Python? → high level language
human readable syntax

Python Fundamentals

Places where python is used ?

1. Data Science
2. AI
3. Website
4. Data Engineering.
5. Gaming Industry.
6. IoT
7. Mobile Development
8. API / Microservices

Topics

- Running Python code
- Using Python Development Tools (IDEs and command line tools)
- Working with Python and iPython shells as well as iPython Notebook

Hello World

class HW {

p — s — ( ) {

System.out.println("Hello World");

} }

javac H-W.java

java -HelloW -

Java v/s Python

→ Compiled → Interpreted

print("Hello World").

Running Python code

What is Python?

class, object

- An interpreted, object-oriented, high-level programming language with dynamic semantics.
- Has high-level built in data structures, combined with dynamic typing and dynamic binding.
- Very attractive for:
  - Rapid Application Development
  - Use as a scripting or glue language to connect existing components together.
- Has a simple, easy to learn syntax which:
  - Emphasizes readability and therefore reduces the cost of program maintenance.

focus on human readability

Low High

Human Readable

Python, Java, JS → High

C → Low

Machine Readable

Python

Java

Java

schedule

4 pm

.py

What is Python?

package → module

single file

collection of multiple files

- Supports modules and packages, which:
  - Encourages program modularity and code reuse.
- The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

'jar' → binary / source

import a module

from a package import a module

Running Python Code?

- A Python Interpreter is needed to run Python code.
- Python code can be run in different ways such as [but not limited to]:
  - Using the Command-Line
  - Using an IDE
  - Using a Text Editor

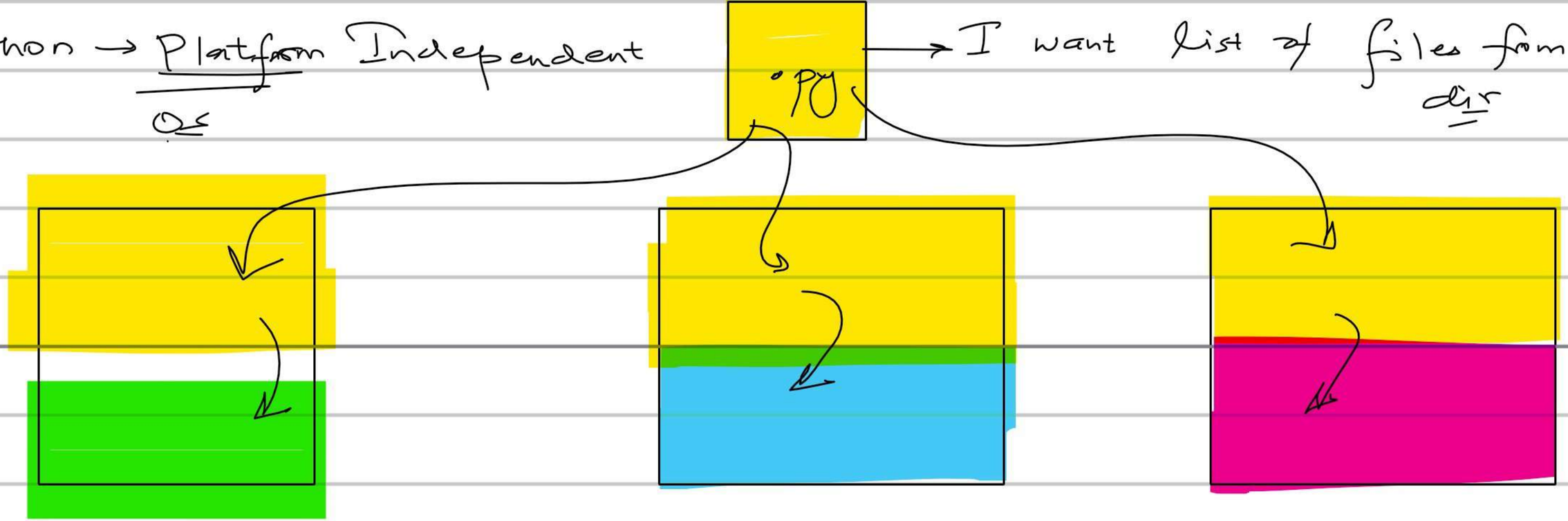
What is Python Interpreter?

Source

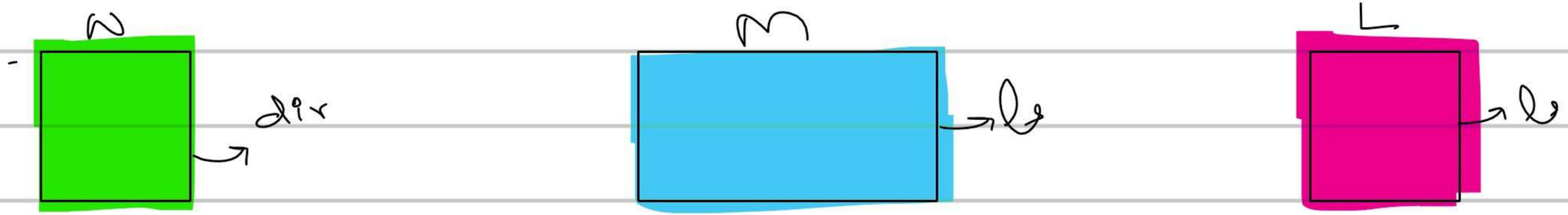
- Is the program you'll need to run Python code and scripts.
- Is a layer of software that works between your program and your computer hardware to get your code running.
- The interpreter is able to run Python code in two different ways:
  - As a:
    - Script: A plain text file containing Python code [logical sequence of orders] that is executed directly.
    - Module: A plain text file, which contains Python code that is designed to be imported and used from another Python file.
  - As a piece of code typed into an interactive session

Python → Platform Independent

OS



→ I want list of files from a dir



How Does the Interpreter Run Python Scripts?

Abc.java → high
↓
compile
↓
Abc.class → low

- When you try to run Python scripts, a multi-step process begins.
- In this process the interpreter will:
  - Process the statements of your script in a sequential fashion
  - Compile the source code to an intermediate format known as bytecode
    - Bytecode is a translation of the code into a lower-level language that's platform-independent.
    - Its purpose is to optimize code execution.
    - So, the next time the interpreter runs your code, it'll bypass this compilation step.
    - Strictly speaking, code optimization is only for modules (imported files), not for executable scripts.

How Does the Interpreter Run Python Scripts?

module

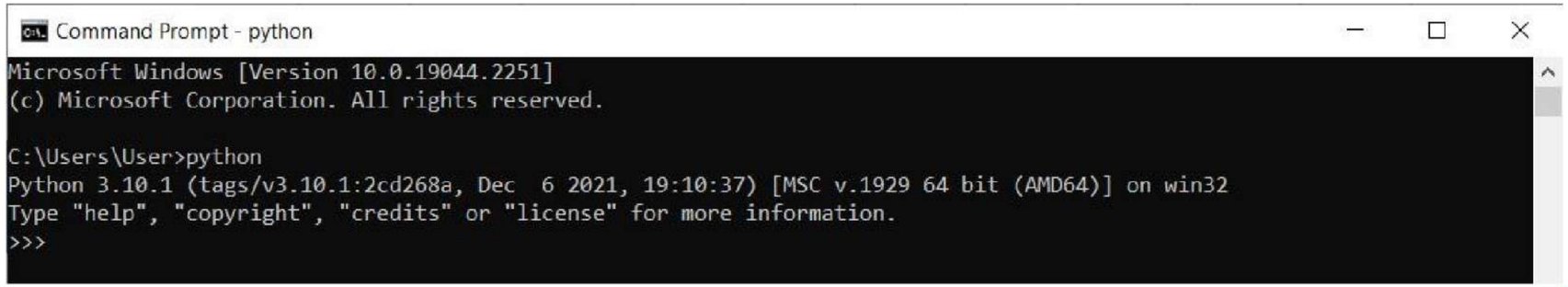
1, 3, 4, 9

big-number.py

- Ship off the code for execution
  - At this point, something known as a Python Virtual Machine (PVM) comes into action.
  - The PVM is the runtime engine of Python.
  - It is a cycle that iterates over the instructions of your bytecode to run them one by one.
  - The PVM is not an isolated component of Python.
  - It's a part of the Python system you've installed on your machine.
  - Technically, the PVM is the last step of what is called the Python interpreter.

Running Python Code Interactively

- A widely used way to run Python code is through an **interactive session.**
- To start a Python interactive session:
  - Open a command-line or terminal
  - Type in **python**, or **python3** depending on your Python installation
  - Then press **Enter.**
  - The standard prompt for the interactive mode is "**>>>**"



Running Python Code Interactively

- Now, you can start typing your code lines

Command Prompt - python

```
Microsoft Windows [Version 10.0.19044.2251]
(c) Microsoft Corporation. All rights reserved.

C:\Users\User>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Welcome to Python')
Welcome to Python
>>>
```

- To exit interactive mode, you can use one of the following options:
  - **quit**() or **exit**(), which are built-in functions
  - The **Ctrl+Z** and **Enter** key combination on Windows

① Command prompt

② Let's write a script →

③ Use Notebooks

Running Python Scripts Using the Command-Line

- A Python interactive session will allow you to write a lot of lines of code
- But once you close the session, you lose everything you've written
- That's why the usual way of writing Python programs is by using plain text files
- By convention, those files will use the **.py** extension
- On Windows systems the extension can also be **.pyw**

Running Python Scripts Using the Command-Line

- You can use any Text Editor to write and save your code, such as:

- Then, execute the file through **cmd**, using the **python** or **python3** commands

```
C:\Users\User\source\repos\MyPy>python FirstHello.py
Welcome to my Python file
```

Running Python Scripts Interactively

- It is also possible to run Python scripts and modules from an interactive session.
  - Taking Advantage of import
    - When you import a module, its contents get loaded for later access and use.
    - The interesting thing about this process is that import runs the code as its final step.

```
C:\Users\User\source\repos\MyPy>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import FirstHello
Welcome to my Python file
>>>
```

  - Hacking the **exec** command

```
>>> exec(open('FirstHello.py').read())
Welcome to my Python file
>>>
```

Running Python Scripts Interactively

- It is also possible to run Python scripts and modules from an interactive session.
  - Using runpy.run_module() and runpy.run_path()
    - The Standard Library includes a module called runpy.
    - In this module, you can find run_module(), which is a function that allows you to run modules without importing them first.
    - This function returns the globals dictionary of the executed module.

```
C:\Users\User\source\repos\MyPy>python
Python 3.10.1 (tags/v3.10.1:2cd268a, Dec  6 2021, 19:10:37) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> runpy.run_module(mod_name='FirstHello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'runpy' is not defined
>>> import runpy
>>> runpy.run_module(mod_name='FirstHello')
Welcome to my Python file
{'__name__': 'FirstHello', '__file__': 'C:\\Users\\User\\source\\repos\\MyPy\\FirstHello.py', '__
User\\source\\repos\\MyPy\\__pycache__\\FirstHello.cpython-310.pyc', '__doc__': None, '__loader__
xternal.SourceFileLoader object at 0x0000021130F5A290>, '__package__': '', '__spec__': ModuleSpec
der=<_frozen_importlib_external.SourceFileLoader object at 0x0000021130F5A290>, origin='C:\\Users
MyPy\\FirstHello.py'), '__builtins__': {'__name__': 'builtins', '__doc__': "Built-in functions, ex
jects.\n\nNoteworthy: None is the `nil' object; Ellipsis represents `...' in slices.", '__package_
<class '_frozen_importlib.BuiltinImporter'>, '__spec__': ModuleSpec(name='builtins', loader=<clas
```

Using Python Development Tools (IDEs and command line tools)

Using IDEs or a Text Editor

- When developing larger and more complex applications, it is recommended that you use an:
  - Integrated development environment (IDE) or
  - Advanced text editor.
- Most of these programs offer the possibility of running your scripts from inside the environment itself.
- It is common for them to include a *Run* or *Build* command, which is usually available from the tool bar or from the main menu.
- Python's standard distribution includes **IDLE** as the default IDE, and you can use it to write, debug, modify, and run your modules and scripts.

Using IDEs or a Text Editor

- Other IDEs that allow you to run Python scripts from inside the environment such as:
  - Eclipse-PyDev
  - PyCharm
  - Eric
  - NetBeans
- Advanced text editors that allow you to run your scripts are like:
  - Sublime Text
  - Visual Studio Code

Working with Python and iPython shells as well as iPython Notebook

Using IPython (Interactive Python)

- A command shell for interactive computing in multiple programming languages
- Originally developed for the Python programming language
- Offers:
  - Introspection
  - Rich media
  - Shell syntax
  - Tab completion
  - History

Using IPython (Interactive Python)

- IPython provides the following features:
  - Interactive shells (terminal and Qt-based).

```
(base) C:\Users\User>ipython
Python 3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.22.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('Hello again')
   ...: print('Done..!')
Hello again
Done..!

In [2]:
```

```
In [3]: exit()

(base) C:\Users\User>
```

  - A browser-based notebook interface with support for code, text, mathematical expressions, inline plots and other media.
  - Support for interactive data visualization and use of GUI toolkits.
  - Flexible, embeddable interpreters to load into one's own projects.
  - Tools for parallel computing.

Installing IPython

- Quick Install
  - With pip already installed, type the following command:

```
pip install ipython
```

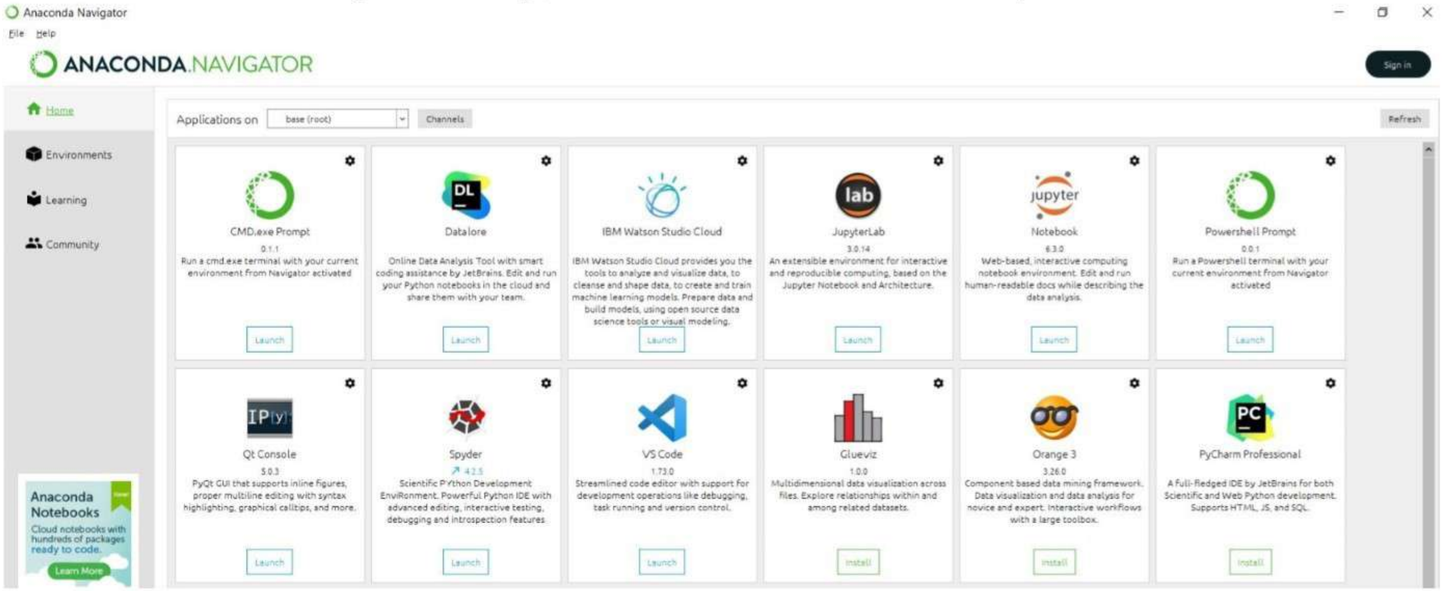
  - Or download and install Continuum's Anaconda
  - Then, update both if needed from within the Anaconda prompt:

  - Using the following commands:

```
conda update conda
conda update ipython
```

Using IPython Notebook

- Is now known as the Jupyter Notebook.
- It is an interactive computational environment, in which you can combine code execution, rich text, mathematics, plots and rich media.
- If you have already downloaded Anaconda, you will be able to use Jupyter Notebook right away, because it comes as part of it's installation.



Python Fundamentals

Topics

- Integers and floats
- Strings and bytes
- Tuples and lists
- Dictionaries and ordered dictionaries
- Sets and frozen sets

Data Types & Data Structures in Python

// comment in Java

comment

Integers and floats

Numbers

- Number data types store numeric values.
- They are immutable data types:
  - Meaning that changing the value of a number data type results in a newly allocated object.
- Number objects are created when you assign a value to them:

```
var1 = 1
var2 = 10
```

- You can also delete the reference to a number object by using the **del** statement:

```
del var
del var_a, var_b
```

Numerical Data Types

- Python supports four different numerical types:
  - **int (signed integers):**
    - They are often called just integers or ints, are positive or negative whole numbers with no decimal point.
  - **long (long integers):**
    - Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.
  - **float (floating point real values):**
    - Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts.
    - Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 10^2 = 250).
  - **complex (complex numbers):**
    - Are of the form a + bJ, where a and b are floats and J (or j) represents the square root of -1 (which is an imaginary number).

Numerical Data Types

```
# integer variable.
a=100
print("The type of variable having value", a, " is ", type(a))

# float variable.
b=20.345
print("The type of variable having value", b, " is ", type(b))
```

Number Type Conversion

- Python converts numbers internally in an expression containing mixed types to a common type for evaluation.
- But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.
- You can use one of the following methods:
  - Type **`int(x)`** to convert x to a plain integer.
  - Type **`long(x)`** to convert x to a long integer.
  - Type **`float(x)`** to convert x to a floating-point number.

Number Type Conversion

```
a = int(1)     # a will be 1
b = int(2.2)   # b will be 2
c = int("3")   # c will be 3

print (a)
print (b)
print (c)
```

This produce the following result

```
1
2
3
```

```
a = float(1)     # a will be 1.0
b = float(2.2)   # b will be 2.2
c = float("3.3") # c will be 3.3

print (a)
print (b)
print (c)
```

This produce the following result

```
1.0
2.2
3.3
```

Strings and bytes

Strings

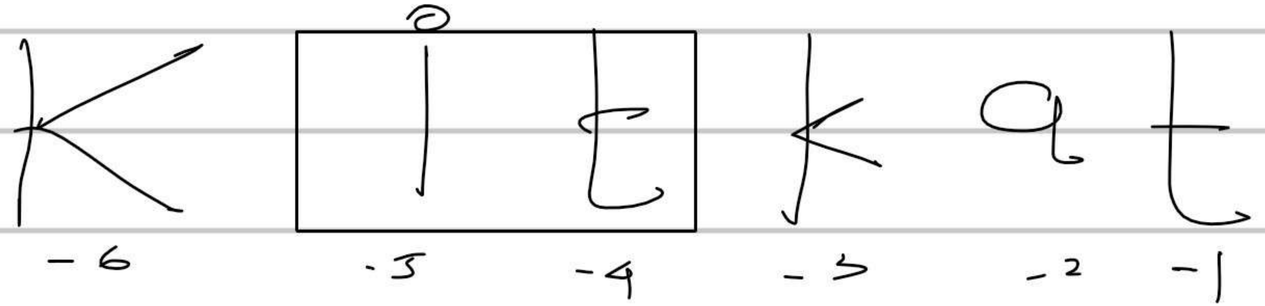
- Strings are amongst the most popular types in Python.
- We can create them simply by enclosing characters in quotes.
- Python treats single quotes the same as double quotes.

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

word[start_index : end_index]

0 based 1 based

1 → 1 2 $\boxed{3}$ 4 5 6



0 → 0 $\boxed{1}$ 2 3 4 5

word[1, 3]

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

word = "I n t e r n a t i o n a l i z a t i o n"

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Inter [0:5] at [6:8] [15:17]

nation [5:11] ion [8:11] [17:] [-3:]

national [5:13] liz [12:15]

Intern [0:6] on [9:11] [18:] [-2:]

Accessing Values in Strings

- Python does not support a character type.
- These are treated as strings of length one, thus also considered a substring.
- To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring.

```
#!/usr/bin/python

var1 = 'Hello World!'
var2 = "Python Programming"

print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

When the above code is executed, it produces the following result

```
var1[0]:  H
var2[1:5]:  ytho
```

Updating Strings

- You can "update" an existing string by (re)assigning a variable to another string.
- The new value can be related to its previous value or to a completely different string altogether.

```
#!/usr/bin/python

var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

When the above code is executed, it produces the following result

```
Updated String :-  Hello Python
```

String Special Operators

Assume string variable **a** holds 'Hello' and variable **b** holds 'Python', then −

Operator	Description	Example
+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell
in	Membership - Returns true if a character exists in the given string	H in a will give 1
not in	Membership - Returns true if a character does not exist in the given string	M not in a will give 1
r/R	Raw String - Suppresses actual meaning of Escape characters. The syntax for raw strings is exactly the same as for normal strings with the exception of the raw string operator, the letter "r," which precedes the quotation marks. The "r" can be lowercase (r) or uppercase (R) and must be placed immediately preceding the first quote mark.	print r'\n' prints \n and print R'\n'prints \n
%	Format - Performs String formatting	See at next section

String Formatting Operator

- One of Python's coolest features is the string format operator %.
- This operator is unique to strings and makes up for the pack of having functions from C's printf() family.

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

When the above code is executed, it produces the following result

```
My name is Zara and weight is 21 kg!
```

String Formatting Operator

Here is the list of complete set of symbols which can be used along with % −

Format Symbol	Conversion
%c	character
%s	string conversion via str() prior to formatting
%i	signed decimal integer
%d	signed decimal integer
%u	unsigned decimal integer
%o	octal integer
%x	hexadecimal integer (lowercase letters)
%X	hexadecimal integer (UPPERcase letters)
%e	exponential notation (with lowercase 'e')
%E	exponential notation (with UPPERcase 'E')
%f	floating point real number
%g	the shorter of %f and %e
%G	the shorter of %f and %E

Triple Quotes

- Python's triple quotes comes to the rescue by allowing strings to span multiple lines, including verbatim NEWLINEs, TABs, and any other special characters.
- The syntax for triple quotes consists of three consecutive **single or double** quotes.

```
#!/usr/bin/python

para_str = """this is a long string that is made up of
several lines and non-printable characters such as
TAB ( \t ) and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [ \n ], or just a NEWLINE within
the variable assignment will also show up.
"""
print para_str
```

Some Built-in String Methods

capitalize()

Capitalizes first letter of string

len(string)

Returns the length of the string

lstrip()

Removes all leading whitespace in string.

count(str, beg= 0,end=len(string))

Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.

rstrip()

Removes all trailing whitespace of string.

find(str, beg=0 end=len(string))

Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.

index(str, beg=0, end=len(string))

Same as find(), but raises an exception if str not found.

isdigit()

Returns true if string contains only digits and false otherwise.

isalpha()

Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.

replace(old, new [, max])

Replaces all occurrences of old in string with new or at most max occurrences if max given.

String Methods – Find()

- The python string **find()** method is used to return the index of the created string where the substring is found.
- Syntax:

```
str.find(str, beg=0, end=len(string))
```

- Parameters:

The following are the parameters of the python string **find()** method.

  - **str** − This parameter specifies the string that is to be searched.
  - **beg** − This parameter specifies the starting index. The default value is '0'.
  - **end** − This parameter specifies the ending index. The default value is the length of the string.

- Return value:

This function returns the index if found and **-1** otherwise.

String Methods – Find()

- Examples:

```
#!/usr/bin/python
str1 = "Hello! Welcome to Tutorialspoint."
str2 = "to";
result= str1.find(str2)
print("The index where the substring is found:", result)
```

Output:

```
The index where the substring is found: 15
```

```
#!/usr/bin/python
str1 = "Hello! Welcome to Tutorialspoint."
str2 = " ";
result= str1.find(str2, 12, 15)
print("The index where the substring is found:", result)
```

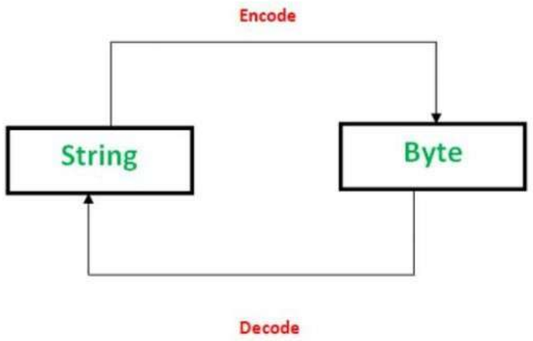
```
The index where the substring is found: 14
```

Bytes

- Byte objects are sequence of **Bytes**, whereas Strings are sequence of **characters**.
- Byte objects are in **machine readable** form internally, Strings are only in **human readable** form.
- Since Byte objects are machine readable, they can be **directly stored on the disk**.
- Whereas, Strings **need encoding** before which they can be stored on disk.

Strings to Bytes

- There are methods to convert a byte object to String and String to byte objects.
- The processes that can be performed to change from one type to another are known as:
  - Encoding
  - Decoding



Encoding

- PNG, JPEG, MP3, WAV, ASCII, UTF-8 etc are different forms of encodings.
- An encoding is a format to represent audio, images, text, etc in bytes.
- Converting **Strings to byte** objects is termed as encoding.
- This is necessary so that the text can be stored on disk using mapping using **ASCII** or **UTF-8** encoding techniques.
- This task is achieved using **encode()**.
- It take encoding technique as argument.
- Default technique is "**UTF-8**" technique.

Encoding

- Example

```
# Python code to demonstrate String encoding

# initialising a String
a = 'GeeksforGeeks'

# initialising a byte object
c = b'GeeksforGeeks'

# using encode() to encode the String
# encoded version of a is stored in d
# using ASCII mapping
d = a.encode('ASCII')

# checking if a is converted to bytes or not
if (d==c):
    print ("Encoding successful")
else : print ("Encoding Unsuccessful")
```

Output:

```
Encoding successful
```

Decoding

- Similarly, Decoding is process to convert a **Byte object to String**.
- It is implemented using **decode()**.
- A byte string can be decoded back into a character string, if you know which encoding was used to encode it.
- Encoding and Decoding are **inverse** processes.

Decoding

- Example:

```
# Python code to demonstrate Byte Decoding

# initialising a String
a = 'GeeksforGeeks'

# initialising a byte object
c = b'GeeksforGeeks'

# using decode() to decode the Byte object
# decoded version of c is stored in d
# using ASCII mapping
d = c.decode('ASCII')

# checking if c is converted to String or not
if (d==a):
    print ("Decoding successful")
else : print ("Decoding Unsuccessful")
```

Output:

```
Decoding successful
```

Java

```
if (condition) {
    statements
}
```

▽ → space

Python

```
if (condition):
▽▽▽▽statement
▽▽▽▽
▽▽▽▽
▽▽▽▽
else:
▽▽▽▽statement
print(
)
```

Conventions in Python

one two three fou

class → Pascal Casing → OneTwoThreeFour

variable, function → snake_casing → one_two_three_four

and

T	T	T
T	F	F
F	T	F
F	F	F

or

T	T	T
T	F	T
F	T	T
F	F	F

Java

condition ? output_if_true : output_if_false

output_if_true if condition else output_if_false

5

for :

1

2

3 → successful ✓

4

5

else : → execute only if the for loops runs completely without break

functions.

def name_of_function (parameters):

optional

return something.

A <u>function</u>

input

5 6 7

(1 2 3 4)

(5 6, 27, 58 9)

varying arguments

(5, 6, 7)

(1, 2, 3, 4)

(5, 6, 7, 8, 9)

```python
# cart=[("Name","Price","Qty")]

cart=[("Bread",15,2),("Butter",7,3),("Eggs",3,1)]

print("Cart:",cart)
```

```
Cart: [('Bread', 15, 2), ('Butter', 7, 3), ('Eggs', 3, 1)]
```

```python
cart.sort()
print("Sorted:",cart)
```

```
Sorted: [('Bread', 15, 2), ('Butter', 7, 3), ('Eggs', 3, 1)]
```

```python
sorted_cart=sorted(cart)
print("Sorted Cart:",sorted_cart)
```

```
Sorted Cart: [('Bread', 15, 2), ('Butter', 7, 3), ('Eggs', 3, 1)]
```

```python
def key_for_sorting(item):
    return item[1]
```

('Bread', 15, 2)
(B, 7, 3)
(E, 3, 1)

```python
cart.sort(key=key_for_sorting)
print("Sorted:",cart)
```

cart = 3

B, 15, 2 | B, 7, 3 | E, 3, 1

def 1

Lambda Functions

→ anonymous function

→ function without any names

No return statement

lambda parameter : expression

filter(,)

[1, 2, 3, 4, 5, 6]

%3

map( )

filter(what, source)

%3

map( )

Higher Order functions

filter

map

list

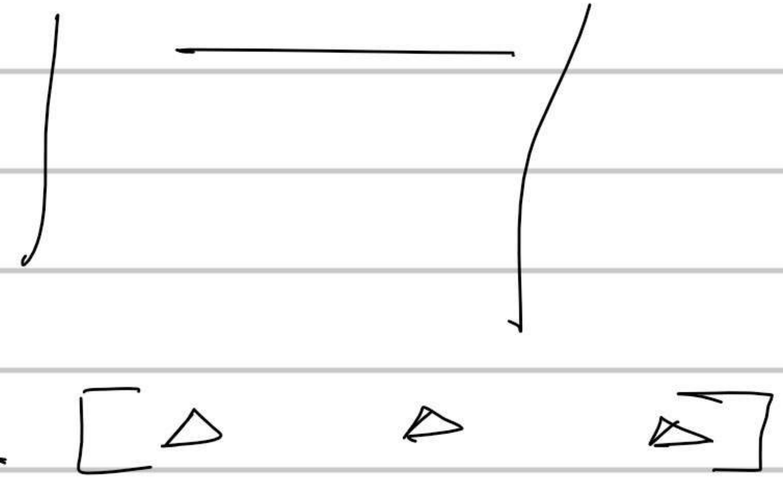
filter (function_where_filter_logic_is_written,
collection on which function).

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9]

function → def filter_divisible_by_3(number):
return number % 3 == 0.

filter (f — , numbers).

map (function, source)



[0, 0, 0]

[△ △ △]

numbers = [1, 2, 3, 4, 5]

[1, 4, 9

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

# map(function_where_transformation_logic_is_written,source_collection)

squared_numbers_lambda=list(map(lambda number: pow(number,3),numbers))

print("O:",numbers)
print("Mapped Numbers:",squared_numbers_lambda)
```

1 8 [1, 2, 3, 4, 5]

numbers - [1, 2, 3, 4, 5]

Tuples and lists

Tuples

- A tuple is an ordered and immutable collection of objects.
- Tuples are sequences, just like lists.
- Differences between tuples and lists:
  - Unlike lists, tuples cannot be changed
  - Tuples use parentheses, whereas lists use square brackets
- Examples:

```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

```
tup1 = ();  # Declared without initialization
```

Accessing Values in Tuples

- To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index.

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5, 6, 7 );
print "tup1[0]: ", tup1[0];
print "tup2[1:5]: ", tup2[1:5];
```

Output:

```
tup1[0]:  physics
tup2[1:5]:  [2, 3, 4, 5]
```

Updating Tuples

- Tuples are immutable which means you cannot update or change the values of tuple elements.
- You are able to take portions of existing tuples to create new tuples as the following example demonstrates:

```
#!/usr/bin/python

tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

Output:

```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

- Removing individual tuple elements is not possible.
- There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.
- To explicitly remove an entire tuple, just use the **del** statement.

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

Output:

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
   File "test.py", line 9, in <module>
      print tup;
NameError: name 'tup' is not defined
```

Basic Tuples Operations

- Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

Indexing, Slicing, and Matrixes

- Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings.
- Assuming following input:

```
L = ('spam', 'Spam', 'SPAM!')
```

Python Expression	Results	Description
L[2]	'SPAM!'	Offsets start at zero
L[-2]	'Spam'	Negative: count from the right
L[1:]	['Spam', 'SPAM!']	Slicing fetches sections

No Enclosing Delimiters

- Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

Output:

```
abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

Built-in Tuple Functions

cmp(tuple1, tuple2)	Compares elements of both tuples.
len(tuple)	Gives the total length of the tuple.
max(tuple)	Returns item from the tuple with max value.
min(tuple)	Returns item from the tuple with min value.
tuple(seq)	Converts a list into tuple.

Lists

- The list is the most versatile datatype available in Python.
- Can be written as a list of comma-separated values (items) between square brackets.
- Important thing about a list is that items in a list need not be of the same type.
- Example:

```
list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5 ];
list3 = ["a", "b", "c", "d"]
```

Accessing Values in Lists

- To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index.

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
list2 = [1, 2, 3, 4, 5, 6, 7 ];
print "list1[0]: ", list1[0]
print "list2[1:5]: ", list2[1:5]
```

Output:

```
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

- You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator
- You can add to elements in a list with the append() method.

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];
print "Value available at index 2 : "
print list[2]
list[2] = 2001;
print "New value available at index 2 : "
print list[2]
```

Output:

```
Value available at index 2 :
1997
New value available at index 2 :
2001
```

```
#!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc'];
aList.append( 2009 );
print "Updated List : ", aList
```

Output:

```
Updated List :  [123, 'xyz', 'zara', 'abc', 2009]
```

Deleting Lists

- To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];
print list1
del list1[2];
print "After deleting value at index 2 : "
print list1
```

Output:

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :
['physics', 'chemistry', 2000]
```

```
#!/usr/bin/python

aList = [123, 'xyz', 'zara', 'abc', 'xyz'];
aList.remove('xyz');
print "List : ", aList
aList.remove('abc');
print "List : ", aList
```

Output:

```
List :  [123, 'zara', 'abc', 'xyz']
List :  [123, 'zara', 'xyz']
```

Built-in List Functions & Methods

cmp(list1, list2)

Compares elements of both lists.

len(list)

Gives the total length of the list.

max(list)

Returns item from the list with max value.

min(list)

Returns item from the list with min value.

list(seq)

Converts a tuple into list.

list.append(obj)

Appends object obj to list

list.count(obj)

Returns count of how many times obj occurs in list

list.extend(seq)

Appends the contents of seq to list

list.index(obj)

Returns the lowest index in list that obj appears

list.insert(index, obj)

Inserts object obj into list at offset index

list.pop(obj=list[-1])

Removes and returns last object or obj from list

list.remove(obj)

Removes object obj from list

list.reverse()

Reverses objects of list in place

list.sort([func])

Sorts objects of list, use compare func if given

Dictionaries and ordered dictionaries

Dictionaries

- Each key is separated from its value by a colon (:)
- The items are separated by commas, and the whole thing is enclosed in curly braces.
- An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values might not be.
- The values of a dictionary can be of any type
- The keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

- To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.
- Examples:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Name']: ", dict['Name']
print "dict['Age']: ", dict['Age']
```

Output:

```
dict['Name']:  Zara
dict['Age']:  7
```

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print "dict['Alice']: ", dict['Alice']
```

Output:

```
dict['Alice']:
Traceback (most recent call last):
   File "test.py", line 4, in <module>
      print "dict['Alice']: ", dict['Alice'];
KeyError: 'Alice'
```

Updating Dictionary

- You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.
- Example:

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

Output:

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

- You can either remove individual dictionary elements or clear the entire contents of a dictionary.
- You can also delete entire dictionary in a single operation.

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name']; # remove entry with key 'Name'
dict.clear();     # remove all entries in dict
del dict ;        # delete entire dictionary

print "dict['Age']: ", dict['Age']
print "dict['School']: ", dict['School']
```

Output:

```
dict['Age']:
Traceback (most recent call last):
   File "test.py", line 8, in <module>
      print "dict['Age']: ", dict['Age'];
TypeError: 'type' object is unsubscriptable
```

Some Built-in Dictionary Methods

dict.copy()

Returns a shallow copy of dictionary *dict*

dict.fromkeys()

Create a new dictionary with keys from seq and values *set* to *value*.

dict.get(key, default=None)

For *key* key, returns value or default if key not in dictionary

dict.has_key(key)

Returns *true* if key in dictionary *dict*, *false* otherwise

dict.items()

Returns a list of *dict*'s (key, value) tuple pairs

dict.keys()

Returns list of dictionary dict's keys

dict.setdefault(key, default=None)

Similar to get(), but will set dict[key]=default if *key* is not already in dict

dict.update(dict2)

Adds dictionary *dict2*'s key-values pairs to *dict*

dict.values()

Returns list of dictionary *dict*'s values

Ordered Dictionary

- An **OrderedDict** is a dictionary subclass that remembers the order that keys were first inserted.
- Differences between dict() and OrderedDict():
  - OrderedDict **preserves the order** in which the keys are inserted.
  - A regular dict doesn't track the insertion order and iterating it gives the values in an arbitrary order.
  - By contrast, the order the items are inserted is remembered by OrderedDict.

Ordered Dictionary

```python
# A Python program to demonstrate working of OrderedDict
from collections import OrderedDict

print("This is a Dict:\n")
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 3
d['d'] = 4

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

print("\nThis is an Ordered Dict:\n")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

for key, value in od.items():
    print(key, value)
```

Output:

```
This is a Dict:
a 1
c 3
b 2
d 4

This is an Ordered Dict:
a 1
b 2
c 3
d 4
```

Ordered Dictionary

- **Key value Change:**
  - If the value of a certain key is changed, the position of the key remains unchanged in OrderedDict.

```python
# A Python program to demonstrate working of key
# value change in OrderedDict
from collections import OrderedDict

print("Before:\n")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4
for key, value in od.items():
    print(key, value)

print("\nAfter:\n")
od['c'] = 5
for key, value in od.items():
    print(key, value)
```

Output:

```
Before:

a 1
b 2
c 3
d 4

After:

a 1
b 2
c 5
d 4
```

Ordered Dictionary

- **Deletion and Re-Inserting:**
  - Deleting and re-inserting the same key will push it to the back as OrderedDict, however, maintains the order of insertion.

```python
# A Python program to demonstrate working of deletion
# re-insertion in OrderedDict
from collections import OrderedDict

print("Before deleting:\n")
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
od['d'] = 4

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

print("\nAfter deleting:\n")
od.pop('c')
for key, value in od.items():
    print(key, value)

print("\nAfter re-inserting:\n")
od['c'] = 3
for key, value in od.items():
    print(key, value)
```

Output:

```
a 1
b 2
c 3
d 4

After deleting:

a 1
b 2
d 4

After re-inserting:

a 1
b 2
d 4
c 3
```

Sets and frozen sets

Sets

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data.
- The other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is *unordered*, *unchangeable**, and *unindexed*.
- Sets are written with curly brackets.

* **Note:** Set *items* are unchangeable, but you can remove items and add new items.

- Example: Creating a Set

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Output:

```
{'banana', 'cherry', 'apple'}
```

Sets - Accessing Items

- You cannot access items in a set by referring to an index or a key.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

```
thisset = {"apple", "banana", "cherry"}

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

Output:

```
banana
cherry
apple
```

Check if "banana" is present in the set:

```
thisset = {"apple", "banana", "cherry"}

print("banana" in thisset)
```

Output:

```
True
```

Sets - Adding Items

- To add one item to a set use the add() method.

```
thisset = {"apple", "banana", "cherry"}

thisset.add("orange")

print(thisset)
```

Output:

```
{'orange', 'apple', 'banana', 'cherry'}
```

Sets - Adding Sets

- To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

Output:

```
{'apple', 'mango', 'cherry', 'pineapple', 'banana', 'papaya'}
```

Sets - Removing a Set Item

- To remove an item in a set, you can use the remove() method.

```
thisset = {"apple", "banana", "cherry"}

thisset.remove("banana")

print(thisset)
```

Output:

```
{'apple', 'cherry'}
```

Note: If the item to remove does not exist, `remove()` will raise an error.

Sets - Removing a Set Item

- To remove an item in a set, you can use the discard() method.

```
thisset = {"apple", "banana", "cherry"}

thisset.discard("banana")

print(thisset)
```

Output:

```
{'cherry', 'apple'}
```

Note: If the item to remove does not exist, `discard()` will **NOT** raise an error.

Sets - Removing a Set Item

- You can also use the pop() method to remove an item, but this method will remove the last item.
- Remember that sets are unordered, so you will not know what item that gets removed.
- The return value of the pop() method is the removed item.

```
thisset = {"apple", "banana", "cherry"}

x = thisset.pop()

print(x)

print(thisset)
```

Output:

```
apple
{'banana', 'cherry'}
```

Note: Sets are *unordered*, so when using the `pop()` method, you do not know which item that gets removed.

Sets - Removing a Set Item

The `clear()` method empties the set:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

Output:

```
set()
```

The `del` keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

Output:

```
Traceback (most recent call last):
  File "demo_set_del.py", line 5, in <module>
    print(thisset) #this will raise an error because the set no longer exists
NameError: name 'thisset' is not defined
```

Frozen Set

- **Python frozenset() Method** creates an immutable Set object from an iterable.
- It is a built-in Python function.
- As it is a set object therefore we cannot have duplicate values in the frozenset.

Syntax : *frozenset(iterable_object_name)*

Parameter : *iterable_object_name*

- *This function accepts iterable object as input parameter.*

Return : *Returns an equivalent frozenset object.*

Frozen Set

- **Using frozenset() Method on tuple**
  - If no parameters are passed to frozenset() function, then it returns an empty frozenset type object in Python.

```
# passing an empty tuple
nu = ()

# converting tuple to frozenset
fnum = frozenset(nu)

# printing empty frozenset object
print("frozenset Object is : ", fnum)
```

Output:

```
frozenset Object is :  frozenset()
```

Frozen Set

- **Using frozenset() Method on list**
- Here as a parameter a list is passed and now it’s frozenset object is returned.

```
l = ["Geeks", "for", "Geeks"]

# converting list to frozenset
fnum = frozenset(l)

# printing empty frozenset object
print("frozenset Object is : ", fnum)
```

Output:

```
frozenset Object is :  frozenset({'Geeks', 'for'})
```

Frozen Set

- **Using frozenset() Method on Dictionary**
  - Since frozenset objects are immutable, they are mainly used as key in dictionary or elements of other sets.
  - The below example explains it clearly.

```
# creating a dictionary
Student = {"name": "Ankit", "age": 21, "sex": "Male",
           "college": "MNNIT Allahabad", "address": "Allahabad"}

# making keys of dictionary as frozenset
key = frozenset(Student)

# printing dict keys as frozenset
print('The frozen set is:', key)
```

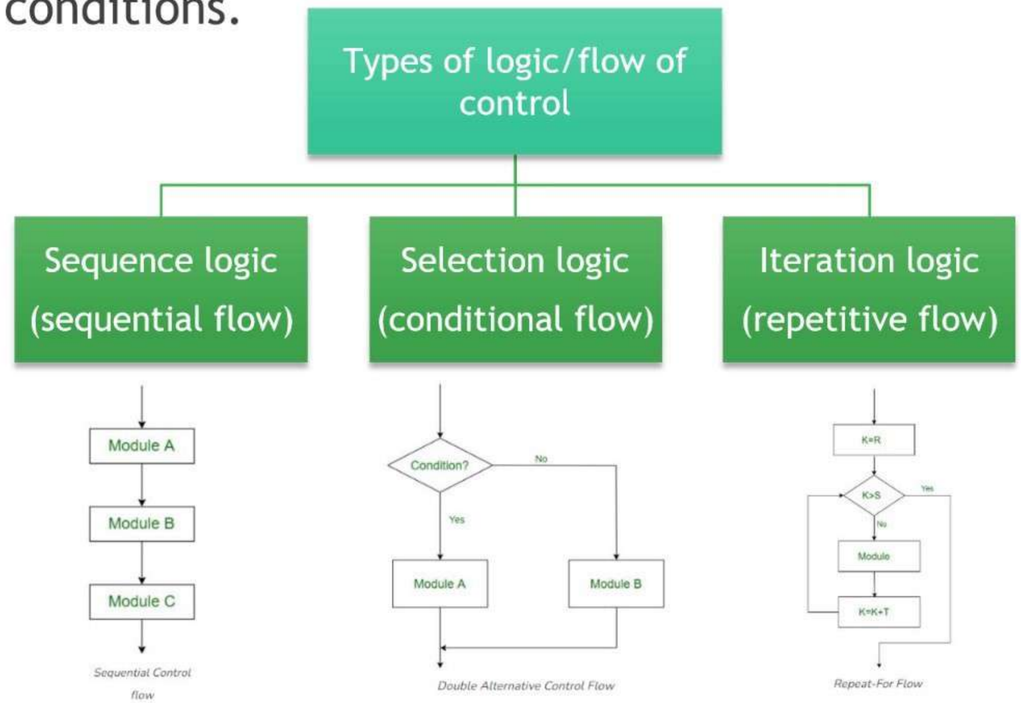
Output:

```
The frozen set is: frozenset({'address', 'name', 'age', 'sex', 'college'})
```

Control Structures

Control Structures

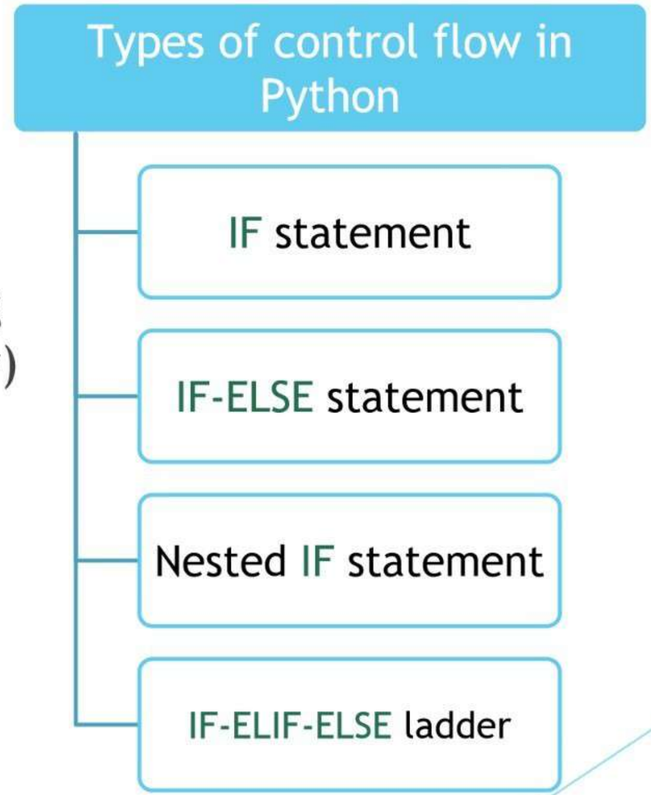
- Are just a way to specify flow of control in programs.
- They help making any algorithm or program more clear and understood.
- They analyze and choose in which direction a program flows based on certain parameters or conditions.



Using IF block and its variations

Control Flow using IF

- Is one of the basic building block when it come to implementing the **Selection logic** in your code
- Help in decision making through performing logical tests and selecting an alternative based on the test outcome
- Decision-making statements in programming languages decide the direction(Control Flow) of the flow of program execution



IF Statement

- The most simple decision-making statement
- Used to decide whether a certain statement or block of statements will be executed or not

Syntax:

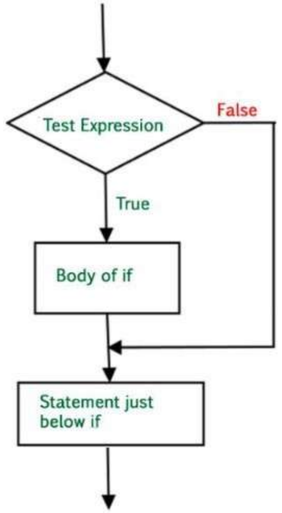
```
if condition:
    # Statements to execute if
    # condition is true
```

Syntax:

```
if condition:
    statement1
statement2

# Here if the condition is true, if block
# will consider only statement1 to be inside
# its block.
```

python uses indentation to identify a block

Flowchart of Python if statement



Example

```
# python program to illustrate If statement

i = 10

if (i > 15):
    print("10 is less than 15")
print("I am Not in if")
```

Output:

```
I am Not in if
```

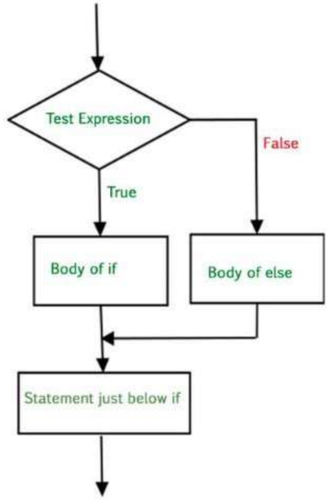
IF-ELSE Statement

- The if statement alone executes statement(s) if a condition is true
- To do something else if the condition is false, we can add the else statement to the if statement

Syntax:

```
if (condition):
    # Executes this block if
    # condition is true
else:
    # Executes this block if
    # condition is false
```

Flowchart of Python if-else statement



Example

```
# python program to illustrate If else statement
#!/usr/bin/python

i = 20
if (i < 15):
    print("i is smaller than 15")
    print("i'm in if Block")
else:
    print("i is greater than 15")
    print("i'm in else Block")
print("i'm not in if and not in else Block")
```

Output:

```
i is greater than 15
i'm in else Block
i'm not in if and not in else Block
```

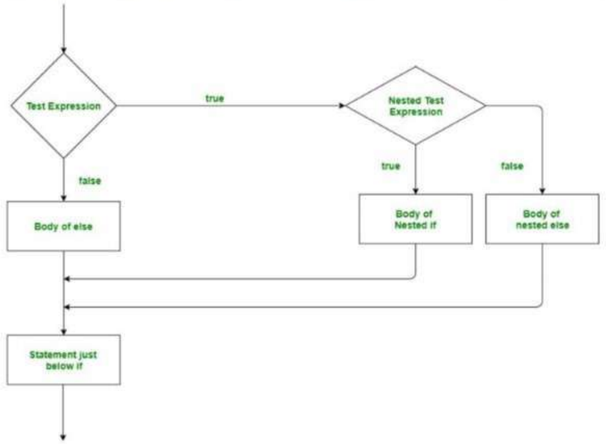
Nested IF Statement

- Is an if statement that is the target of another if statement
- It means an if statement is inside another if statement

Syntax:

```
if (condition1):
   # Executes when condition1 is true
   if (condition2):
      # Executes when condition2 is true
   # if Block is end here
# if Block is end here
```

Flowchart of Python Nested if Statement



Example

```
# python program to illustrate nested If statement

i = 10
if (i == 10):

    #  First if statement
    if (i < 15):
        print("i is smaller than 15")

    # Nested - if statement
    # Will only be executed if statement above
    # it is true
    if (i < 12):
        print("i is smaller than 12 too")
    else:
        print("i is greater than 15")
```

Output:

```
i is smaller than 15
i is smaller than 12 too
```

IF-ELIF-ELSE Statement

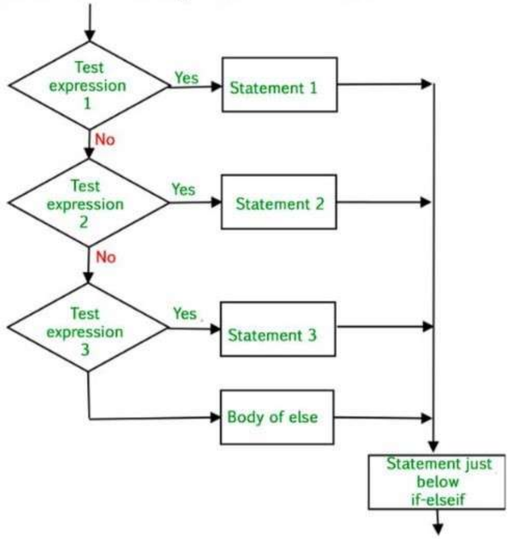
- user can decide among multiple options.
- The if statements are executed from the top down.
- If one of the conditions controlling the if is true Example
  - The statement associated with that if is executed
  - The rest of the ladder is bypassed
- If none of the conditions is true, then the final else statement will be executed.

IF-ELIF-ELSE Statement

Syntax:

```
if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement
```

Flowchart of Python if-elif-else ladder



Example

```python
# Python program to illustrate if-elif-else ladder
#!/usr/bin/python

i = 20
if (i == 10):
    print("i is 10")
elif (i == 15):
    print("i is 15")
elif (i == 20):
    print("i is 20")
else:
    print("i is not present")
```

Output:

```
i is 20
```

Short-hand forms for IF Statement

Short Hand if statement

Used when a single statement is to be executed inside the if.
The statement can be put on the same line as the if statement.

Syntax:

```
if condition: statement
```

Example

```
# Python program to illustrate short hand if
i = 10
if i < 15: print("i is less than 15")
```

Output:

```
i is less than 15
```

Short Hand if-else statement

Used to write the if-else statements in a single line where only one statement is needed in both the if and else blocks.

Syntax:

```
statement_when_True if condition else statement_when_False
```

Example

```
# Python program to illustrate short hand if-else
i = 10
print(True) if i < 15 else print(False)
```

Output:

```
True
```

Using MATCH to replace SWITCH CASE

SWITCH CASE Replacement

- Unlike every other programming language we have used before, Python does not have a switch or case statement
- There are many ways to go around this issue
- One way is by using the **match** statement
  - In Python 3.10 and after that, Python will support this by using **match** in place of **switch**

Example

```python
# This code runs only in python 3.10 or above versions
def number_to_string(argument):
    match argument:
        case 0:
            return "zero"
        case 1:
            return "one"
        case 2:
            return "two"
        case default:
            return "something"


head = number_to_string(2)
print(head)
```

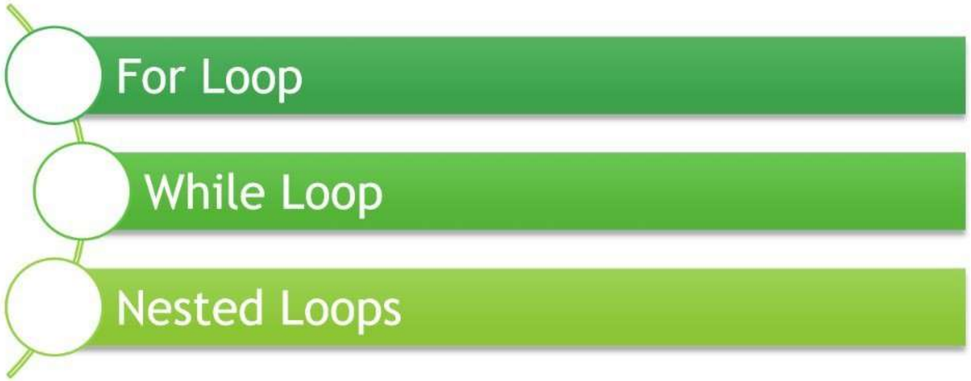
Output:

```
two
```

Constructing Loops [FOR – WHILE]

Looping Constructs

- Python programming language provides various types of loops to handle looping requirements.
- Python provides three ways for executing the loops



- While all the ways provide similar basic functionality, they differ in their syntax and condition-checking time.

For in Loop

- Used for sequential traversal.
- For example: traversing a list or string or array etc.
- In Python, there is “for in” loop which is similar to for each loop in other languages.

Syntax:

```
for iterator_var in sequence:
    statements(s)
```

For in Loop

Example

```
# Python program to illustrate
# Iterating over range 0 to n-1

n = 4
for i in range(0, n):
    print(i)
```

Output:

```
0
1
2
3
```

Example

```
# Python program to illustrate
# Iterating over a list
print("List Iteration")
l = ["geeks", "for", "geeks"]
for i in l:
    print(i)
```

Output:

```
List Iteration
geeks
for
geeks
```

Example

```
# Python program to illustrate
# Iterating by index

list = ["geeks", "for", "geeks"]
for index in range(len(list)):
    print(list[index])
```

Output:

```
geeks
for
geeks
```

WHILE loop

- Used to execute a block of statements repeatedly until a given condition is satisfied
- When the condition becomes false, the line immediately after the loop in the program is executed

Syntax:

```
while expression:
    statement(s)
```

All statements indented by the same number of character spaces after a programming construct are part of a single block of code.

Python uses indentation as its method of grouping statements.

WHILE loop

Example

```
# Python program to illustrate while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
```

Output:

```
Hello Geek
Hello Geek
Hello Geek
```

Nested Loops

- Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```
for iterator_var in sequence:
   for iterator_var in sequence:
       statements(s)
   statements(s)
```

Syntax:

```
while expression:
   while expression:
       statement(s)
   statement(s)
```

can put any type of loop inside of any other type of loop

Example

```
# Python program to illustrate
# nested for loops in Python
from __future__ import print_function
for i in range(1, 5):
    for j in range(i):
        print(i, end=' ')
    print()
```

Output:

```
1
2 2
3 3 3
4 4 4 4
```

Loop Control Statements

- Loop control statements change execution from their normal sequence
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed

Continue Statement

Returns the control to the beginning of the loop

Example

```
# Prints all letters except 'e' and 's'
for letter in 'geeksforgeeks':
    if letter == 'e' or letter == 's':
        continue
    print('Current Letter :', letter)
```

Output:

```
Current Letter : g
Current Letter : k
Current Letter : f
Current Letter : o
Current Letter : r
Current Letter : g
Current Letter : k
```

Loop Control Statements

- Loop control statements change execution from their normal sequence
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed

Break Statement

Brings control out of the loop

Example

```
for letter in 'geeksforgeeks':

    # break the loop as soon it sees 'e'
    # or 's'
    if letter == 'e' or letter == 's':
        break

print('Current Letter :', letter)
```

Output:

```
Current Letter : e
```

Loop Control Statements

- Loop control statements change execution from their normal sequence
- When execution leaves a scope, all automatic objects that were created in that scope are destroyed

Pass Statement

Used to write empty loops
Also used for empty control statements [e.g. functions and classes]

Example

```
# An empty loop
for letter in 'geeksforgeeks':
    pass
print('Last Letter :', letter)
```

Output:

```
Last Letter : s
```

Python Fundamentals

Organizing and Distributing Code

Topics

- Creating modules and packages
- Distributing code to repositories

Creating modules and packages

Modules

- A module is a simple Python file that contains collections of functions and global variables.
- A module has a .py extension file.
- A module is an executable file.
- Modules are organized and grouped in packages.
- Examples of modules:
  - Datetime
  - Regex
  - Random etc..

Modules - Example

1. Save the code in a file called demo_module.py

```
def myModule(name):
    print("This is My Module : "+ name)
```

2. Import module named demo_module and call the myModule function inside it.

```
import demo_module

demo_module.myModule("Math")
```

Output:

```
This is My Module : Math
```

Modules

Create a Module

To create a module just save the code you want in a file with the file extension `.py` :

Save the code in a file called demo_module.py

```
def myModule(name):
    print("This is My Module : "+ name)
```

Use a Module

Now we can use the module we just created, by using the `import` statement:

Import module named demo_module and call the myModule function inside it.

```
import demo_module

demo_module.myModule("Math")
```

Note: When using a function from a module, use the syntax: *module_name.function_name*.

Variables in Modules

- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

1 Save this code in the file mymodule.py

```
person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

2 Import the module named mymodule, and access the person1 dictionary:

```
import mymodule

a = mymodule.person1["age"]
print(a)
```

Output:

```
36
```

Naming Modules

- Naming a Module:
  - You can name the module file whatever you like, but it must have the file extension .py
- Renaming a Module:
  - You can create an alias when you import a module, by using the as keyword:

Create an alias for mymodule called mx :

```
import mymodule as mx

a = mx.person1["age"]
print(a)
```

Output:

```
36
```

Importing from Modules

- You can choose to import only parts from a module, by using the from keyword.

1 The module named `mymodule` has one function and one dictionary:

```
def greeting(name):
  print("Hello, " + name)

person1 = {
  "name": "John",
  "age": 36,
  "country": "Norway"
}
```

2 Import only the person1 dictionary from the module:

```
from mymodule import person1

print (person1["age"])
```

Output:

```
36
```

Note: When importing using the `from` keyword, do not use the module name when referring to elements in the module. Example: `person1["age"]`, **not** ~~`mymodule.person1["age"]`~~

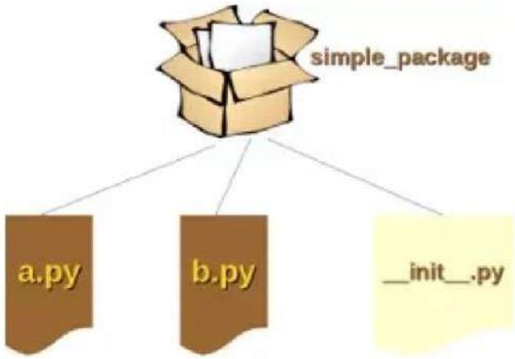
Packages

- A package is a simple directory having collections of modules.
- This directory contains Python modules and also having __init__.py file by which the interpreter interprets it as a Package.
- The package is simply a namespace.
- The package also contains sub-packages inside it.
- Examples of Packages:
  - Numpy
  - Pandas

```
Student(Package)
| __init__.py (Constructor)
| details.py (Module)
| marks.py (Module)
| collegeDetails.py (Module)
```

ecommerce
| - __init.py
| sale.py
| - customer
__init__.py
contact.py

Example – Creating a Package



- First, we need a directory:
  - The package name will be the directory name.
  - We will call our package "simple_package".
- Then, we create the __init__.py file:
  - This file can be empty, or it can contain valid Python code.
  - This code will be executed when a package is imported, so it can be used to initialize a package, e.g. to make sure that some other modules are imported or some values set.
- Now, we add Python files [modules] into this directory:
  - Create two simple files a.py and b.py just for the sake of filling the package with modules.

Example – Creating a Package

The content of a.py:

```
def bar():
    print("Hello, function 'bar' from module 'a' calling")
```

The content of b.py:

```
def foo():
    print("Hello, function 'foo' from module 'b' calling")
```

- Let's see what happens, when we import simple_package from the interactive Python shell, assuming that the directory simple_package is either in the directory from which you call the shell or that it is contained in the search path or environment variable "PYTHONPATH" (from your operating system):

```
import simple_package
```

```
simple_package/a
```

OUTPUT:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-347df8a711cc> in <module>
----> 1 simple_package/a
NameError: name 'a' is not defined
```

```
simple_package/b
```

OUTPUT:

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-4-e71d2904d2bd> in <module>
----> 1 simple_package/b
NameError: name 'b' is not defined
```

Example – Creating a Package

- We can see that the package simple_package has been loaded but neither the module "a" nor the module "b“.
- We can import the modules a and b in the following way:

```
from simple_package import a, b
a.bar()
b.foo()
```

OUTPUT:

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

Example – Creating a Package

- As we have seen at the beginning of the chapter, we can't access neither "a" nor "b" by solely importing simple_package.
- Yet, there is a way to automatically load these modules. We can use the file __init__.py for this purpose.
- All we have to do is add the following lines to the so far empty file __init__.py:

```
import simple_package.a
import simple_package.b
```

- Now, it will work:

```
import simple_package
simple_package.a.bar()
simple_package.b.foo()
```

OUTPUT:

```
Hello, function 'bar' from module 'a' calling
Hello, function 'foo' from module 'b' calling
```

Using the dir() function

There is a built-in function to list all the function names (or variable names) in a module. The `dir()` function:

List all the defined names belonging to the platform module:

```
import platform

x = dir(platform)
print(x)
```

Output:

```
['DEV_NULL', '_UNIXCONFDIR', 'WIN32_CLIENT_RELEASES', 'WIN32_SERVER_RELEASES', '__bu
```

And with packages as well to check if sub-packages or modules have been imported or not:

```
for mod in ['foobar', 'effects', 'filters', 'formats']:
    print(mod, mod in dir())
```

OUTPUT:

```
foobar True
effects True
filters True
formats True
```

Distributing code to repositories

Packaging your project

- To have your project installable from a Package Index like PyPI, you'll need to create a Distribution (aka "Package") for your project.
- Before you can build wheels and sdists for your project, you'll need to install the build package:

Windows

```
py -m pip install build
```

OR

```
python -m pip install build
```

- you should create a Source Distribution:

Windows

```
py -m build --sdist
```

Source Distribution

- A “source distribution” is unbuilt (i.e. it’s not a Built Distribution), and requires a build step when installed by pip.
- Even if the distribution is pure Python (i.e. contains no extensions), it still involves a build step to build out the installation metadata from setup.py and/or setup.cfg.

Wheels

- You should also create a wheel for your project.
- A wheel is a built package that can be installed without needing to go through the “build” process.
- Installing wheels is substantially faster for the end user than installing from a source distribution.

Windows

```
py -m build --wheel
```

setup.py

- The most important file is setup.py which exists at the root of your project directory:
- setup.py serves two primary functions:
  - It’s the file where various aspects of your project are configured.
    - The primary feature of setup.py is that it contains a global setup() function.
    - The keyword arguments to this function are how specific details of your project are defined.
  - It’s the command line interface for running various commands that relate to packaging tasks.
    - To get a listing of available commands, run python setup.py --help-commands.

Uploading your Project to PyPI

- When you run the command to create your distribution, a new directory dist/ gets created under your project's root directory.
- That's where you'll find your distribution file(s) to upload.
- Create an account
  - First, you need a PyPI user account.
  - You can create an account using the form on the PyPI website.
  - Go to https://pypi.org/manage/account/#api-tokens and create a new API token.
  - Don't limit its scope to a particular project, since you are creating a new project.

Don't close the page until you have copied and saved the token — you won't see that token again.

Upload your distributions

- Once you have an account you can upload your distributions to PyPI using twine.
- The process for uploading a release is the same regardless of whether or not the project already exists on PyPI:
  - If it doesn't exist yet, it will be automatically created when the first release is uploaded.
  - For the second and subsequent releases, PyPI only requires that the version number of the new release differ from any previous releases.

Windows

```
twine upload dist/*
```

Lab – Creating a distribution

Python Fundamentals

Object Oriented and Functional Programming

Topics

- Creating and using functions and classes
- Modifying functions and classes with decorators
- Introducing meta-classes

Creating and using functions and classes

Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

Creating a Function

In Python a function is defined using the `def` keyword:

```
def my_function():
  print("Hello from a function")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

```
def my_function():
  print("Hello from a function")

my_function()
```

Output:

```
Hello from a function
```

Arguments in Functions

- Information can be passed into functions as arguments.
- Arguments are specified after the function name, inside the parentheses.
- In case of many arguments, separate them with a comma.

```python
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

Output:

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Parameters or Arguments?

- The terms *parameter* and *argument* can be used for the same thing: information that are passed into a function.

  From a function's perspective:

  A parameter is the variable listed inside the parentheses in the function definition.

  An argument is the value that is sent to the function when it is called.

- Number of Arguments
  - By default, a function must be called with the correct number of arguments.

This function expects 2 arguments, but gets only 1:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil")
```

Output:

```
Traceback (most recent call last):
  File "demo_function_args_error.py", line 4, in <module>
    my_function("Emil")
TypeError: my_function() missing 1 required positional argument: 'lname'
```

This function expects 2 arguments, and gets 2 arguments:

```
def my_function(fname, lname):
  print(fname + " " + lname)

my_function("Emil", "Refsnes")
```

Arbitrary Arguments, *args

- If you do not know how many arguments that will be passed into your function, add a * before the parameter name in the function definition.
- This way the function will receive a tuple of arguments, and can access the items accordingly:

If the number of arguments is unknown, add a * before the parameter name:

```
def my_function(*kids):
  print("The youngest child is " + kids[2])

my_function("Emil", "Tobias", "Linus")
```

Output:

```
The youngest child is Linus
```

Arbitrary Arguments are often shortened to **args* in Python documentations.

Keyword Arguments

- You can also send arguments with the *key* = *value* syntax.
- This way the order of the arguments does not matter.

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

Output:

```
The youngest child is Linus
```

The phrase *Keyword Arguments* are often shortened to *kwargs* in Python documentations.

Arbitrary Keyword Arguments, **kwargs

- If you do not know how many keyword arguments that will be passed into your function, add two asterisks ** before the parameter name in the function definition.
- This way the function will receive a dictionary of arguments, and can access the items accordingly:

If the number of keyword arguments is unknown, add a double ** before the parameter name:

```
def my_function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Tobias", lname = "Refsnes")
```

Output:

```
His last name is Refsnes
```

Arbitrary Kword Arguments are often shortened to ***kwargs* in Python documentations.

Default Parameter Value

- The following example shows how to use a default parameter value.
- If we call the function without argument, it uses the default value:

```
def my_function(country = "Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

Output:

```
I am from Sweden
I am from India
I am from Norway
I am from Brazil
```

Passing a List as an Argument

- You can send any data types of argument to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.
- E.g. if you send a List as an argument, it will still be a List when it reaches the function:

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana", "cherry"]

my_function(fruits)
```

Output:

```
apple
banana
cherry
```

Return Values

- To let a function return a value, use the return statement:

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

Output:

```
15
25
45
```

The pass Statement

- function definitions cannot be empty, but if you for some reason have a function definition with no content, put in the pass statement to avoid getting an error.

```
def myfunction():
  pass
```

Object Orientation in Python

Classes

- Python is an object oriented programming language.
- Almost everything in Python is an object, with its properties and methods.
- A Class is like an object constructor, or a "blueprint" for creating objects.

Create a Class

To create a class, use the keyword `class` :

Create a class named MyClass, with a property named x:

```
class MyClass:
  x = 5
```

```
print(MyClass)
```

Output:

```
<class '__main__.MyClass'>
```

Create Object

Now we can use the class named MyClass to create objects:

Create an object named p1, and print the value of x:

```
p1 = MyClass()
print(p1.x)
```

Output:

```
5
```

Point draw()

x & y

```
class Point {
    int x=10;
    int y=20;
    public void draw( ) {
        s ———— (x),
        s ———— (y),
    }
}
```

self

```
class Point :
    def draw(self):
        self.x = 10
        self.y = 20
        print("Draw").
```

```
p = Point():
p.draw()
print (p.x, p.y)
```

The __init__() Function

→ Constructor

__init__( )

- To understand the meaning of classes we have to understand the built-in __init__() function.
- All classes have a function called __init__(), which is always executed when the class is being initiated.
- Use the __init__() function to assign values to object properties, or other operations that are necessary to do when the object is being created:

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Output:

```
John
36
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

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

    def details(self):
        print(f"Person <{self.name},{self.age}>")
```

actual variables

parameters

```
tom=Person("Tom",25)
tom.details()
```

```
Person <Tom,25>
```

abstract class

Car

start()
stop()

no-body.

abstract

Honda City

implementation

overriding
start()
stop()

abc → abstract base class

abc.ABC → Abstract Base Class

abc.@abstractmethod → which make function abstract

Modules & Packages

→ a file by name of ~~.py~~

greetings → English
→ French
→ German

The __str__() Function

- The __str__() function controls what should be returned when the class object is represented as a string.
- If the __str__() function is not set, the string representation of the object is returned as:

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

p1 = Person("John", 36)

print(p1)
```

Output:

```
<__main__.Person object at 0x15039e602100>
```

The __str__() Function

- If the __str__() function is set, the string representation of the object is returned as:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

Output:

```
John(36)
```

Object Methods

- Objects can also contain methods. Methods in objects are functions that belong to the object.
- Let us create a method in the Person class:

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

  def myfunc(self):
    print("Hello my name is " + self.name)

p1 = Person("John", 36)
p1.myfunc()
```

Output:

```
Hello my name is John
```

Note: The `self` parameter is a reference to the current instance of the class, and is used to access variables that belong to the class.

The self Parameter

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class:

Use the words *mysillyobject* and *abc* instead of *self*:

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Output:

```
Hello my name is John
```

Working with Object Properties

1 Modify Object Properties

You can modify properties on objects like this:

Set the age of p1 to 40:

```
p1.age = 40
```

2 Delete Object Properties

You can delete properties on objects by using the `del` keyword:

Delete the age property from the p1 object:

```
del p1.age
```

```
print(p1.age)
```

Output:

```
Traceback (most recent call last):
  File "demo_class7.py", line 13, in <module>
    print(p1.age)
AttributeError: 'Person' object has no attribute 'age'
```

3 Delete Objects

You can delete objects by using the `del` keyword:

Delete the p1 object:

```
del p1
```

```
print(p1)
```

Output:

```
Traceback (most recent call last):
  File "demo_class8.py", line 13, in <module>
    print(p1)
NameError: 'p1' is not defined
```

The pass Statement

- class definitions cannot be empty, but if you for some reason have a class definition with no content, put in the pass statement to avoid getting an error.

```
class Person:
  pass
```

Modifying functions and classes with decorators

Python Decorators

- We use a decorator when we need to change the behavior of a function without modifying the function itself.
- A few good examples are when we want to add:
  - Logging
  - Test performance
  - Perform caching
  - Verify permissions,and so on.

Creating a Decorator

- To understand how decorators work, you should understand a few concepts first.
- **A function is an object.**
  - Because of that, a function can be assigned to a variable.
    - The function can be accessed from that variable.

```
def my_function():

    print('I am a function.')

# Assign the function to a variable without parenthesis. We don't want to execute the function.

description = my_function
```

```
# Accessing the function from the variable I assigned it to.

print(description())

# Output

'I am a function.'
```

Creating a Decorator

- **A function can be nested within another function.**

```
def outer_function():

    def inner_function():

        print('I came from the inner function.')

    # Executing the inner function inside the outer function.
    inner_function()
```

```
outer_function()

# Output

I came from the inner function.
```

Note that the `inner_function` is not available outside the `outer_function`. If I try to execute the `inner_function` outside of the `outer_function` I receive a NameError exception.

```
inner_function()

Traceback (most recent call last):
  File "/tmp/my_script.py", line 9, in <module>
    inner_function()
NameError: name 'inner_function' is not defined
```

Creating a Decorator

- **Since a function can be nested inside another function it can also be returned.**

```
def outer_function():
    '''Assign task to student'''

    task = 'Read Python book chapter 3.'
    def inner_function():
        print(task)
    return inner_function

homework = outer_function()
```

```
homework()

# Output

'Read Python book chapter 5.'
```

Creating a Decorator

- **A function can be passed to another function as an argument.**

```
def friendly_reminder(func):
    '''Reminder for husband'''

    func()
    print('Don\'t forget to bring your wallet!')

def action():

    print('I am going to the store buy you something nice.')
```

```
# Calling the friendly_reminder function with the action function used as an argument.

friendly_reminder(action)

# Output

I am going to the store buy you something nice.
Don't forget to bring your wallet!
```

How to Create a Python Decorator

- Create an outer function that takes a function as an argument.
- Create an inner function that wraps around the decorated function.

```
def my_decorator_func(func):

    def wrapper_func():
        # Do something before the function.
        func()
        # Do something after the function.
    return wrapper_func
```

Using a Python Decorator

- To use a decorator:
  - We attach it to a function like you see in the code below.
  - We use a decorator by placing the name of the decorator directly above the function we want to use it on.
  - We prefix the decorator function with an @ symbol.

```
@my_decorator_func
def my_func():

    pass
```

Lab: Creating a function decorator

Introducing meta-classes

Meta-classes

- A metaclass in **Python** is a class of a class that defines how a class behaves.
- A class is itself an instance of a metaclass.
- A class in Python defines how the instance of the class will behave.

Meta-classes

- Things to know prior to working with metaclasses:
- **Everything in Python is an Object**

```
class TestClass():
    pass

my_test_class = TestClass()
print(my_test_class)
```

Output:

```
<__main__.TestClass object at 0x7f6fcc6bf908>
```

Meta-classes

- **Python Classes can be Created Dynamically**
  - `type` in Python enables us to find the type of an object.
  - We can proceed to check the type of object we created above.

```
type(TestClass)
```

Output:

```
type
```

```
type(type)
```

Output:

```
type
```

  - `type` enables us to create classes dynamically.

```
class DataCamp():
    pass
DataCampClass = type('DataCamp', (), {})
print(DataCampClass)
print(DataCamp())
```

Output:

```
<class '__main__.DataCamp'>
<__main__.DataCamp object at 0x7f6fcc66e358>
```

Meta-classes

- **Python Classes can be Created Dynamically**
  - `type` in Python enables us to find the type of an object.
  - We can proceed to check the type of object we created above.

```
type(TestClass)
```

Output:

```
type
```

```
type(type)
```

Output:

```
type
```

  - `type` enables us to create classes dynamically.

```
class DataCamp():
    pass
DataCampClass = type('DataCamp', (), {})
print(DataCampClass)
print(DataCamp())
```

Output:

```
<class '__main__.DataCamp'>
<__main__.DataCamp object at 0x7f6fcc66e358>
```

In the example DataCamp is the class name while DataCampClass is the variable that holds the class reference.

Meta-classes

- When using `type` we can pass attributes of the class using a dictionary as shown below:

```
PythonClass = type('PythonClass', (), {'start_date': 'August 2018', 'instructor': 'John Doe'} )
print(PythonClass.start_date, PythonClass.instructor)
print(PythonClass)
```

Output:

```
August 2018 John Doe
<class '__main__.PythonClass'>
```

- In case we wanted our PythonClass to inherit from the DataCamp class we pass it to our second argument when defining the class using `type`:

```
PythonClass = type('PythonClass', (DataCamp,), {'start_date': 'August 2018', 'instructor': 'John Doe'} )
print(PythonClass)
```

Output:

```
<class '__main__.PythonClass'>
```

Meta-classes

- Now we realize that Python creates the classes using a metaclass.
- We have seen that everything in Python is an object.
- These objects are created by metaclasses.
- Whenever we call `class` to create a class, there is a metaclass that does the magic of creating the class behind the scenes.
- We've already seen type do this.
  - It is similar to `str` that creates strings and `int` that creates integers.
- In Python, the `__class__` attribute enables us to check the type of the current instance.

```
article = 'metaclasses'
article.__class__
```

Output:

```
str
```

We can also check the type using `type(article)`.

```
type(article)
```

Output: `str`

When we check the type of `str`, we also find out that it's type.

```
type(str)
```

Output: `type`

Meta-classes

- When we check the type for `float`, `int`, `list`, `tuple`, and `dict`, we will have a similar output.
- This is because all of these objects are of type `type`.

```
print(type(list),type(float), type(dict), type(tuple))
```

Output:

```
<class 'type'> <class 'type'> <class 'type'> <class 'type'>
```

- We've already seen `type` creates classes.
- Hence when we check the `__class__` of `__class__` it should return `type`.

```
article.__class__.__class__
```

Output:

```
type
```

Creating Custom Metaclasses

- In Python, we can customize the class creation process by passing the `metaclass` keyword in the class definition.
- This can also be done by inheriting a class that has already passed in this keyword.

```
class MyMeta(type):
    pass

class MyClass(metaclass=MyMeta):
    pass

class MySubclass(MyClass):
    pass
```

```
print(type(MyMeta))
print(type(MyClass))
print(type(MySubclass))
```

Output:

```
<class 'type'>
<class '__main__.MyMeta'>
<class '__main__.MyMeta'>
```

- When defining a class and no metaclass is defined the default `type` metaclass will be used.
- If a metaclass is given and it is not an instance of `type()`, then it is used directly as the metaclass.

Python Fundamentals

Error Handling and Testing

Topics

- Handling and raising exceptions
- Writing and executing tests (doc tests and unit tests)
- Checking code coverage by tests

Exceptions → are unexpected erroneous event which breaks the flow of the code

Handling and raising exceptions

Exceptions

- An exception is an event, which occurs during the execution of a program that disrupts the normal flow of the program's instructions.
- In general, when a Python script encounters a situation that it cannot cope with, it raises an exception.
- An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either:
  - Handle the exception immediately or
  - It terminates and quits.

Handling an exception

- If you have some ***suspicious*** code that may raise an exception, you can defend your program by placing the suspicious code in a **`try:`** block.
- After the **`try:`** block, include an **`except:`** statement, followed by a block of code which handles the problem as elegantly as possible.
- Syntax:

```
try:
   You do your operations here;
   ......................
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

Handling an exception

try :

- Here are few important points about the above-mentioned syntax:
  - A single `try` statement can have multiple `except` statements.
    - This is useful when the `try` block contains statements that may throw different types of exceptions.
  - You can also provide a generic `except` clause, which handles any exception.
  - After the `except` clause(s), you can include an `else`-clause.
    - The code in the `else`-block executes if the code in the `try:` block does not raise an exception.
  - The `else`-block is a good place for code that does not need the `try:` block's protection.

except

except.

else:

Handling an exception - Example

- This example opens a file, writes content in the, file and comes out gracefully because there is no problem at all:

```
#!/usr/bin/python

try:
   fh = open("testfile", "w")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
   fh.close()
```

Output:

```
Written content in the file successfully
```

Handling an exception - Example

- This example tries to open a file where you do not have write permission, so it raises an exception:

```
#!/usr/bin/python

try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

Output:

```
Error: can't find file or read data
```

The *except* Clause with No Exceptions

- You can also use the except statement with no exceptions defined

```
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

- This kind of a **try-except** statement catches all the exceptions that occur.
  - **But** is not considered a good programming practice, because it catches all exceptions but does not help the programmer identify the root cause of the problem that may occur.

The *except* Clause with Multiple Exceptions

- You can also use the same *except* statement to handle multiple exceptions

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

→ housekeeping

- You can use a **finally:** block along with a **try:** block.
- The **finally** block is a place to put any code that must execute, whether the **try**-block raised an exception or not.

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

try:
open("file")
except:
file should
finally:

Resource Handling

try

file = open(———)

any resource opened here is closed

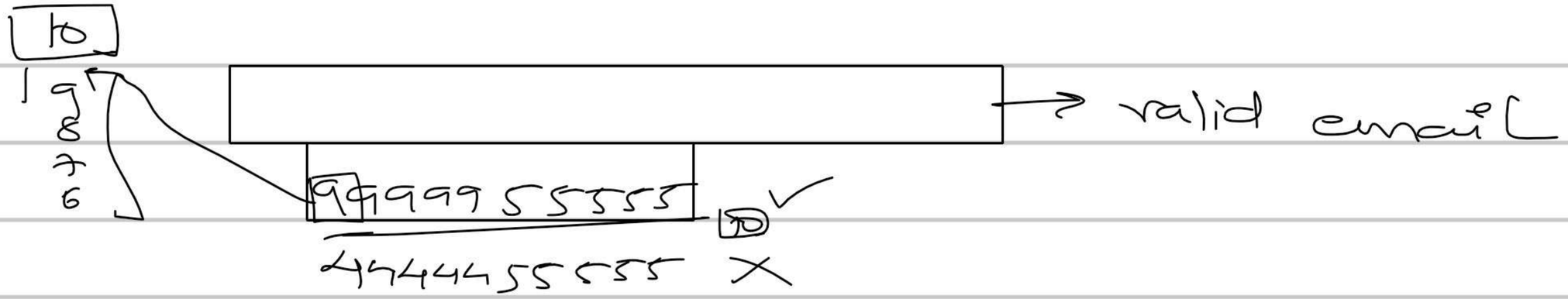
try:

closed

with open(" — ") as file, ——— :

Regex → Regular Expression

→ sequence of character that forms a search pattern



Symbol	What it represents?
*	Zero or more occurrences of the character that preceeds this asterisk
.	A wild card that represents any character
\s	Represents whitespace
[pqr]	A single character which can be either a ‘p’ , ’q’ or an ‘r’
[a-d]	A single character that falls in the range ‘a-d’
[^pq]	A single character that is neither ‘p’ nor ‘q’
^pattern	^ is an anchor tag that represents the beginning of line
pattern$	$i s an anchor tag that represents the end of line

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ fooaaaabar
- ☑ fooabar
- ☑ foobar
- ☑ fooaabar
- ☒ fooxxxbar
- ☒ fooxbar

- ☑ foo aaaa bar
- ☑ foo a bar
- ☑ foo bar
- ☑ foo aa bar

foo a* bar

a* - Zero or more occurrences o 'a' (The character just preceeding the asterisk)

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ fooaaaabar
- ☑ fooabar
- ☑ foobar
- ☑ fooaabar
- ☒ fooxxxbar
- ☒ fooxbar

- ☑ foo aaaa bar
- ☑ foo a bar
- ☑ foo bar
- ☑ foo aa bar

foo a* bar

Z – N

a* - Zero or more occurrences o 'a' (The character just preceeding the asterisk)

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

foo bar

- ☑ fooabar
- ☑ fooxbar
- ☒ baryfoo
- ☒ foobar
- ☒ fooxybar
- ☑ foocbar

☑ foo	a	bar
☑ foo	x	bar
☑ foo	c	bar

foo . bar

. - Single wildcard . Can represent only ONE character [any] in single position

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

foo*.bar

foo.*.bar

z-m

- ☑ foobar
- ☒ barfoo
- ☑ fooabcbar
- ☑ fooxcbar
- ☒ barcbyfoo
- ☑ foozbar
- ☒ barafoo
- ☒ barabfoo

☑ foo		0	bar
☑ foo	abc	3	bar
☑ foo	xc	2	bar
☑ foo	z	1	bar

foo.*bar

z-m

a

.→ any value

* → zero to more

.* - Zero or more occurrences of wildcard, which means zero or more occurrences of any character

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ fooxxxbar
- ☑ foo _bar
- ☒ fooxbar
- ☒ fooxxbar
- ☑ foo bar
- ☑ foo ___bar
- ☑ foobar
- ☒ fooyyybar

☑ foo	<3 spaces>	bar
☑ foo	<1 spaces>	bar
☑ foo	<6 spaces>	bar
☑ foo	<0 spaces>	bar

foo \s* bar

Zero or more

\s represents whitespace. \s* represents zero or more occurrence of whitespace

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

[9876]

10

[9999988888 → ✓

4444488888 — ✗

8888899999 — ✓

7721209139 85 — ✓

[abc] → either of a, b or c

- ☑ foo
- ☒ moo
- ☑ coo
- ☒ doo
- ☒ poo
- ☑ loo
- ☒ boo
- ☒ hoo

☑ f	oo
☑ c	oo
☑ l	oo

[fcl]oo

[abc] - Character class. One of the character inside the square brackets - a,b or c

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

[^abc] → should not be a,b or c

- ☑ foo
- ☒ moo
- ☑ coo
- ☑ doo
- ☑ poo
- ☑ loo
- ☑ boo
- ☒ hoo

☒ m	oo
☒ h	oo

[fcdplb]oo

[^mh]oo

[^abc] - Any character EXCEPT any of the one inside the square brackets, in a single

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ joo
- ☒ moo
- ☑ koo
- ☑ loo
- ☒ poo
- ☒ zoo
- ☑ moo
- ☒ hoo

☑ j	oo
☑ k	oo
☑ l	oo
☑ m	oo

[jklm] oo

[j-m] oo

[a-c] - One of the character falling in the range given in square brackets

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ joo
☒ moo
☑ koo
☑ loo
☒ poo
☑ moo
☑ zoo
☒ hoo

☑ j oo
☑ k oo
☑ l oo
☑ m oo
☑ z oo

[jklmz]oo

[j-mz]oo

[a-cx] - One of the character falling in the range given in square brackets OR any of the other choices given in square brackets - a,b, c,x

1. Understand the Requirement: What needs to be included or excluded
2. Identify the pattern in inclusion or exclusion list
3. Final Regular Expression

☑joo
☒moo
☑Koo
☑Loo
☒poo
☑moo
☑zoo
☒hoo

☑j oo
☑K oo
☑L oo
☑m oo
☑z oo

[j-m]-M]z]oo

[a-cACx] - One of the character falling in the range given in square brackets OR any of the other choices given in square brackets - a,b, c,A,B,C,x

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ x#y

☑ x:y

☑ x.y

☒ x&y

☒ x%y

☑ x	#	y
☑ x	:	y
☑ x	.	y

x[#:.]y

If a '.' is inside square brackets, it need not to be escaped

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

Escape → \

☑ x#y

☑ x:y

☑ x^y

☒ x&y

☒ x%y

☑	x	#	y
☑	x	:	y
☑	x	^	y

x[#:\^]y

If a '^ - ' is inside square brackets, it need to be escaped with backslash

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ x#y

☑ x\y

☑ x^y

☒ x&y

☒ x%y

☑ x	#	y
☑ x	\	y
☑ x	^	y

x[#\\^]y

x[^&%]y

If a '\' is inside square brackets, it need to be escaped with another backslash

1. Understand the Requirement: What needs to be included or excluded
2. Identify the pattern in inclusion or exclusion list
3. Final Regular Expression

☑ foo bar baz
☒ bar foo baz
☒ baz foo bar
☒ bar baz foo
☑ foo baz bar
☒ baz bar foo

☑ foo	bar baz
☑ foo	baz bar

^foo.*

^ is a placeholder that signifies beginning of line. Inside [] a ^ means negation, but ouside brackets it acts as an anchor

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ foo bar baz
- ☒ bar foo baz
- ☑ baz foo bar
- ☒ bar baz foo
- ☑ foo baz bar
- ☒ baz bar foo

☑ baz	foo bar
☑ foo	baz bar

.* bar$

$ is a placeholder which signifies end of line

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☑ foo
- ☒ bar foo
- ☒ baz foo
- ☒ bar baz foo
- ☒ foo baz bar
- ☒ baz bar foo

☑ foo

^foo $

^ signifies start of the line ; $ is a placeholder which signifies end of line

Symbol	What it represents?
+	One or more occurences of the character that preceeds + symbol
?	Zero or one occurence of the character that preceeds + symbol
pat1 | pat2	Match either the pattern pat1 or pattern pat2
()	Divides te pattern into groups
{m}	Exactly ‘m’ ocuurences of whatever preceedes
{m,n}	Atleast m and atmost n occurrences of whatever preceedes

Email Pattern

1. It should have one @ symbol in between
2. Before @ it can be any number of a-z,A-Z, _,.
3. After @ there should be minimum one number of a-z,A-Z,0-9,-
4. After @ and previous combination there should be minimum one period(.) sign
5. It should end with either com,org or net

Indian Mobile Number Pattern

1. It should have 10 numbers
2. Starting number should be either 7,8, or 9

Basic Syntax and Symbols in RegEx:

Symbol	Meaning	Example
`.`	Any character except a newline	`a.b` matches `acb`
`*`	0 or more repetitions of the preceding char	`ab*` matches `a`, `ab`, `abb`
`+`	1 or more repetitions of the preceding char	`ab+` matches `ab`, `abb`
`?`	0 or 1 occurrence of the preceding char	`ab?` matches `a`, `ab`
`\d`	Matches any digit (0-9)	`\d+` matches `123`
`\w`	Matches any alphanumeric character (or _)	`\w+` matches `abc123`
`\s`	Matches any whitespace character	`\s` matches spaces, tabs
`^`	Matches the beginning of the string	`^a` matches `apple`
`$`	Matches the end of the string	`e$` matches `apple`
`[]`	Matches any one character inside the brackets	`[aeiou]` matches `a` in `cat`
`` ` ``	`` ` ``	Matches either the pattern on the left or right
`{}`	Specifies number of repetitions	`a{3}` matches `aaa`

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

\d

- ☑ 834
- ☑ 519
- ☒ 4874
- ☒ 5
- ☒ 89
- ☒ 45687
- ☒ 25
- ☑ 645

☑ ^	[0-9][0-9][0-9]	$
☑ ^	[0-9][0-9][0-9]	$
☑ ^	[0-9][0-9][0-9]	$

^\d{3} $

a{m} represents exactly 'm' repetations of whateer immediately precedess this

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

☑ lion	☑ ^	[a-z]{4}	$
☑ tiger			
☒ leopard – 7	☑ ^	[a-z]{5}	$
☒ fox – 3			
☒ kangaroo – 8	☑ ^	[a-z]{5}	$
☒ cat – 3			
☑ mouse 5	☑ ^	[a-z]{6}	$
☑ cuckoo 5			
☑ deer 5	☑ ^	[a-z]{4}	$

^[a-z]{4,6}$

a{m,n} represents atleast 'm' and atmost 'n' repetations of whateer immediately precedess this

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

- ☒ ha
- ☑ hahahahaha
- ☒ hahaha
- ☑ hahahaha
- ☒ haha
- ☑ hahahahahaha
- ☑ hahahahahahahaha
- ☑ hahahahahahahahaha

- ☑ ha{5}
- ☑ ha{4}
- ☑ ha{6}
- ☑ ha{8}
- ☑ ha{9}

(ha){4,9}

paranthesis is used for grouping and treating as a single entity
{m,} represents atleast m repeatations of whatever preceeds

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

+ → one or more

- ☑ fooaaaabar
- ☑ fooabar
- ☒ foobar
- ☒ fooxxxbar
- ☒ fooxbar

☑ foo	aaaa	bar
☑ foo	a	bar
☑ foo	aa	bar

foo a+ bar

a+ One or more occurences of 'a'

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

? → zero or one occurence

☑https://website
☑http://website
☒httpss://website
☒httpx://website
☒httpxx://website

☑http s ://website

☑http ://website

http s? ://website

a? Zero or One occurence of 'a'

1 Understand the Requirement: What needs to be included or excluded

2 Identify the pattern in inclusion or exclusion list

3 Final Regular Expression

[a-z]{3} wood

- ☒ redwood
- ☑ logwood
- ☒ sapwood
- ☒ rosewood
- ☑ plywood
- ☒ teakwood

☑ log	wood
☑ ply	wood

(ply| log)wood

(a|b) represents either a or b, where a and b can be multi character strings

`\A` - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
`\Athe`	`the sun`	Match
	`In the sun`	No match

`\Z` - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
`Python\Z`	`I like Python`	1 match
	`I like Python Programming`	No match
	`Python is fun.`	No match

`\b` - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
`\bfoo`	`football`	Match
	`a football`	Match
	`afootball`	No match
`foo\b`	`the foo`	Match
	`the afoo test`	Match

`\B` - Opposite of `\b`. Matches if the specified characters are **not** at the beginning or end of a word.

Expression	String	Matched?
`\Bfoo`	`football`	No match
	`a football`	No match
	`afootball`	Match
`foo\B`	`the foo`	No match
	`the afoo test`	No match
	`the afootest`	Match

`\d` - Matches any decimal digit. Equivalent to `[0-9]`

Expression	String	Matched?
`\d`	`12abc3`	3 matches (at `12abc3`)
	`Python`	No match

`\D` - Matches any non-decimal digit. Equivalent to `[^0-9]`

Expression	String	Matched?
`\D`	`1ab34"50`	3 matches (at `1ab34"50`)
	`1345`	No match

`\s` - Matches where a string contains any whitespace character. Equivalent to `[ \t\n\r\f\v]`.

Expression	String	Matched?
`\s`	`Python RegEx`	1 match
	`PythonRegEx`	No match

`\S` - Matches where a string contains any non-whitespace character. Equivalent to `[^ \t\n\r\f\v]`.

Expression	String	Matched?
`\S`	`a b`	2 matches (at `a b`)
		No match

`\w` - Matches any alphanumeric character (digits and alphabets). Equivalent to `[a-zA-Z0-9_]`. By the way, underscore `_` is also considered an alphanumeric character.

Expression	String	Matched?
`\w`	`12&": ;c`	3 matches (at `12&": ;c`)
	`%"> !`	No match

`\W` - Matches any non-alphanumeric character. Equivalent to `[^a-zA-Z0-9_]`

Expression	String	Matched?
`\W`	`1a2%c`	1 match (at `1a2%c`)
	`Python`	No match

Email Pattern

1. It should have one @ symbol in between
2. Before @ it can be any number of a-z,A-Z, _,.
3. After @ there should be minimum one number of a-z,A-Z,0-9,-
4. After @ and previous combination there should be minimum one period(.) sign
5. It should end with either com,org or net

```
word_list = ['zartab@codewithz.com',
             'zartab.n@uni.edu',
             'zartab-n-1312@my-company.net',
             'zartab#n#1312@my-company.net']
```

Indian Mobile Number Pattern

South Africa +27

10

1. It should have 10 numbers
2. Starting number should be either 7,8, or 9

+91

(7|8|9)

^[7-9]{1}\d{9}$

^0\d{9}$

^[a-zA-Z][a-zA-Z0-9_.-]*@[a-zA-Z][a-zA-Z0-9.-]*\.(com|org|net)

Argument of an Exception

- An exception can have an *argument*, which is a value that gives additional information about the problem.
- The contents of the argument vary by exception.
- You capture an exception's argument by supplying a variable in the except clause.

```
try:
   You do your operations here;
   ......................
except ExceptionType as Argument:
   You can print value of Argument here...
```

Raising an Exceptions

- You can raise exceptions in several ways by using the `raise` statement.
- The general syntax for the `raise` statement is as follows:

```
raise [Exception [, args [, traceback]]]
```

- Example:

Raise an error and stop the program if x is lower than 0:

```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

Output:

```
Traceback (most recent call last):
  File "demo_ref_keyword_raise.py", line 4, in <module>
    raise Exception("Sorry, no numbers below zero")
Exception: Sorry, no numbers below zero
```

User-Defined Exceptions

- Python also allows you to create your own exceptions by deriving classes from the standard built-in exceptions.

```python
# A python program to create user-defined exception
# class MyError is derived from super class Exception
class MyError(Exception):

    # Constructor or Initializer
    def __init__(self, value):
        self.value = value

    # __str__ is to print() the value
    def __str__(self):
        return(repr(self.value))


try:
    raise(MyError(3*2))

# Value of Exception is stored in error
except MyError as error:
    print('A New Exception occurred: ', error.value)
```

Output:

```
('A New Exception occurred: ', 6)
```

Writing and executing tests (doc tests and unit tests)

Is testing important?

Google Weather

weather_api() Cape Town

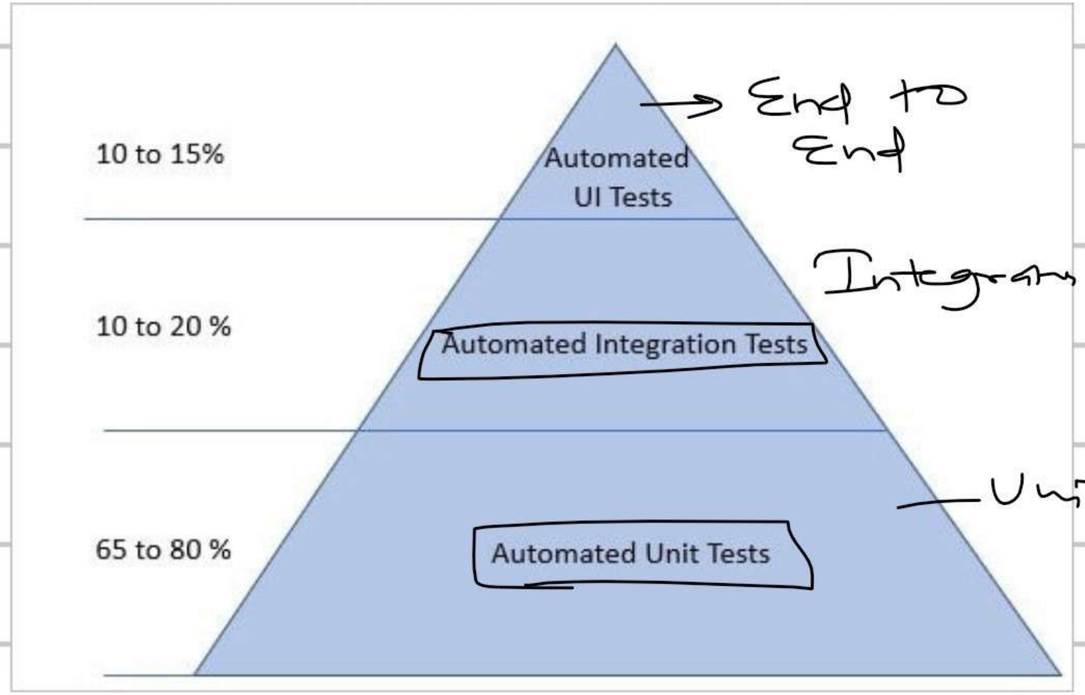
✓

Out of range

google-weather is

happy case

sad case



End to End

Integration

Unit

Unit Test → involves testing individual components (functions) to ensure that each part is working as expected

Ford, BMW, Toyota ?
[are they manufacturing or assembling]

Parts → car
↳ Test → Assembler ✗
→ Manufacturer ✓

Application
⇓ Car
Parts → function
Unit tests

Given / When / Then | Arrange / Act / Assert.

Given/Arrange → Set up the conditions & input for tests

When / Act → Call the method/function which we want to test

Then/Assert → Verify that the outcome matches the expected result

Unit Test

- Unit testing is a method for testing software that looks at the smallest testable pieces of code, called units, which are tested for correct operation.
- By doing unit testing, we can verify that each part of the code, including helper functions that may not be exposed to the user, works correctly and as intended.
- The idea is that we are independently checking each small piece of our program to ensure that it works.

Using Python Built-in PyUnit Framework

- You might be wondering, why do we need unit testing frameworks since Python and other languages offer the assert keyword?
- Unit testing frameworks help automate the testing process and allow us to run multiple tests on the same function with different parameters, check for expected exceptions, and many others.
- PyUnit is Python's built-in unit testing framework and Python's version of the corresponding JUnit testing framework for Java.
- To get started building a test file, we need to import the unittest library to use PyUnit:

```
import unittest
```

Using Python Built-in PyUnit Framework

- Then, we can get started writing out first unit test.
- Unit tests in PyUnit are structured as subclasses of the unittest.TestCase class, and we can override the runTest() method to perform our own unit tests which check conditions using different assert functions in unittest.TestCase:

```
class TestGetAreaRectangle(unittest.TestCase):
    def runTest(self):
        rectangle = Rectangle(2, 3)
        self.assertEqual(rectangle.get_area(), 6, "incorrect area")
```

Using Python Built-in PyUnit Framework

- Then, we can get started writing out first unit test.
- Unit tests in PyUnit are structured as subclasses of the unittest.TestCase class, and we can override the runTest() method to perform our own unit tests which check conditions using different assert functions in unittest.TestCase:

```
class TestGetAreaRectangle(unittest.TestCase):
    def runTest(self):
        rectangle = Rectangle(2, 3)
        self.assertEqual(rectangle.get_area(), 6, "incorrect area")
```

- Then, to run the unit test, we call unittest.main() in our program

```
...
unittest.main()
```

Output:

```
.
----------------------------------------
Ran 1 test in 0.003s

OK
```

Unit Test Output

- There are various options through which we can control what to see in the output of each test we run.
- Those options are part of the Verbosity argument that we pass to the “main” function, and they are:

Verbosity = 0 (quiet) – get the total numbers of tests executed and the global result

```
unittest.main(argv=['''], verbosity=0,exit=False)
```

```
======================================================================
FAIL: test_slow_is_equal_to_quick_at_2 (__main__.FibonaticsTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-18-8fb699d9c807>", line 66, in test_slow_is_equal_to_quick_at_2
    self.assertEqual(quick_steps+1, slow_steps)
AssertionError: 4 != 3

----------------------------------------------------------------------
Ran 9 tests in 0.014s

FAILED (failures=1)
```

```
<unittest.main.TestProgram at 0x20a5f3e7788>
```

Unit Test Output

Verbosity = 1 (default) – quiet + a dot for every successful test or a F for every failure

```
unittest.main(argv=['''], verbosity=1,exit=False)
```

```
....F....
======================================================================
FAIL: test_slow_is_equal_to_quick_at_2 (__main__.FibonaticsTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-18-8fb699d9c807>", line 66, in test_slow_is_equal_to_quick_at_2
    self.assertEqual(quick_steps+1, slow_steps)
AssertionError: 4 != 3

----------------------------------------------------------------------
Ran 9 tests in 0.026s

FAILED (failures=1)
```

```
<unittest.main.TestProgram at 0x20a5f3c2a88>
```

Unit Test Output

Verbosity = 2 (verbose) – help string of every test and the result

```
unittest.main(argv=['''], verbosity=2,exit=False)
```

```
test_for_equality_with_10 (__main__.FibonaticsTests) ... ok
test_for_equality_with_11 (__main__.FibonaticsTests) ... ok
test_negative_value_1 (__main__.FibonaticsTests) ... ok
test_negative_value_20 (__main__.FibonaticsTests) ... ok
test_slow_is_equal_to_quick_at_2 (__main__.FibonaticsTests) ... FAIL
test_slow_needs_more_steps_with_11 (__main__.FibonaticsTests) ... ok
test_value_10 (__main__.FibonaticsTests) ... ok
test_value_11 (__main__.FibonaticsTests) ... ok
test_value_slow_20 (__main__.FibonaticsTests) ... ok

======================================================================
FAIL: test_slow_is_equal_to_quick_at_2 (__main__.FibonaticsTests)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-18-8fb699d9c807>", line 66, in test_slow_is_equal_to_quick_at_2
    self.assertEqual(quick_steps+1, slow_steps)
AssertionError: 4 != 3

----------------------------------------------------------------------
Ran 9 tests in 0.028s

FAILED (failures=1)
```

```
<unittest.main.TestProgram at 0x20a5f3f2ec8>
```

Doctest

- The doctest module is a lightweight testing framework that provides quick and straightforward test automation.
- It can read the test cases from your project's documentation and your code's docstrings.
- This framework is shipped with the Python interpreter and adheres to the batteries-included philosophy.

Doctest

- You can use doctest from either your code or your command line.
- To find and run your test cases, doctest follows a few steps:
  - Searches for text that looks like Python interactive sessions in your documentation and docstrings
  - Parses those pieces of text to distinguish between executable code and expected results
  - Runs the executable code like regular Python code
  - Compares the execution result with the expected result

Doctest

- The doctest framework searches for test cases in your documentation and the docstrings of packages, modules, functions, classes, and methods.
- It doesn’t search for test cases in any objects that you import.
- In general, doctest interprets as executable Python code all those lines of text that start with the primary (>>>) or secondary (...) REPL prompts.
- The lines immediately after either prompt are understood as the code’s expected output or result.

Doctest Benefits

- Writing quick and effective test cases to check your code as you write it
- Running acceptance, regression, and integration test cases on your projects, packages, and modules
- Checking if your docstrings are up-to-date and in sync with the target code
- Verifying if your projects' documentation is up-to-date
- Writing hands-on tutorials for your projects, packages, and modules
- Illustrating how to use your projects' APIs and what the expected input and output must be

Creating doctest Tests for Checking Returned and Printed Values

- The first and probably most common use case of code testing is checking the return value of functions, methods, and other callables.
- You can do this with doctest tests.

Original Code

```
# calculations.py

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

Adding docstrings to the code

```
# calculations.py

def add(a, b):
    """Compute and return the sum of two numbers.

    Usage examples:
    >>> add(4.0, 2.0)
    6.0
    >>> add(4, 2)
    6.0
    """
    return float(a + b)
```

Running doctest Tests

- There are various ways to run the tests, but we will focus on running the test through the Jupyter Notebook.
- To do that we need to import the doctest library.

Original Code

```
# calculations.py

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

Adding docstrings to the code

```
# calculations.py

def add(a, b):
    """Compute and return the sum of two numbers.

    Usage examples:
    >>> add(4.0, 2.0)
    6.0
    >>> add(4, 2)
    6.0
    """
    return float(a + b)
```

Lab: Conducting Tests

Checking code coverage by tests

What is Code Coverage?

- This measures the number of lines of source code executed during a given test suite for a program.
- Tools that measure code coverage normally express this metric as a percentage.
- Many use the terms “code coverage” and “test coverage” interchangeably.
- However, they are two different things.
- Test coverage is a measurement of the degree to which a test or testing suite actually checks the full extent of a program’s functionality.

Coverage.py

- Is one of the most popular code coverage tools for Python.
- It uses code analysis tools and tracing hooks provided in Python standard library to measure coverage.
- It runs on major versions of CPython, PyPy, Jython and IronPython.
- You can use Coverage.py with both unittest and Pytest.
- You can either download the latest version of Coverage and install it manually, or you can install it with pip as:

```
pip install coverage
```

- Now run your program with coverage as:

```
coverage run my_program.py arg1 arg2
```

Coverage.py

- To get coverage data output use the command:

```
coverage report -m
```

Output:

```
Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
my_program.py          20      4    80%   33-35, 39
my_other_module.py     56      6    89%   17-23
-------------------------------------------------
TOTAL                  76     10    87%
```

- To generate report in html file, use the command:

```
coverage html
```

Coverage report: 37.59%

Module ↓	statements	missing	excluded	branches	partial	coverage
cogapp/__init__.py	2	0	0	0	0	100.00%
cogapp/__main__.py	3	3	0	0	0	0.00%
cogapp/backward.py	19	8	0	2	1	57.14%
cogapp/cogapp.py	427	197	4	176	26	47.10%
cogapp/makefiles.py	28	20	3	14	0	19.05%
cogapp/test_cogapp.py	704	486	6	6	0	30.99%
cogapp/test_makefiles.py	55	55	0	6	0	0.00%
cogapp/test_whiteutils.py	69	69	0	0	0	0.00%
cogapp/whiteutils.py	45	3	0	32	3	92.21%
Total	**1352**	**841**	**13**	**236**	**30**	**37.59%**

Python Fundamentals

Working with Files and Directories

Topics

- Accessing different types of files and file handling principles
- Creating, reading, updating and deleting files (including regular text files, csv, as well as Microsoft Word and Microsoft Excel files)
- Extracting data from text files using Regular Expressions
- Creating and deleting directories, listing and searching for files

Accessing different types of files and file handling principles

File Handling

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the `open()` function.
- The `open()` function takes two parameters:
  - `Filename`
  - `mode`

File Handling

- There are four different methods (modes) for opening a file:

```
"r" - Read - Default value. Opens a file for reading, error if the file does not exist

"a" - Append - Opens a file for appending, creates the file if it does not exist

"w" - Write - Opens a file for writing, creates the file if it does not exist

"x" - Create - Creates the specified file, returns an error if the file exists
```

- In addition you can specify if the file should be handled as binary or text mode:

```
"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)
```

File Handling

- Syntax
- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

- You can also specify the mode:

```
f = open("demofile.txt", "rt")
```

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Note: Make sure the file exists, or else you will get an error.

Creating, reading, updating and deleting files

Reading a File

- To open the file, use the built-in `open()` function.
- The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

```
f = open("demofile.txt", "r")
print(f.read())
```

Output:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

- If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Output:

```
Welcome to this text file!
This file is located in a folder named "myfiles", on the D drive.
Good Luck!
```

Read Only Parts of the File

- By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Output: `Hello`

Read Lines

- You can return one line by using the `readline()` method:

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

Output:

```
Hello! Welcome to demofile.txt
```

- By calling `readline()` two times, you can read the two first lines:

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Output:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
```

Read Lines

- By looping through the lines of the file, you can read the whole file, line by line:

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Output:

```
Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!
```

Close Files

- It is a good practice to always close the file when you are done with it.

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Note: You should always close your files, in some cases, due to buffering, changes made to a file may not show until you close the file.

Write to an Existing File

- To write to an existing file, you must add a parameter to the `open()` function:
  - "a" - Append:
    - will append to the end of the file

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Output:

```
Hello! Welcome to demofile2.txt
This file is for testing purposes.
Good Luck!Now the file has more content!
```

Write to an Existing File

- To write to an existing file, you must add a parameter to the `open()` function:
  - "w" - Write:
    - will overwrite any existing content

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile3.txt", "r")
print(f.read())
```

Output:

```
Woops! I have deleted the content!
```

Note: the "w" method will overwrite the entire file.

Create a New File

- To create a new file in Python, use the `open()` method, with one of the following parameters:
- "x" - Create:
  - will create a file, returns an error if the file exist
- "a" - Append:
  - will create a file if the specified file does not exist
- "w" - Write:
  - will create a file if the specified file does not exist

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Create a new file if it does not exist:

```
f = open("myfile.txt", "w")
```

Delete a File

- To delete a file, you must import the OS module, and run its `os.remove()` function:

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

- To avoid getting an error, you might want to check if the file exists before you try to delete it:

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Extracting data from text files using Regular Expressions

Regular Expressions in Python

- Regular expressions can look scary, but are pretty simple once you understand the rules.
- The syntax for regular expressions appeared and was standardised in the Perl language, and now nearly all programming languages support “Perl Compatible Regular Expressions” (PCRE).
- Python provides the re and regexp modules, that support most of PCRE.

Regular Expressions in Python

- There are a lot of special characters. They are:
  - `\d` Match any digit (number)
  - `\s` Match a space
  - `\w` Match any word character (alphanumeric and "_")
  - `\S` Match any non-whitespace character
  - `\D` Match any non-digit character
  - `.` Match any character
  - `\t` Match a tab
  - `\n` Match a newline

Regular Expressions in Python

- You can control which characters are matched in the square brackets using;
  - `[abc]` Match a, b or c
  - `[a-z]` Match any character between a to z
  - `[A-Z]` Match any character between A to Z
  - `[a-zA-Z]` Match any character from a to z and A to Z (any letter)
  - `[0-9]` Match any digit
  - `[02468]` Match any even digit
  - `[^0-9]` Matches NOT digits (^ means NOT)

Regular Expressions in Python

- You can also use repetition in your matching.
  - `*` Match 0 or more times, e.g. `\w*` means match 0 or more word characters
  - `+` Match 1 or more times, e.g. `\w+` means match 1 or more word characters
  - `?` Match 0 or 1 times, e.g. `\w?` means match 0 or 1 word characters
  - `{n}` Match exactly n times, e.g. `\w{3}` means match exactly 3 word characters
  - `{n,}` Match at least n times, e.g. `\w{5,}` means match at least 5 word characters
  - `{m,n}` Match between m and n times, e.g. `\w{5,7}` means match 5-7 word characters

Python Regex – Get List of all Numbers from String

- To get the list of all numbers in a String, use the regular expression **'[0-9]+'** with `re.findall()` method.
- **[0-9]** represents a regular expression to match a single digit in the string.
- **[0-9]+** represents continuous digit sequences of any length.

```
import re

str = 'We live at 9-162 Malibeu. My phone number is 666688888.'
#search using regex
x = re.findall('[0-9]+', str)
print(x)
```

Output

```
['9', '162', '666688888']
```

Creating and deleting directories, listing and searching for files

OS module

- The `OS` module in Python provides functions for interacting with the operating system.
- `OS` comes under Python's standard utility modules.
- This module provides a portable way of using operating system dependent functionality.
- The `os` and `os.path` modules include many functions to interact with the file system.
- All functions in `os` module raise `OSError` in the case of invalid or inaccessible file names and paths, or other arguments that have the correct type but are not accepted by the operating system.

Create a Directory Using os.mkdir()

- `os.mkdir()` method in Python is used to create a directory named path with the specified numeric mode.
- This method raise `FileExistsError` if the directory to be created already exists.

Syntax: *os.mkdir(path, mode = 0o777, *, dir_fd = None)*

Parameter:

path: *A path-like object representing a file system path. A path-like object is either a string or bytes object representing a path.*

mode (optional): *A Integer value representing mode of the directory to be created. If this parameter is omitted then default value 0o777 is used.*

dir_fd (optional): *A file descriptor referring to a directory. The default value of this parameter is None.*

If the specified path is absolute then dir_fd is ignored.

Note: *The '*' in parameter list indicates that all following parameters (Here in our case 'dir_fd') are keyword-only parameters and they can be provided using their name, not as positional parameter.*

Return Type: *This method does not return any value.*

Create a Directory Using os.mkdir()

- Example:

```python
# importing os module
import os

# Directory
directory = "GeeksforGeeks"

# Parent Directory path
parent_dir = "D:/Pycharm projects/"

# Path
path = os.path.join(parent_dir, directory)

# Create the directory
# 'GeeksForGeeks' in
# '/home / User / Documents'
os.mkdir(path)
print("Directory '% s' created" % directory)
```

Output:

```
Directory 'GeeksforGeeks' created
```

```python
# importing os module
import os

# Directory
directory = "Geeks"

# Parent Directory path
parent_dir = "D:/Pycharm projects"

# mode
mode = 0o666

# Path
path = os.path.join(parent_dir, directory)

# Create the directory
# 'GeeksForGeeks' in
# '/home / User / Documents'
# with mode 0o666
os.mkdir(path, mode)
print("Directory '% s' created" % directory)
```

Output:

```
Directory 'Geeks' created
```

Create a Directory Using os.mkdir()

- Example:

Handling error while using `os.mkdir()` method.

```
# Python program to explain os.mkdir() method

# importing os module
import os

# path
path = 'D:/Pycharm projects / GeeksForGeeks'

# Create the directory
# 'GeeksForGeeks' in
# '/home / User / Documents'
try:
    os.mkdir(path)
except OSError as error:
    print(error)
```

Output:

```
[WinError 183] Cannot create a file when that file/
              /already exists: 'D:/Pycharm projects/GeeksForGeeks'
```

Create a Directory Using os.mkdirs()

- `os.makedirs()` method in Python is used to create a directory recursively.
- That means while making leaf directory if any intermediate-level directory is missing, `os.makedirs()` method will create them all.

```
D:/Pycharm projects/GeeksForGeeks/Authors/Nikhil
```

  - Suppose we want to create directory ‘Nikhil’ but Directory ‘GeeksForGeeks’ and ‘Authors’ are unavailable in the path.
  - Then `os.makedirs()` method will create all unavailable/missing directories in the specified path.
  - ‘GeeksForGeeks’ and ‘Authors’ will be created first then ‘Nikhil’ directory will be created.

Create a Directory Using os.mkdirs()

Syntax: *os.makedirs(path, mode = 0o777, exist_ok = False)*

Parameter:

path: *A path-like object representing a file system path. A path-like object is either a string or bytes object representing a path.*

mode (optional): *A Integer value representing mode of the newly created directory. If this parameter is omitted then the default value 0o777 is used.*

exist_ok (optional): *A default value False is used for this parameter. If the target directory already exists an OSError is raised if its value is False otherwise not.*

Return Type: *This method does not return any value.*

Create a Directory Using os.mkdirs()

Example

Use of `os.makedirs()` method to create directory.

1

```
# Python program to explain os.makedirs() method

# importing os module
import os

# Leaf directory
directory = "Nikhil"

# Parent Directories
parent_dir = "D:/Pycharm projects/GeeksForGeeks/Authors"

# Path
path = os.path.join(parent_dir, directory)

# Create the directory
# 'Nikhil'
os.makedirs(path)
print("Directory '% s' created" % directory)

# Directory 'GeeksForGeeks' and 'Authors' will
# be created too
# if it does not exists
```

2

```
# Leaf directory
directory = "c"

# Parent Directories
parent_dir = "D:/Pycharm projects/GeeksforGeeks/a/b"

# mode
mode = 0o666

path = os.path.join(parent_dir, directory)

# Create the directory 'c'

os.makedirs(path, mode)
print("Directory '% s' created" % directory)
```

Output:

```
Directory 'Nikhil' created
Directory 'c' created
```

Delete a Directory using os.rmdir()

- `os.rmdir()` method in Python is used to remove or delete an empty directory.
- `OSError` will be raised if the specified path is not an empty directory.

Syntax: *os.rmdir(path, *, dir_fd = None)*

Parameter:

- ***path:*** *A path-like object representing a file path. A path-like object is either a string or bytes object representing a path.*
- ***dir_fd (optional):*** *A file descriptor referring to a directory. The default value of this parameter is None. If the specified path is absolute then dir_fd is ignored.*

Note: *The '*' in parameter list indicates that all following parameters (Here in our case 'dir_fd') are keyword-only parameters and they can be provided using their name, not as positional parameter.*

Return Type: *This method does not return any value.*

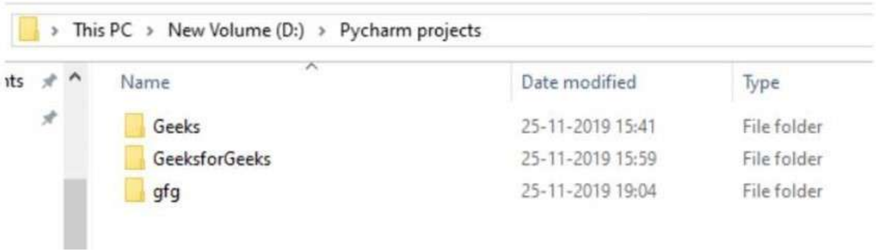
Note: You can only remove *empty* folders.

Delete a Directory using os.rmdir()

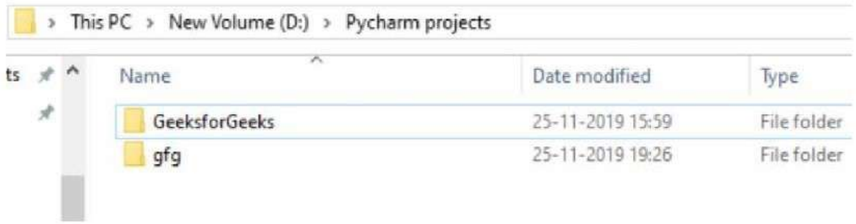
- Example:

```
# importing os module
import os

# Directory name
directory = "Geeks"

# Parent Directory
parent = "D:/Pycharm projects/"

# Path
path = os.path.join(parent, directory)

# Remove the Directory
# "Geeks"
os.rmdir(path)
```

Before



After



Delete a Directory using os.rmdir()

- Example:

Error Handling while deleting a directory

```
# importing os module
import os

# Directory name
directory = "GeeksforGeeks"

# Parent Directory
parent = "D:/Pycharm projects/"

# Path
path = os.path.join(parent, directory)

# Remove the Directory
# "GeeksforGeeks"
try:
    os.rmdir(path)
    print("Directory '% s' has been removed successfully" % directory)
except OSError as error:
    print(error)
    print("Directory '% s' can not be removed" % directory)
```

Output:

```
[WinError 145] The directory is not empty: 'D:/Pycharm projects/GeeksforGeeks'
Directory 'GeeksforGeeks' can not be removed
```

Listing Files

- Python now supports a number of APIs to list the directory contents.
- We can use:
  - `Path.iterdir`
  - `os.scandir`
  - `os.walk`
  - `Path.rglob`
  - `os.listdir`

Listing Files

- `os.listdir()` method gets the list of all files and directories in a specified directory.
- By default, it is the current directory.
- Beyond the first level of folders, `os.listdir()` does not return any files or folders.

Syntax: *os.listdir(path)*

Parameters:

- *Path of the directory*

Return Type: *returns a list of all files and directories in the specified path*

Listing Files

- **Example:**

```
# import OS module
import os

# Get the list of all files and directories
path = "C://Users//Vanshi//Desktop//gfg"
dir_list = os.listdir(path)

print("Files and directories in '", path, "' :")

# prints all files
print(dir_list)
```

Output:

```
In [19]: runfile('C:/Users/Vanshi/Desktop/gfg/untitled5.py', wdir='C:/Users/Vanshi/
Desktop/gfg')
Files and directories in ' C://Users//Vanshi//Desktop//gfg ' :
['bestsellers.csv', 'country_wise_latest.csv', 'covid_19_clean_complete.csv',
'cumulative.csv', 'day_wise.csv', 'full_grouped.csv', 'GFG.png', 'gfgcopy.txt',
'my_pdf.pdf', 'output_folder', 'temperature_dataframe_editUS.csv', 'test.txt',
'test1.txt', 'titanic_train.csv', 'untitled5.py', 'usa_county_wise.csv',
'worldometer_data.csv']
```

Listing Files

- **Example:**

  To get all the files, and no folders.

```
import os

print("Python Program to print list the files in a directory.")

Direc = input(r"Enter the path of the folder: ")
print(f"Files in the directory: {Direc}")

files = os.listdir(Direc)
files = [f for f in files if os.path.isfile(Direc+'/'+f)] #Filtering only the files.
print(*files, sep="\n")
```

Listing Files

- **Example:**

  To get only .txt files.

```
# import OS
import os

for x in os.listdir():
    if x.endswith(".txt"):
        # Prints only text file present in My Folder
        print(x)
```

Output:

```
In [5]: runfile('C:/Users/Vanshi/Desktop/gfg/untitled5.py', wdir='C:/Users/Vanshi/
Desktop/gfg')
gfgcopy.txt
test.txt
test1.txt
```

Listing Files

- `os.scandir()` is supported for Python 3.5 and greater.

Syntax: *os.scandir(path = '.')*

Return Type: *returns an iterator of os.DirEntry object.*

Listing Files

- **Example:**

```python
# import OS module
import os

# This is my path
path = "C://Users//Vanshi//Desktop//gfg"

# Scan the directory and get
# an iterator of os.DirEntry objects
# corresponding to entries in it
# using os.scandir() method
obj = os.scandir()

# List all files and directories in the specified path
print("Files and Directories in '% s':" % path)
for entry in obj:
    if entry.is_dir() or entry.is_file():
        print(entry.name)
```

Output:

```
In [9]: runfile('C:/Users/Vanshi/Desktop/gfg/untitled5.py', wdir='C:/Users/Vanshi/
Desktop/gfg')
Files and Directories in 'C://Users//Vanshi//Desktop//gfg':
bestsellers.csv
country_wise_latest.csv
covid_19_clean_complete.csv
cumulative.csv
day_wise.csv
full_grouped.csv
GFG.png
gfgcopy.txt
my_pdf.pdf
output_folder
temperature_dataframe_editUS.csv
test.txt
test1.txt
titanic_train.csv
untitled5.py
usa_county_wise.csv
worldometer_data.csv
```

Python Fundamentals

Accessing Databases

Topics

- Selecting, inserting, updating and deleting data
- Generic database API based on SQLite 3, PostgreSQL and MySQL
- Using the Object Relational Mapper (SQLAlchemy)
- Working with NoSQL databases

Selecting, inserting, updating and deleting data

Databases in Python

- The Python programming language has powerful features for database programming.
- Python supports various databases like SQLite, MySQL, Oracle, Sybase, PostgreSQL, etc.
- Python also supports Data Definition Language (DDL), Data Manipulation Language (DML) and Data Query Statements.
- The Python standard for database interfaces is the Python DB-API.
- Most Python database interfaces adhere to this standard.
- Here is the list of available Python database interfaces: Python Database Interfaces and APIs.
- You must download a separate DB API module for each database you need to access.

Connecting to Database - Using SQLite3

- Connecting to SQLite database in python is done by using:
  - python's inbuilt, sqlite3 module.
- To do that you should:
  - Create a connection object that represents the database
  - Then create some cursor objects to execute SQL statements

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')

print "Opened database successfully";
```

Output:

```
$chmod +x sqlite.py
$./sqlite.py
Open database successfully
```

Create a Table

- Following Python program will be used to create a table in the previously created database.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute('''CREATE TABLE COMPANY
         (ID INT PRIMARY KEY     NOT NULL,
         NAME           TEXT    NOT NULL,
         AGE            INT     NOT NULL,
         ADDRESS        CHAR(50),
         SALARY         REAL);''')
print "Table created successfully";

conn.close()
```

Output:

```
Opened database successfully
Table created successfully
```

Insert Operation

- Following Python program shows how to create records in the COMPANY table created

```
import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (1, 'Paul', 32, 'California', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (2, 'Allen', 25, 'Texas', 15000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (3, 'Teddy', 23, 'Norway', 20000.00 )");

conn.execute("INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY) \
      VALUES (4, 'Mark', 25, 'Rich-Mond ', 65000.00 )");

conn.commit()
print "Records created successfully";
conn.close()
```

Output:

```
Opened database successfully
Records created successfully
```

Select Operation

- Following Python program shows how to fetch and display records from the COMPANY table created

```
import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

Output:

```
Opened database successfully
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Update Operation

- Following Python code shows how to use UPDATE statement to update any record and then fetch and display the updated records from the COMPANY table.

```
import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("UPDATE COMPANY set SALARY = 25000.00 where ID = 1")
conn.commit
print "Total number of rows updated :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

Output:

```
Opened database successfully
Total number of rows updated : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 25000.0

ID = 2
NAME = Allen
ADDRESS = Texas
SALARY = 15000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Delete Operation

- Following Python code shows how to use DELETE statement to delete any record and then fetch and display the remaining records from the COMPANY table.

```
#!/usr/bin/python

import sqlite3

conn = sqlite3.connect('test.db')
print "Opened database successfully";

conn.execute("DELETE from COMPANY where ID = 2;")
conn.commit()
print "Total number of rows deleted :", conn.total_changes

cursor = conn.execute("SELECT id, name, address, salary from COMPANY")
for row in cursor:
   print "ID = ", row[0]
   print "NAME = ", row[1]
   print "ADDRESS = ", row[2]
   print "SALARY = ", row[3], "\n"

print "Operation done successfully";
conn.close()
```

Output:

```
Opened database successfully
Total number of rows deleted : 1
ID = 1
NAME = Paul
ADDRESS = California
SALARY = 20000.0

ID = 3
NAME = Teddy
ADDRESS = Norway
SALARY = 20000.0

ID = 4
NAME = Mark
ADDRESS = Rich-Mond
SALARY = 65000.0

Operation done successfully
```

Generic database API based on SQLite 3, PostgreSQL and MySQL

Using the Object Relational Mapper (SQLAlchemy)

SQLAlchemy

ORM

Student

- SQLAlchemy is a popular SQL toolkit and **Object Relational Mapper**.
- It is written in **Python** and gives full power and flexibility of SQL to an application developer.
- It is an **open source** and **cross-platform software** released under MIT license.
- SQLAlchemy is famous for its object-relational mapper (ORM), using which:
  - Classes can be mapped to the database,
    - allowing the object model and database schema to develop in a cleanly decoupled way from the beginning.

Expression Language

Sqlite
Postgre
MySQL

- Expression Language is one of the core components of SQLAlchemy.
- It allows the programmer to specify SQL statements in Python code and use it directly in more complex queries.
- Expression language is independent of backend and comprehensively covers every aspect of raw SQL.
- It is closer to raw SQL than any other component in SQLAlchemy.
- Statements of Expression language will be translated into corresponding raw SQL queries by SQLAlchemy engine.

Connecting to Database using SQLAlchemy

sqlalchemy
not
internal

pip install sqlalchemy

- The engine class connects a **Pool and Dialect together** to provide a source of database **connectivity and behavior**.
- An object of Engine class is instantiated using the **`create_engine()`** function.
- The **`create_engine()`** function takes the database as one argument.
- The database is not needed to be defined anywhere.
- The standard calling form has to send the URL as the first positional argument, usually a string that indicates database dialect and connection arguments.

Using the code given below, we can create a database

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite:///college.db', echo = True)
```

*The **echo flag** is a shortcut to set up SQLAlchemy logging, which is accomplished via Python's standard logging module.* To hide the verbose output, set echo attribute to **None**.

Connecting to Database using SQLAlchemy

- The `create_engine()` function returns an **Engine object.**
- Some important methods of Engine class are:

connect()

Returns connection object

execute()

Executes a SQL statement construct

dispose()

Disposes of the connection pool used by the Engine

begin()

Returns a context manager delivering a Connection with a Transaction established. Upon successful operation, the Transaction is committed, else it is rolled back

driver()

Driver name of the Dialect in use by the Engine

transaction()

Executes the given function within a transaction boundary

table_names()

Returns a list of all table names available in the database

Creating tables using SQLAlchemy

- To create a table, we have to:
  - Use the `Table()` function, which requires:
    - **Metadata:** `MetaData` object that will hold this table
    - **Classes:** One or more objects of `column` class
- SQLAlchemy Column object:
  - Represents a column in a database table which is in turn represented by a `Tableobject`.
- Metadata contains:
  - Definitions of tables and associated objects such as index, view, triggers, etc.

Creating tables using SQLAlchemy

- Metdata
  - An object of MetaData class is a collection of Table objects and their associated schema constructs.
  - It holds a collection of Table objects as well as an optional binding to an Engine or Connection.

```
from sqlalchemy import MetaData
meta = MetaData()
```

Creating tables using SQLAlchemy

- Column object represents a **column** in a **database table**.
- Constructor takes `name`, `type` and other parameters such as `primary_key`, `autoincrement` and other constraints.
- SQLAlchemy matches Python data to the best possible generic column data types defined in it.
- Some of the generic data types are:
  - BigInteger
  - Boolean
  - Date
  - DateTime
  - Float
  - Integer
  - Numeric
  - SmallInteger
  - String
  - Text
  - Time

Creating tables using SQLAlchemy

- To create a **students table** in college database, use the following snippet:

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String
engine = create_engine('sqlite:///college.db', echo = True)
meta = MetaData()

students = Table(
   'students', meta,
   Column('id', Integer, primary_key = True),
   Column('name', String),
   Column('lastname', String),
)
meta.create_all(engine)
```

The `create_all()` function uses the engine object to create all the defined table objects and stores the information in metadata.

Output:

```
CREATE TABLE students (
   id INTEGER NOT NULL,
   name VARCHAR,
   lastname VARCHAR,
   PRIMARY KEY (id)
)
```

*Because echo attribute of `create_engine()` function is set to **True**, the console will display the actual SQL query for table creation*

Creating tables using SQLAlchemy

- To create a **students table** in college database, use the following snippet:

```
from sqlalchemy import create_engine, MetaData, Table, Column, Integer, String
engine = create_engine('sqlite:///college.db', echo = True)
meta = MetaData()

students = Table(
   'students', meta,
   Column('id', Integer, primary_key = True),
   Column('name', String),
   Column('lastname', String),
)
meta.create_all(engine)
```

The `create_all()` function uses the engine object to create all the defined table objects and stores the information in metadata.

Output:

```
CREATE TABLE students (
   id INTEGER NOT NULL,
   name VARCHAR,
   lastname VARCHAR,
   PRIMARY KEY (id)
)
```

*Because echo attribute of `create_engine()` function is set to **True**, the console will display the actual SQL query for table creation*

Inserting data using SQLAlchemy

- the `INSERT` statement is created by executing `insert()` method:

```
ins = users.insert().values(name = 'Karan')
```

- The result of above method is an insert object that can be verified by using `str()` function:

```
str(ins)
```

Output:

```
'INSERT INTO users (name) VALUES (:name)'
```

- Similarly, methods like `update()`, `delete()` and `select()` create `UPDATE`, `DELETE` and `SELECT` expressions respectively.

Executing Expressions

- In order to execute the resulting SQL expressions, we have to **obtain a connection object representing an actively checked out DBAPI connection resource** and then **feed the expression object** as shown in the code below.

```
conn = engine.connect()
```

- The following `insert()` object can be used for `execute()` method

```
ins = students.insert().values(name = 'Ravi', lastname = 'Kapoor')
result = conn.execute(ins)
```

Output:

```
INSERT INTO students (name, lastname) VALUES (?, ?)
('Ravi', 'Kapoor')
COMMIT
```

Executing Expressions

- To issue many inserts using DBAPI's execute many() method, we can send in a list of dictionaries each containing a distinct set of parameters to be inserted.

```
conn.execute(students.insert(), [
   {'name':'Rajiv', 'lastname' : 'Khanna'},
   {'name':'Komal','lastname' : 'Bhandari'},
   {'name':'Abdul','lastname' : 'Sattar'},
   {'name':'Priya','lastname' : 'Rajhans'},
])
```

Selecting data using SQLAlchemy

- The **`select()`** method of table object enables us to **construct `SELECT` expression.**

```
s = students.select()
```

- The select object translates to **`SELECT` query by `str(s)` function** as shown below:

```
'SELECT students.id, students.name, students.lastname FROM students'
```

- We can use this select object as a parameter to the **`execute()`** method of the connection object

```
result = conn.execute(s)
```

Selecting data using SQLAlchemy

- The resultant variable is an equivalent of cursor in DBAPI. We can now fetch records using **`fetchone()` method.**

```
row = result.fetchone()
```

- All selected rows in the table can be printed by a **`for` loop:**

```
for row in result:
   print (row)
```

- Here, we have to note that select object can also be obtained by **`select()`** function in **`sqlalchemy.sql`** module.
- The **`select()`** function requires the table object as argument.

```
from sqlalchemy.sql import select
s = select([users])
result = conn.execute(s)
```

Updating data using SQLAlchemy

- The **update()** method on target table object constructs equivalent UPDATE SQL expression.

```
stmt = students.update().where(students.c.lastname == 'Khanna').values(lastname = 'Kapoor')
```

- The **values()** method on the resultant update object is used to specify the SET conditions of the UPDATE.
- The stmt object is an update object that translates to:

```
'UPDATE students SET lastname = :lastname WHERE students.lastname = :lastname_1'
```

- The bound parameter **lastname_1** will be substituted when **execute()** method is invoked.

Deleting data using SQLAlchemy

- The delete operation can be achieved by running delete() method on target table object:

```
stmt = students.delete().where(students.c.id > 2)
```

- The resultant SQL expression will have a bound parameter which will be substituted at runtime when the statement is executed.

```
'DELETE FROM students WHERE students.id > :id_1'
```

Working with NoSQL databases

NoSQL and Python

- As more and more data become available as unstructured or semi-structured, the need of managing them through NoSql database increases.
- Python can also interact with NoSQL databases in a similar way as is interacts with Relational databases.
- we will use python to interact with MongoDB as a NoSQL database.
- MongoDB is an open-source document database and leading NoSQL database.
- MongoDB is written in C++.

Connecting to MongoDB

- In order to connect to MongoDB, python uses a library known as **pymongo**.
- You can add this library to your python environment, using the below command from the Anaconda environment.

```
conda install pymongo
```

- This library enables python to connect to MOngoDB using a db client.
- Once connected we select the db name to be used for various operations.

Inserting Data - MongoDB

- To insert data into MongoDB we use the `insert()` method which is available in the database environment.
- First we connect to the db using python code shown below and then we provide the document details in form of a series of key-value pairs.

```
# Use the insert method
result = employee.insert_one(employee_details)
```

Querying Data - MongoDB

- To query data from MongoDB we use the `find_one()` method which returns a single record from the collection.
- To better display the record content, use the `pprint()` because it has better format when displaying complex objects compared to the normal `print()`.

```
# Query for the inserted document.
Queryresult = employee.find_one({'Age': '42'})
pprint(Queryresult)
```

- To use `pprint()` you need to import its corresponding module:

```
from pprint import pprint
```

Updating Data - MongoDB

- Updating an existing MongoDB data is similar to inserting. We use the update() method which is native to mongoDB.

```
# Use the condition to choose the record
# and use the update method
db.employee.update_one(
        {"Age":'42'},
        {
        "$set": {
            "Name":"Srinidhi",
            "Age":'35',
            "Address":"New Omsk, WC"
        }
        }
    )
```

Deleting Data - MongoDB

- Deleting a record is also straight forward where we use the delete method.
- Here also we mention the condition which is used to choose the record to be deleted.

```
# Use the condition to choose the record
# and use the delete method
db.employee.delete_one({"Age":'35'})
```

Python Fundamentals

Conquering the Web

Topics

- Retrieving web pages → request
- Parsing HTML and XML → BeautifulSoup
- Filling web forms automatically → Selenium
- Creating web applications in Python → django or flask

Collections

collection

movie

movie

collection

movie

movie

Retrieving web pages

Retrieving Web pages

- Python is used for a number of things, from data analysis to server programming.
- And one exciting use-case of Python is Web Scraping.
- We need to send a request in order to retrieve a web page from a webserver.
- The `requests` module allows you to send HTTP requests using Python.
- The HTTP request returns a Response Object with all the response data (content, encoding, status, and so on).

```
import requests

res = requests.get('https://codedamn.com')

print(res.text)
print(res.status_code)
```

Parsing HTML and XML

Parsing HTML

- `BeautifulSoup` library is used in Python to do web scraping.
- Some features that make `BeautifulSoup` a powerful solution are:
- It provides a lot of simple methods and Pythonic idioms for navigating, searching, and modifying a DOM tree.
- It doesn't take much code to write an application
- `BeautifulSoup` sits on top of popular Python parsers like `lxml` and `html5lib`, allowing you to try out different parsing strategies or trade speed for flexibility.

```
from bs4 import BeautifulSoup

page = requests.get("https://codedamn.com")
soup = BeautifulSoup(page.content, 'html.parser')
title = soup.title.text # gets you the text of the <title>(...)</title>
```

Select with BeautifulSoup

- Once you have the soup variable (like in the previous example), you can work with `.select` on it which is a `CSS` selector inside `BeautifulSoup`.
- That is, you can reach down the `DOM` tree just like how you will select elements with `CSS`.

```
import requests
from bs4 import BeautifulSoup

# Make a request
page = requests.get(
    "https://codedamn-classrooms.github.io/webscraper-python-codedamn-classroom-website/")
soup = BeautifulSoup(page.content, 'html.parser')

# Extract first <h1>(...)</h1> text
first_h1 = soup.select('h1')[0].text
```

Lab: Working with the Web

Retrieving webpages, scraping, parsing and extracting HTML content

Parsing XML

- What is XML?
  - The Extensible Markup Language (XML) is a markup language much like HTML or SGML.
  - This is recommended by the World Wide Web Consortium and available as an open standard.
  - XML is a portable, open source language that allows programmers to develop applications that can be read by other applications, regardless of operating system and/or developmental language.
  - XML is extremely useful for keeping track of small to medium amounts of data without requiring a SQL-based backbone.

Parsing XML

- The Python standard library provides a minimal but useful set of interfaces to work with XML.
- The two most basic and broadly used APIs to XML data are the `SAX` and `DOM` interfaces.
  - **Simple API for XML (SAX):**
    - Here, you register callbacks for events of interest and then let the parser proceed through the document.
    - This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.
  - **Document Object Model (DOM) API:**
    - This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

Parsing XML

- SAX obviously cannot process information as fast as DOM can when working with large files.
- On the other hand, using DOM exclusively can really kill your resources, especially if used on a lot of small files.
- SAX is read-only, while DOM allows changes to the XML file.
- Since these two different APIs literally complement each other, there is no reason why you cannot use them both for large projects.

Parsing XML with DOM APIs

- The Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents.
- The `DOM` is extremely useful for random-access applications.
  - `SAX` only allows you a view of one bit of the document at a time.
  - If you are looking at one `SAX` element, you have no access to another.
- Here is the easiest way to quickly load an XML document and to create a `minidom` object using the `xml.dom` module.
- The `minidom` object provides a simple parser method that quickly creates a `DOM` tree from the XML file.
- The sample phrase calls the `parse( file [,parser] )` function of the `minidom` object to parse the XML file designated by file into a `DOM` tree object.

Lab: Parsing XML Data using DOM API

Filling web forms automatically

Autofilling Web Forms

- Python can be used to achieve a state where the interaction with a webpage can be automated.
- This type of automation can be achieved through the use of Selenium package
  - Selenium Python bindings provides a simple API to write functional/acceptance tests using Selenium WebDriver.
- Through Selenium Python API you can access all functionalities of Selenium WebDriver in an intuitive way.
- Selenium Python bindings provide a convenient API to access Selenium WebDrivers like Firefox, IE, Chrome, Remote etc.
- The current supported Python versions are 3.5 and above.

Installing Python bindings for Selenium

- Use pip to install the selenium package. Python 3 has pip available in the standard library.
- Using *pip*, you can install selenium like this:

```
pip install selenium
```

Drivers

- Selenium requires a driver to interface with the chosen browser.
- Firefox, for example, requires geckodriver, which needs to be installed before the code can be run.
- Make sure it's in your *PATH*, e. g., place it in */usr/bin* or */usr/local/bin*.
- Failure to observe this step will give you an error:
  - `selenium.common.exceptions.WebDriverException: Message: 'geckodriver' executable needs to be in PATH`.
- Other supported browsers will have their own drivers available. Links to some of the more popular browser drivers follow.

Chrome:	https://sites.google.com/chromium.org/driver/
Edge:	https://developer.microsoft.com/en-us/microsoft-edge/tools/webdriver/
Firefox:	https://github.com/mozilla/geckodriver/releases
Safari:	https://webkit.org/blog/6900/webdriver-support-in-safari-10/

Using Selenium

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.by import By
```

- The *selenium.webdriver* module provides all the WebDriver implementations.
- Currently supported WebDriver implementations are Firefox, Chrome, IE and Remote.
- The *Keys* class provide keys in the keyboard like RETURN, F1, ALT etc.
- The *By* class is used to locate elements within a document.

Using Selenium – WebDriver instance

- Next, the instance of Firefox `WebDriver` is created.

```
driver = webdriver.Firefox()
```

- The `driver.get` method will navigate to a page given by the URL.
- WebDriver will wait until the page has fully loaded (that is, the "`onload`" event has fired) before returning control to your code.

```
driver.get("http://www.python.org")
```

Using Selenium – Finding Elements

- WebDriver offers a number of ways to find elements using the `find_element` method.
- For example, the input text element can be located by its `name` attribute using the `find_element` method and using `By.NAME` as its first parameter.

```
elem = driver.find_element(By.NAME, "q")
```

Using Selenium – Sending Input

- Next, we are sending keys, this is similar to entering keys using your keyboard.
- Special keys can be sent using the `Keys` class imported from `selenium.webdriver.common.keys`.
- To be safe, we'll first clear any pre-populated text in the input field (e.g. "Search") so it doesn't affect our search results:

```
elem.clear()
elem.send_keys("pycon")
elem.send_keys(Keys.RETURN)
```

Using Selenium – Waits

- These days, most of the web apps are using AJAX techniques.
- When a page is loaded by the browser, the elements within that page may load at different time intervals.
- This makes locating elements difficult: if an element is not yet present in the DOM, a locate function will raise an `ElementNotVisibleException` exception.
- Using waits, we can solve this issue.
- Waiting provides some slack between actions performed - mostly locating an element or any other operation with the element.

Using Selenium – Waits

- Selenium Webdriver provides two types of waits
  - Implicit:
    - Makes `WebDriver` poll the DOM for a certain amount of time when trying to locate an element.
  - Explicit:
    - Makes `WebDriver` wait for a certain condition to occur before proceeding further with execution.

Using Selenium – Explicit Waits

- Waits for a certain condition to occur before proceeding further in the code.
- The extreme case of this is `time.sleep()`:
  - Sets the condition to an exact time period to wait.
- There are some convenience methods provided that help you write code that will wait only as long as required.
  - `WebDriverWait` in combination with `ExpectedCondition` is one way this can be accomplished.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

driver = webdriver.Firefox()
driver.get("http://somedomain/url_that_delays_loading")
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "myDynamicElement"))
    )
finally:
    driver.quit()
```

Using Selenium – Explicit Waits

- There are some common conditions that are frequently of use when automating web browsers.

  - title_is
  - title_contains
  - presence_of_element_located
  - visibility_of_element_located
  - visibility_of
  - presence_of_all_elements_located
  - text_to_be_present_in_element
  - text_to_be_present_in_element_value
  - frame_to_be_available_and_switch_to_it
  - invisibility_of_element_located
  - element_to_be_clickable
  - staleness_of
  - element_to_be_selected
  - element_located_to_be_selected
  - element_selection_state_to_be
  - element_located_selection_state_to_be
  - alert_is_present

Using Selenium – Implicit Waits

- An implicit wait tells WebDriver to poll the DOM for a certain amount of time when trying to find any element (or elements) not immediately available.
- The default setting is 0 (zero).
- Once set, the implicit wait is set for the life of the WebDriver object.

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.implicitly_wait(10) # seconds
driver.get("http://somedomain/url_that_delays_loading")
myDynamicElement = driver.find_element_by_id("myDynamicElement")
```

Lab: Auto-fill a Web Form

Creating web applications in Python

Web Development

- Python is one of the most in-demand programming languages.
  - It is because of its ease of use and easy-to-read syntax.
- Being a popular and easy language, Python is used in several fields to develop various kinds of applications such as:
  - Desktop applications
  - Machine Learning models
  - Web Development

Web Development

- **Web Development** means developing applications that can be accessed via the internet on a web browser.
- A web application has two components:
  - **Frontend**
    - Is the part that deals with the users and their interactions.
    - It is what the users see on their web browser when they visit a URL to access a web application.
  - **Backend**
    - Is the part that deals with the server-side aspects.
    - The backend handles storing, retrieving, and formatting data in an agreed-upon format so that the data can be parsed and understood by other applications.

What makes Python suitable for Web Development?

- Short Learning Curve
- Rich Ecosystem
- Development Speed
- Large Community

Python Web Frameworks

- There are multiple Python Web Development Frameworks for backend development, ranging from small, focused, and micro frameworks to big and batteries-included frameworks.
- Some of the most popular Python web frameworks:
  - Flask
  - Django

Flask

- Flask is one of the most popular HTTP Python Web Development Framework.
- Since it is a micro framework, it does not have a lot of features baked into it that the other web frameworks might have, such as templating, account authorization, authentication, etc.
- However, Flask gives you the freedom to use any library or even custom code to deal with those concerns.
- It is used by huge companies including Netflix, LinkedIn, Uber, etc.

Django

- Django is a Python HTTP framework for building the backend of web applications.
- It is a batteries-included framework that includes a lot of components, such as ORM, templating engine serializer, etc., for implementing various tasks.
- One of the reasons for Django's popularity is that it is quite easy to learn and use, especially because of its pluggable architecture.
- Django allows you to build small decoupled apps that can be included or plugged into larger projects.
- In Django, you get apps and projects, where an app is a small self-contained codebase, while a project is a collection of multiple apps.

Lab: Creating a webpage using Flask