)
['1', '2', '1', '1']
>>>
```

All 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']
>>>
>>> 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. Carat ^

This metacharacter will simply match the pattern that starts with character(s) followed after ‘^’ i.e., it matches at the beginning of line.

```
>>> text = 'The Army Of Dead'
>>> pattern = '^T'
>>> re.findall (pattern, text)
['T']
>>>
>>> text = 'In the southern principality of Dorne, Ellaria Sand (Indira Varma) seeks vengeance
against the Lannisters.'
>>> 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 = 'Later seasons? In later seasons, critics pointed out that certain characters had developed
"plot armor" to survive in unlikely circumstances; and attributed this to Game of Thrones deviating
from the novels to become more of a traditional television series. The series also reflects the
substantial death rates in war!'
>>> pattern = '[^!?.;,"]'
>>> re.findall (pattern, text)
['L', 'a', 't', 'e', 'r', ' ', 's', 'e', 'a', 's', 'o', 'n', 's', ' ', 'I', 'n', ' ', 'l', 'a', 't', 'e', 'r', ' ', 's', 'e', 'a', 's', 'o', 'n', 's', ' ', 'c',
'r', 'i', 't', 'i', 'c', 's', ' ', 'p', 'o', 'i', 'n', 't', 'e', 'd', ' ', 'o', 'u', 't', ' ', 't', 'h', 'a', 't', ' ', 'c', 'e', 'r', 't', 'a', 'i', 'n', ' ',
'c', 'h', 'a', 'r', 'a', 'c', 't', 'e', 'r', 's', ' ', 'h', 'a', 'd', ' ', 'd', 'e', 'v', 'e', 'l', 'o', 'p', 'e', 'd', ' ', 'p', 'l', 'o', 't', ' ', 'a',
'r', 'm', 'o', 'r', ' ', 't', 'o', ' ', 's', 'u', 'r', 'v', 'i', 'v', 'e', ' ', 'i', 'n', ' ', 'u', 'n', 'l', 'i', 'k', 'e', 'l', 'y', ' ', 'c', 'i', 'r', 'c',
'u', 'm', 's', 't', 'a', 'n', 'c', 'e', 's', ' ', 'a', 'n', 'd', ' ', 'a', 't', 't', 'r', 'i', 'b', 'u', 't', 'e', 'd', ' ', 't', 'h', 'i', 's', ' ', 't', 'o',
' ', 'G', 'a', 'm', 'e', ' ', 'o', 'f', ' ', 'T', 'h', 'r', 'o', 'n', 'e', 's', ' ', 'd', 'e', 'v', 'i', 'a', 't', 'i', 'n', 'g', ' ', 'f', 'r', 'o', 'm',
' ', 't', 'h', 'e', ' ', 'n', 'o', 'v', 'e', 'l', 's', ' ', 't', 'o', ' ', 'b', 'e', 'c', 'o', 'm', 'e', ' ', 'm', 'o', 'r', 'e', ' ', 'o', 'f', ' ', 'a', '
', 't', 'r', 'a', 'd', 'i', 't', 'i', 'o', 'n', 'a', 'l', ' ', 't', 'e', 'l', 'e', 'v', 'i', 's', 'i', 'o', 'n', ' ', 's', 'e', 'r', 'i', 'e', 's', ' ', 'T', 'h',
'e', ' ', 's', 'e', 'r', 'i', 'e', 's', ' ', 'a', 'l', 's', 'o', ' ', 'r', 'e', 'f', 'l', 'e', 'c', 't', 's', ' ', 't', 'h', 'e', ' ', 's', 'u', 'b', 's', 't',
'a', 'n', 't', 'i', 'a', 'l', ' ', 'd', 'e', 'a', 't', 'h', ' ', 'r', 'a', 't', 'e', 's', ' ', 'i', 'n', ' ', 'w', 'a', 'r']
>>>
```

4. Dollar $

It matches at the end of a line, which is defined as either the end of the 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', 'oa', 't', 's', 'a', 'n', 'd', 'ba', 'l', 'ma', 'caa', 'n', 's', 'i', 'n', 's', 'i', 'l', 'v', 'e', 'r']
>>>
>>> text = 'He suffers from hippopotomonstrosesquipedaliophobia.'
>>>
>>> # ‘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 hippopotomonstrosesquipedaliophobia.'
>>>
>>> #‘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', 'oa', 't', 's', 'a', 'n', 'd', 'ba', 'l', 'ma', 'ca', 'a', 'n', 's', 'i', 'n',
's', 'i', 'l', 'v', 'e', 'r']
>>>
>>> text = 'He suffers from hippopotomonstrosesquipedaliophobia.'
>>>
>>> #‘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 = 'Zxzxzzxzzzxzzzzxxz'
>>> 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, it's also used to escape all the metacharacters so you can still match them in patterns.

Unfortunately, a backslash must itself be escaped in normal Python strings, and that results in expressions that are difficult to read.

Using *raw strings*, created by prefixing the literal value with *r*, eliminates this problem and maintains 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 regexes, A|B will match any string that matches either A or B.
To match a literal '|', use \|, or 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)

This method returns a *match object* containing information about the match, where it starts and ends, the substring it matched,
and more; if match is found, else returns a None object.

```
>>> text = 'Dark Wings Dark Words'
>>> pattern = 'Dark'
>>> re.search (pattern, text)
<re.Match object; span=(0, 4), match='Dark'>
>>>
```

.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 the ***( )*** metacharacters.

Parentheses group together the expressions contained inside them, and you 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'
>>>
```

In the above example if you want to separate the *Batch*, *Branch* and *id* of a student you can use grouping in you regex pattern.

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 of a pattern splits a string apart wherever the regex matches, returning a list of the pieces.

```
>>> text = 'Floccinaucinihilipilification'
>>> pattern = 'i'
>>> re.split (pattern, text)
['Flocc', 'nauc', 'n', 'h', 'l', 'p', 'l', 'f', 'cat', 'on']
>>>
>>> text = 'Dd38b98bb497ba994e9a2b368cf0b7c7282323328964dd63f'
>>> pattern = r'\d'
>>> re.split (pattern, text)
['Dd', '', 'b', '', 'bb', '', '', 'ba', '', '', 'e', 'a', 'b', '', '', 'cf', 'b', 'c', '', '', '', '', '', '', '', '', '', '', '', '', 'dd', '', 'f']
>>>
>>> pattern = r'\d\d'
>>> re.split (pattern, text)
['Dd', 'b', 'bb', '7ba', '4e9a2b', '8cf0b7c', '', '', '', '', '', '4dd', 'f']
>>>
```

2.4 Contact Extractor

Write a script to extract all emails and contact numbers from a given text file.

contact_extractor.py

```
import re

with open ('your_text_file') 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]+.com"

email_list = re.findall(email_pattern, text_doc)
```

Grow through,
what you go through

1. Palindrome

Write a function that takes a string as argument and returns True if it is palindrome i.e., a string that is equal to its reversed version.
e.g 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
```

You can easily reduce this code to one line, if you use slicing but you should first try to develop the logic how you can approach to a certain problem manually.

palindrome.py

```
def palindrome (string):
    return (string.lower () == string [::-1].lower ())
```

2. Armstrong Number

Write a function that takes an integer as argument and returns True if it is an Armstrong number i.e., sum of digits raised to power of total digits equals the number itself.

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

    else:
        return False


number = armstrong (371)
print (number)
```

3. Count Vowels

Write a function that takes a string as argument and counts the number of vowels in the text.

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 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
```

You can also use *dict.setdefault()* method

character_count.py

```
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
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. Tower of hanoi

Three pegs *A*, *B*, *C* and different sized disks are given.

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 to be followed during transfer:

1. 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.

2. Only one disk can be moved at a time.

3. 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} \n")
      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
```

7. 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
```

8. String reversal

Write a function that takes a string as argument and returns a reversed copy of it.

string_rev.py

```
def reverse (string):
   s = ''

   for _ in range (len (string)):
      s += string [len (string) - 1]
      string = string [:(len (string) - 1)]

   return s
```

string_rev.py

```
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)
```

9. Integer reversal

Write a function that takes an integer as argument and returns reverse order of digits of that integer.

int_reversal.py

```
def rev_int (num):
    string = ""

    if num == 0:
        return 0

    if 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)
```

10. Anagrams

Write a function that takes two strings as its arguments and returns True if those two strings are anagrams of each other i.e., you can create string2 by rearranging the letters of string1.

e .g *public relations* is anagram of *crap built on lies*

anagram.py

```
def Anagram(s1, s2):
    s1 = s1.replace(' ', '').lower()
    s2 = s2.replace(' ', '').lower()

    if len(s1) == len(s2):

        for i in s1:
            if i not in s2:
                return False

        return True

    return False
```

11. 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 this without slicing.

e.g. 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
```

12. Collatz Conjecture

Start with a number 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.

collat_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.')
```

13. Largest Continuous Sum

Write a function that takes a list as 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
```

14. Most Repeated Character

Write a function that takes a string as 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
```

15. Capitalized String

Write a function that takes a string as argument and returns the capitalized copy of it.

You can use *str.title()* which is built-in string method that automatically capitalizes each word of the string.

capitalized_string.py

```
def capitalized_string (string):
    string = string.replace (string[0], string[0].upper())

    for i in range (len(string)):
        if string[i] == ' ':
            string = string.replace (string[i+1], string[i+1].upper())

    return string
```

capitalized_string.py

```
def capitalized_string (string):
    list_ = string.split()

    list_ = list (map (str.capitalize, list_ ))

    return ' '.join(list_)
```

16. Unique Characters

Write a function that takes a string as 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
```

unique_char.py

```
def uni_char (s):
    return (len (set (s)) == len (s))
```

17. Linear Search

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.

Write a function to implement the linear search algorithm (also known as sequential search).

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 sorted.
It is among the famous computer algorithms.

binary_search.py

```
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
```

binary_search.py

```
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

calculator.py

```
def calculate(num1, op, num2):
    if op == "+":
        return num1+num2

    if op == "-":
        return num1-num2

    if op == "*":
        return num1*num2

    if op == "/":
        return num1/num2

    if op == "//":
        return num1//num2

    if 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}")
```

```
Hit "Ctrl+C" to exit.

Number: 11
Operator: +
Number: 22
Ans: 33
Operator: *
Number: 33
Ans: 1089
Operator: ^CTraceback (most recent call last):
  File "test.py", line 31, in <module>
    op = input ('Operator: ')
KeyboardInterrupt
>>>
```

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 randomly generates 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):
        playerNumber = int (input ("Try your luck! \t \t [Enter the number between 1 to 10] \n"))


            if (playerNumber == secretNumber):
                  print ("\nYou were lucky this time \n")
                  break

            elif (playerNumber > secretNumber):
               print ("\nHard Luck! \t \t [ Hint: Try a smaller number ] \nTries left: ", i-2)

            elif (playerNumber < secretNumber):
               print ("\nHard Luck! \t \t [ Hint:  Try a higher value ]\nTries left: ", i-2)

            else:
                  print ("You're not entering a valid value")


 #if the loop is executed completely without being broken else will execute
      else:
            print("\nGo and watch pogo\n")


playerName = input ("If you're lame, then don't Enter your Name:\n")
print (f"{playerName}! Welcome  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 looser!")
            break
```

21. Tic Tac Toe

This is your milestone project and before looking at the source code try to solve this problem by yourself. Remember errors are your best friends so dont 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 first goal must be to solve a problem no matter how long your logic is and then you can revisit your script and optimize your code.

TicTacToe.py

```
from random import randint

#--------------------------- main starts ----------------------------#

print ('\t\t\tWelcome to Tic Tac Toe!\n')
print (""" These are the locations on the board

        |----------------|----------------|----------------|
        |                |                |                |
        |       1        |       2        |       3        |
        |                |                |                |
        |----------------|----------------|----------------|
        |                |                |                |
        |       4        |       5        |       6        |
        |                |                |                |
        |----------------|----------------|----------------|
        |                |                |                |
        |       7        |       8        |       9        |
        |                |                |                |
        |----------------|----------------|----------------|

        """)


name1 = input ('Player One: ')
name2 = input ('Player Two: ')

print (f"\n{name1}! You're 'X'. \t {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)]
#Main will continue after function definitions . . .
```

```
#---------------------------- function definitions -----------------------------#

#Board
def display_board (  ):
        print (f"""
                |----------------|-----------------|-----------------|
                |                |                 |                 |
                |    {B[0]}      |    {B[1]}       |    {B[2]}       |
                |                |                 |                 |
                |----------------|-----------------|-----------------|
                |                |                 |                 |
                |    {B[3]}      |    {B[4]}       |    {B[5]}       |
                |                |                 |                 |
                |----------------|-----------------|-----------------|
                |                |                 |                 |
                |    {B[6]}      |    {B[7]}       |    {B[8]}       |
                |                |                 |                 |
                |----------------|-----------------|-----------------|


              \n""")



#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"):
        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 psuedo programming language you can say.
It recognizes only on 8 operators **`[ ] < > , . + −`**
everything else is ignored by its interpreter i.e., taken as comment.

, 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 Google *BrainFuck visualizer*, to visualize how it interprets everything based on ASCII values.

This is the only project I’m leaving to you.

Sorting Algorithms

(Implementation only)

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
```

bubble_sort.py

```
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
```

bubble_sort.py

```
#yen optimization

def bubble_sort_optimized (arr):
    n = len (arr)
    swapping = 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
```

selection_sort.py

```
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"Merging...\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


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
```

Data Structures

(Implementation only)

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()}')
```

```
a.isEmpty(): True
a.pop(): Stack is empty!
a.top(): Stack is empty!
a.top(): 32
a.pop(): 32
a.isEmpty(): True
```

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)}')
```

```
a.isEmpty(): True
a.rear(): 60
a.isEmpty(): False
len(a): 6
a.dequeue(): 10
a.front(): 20
len(a): 5
```

---

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()}')
```

```
a.is_empty(): True
len(a): 0
a.remove_front(): 40
a.remove_rare(): 80
a.remove_front(): 30
a.remove_rare(): 10
a.remove_front(): 20
a.remove_rare(): Queue is Empty!
a.remove_front(): Queue is Empty!
```

---

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()}')
```

```
55
44
33
22
11
a.dequeue(): 11
99
55
44
33
22
a.dequeue(): 22
a.dequeue(): 33
a.dequeue(): 44
111
99
55
a.dequeue(): 55
a.dequeue(): 99
a.dequeue(): 111
a.dequeue(): Empty!
```

---

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



a = SLL (4)

print (f"Root: {a.root.data} \nLeaf: {a.leaf.data}")

a.display()

print ('\n')
a.insert (55,5)
a.display()

print ('\n')
a.insert (444,5)
a.display()

print (f"\nRoot: {a.root.data} \nLeaf: {a.leaf.data}")

print ('\n')
a.insert (0,1)
a.display()

print (f"\nRoot: {a.root.data} \nLeaf: {a.leaf.data}")

print ('\n')
a.delete (1)
a.display()

print (f"\nRoot: {a.root.data} \nLeaf: {a.leaf.data}")

print ('\n')
a.delete (6)
a.display()

print (f"\nRoot: {a.root.data} \nLeaf: {a.leaf.data}")

print ('\n')
a.delete (3)
a.display()
```

```
Node 1: 11
Node 2: 22
Node 3: 33
Node 4: 44
Root: 11
Leaf: 44
|11|   |22|   |33|   |44|

|11|   |22|   |33|   |44|   |55|

|11|   |22|   |33|   |44|   |444|   |55|
Root: 11
Leaf: 55


|0|   |11|   |22|   |33|   |44|   |444|   |55|
Root: 0
Leaf: 55


|11|   |22|   |33|   |44|   |444|   |55|
Root: 11
Leaf: 55


|11|   |22|   |33|   |44|   |444|
Root: 11
Leaf: 444


|11|   |22|   |44|   |444|
```

6. Hash Tables

This hash table takes integer keys only.

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]
            #return True

        next_index = self.rehash (index)

        while (next_index != index):

            if self.key[next_index] == key:
                return self.data [next_index]
                #return True

            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])
```

```


Apple added!
Banana added!
Cherry added!
Grapes added!
Lemon added!


8 updated!


8 added!


Apple
Banana
Cherry
Grapes
Orange
8
```

---

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()
```

```
56 inserted!
40 inserted!
60 inserted!
60
```

---

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 node ---> this is root node
    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')
```

```
right-left heavy situation
Rotating to right on node 78
rotating to left on node 70
left-right heavy situation
rotating to left on node 17
Rotating to right on node 44


5
8
17
20
30
35
40
44
70
75
78


5 is removed from tree
8 is removed from tree
20 is removed from tree
rotating to left on node 30


17
30
35
40
44
70
75
78
```

---

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())


    #__specialmethods__

    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)
```

```
A added to graph!
B added to graph!
C added to graph!
D added to graph!
E added to graph!
F added to graph!
G added to graph!
H added to graph!


A is connected to E


B is connected to C
B is connected to E
B is connected to F


C is connected to B
C is connected to D
C is connected to F
C is connected to G


D is connected to G
D is connected to H
D is connected to F
D is connected to C


E is connected to F
E is connected to A
E is connected to B


F is connected to B
F is connected to E
F is connected to D
F is connected to C


G is connected to D
G is connected to C


H is connected to D

D
```

10. Breadth First Search and Depth First Search

graph_traversal.py

```
class Node():
    def __init__(self,name):
        self.name = name
        self._adjacent_vertices = []
        self.visited = False
        self.predcessor = 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)
print ('\n')
```

```
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 ('Depth First Search:')
DFS (node_1)
```

```
Breadth First Search:
A
B
E
C
D


Depth First Search:
A
B
D
E
C
```

---

11. 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())



    #__specialmethods__
    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)
```

```
A added to graph!
B added to graph!
C added to graph!
D added to graph!
E added to graph!
F added to graph!


A is connected to B
A is connected to C
B is connected to E
B is connected to C
C is connected to D
D is connected to E
D is connected to F
E is connected to F


Shortest path costs: 9 units
F
E
D
C
B
A
```

---

12. 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 dectected")
            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)
```

```
F
E
B
C
D
A

Shortest path cost: 3 units


XYZ graph:
Negative weighted cycle dectected
```

Glossary

(source: pydocs)

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, it must be 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 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 are 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.

What next?

Hacker rank, code chef, geeks for geeks, solo learn and lot of other online resources are available where you can find practice problems. All you need after reading this book is a good internet connection to solve real problems that will brush up your skills.

If you ever get stuck at some point, visit *stackoverflow* or some other online community. Most probably your question may be already answered there. You can also join *github* where you can explore other people’s projects.