

PYTHON LANGUAGE TRAINING

Mohamad Rafiq

PYTHON PROGRAMMING

2

- Introduction to python language
- Download & Install python
- Python Syntax and comments
- Python Keywords and Identifiers
- Python Data Types, Variables
- Python Operators
- Control flow – Decision making
- Control flow – Looping, Branching

WHY PYTHON

- Easy and Powerful
- High level Language
- Interpreted language
- Object Oriented language
- Portable
- Extensible
- Embeddable
- Extensive libraries

EASY AND POWERFUL

To print Helloworld:

Java:

```
public class HelloWorld
{
    p s v main(String[] args)
    {
        SOP("Hello world");
    }
}
```

C:

```
#include<stdio.h>
void main()
{
    print("Hello world");
}
```

Python:

```
print("Hello World")
```

EASY AND POWERFUL

To print the sum of 2 numbers

Java:

```
public class Add
{
    public static void main(String[] args)
    {
        int a,b;
        a =10;
        b=20;
        System.out.println("The Sum:"+a+b));
    }
}
```

C:

```
#include <stdio.h>

void main()
{
    int a,b;
    a =10;
    b=20;
    printf("The Sum:%d",a+b));
}
```

Python:

```
a=10
b=20
print("The Sum:",a+b))
```

ABOUT PYTHON

Named after TV show *Monty Python's Flying Circus* broadcasted in BBC from 1969 to 1974.

Guido van Rossum developed Python language by taking almost all programming features from different languages

1. Functional Programming Features from C
2. Object Oriented Programming Features from C++
3. Scripting Language Features from Perl and Shell Script



USE OF PYTHON

Where we can use Python:

Almost everywhere.

1. Developing Web Applications – Django/Flask
2. AI/ML – Tensorflow/pytorch
3. IoT Applications – Raspberry Pi
4. Test Automation – Pytest/Robot
5. Game development – pygame/pyopenGL
6. Data Science – Pandas/numpy

INSTALL PYTHON

Python installer download

- www.python.org

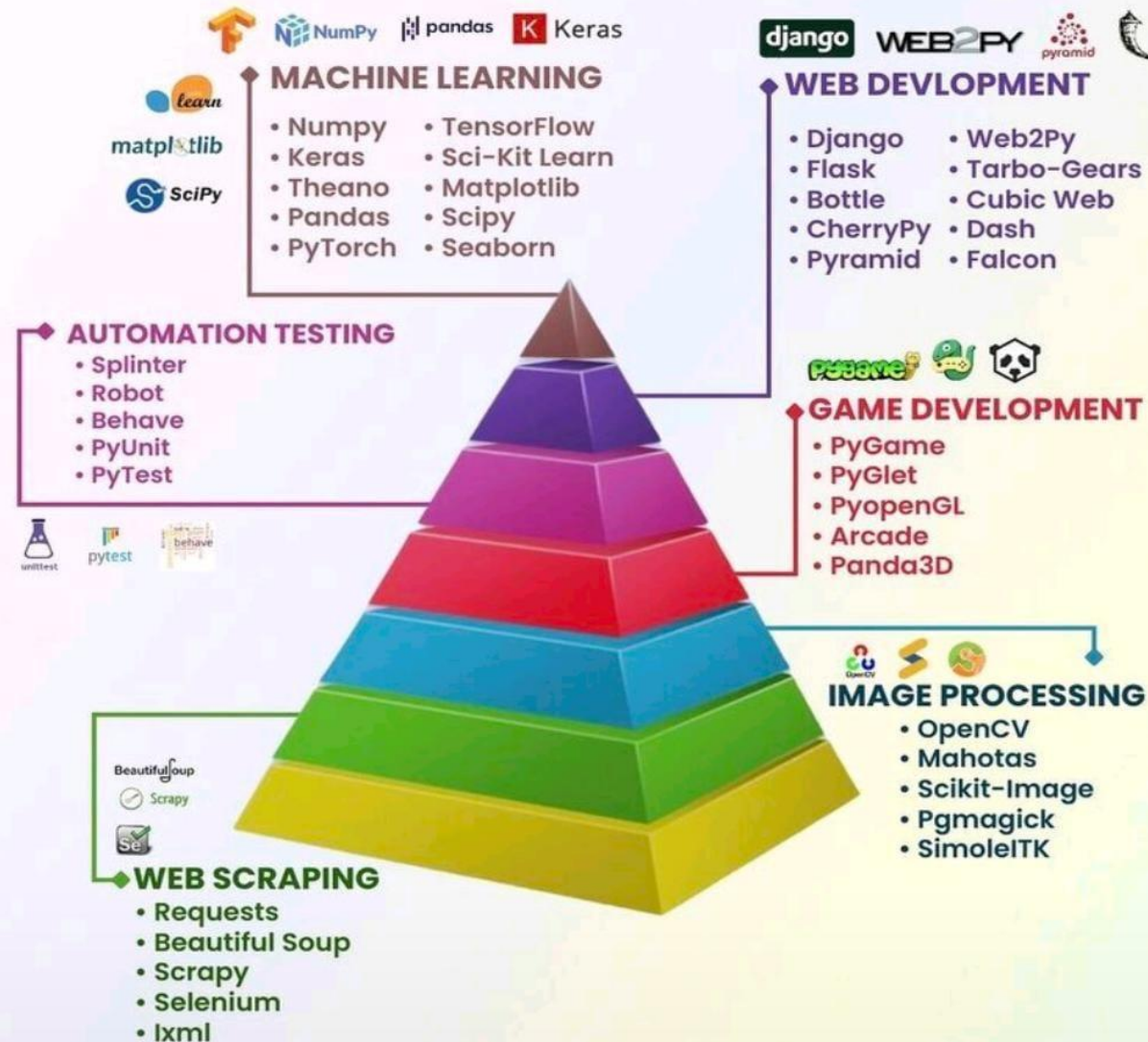
Python libraries

- www.pypi.org

Alternative implementations:

- [IronPython](#) (Python running on .NET)
- [Jython](#) (Python running on the Java Virtual Machine)
- [PyPy](#) (A [fast](#) python implementation with a JIT compiler)

PYTHON LIBRARIES AND FRAMEWORKS



PYTHON

There are libraries for regular expressions, documentation-generation, unit-testing, web browsers, threading, databases, CGI, email, image manipulation, and a lot of other functionality.

Dynamically Typed:

No need to declare type for variables. Whenever we are assigning the value, based on value, type will be allocated automatically

PYTHON SYNTAX, INDENTATION, COMMENTS

- Syntax is similar to other programming languages
 - Indentation
 - use equal number of space for every section of code
 - Comments
 - #(hash) for single line comment
 - Triple quotes `'''`, `"""` to comment multiple lines
- ```
'''

multi
line
comment
'''
```

# PYTHON KEYWORDS

*import keyword*

*keyword.kwlist*

*False, None, True, and, as, assert,  
async, await, break, class, continue,  
def, del, elif, else, except, finally,  
for, from, global, if, import, in, is,  
lambda, nonlocal, not, or, pass, raise,  
return, try, while, with, yield*

# SESSION 2: VIRTUALENV, DATATYPES

13

- **Virtual Environment**
- **Data Types**
- **Mutable/Immutable objects**

# VIARTUALENV

- `python -m venv /path/to/new/virtual/environment`  
Ex: `python -m venv testenv`
- `.\testenv\Scripts\activate`
- `which python`
- `which pip`
- `pip install pip tz`
- `pip list`
- `pip freeze --local > requirements.txt`
- `deactivate`

# PYTHON DATATYPES

## Primitive Data Structures

- These are the most primitive or the basic data structures. They are the building blocks for data manipulation.
- Python has four primitive variable types:
  - Integers
  - Float
  - Strings
  - Boolean

# PYTHON DATATYPES

16

## Integers

- You can use an integer represent numeric data, and more specifically, whole numbers from negative infinity to infinity, like 4, 5, or -1.

## Float

- "Float" stands for 'floating point number'. You can use it for rational numbers, usually ending with a decimal figure, such as 1.11 or 3.14.

## String

- Strings are collections of alphabets, words or other characters. You can create strings by enclosing a sequence of characters within a pair of single or double quotes. For example: 'cake', "cookie", etc.

## Boolean

- This built-in data type that can take up the values: True and False, which often makes them interchangeable with the integers 1 and 0.



# PYTHON DATATYPES

17

## Integers

- You can use an integer represent numeric data, and more specifically, whole numbers from negative infinity to infinity, like 4, 5, or -1.

## Float

- "Float" stands for 'floating point number'. You can use it for rational numbers, usually ending with a decimal figure, such as 1.11 or 3.14.

## String

- Strings are collections of alphabets, words or other characters. You can create strings by enclosing a sequence of characters within a pair of single or double quotes. For example: 'cake', "cookie", etc.

## Boolean

- This built-in data type that can take up the values: True and False, which often makes them interchangeable with the integers 1 and 0.

# MUTABLE/IMMUTABLE OBJECTS

## Mutable Datatypes

- Lists
- Dictionary
- Sets
- Arrays

## Immutable Datatypes

- Strings
- Tuples
- Integers
- Floats
- Boolean
- Frozenset

# TUPLES

19

*a tuple is a built-in data type that allows you to create immutable sequences of values. The values or items in a tuple can be of any type.*

**Ordered:** *They contain elements that are sequentially arranged according to their specific insertion order.*

*Lightweight, Indexable through a zero-based index*

**Immutable:** *They don't support in-place mutations or changes to their contained elements. They don't support growing or shrinking operations.*

**Heterogeneous , Nestable, Iterable, Sliceable**

# SESSION 3: OPERATORS & STRINGS

20

- **Operators in python**
- **Assert statement**
- **String Operations**

# PYTHON OPERATORS

21

+ (plus)

Adds two objects

- (minus)

\* (multiply)

\*\* (power)

Returns x to the power of y

/ (divide)

// (divide and floor)

Divide x by y and round the answer *down* to the nearest integer value. Note that if one of the values is a float, you'll get back a float.

13 // 3 gives 4

9//1.81 gives 4.0

% (modulo)

Returns the remainder of the division

13 % 3 gives 1 . -25.5 % 2.25 gives 1.5 .

# PYTHON OPERATORS

<< (left shift)

Shifts the bits of the number to the left by the number of bits specified.

2 << 2 gives 8 . 2 is represented by 10 in bits.

Left shifting by 2 bits gives 1000 which represents the decimal 8 .

>> (right shift)

Shifts the bits of the number to the right by the number of bits specified.

11 >> 1 gives 5 .

11 is represented in bits by 1011 which when right shifted by 1 bit gives 101 which is the decimal 5 .

& (bit-wise AND)

Bit-wise AND of the numbers: if both bits are 1 , the result is 1 . Otherwise, it's 0 .

5 & 3 gives 1 ( 0101 & 0011 gives 0001 )

| (bit-wise OR)

Bitwise OR of the numbers: if both bits are 0 , the result is 0 . Otherwise, it's 1 .

5 | 3 gives 7 ( 0101 | 0011 gives 0111 )

^ (bit-wise XOR)

Bitwise XOR of the numbers: if both bits ( 1 or 0 ) are the same, the result is 0 . Otherwise, it's 1 .

5 ^ 3 gives 6 ( 0101 ^ 0011 gives 0110 )

~ (bit-wise invert)

The bit-wise inversion of x is -(x+1)

~5 gives -6 .

# PYTHON OPERATORS

< (less than)

Returns whether x is less than y. All comparison operators return True or False . Note the capitalization of these names.

5 < 3 gives False and 3 < 5 gives True .

Comparisons can be chained arbitrarily: 3 < 5 < 7 gives True .

> (greater than)

Returns whether x is greater than y

5 > 3 returns True . If both operands are numbers, they are first converted to a common type.

Otherwise, it always returns

False .

# PYTHON OPERATORS

24

`<=` (less than or equal to)

`x = 3; y = 6; x <= y` returns True

`>=` (greater than or equal to)

`x = 4; y = 3; x >= 3` returns True

`==` (equal to)

Compares if the objects are equal

`x = 2; y = 2; x == y` returns True

`x = 'str'; y = 'stR'; x == y` returns False

**is**

`!=` (not equal to)

`x = 2; y = 3; x != y` returns True

`not` (boolean NOT)

`x = True; not x` returns False .

`and` (boolean AND)



# ASSERT STATEMENT

**assert** expression[, assertion\_message]

expression can be any valid Python expression or object,

which is then tested for truthiness.

If expression is false, then the statement throws an **AssertionError**.

# SESSION 4: CONTROL FLOW

26

- Python Program Flow
  - *if, elif, else* statements
  - *while* loop
  - *for* loop
- Control statements: *break, continue* and *pass*
- *range(start, end, step)*
- Examples for looping

# CONTROL FLOW

27

There are three control flow statements in Python - if , for and while .

## Conditional statements:

*if <expression>:*

*if else*

*if elif else*

## Looping:

*for*

*for with else*

*while*

*while with else*

## Control statements:

*break*

*continue*

*pass*

# CONTROL FLOW: break, continue & pass

| Statement       | Action                                               | Use Case                                   |
|-----------------|------------------------------------------------------|--------------------------------------------|
| <i>pass</i>     | Does nothing                                         | Placeholder for future code                |
| <i>continue</i> | Skips the rest of the loop for the current iteration | When you want to skip a specific iteration |
| <i>break</i>    | Terminates the loop                                  | When you want to end the loop prematurely  |

# SESSION 5&6 : FUNCTIONS AND MODULES

29

- **Built-in Functions**
- **User-defined Functions**
- **Modules**

# BUILT-IN FUNCTIONS

Built-in functions are pre-defined functions provided by the Python language that can be used to perform common tasks.

*len, dir*

*range, sum*

*print, enumerate etc.,*

# BUILT-IN FUNCTIONS

- Anonymous functions

- A map() function

**map(function, iterable[ iterable1, iterable2,..., iterableN])**

- A filter() function

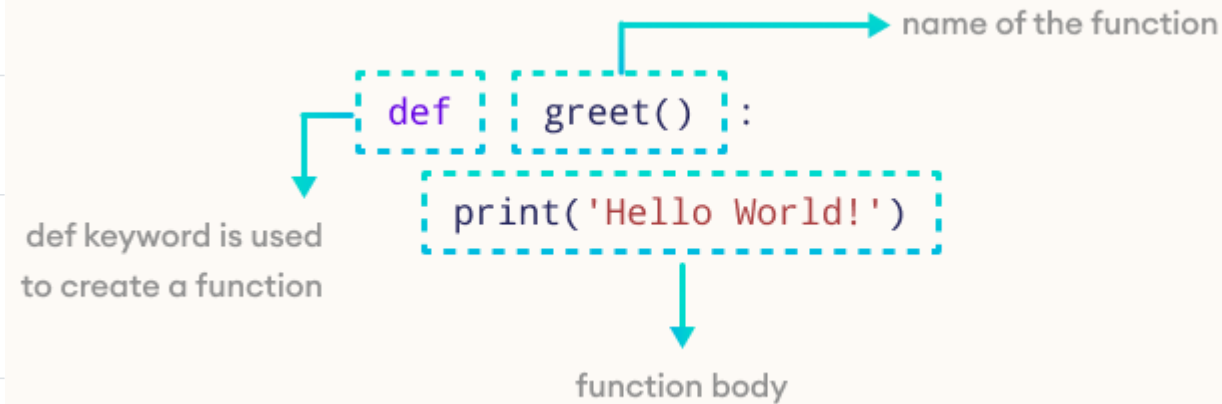
- A reduce() function

# USER-DEFINED FUNCTIONS

32

```
def <function_name>([<parameters>]):
 <statement(s)>
```

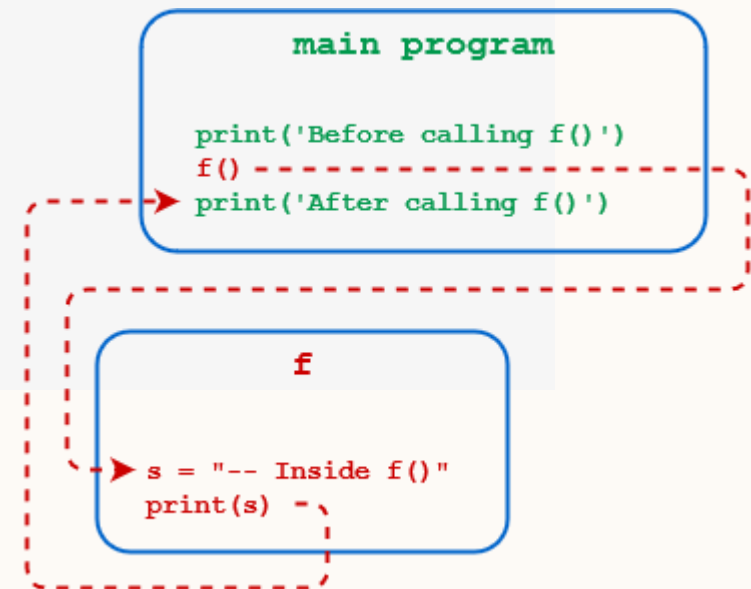
| Component       | Meaning                                                                            |
|-----------------|------------------------------------------------------------------------------------|
| def             | The keyword that informs Python that a function is being defined                   |
| <function_name> | A valid Python identifier that names the function                                  |
| <parameters>    | An optional, comma-separated list of parameters that may be passed to the function |
| :               | Punctuation that denotes the end of the Python function header                     |
| <statement(s)>  | A block of valid Python statements                                                 |



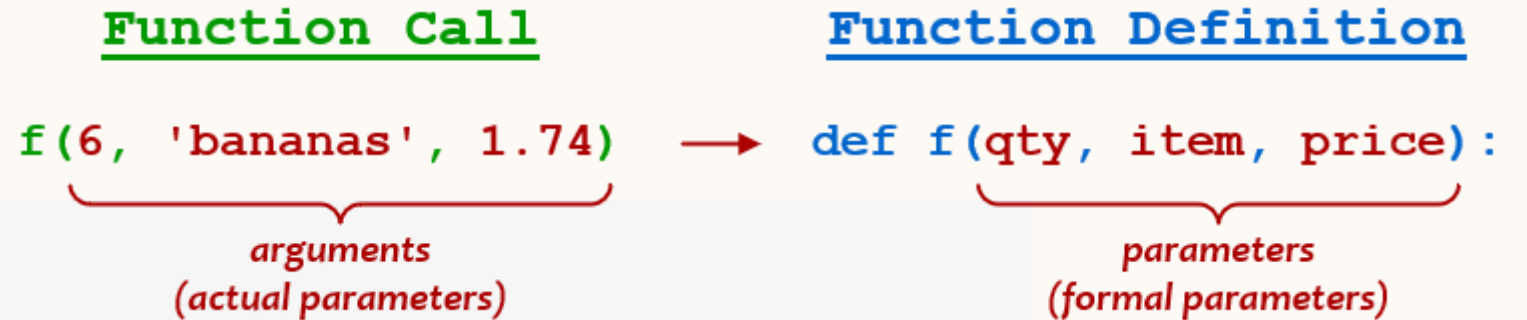


# USER-DEFINED FUNCTIONS

- **Abstraction and Reusability**
- ***Modularity***
- ***Namespace Separation***



# USER-DEFINED FUNCTIONS



## ➤ Argument Passing

1. **Positional arguments** must agree in order and number with the parameters declared in the function definition.
2. **Keyword arguments** must agree with declared parameters in number, but they may be specified in arbitrary order.
3. **Default parameters** allow some arguments to be omitted when the function is called.

## ➤ The return Statement

It immediately terminates the function and passes execution control back to the caller.

It provides a mechanism by which the function can pass data back to the caller.

## ➤ Variable-Length Arguments

# SESSION 7: FILES AND EXCEPTIONS

- File Handling
- Exception Handling
- Comprehensions
- Generators
- Decorators
- Python debugger ***pdb***

# TYPES OF ERRORS

- *SyntaxError*
- *ModuleNotFoundError*
- *ValueError*
- *KeyError*
- *TypeError*
- *NameError*
- *KeyboardInterrupt*
- *ZeroDivisionError*

# Exceptions Handling

- Errors
  - Exception handling with try
  - handling Multiple Exceptions
  - Writing your own Exception

try:

{

Run this code

except:

{

Execute this code when  
there is an exception

# Exceptions Handling

try:



Run this code

except:



Execute this code when  
there is an exception

else:



No exceptions? Run this  
code.

finally:



Always run this code.

# FILE HANDLING

39

- File handling Modes
  - Reading Files
  - Writing& Appending to Files
  - Handling File Exceptions
  - The *with* statement (Context manager)

# COMPREHENSIONS

40

## List comprehension:

Syntax:

*newList = [ expression(element) for element in oldList if condition ]*

*numbers = [12, 13, 14]*

*doubled = [x \*2 for x in numbers]*

*print(doubled)*

## Dictionary comprehension:

*newDict = [ key: value for element in oldList if condition ]*

*print(newDict)*

## Tuple comprehension:

*mytup = (3, 5, 7, 9, 12,11, 13, 4)*

*new\_tup = (x+1 for x in mytup if x%2==1)*

*print(tuple(new\_tup))*



# GENERATORS

41

*Definition: **A function that returns an iterator.***

*A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the **yield** keyword rather than **return**.*

*If the body of a def contains yield, the function automatically becomes a Python generator function.*

**next**

# PYTHON DEBUGGER

- `pdb`: The module `pdb` defines an interactive source code debugger for Python programs.
- It supports setting (conditional) breakpoints and single stepping at the source line level, inspection of stack frames, source code listing, and evaluation of arbitrary Python code in the context of any stack frame
- The typical usage to break into the debugger is to insert:
- ***import pdb;***
- ***pdb.set\_trace()***
- ***next*** or ***n*** - *Execute the current line and move to the next line ignoring function calls*
- ***step*** or ***s*** - *Step into functions called at the current line*
- ***print*** or ***p*** – *for printing objects*

# SESSION 8: OBJECT-ORIENTED PROGRAMMING

43

- Classes and Objects
- Methods
- Constructors
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

# OBJECT-ORIENTED PROGRAMMING

## Python Class

1. A blueprint to create objects.
2. The "object" is the base class in Python
3. The "class" keyword is used to create a class
4. Python constructor is defined with `__init__()` method.
5. Python doesn't support multiple constructors in a class i.e. No Constructor Overloading.

# OBJECT-ORIENTED PROGRAMMING

**Classes** are the building blocks of object-oriented programming in Python

**Model and solve:** You'll find many situations where the objects in your code map to real-world objects.

**Code Reuse:** You can define hierarchies of related classes. The base classes at the top of a hierarchy provide common functionality that you can reuse later in the subclasses down the hierarchy.

**Encapsulation:** You can use Python classes to bundle together related attributes and methods in a single entity, the object. This helps you better organize your code using modular and autonomous entities that you can even reuse across multiple projects.

**Abstraction:** You can use classes to abstract away the implementation details of core concepts and objects. This will help you provide your users with intuitive interfaces (APIs) to process complex data and behaviors.

**Polymorphism:** You can implement a particular interface in several slightly different classes and use them interchangeably in your code. This will make your code more flexible and adaptable.

**classes can help you write more organized, structured, maintainable, reusable, flexible, and user-friendly code**

# OOPS - INHERITANCE

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

 def speak(self):
 return f"{self.name} makes a sound"
```

```
class Dog(Animal):
 def speak(self):
 return f"{self.name} barks"
 def colour(self):
 def breed(self)
```

# OOPS - INHERITANCE

```
class Parent1:
 def method1(self):
 return "Method from Parent1"
```

```
class Parent2:
 def method2(self):
 return "Method from Parent2"
```

```
class Child(Parent1, Parent2):
 def method3(self):
 return "Method from Child"
```

```
class GrandChild(Child):
 def method4(self):
 return "Method from Child"
```

```
Example usage
child = Child()
```

- Global interpreter Lock

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.



# The GIL

- GIL: “Global Interpreter Lock” - this is a lock which must be acquired for a thread to enter the interpreter’s space.
- Only one thread may be executing within the Python interpreter at once.



# RECURSION

- Algorithmically: A way to design solutions to the problems by divide and conquer
  - *reduce a problem to simpler version of the same problem*
- Semantically: a programming technique where a function calls itself.
  - Ensure there is no infinite recursion
  - Must have 1 or more base cases that are easy to solve
  - Must solve the same problem on some other input with goal of simplifying the larger problem

# RECURSION

```
def factorial(n):
 return_value = 1
 for i in range(2, n + 1):
 return_value *= i
 return return_value
```

```
def factorial(n):
 if n <= 1 :
 return 1
 else:
 return n * factorial(n - 1)
```

# Python Collections Module

Python's **collections** module provide specialized datatypes providing alternatives to general purpose built in datatypes like dict, list, set and tuple

- Write readable and explicit code with ***namedtuple***
- Build efficient queues and stacks with ***deque***
- Count objects quickly with ***Counter***
- Handle missing dictionary keys with ***defaultdict***
- Guarantee the insertion order of keys with ***OrderedDict***
- Manage multiple dictionaries as a single unit with ***ChainMap***

# Python Collections Module

Python's **collections** module provide specialized datatypes providing alternatives to general purpose built in datatypes like dict, list, set and tuple

- Write readable and explicit code with ***namedtuple***
- Build efficient queues and stacks with ***deque***
- Count objects quickly with ***Counter***
- Handle missing dictionary keys with ***defaultdict***
- Guarantee the insertion order of keys with ***OrderedDict***
- Manage multiple dictionaries as a single unit with ***ChainMap***

# Python Collections Module

| Data type           | Description                                                                                                                                                          |
|---------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>deque</b>        | A sequence-like collection that supports efficient addition and removal of items from either end of the sequence                                                     |
| <b>defaultdict</b>  | A dictionary subclass for constructing default values for missing keys and automatically adding them to the dictionary                                               |
| <b>namedtuple()</b> | A factory function for creating subclasses of tuple that provides named fields that allow accessing items by name while keeping the ability to access items by index |
| <b>OrderedDict</b>  | A dictionary subclass that keeps the key-value pairs ordered according to when the keys are inserted                                                                 |
| <b>Counter</b>      | A dictionary subclass that supports convenient counting of unique items in a sequence or iterable                                                                    |
| <b>ChainMap</b>     | A dictionary-like class that allows treating a number of mappings as a single dictionary object                                                                      |

# Python Collections Module

## Namedtuple()

is a factory function that allows you to create tuple subclasses with named fields. These fields give you direct access to the values in a given named tuple using the dot notation, like in `obj.attr`.

```
>>> Point = namedtuple("Point", ["x", "y"])
```

```
>>> point = Point(2, 4)
```

```
>>> point
```

```
Point(x=2, y=4)
```

```
>>> # Access the coordinates
```

```
>>> point.x
```

```
2
```

```
>>> point.y
```

```
4
```

```
>>> point[0]
```

```
2
```

# Python Collections Module

## `deque()`

This sequence-like data type is a generalization of stacks and queues designed to support memory-efficient and fast append and pop operations on both ends of the data structure.

append and pop operations on the beginning or left side of list objects are inefficient, with  $O(n)$  time complexity.

append and pop operations on the right side of a list are normally efficient ( $O(1)$ )



# Python Collections Module

**deque()**

```
>>> from collections import deque
```

```
>>> ticket_queue = deque()
```

```
>>> ticket_queue
```

```
deque([])
```

```
>>> # People arrive to the queue
```

```
>>> ticket_queue.append("Jane")
```

```
>>> ticket_queue.append("John")
```

```
>>> ticket_queue.append("Linda")
```

```
>>> ticket_queue
```

```
deque(['Jane', 'John', 'Linda'])
```

```
>>> # People bought their tickets
```

```
>>> ticket_queue.popleft()
```

```
'Jane'
```

# Python Collections Module

**deque()**

```
>>> from collections import deque
```

```
>>> recent_files = deque(["core.py", "README.md", "__init__.py"], maxlen=3)
```

```
>>> recent_files.appendleft("database.py")
```

```
>>> recent_files
```

```
deque(['database.py', 'core.py', 'README.md'], maxlen=3)
```

```
>>> recent_files.appendleft("requirements.txt")
```

```
>>> recent_files
```

```
deque(['requirements.txt', 'database.py', 'core.py'], maxlen=3)
```

# Python Collections Module

| Method                      | Description                                   |
|-----------------------------|-----------------------------------------------|
| <code>.clear()</code>       | Remove all the elements from a deque          |
| <code>.copy()</code>        | Create a shallow copy of a deque              |
| <code>.count(x)</code>      | Count the number of deque elements equal to x |
| <code>.remove(value)</code> | Remove the first occurrence of value          |

Another interesting feature of deques is the ability to rotate their elements using `.rotate()`:

# Python Collections Module

## defaultdict()

A common problem you'll face when you're working with dictionaries in Python is how to handle missing keys. If you try to access a key that doesn't exist in a given dictionary, then you get a `KeyError`:

```
>>> from collections import defaultdict
>>> counter = defaultdict(int)
>>> counter
defaultdict(<class 'int'>, {})
>>> counter["dogs"]
0
>>> counter
defaultdict(<class 'int'>, {'dogs': 0})
>>> counter["dogs"] += 1
>>> counter["dogs"] += 1
>>> counter["dogs"] += 1
>>> counter["cats"] += 1
>>> counter["cats"] += 1
>>> counter
defaultdict(<class 'int'>, {'dogs': 3, 'cats': 2})
```

# Python Collections Module

## defaultdict()

```
>>> from collections import defaultdict

>>> pets = [
... ("dog", "Affenpinscher"),
... ("dog", "Terrier"),
... ("dog", "Boxer"),
... ("cat", "Abyssinian"),
... ("cat", "Birman"),
...]

>>> group_pets = defaultdict(list)

>>> for pet, breed in pets:
... group_pets[pet].append(breed)
...

>>> for pet, breeds in group_pets.items():
... print(pet, "->", breeds)
...
dog -> ['Affenpinscher', 'Terrier', 'Boxer']
cat -> ['Abyssinian', 'Birman']
```

# Python Collections Module

## OrderedDict()

- OrderedDict iterates over keys and values in the same order keys were first inserted into the dictionary.
- If you assign a new value to an existing key, then the order of the key-value pair remains unchanged.
- If an entry is deleted and reinserted, then it'll be moved to the end of the dictionary.

```
>>> from collections import OrderedDict
>>> life_stages = OrderedDict()
>>> life_stages["childhood"] = "0-9"
>>> life_stages["adolescence"] = "9-18"
>>> for stage, years in life_stages.items():
... print(stage, "->", years)
...
childhood -> 0-9
adolescence -> 9-18
```

# Python Collections Module

```
>>> word = "mississippi"
```

```
>>> counter = {}
```

```
>>> for letter in word:
```

```
... if letter not in counter:
```

```
... counter[letter] = 0
```

```
... counter[letter] += 1
```

```
...
```

```
>>> counter
```

```
{'m': 1, 'i': 4, 's': 4, 'p': 2}
```

# Python Collections Module

```
>>> from collections import defaultdict
>>> counter = defaultdict(int)
>>> for letter in "mississippi":
... counter[letter] += 1
...
>>> counter
defaultdict(<class 'int'>, {'m': 1, 'i': 4, 's': 4, 'p': 2})
```

```
>>> from collections import Counter
>>> Counter("mississippi")
Counter({'i': 4, 's': 4, 'p': 2, 'm': 1})
```



# Python Collections Module

65

- Python's ChainMap groups multiple dictionaries and other mappings together to create a single object that works like a regular dictionary.
- It takes several mappings and makes them logically appear as one.
- ChainMap objects can have several dictionaries with either unique or repeated keys.
- ChainMap allows you to treat all your dictionaries as one. If you have unique keys across your dictionaries, you can access and update the keys as if you were working with a single dictionary.
- If you have repeated keys across your dictionaries, besides managing your dictionaries as one, you can also take advantage of the internal list of mappings to define some sort of access priority.

# Python Collections Module

Python

```
>>> from collections import ChainMap

>>> cmd_proxy = {} # The user doesn't provide a proxy
>>> local_proxy = {"proxy": "proxy.local.com"}
>>> global_proxy = {"proxy": "proxy.global.com"}

>>> config = ChainMap(cmd_proxy, local_proxy, global_proxy)
>>> config["proxy"]
'proxy.local.com'
```

ChainMap objects behave similarly to regular dict objects, they have a `.maps` public attribute that holds the internal list of mappings

Python

```
>>> from collections import ChainMap

>>> numbers = {"one": 1, "two": 2}
>>> letters = {"a": "A", "b": "B"}

>>> alpha_nums = ChainMap(numbers, letters)
>>> alpha_nums.maps
[{'one': 1, 'two': 2}, {'a': 'A', 'b': 'B'}]
```

# Python Collections Module

Python

```
>>> from collections import ChainMap

>>> dad = {"name": "John", "age": 35}
>>> mom = {"name": "Jane", "age": 31}
>>> family = ChainMap(mom, dad)
>>> family
ChainMap({'name': 'Jane', 'age': 31}, {'name': 'John', 'age': 35})

>>> son = {"name": "Mike", "age": 0}
>>> family = family.new_child(son)

>>> for person in family.maps:
... print(person)
...
{'name': 'Mike', 'age': 0}
{'name': 'Jane', 'age': 31}
{'name': 'John', 'age': 35}

>>> family.parents
ChainMap({'name': 'Jane', 'age': 31}, {'name': 'John', 'age': 35})
```

# Python Collections Module

A final feature to highlight in ChainMap is that mutating operations, such as updating keys, adding new keys, deleting existing keys, popping keys, and clearing the dictionary, act on the first mapping in the internal list of mappings

Python

```
>>> from collections import ChainMap

>>> numbers = {"one": 1, "two": 2}
>>> letters = {"a": "A", "b": "B"}

>>> alpha_nums = ChainMap(numbers, letters)
>>> alpha_nums
ChainMap({'one': 1, 'two': 2}, {'a': 'A', 'b': 'B'})

>>> # Add a new key-value pair
>>> alpha_nums["c"] = "C"
>>> alpha_nums
ChainMap({'one': 1, 'two': 2, 'c': 'C'}, {'a': 'A', 'b': 'B'})

>>> # Pop a key that exists in the first dictionary
>>> alpha_nums.pop("two")
2
>>> alpha_nums
ChainMap({'one': 1, 'c': 'C'}, {'a': 'A', 'b': 'B'})
```

# Python Regex

- A RegEx, or Regular Expression, is a sequence of characters that forms a search pattern.
- RegEx can be used to check if a string contains the specified search pattern.
- Python has a built-in package called re, which can be used to work with Regular Expressions.
- Import the **re** module:

```
import re
```

# Python Regex

- **The *match* Function**

- The `re.match` function returns a match object on success, `None` on failure. We use `group(num)` or `groups()` function of match object to get matched expression.

```
re.match(pattern, string, flags=0)
```

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match(r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
 print "matchObj.group() : ", matchObj.group()
 print "matchObj.group(1) : ", matchObj.group(1)
 print "matchObj.group(2) : ", matchObj.group(2)
else:
 print "No match!!"
```

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

# Python Regex

- **The *search* Function**
  - This function searches for first occurrence of RE pattern within string with optional flags.
  - The re.search function returns a match object on success, none on failure. We use group(num) or groups() function of match object to get matched expression.

```
re.search(pattern, string, flags=0)
```

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

searchObj = re.search(r'(.*) are (.*?) .*', line, re.M|re.I)

if searchObj:
 print "searchObj.group() : ", searchObj.group()
 print "searchObj.group(1) : ", searchObj.group(1)
 print "searchObj.group(2) : ", searchObj.group(2)
else:
 print "Nothing found!!"
```

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

# Python Regex

- **Matching Versus Searching**

- Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match(r'dogs', line, re.M|re.I)
if matchObj:
 print "match --> matchObj.group() : ", matchObj.group()
else:
 print "No match!!"

searchObj = re.search(r'dogs', line, re.M|re.I)
if searchObj:
 print "search --> searchObj.group() : ", searchObj.group()
else:
 print "Nothing found!!"
```

```
No match!!
search --> matchObj.group() : dogs
```



# Python Regex

- **Search and Replace**

- One of the most important re methods that use regular expressions is sub.
- This method replaces all occurrences of the RE *pattern* in *string* with *repl*, substituting all occurrences unless *max* provided. This method returns modified string.

```
#!/usr/bin/python
import re

phone = "2004-959-559 # This is Phone Number"

Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num
```

Phone Num : 2004-959-559

Phone Num : 2004959559

# Python Regex

- **The findall() Function**

- The findall() function returns a list containing all matches.
- The list contains the matches in the order they are found.
- If no matches are found, an empty list is returned.

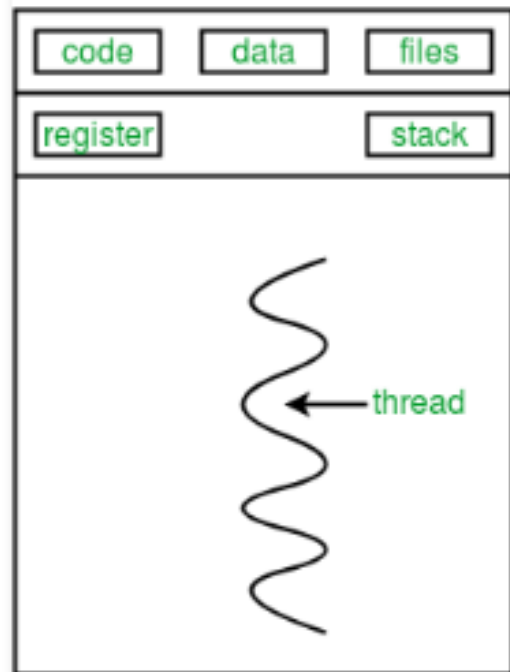
```
import re

str = "The rain in Spain"
x = re.findall("ai", str)
print(x)
```

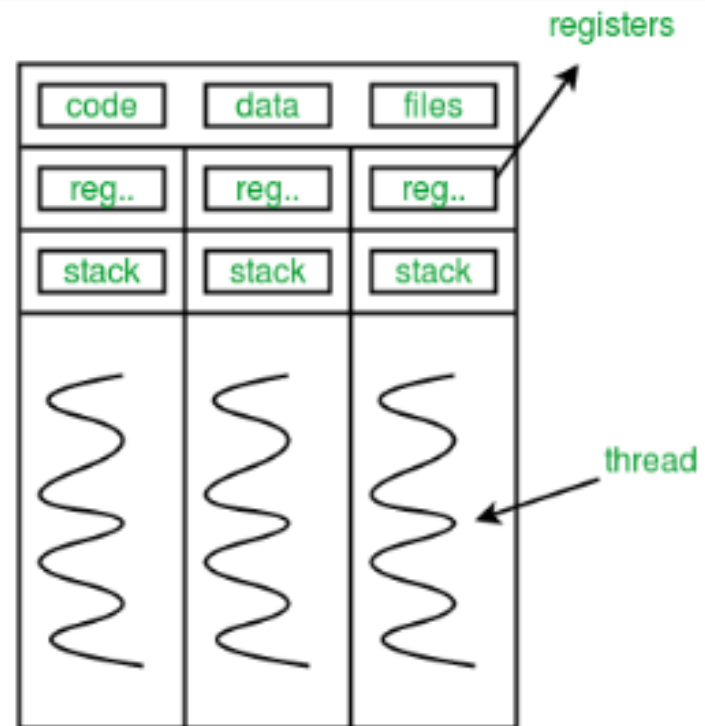
```
C:\Users\My Name>python demo_regex_findall.py
['ai', 'ai']
```

# Python Multithreading

- A thread is a separate flow of execution. This means that your program will have two things happening at once.
- I/O Bound tasks are suitable for threads.
- CPU bound tasks are not suitable for threads.
- Tasks that spend much of their time waiting for external events are generally good candidates for threading. Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all.
- Python standard library provides threading module built-in.
- `simple_thread = threading.Thread(target=thread_function, args=(1,), daemon=True)`
- Starting thread:
- To start a separate thread, you create a Thread instance and then tell it to `.start()`



single-threaded process



multithreaded process

# Python Multithreading

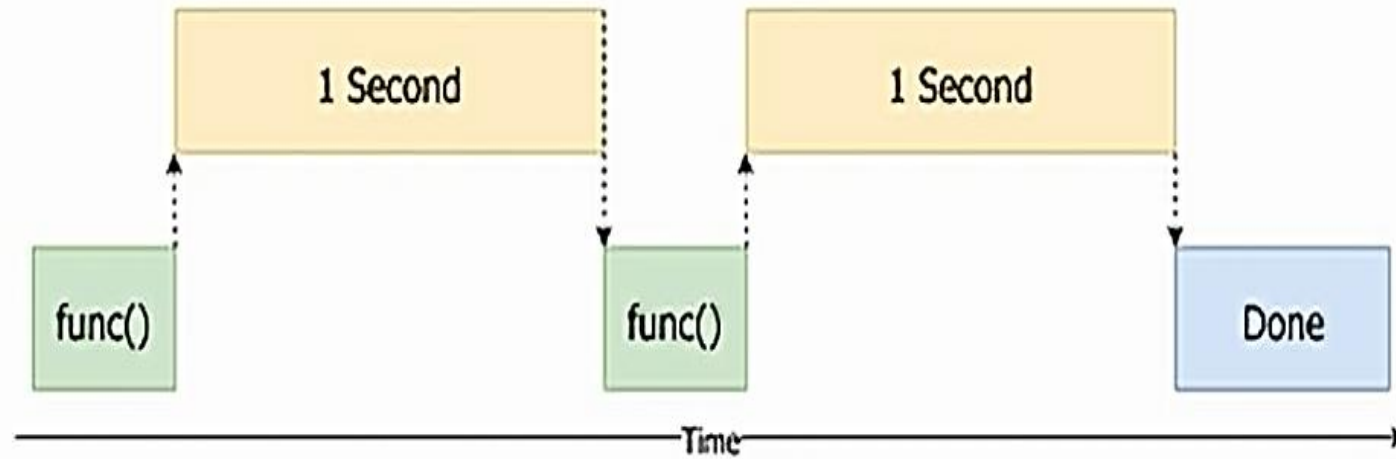
- The `<Thread>` class publishes the following methods.
- `run()`: It is the entry point function for any thread.
- `start()`: The `start()` method triggers a thread when `run` method is called.
- `join([time])`: The `join()` method enables a program to wait for threads to terminate.
- `isAlive()`: The `isAlive()` method verifies an active thread.
- `getName()`: The `getName()` method retrieves the name of a thread.
- `setName()`: The `setName()` method updates the name of a thread.

- Global interpreter Lock

The Python Global Interpreter Lock or GIL, in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

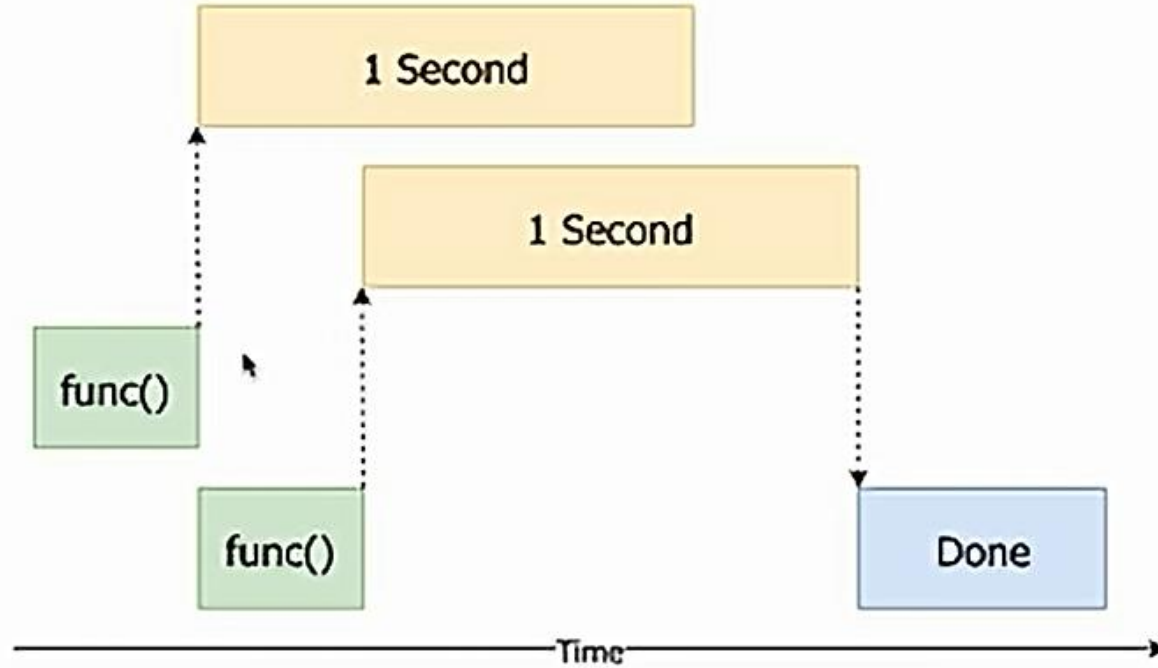
# Single Thread

79



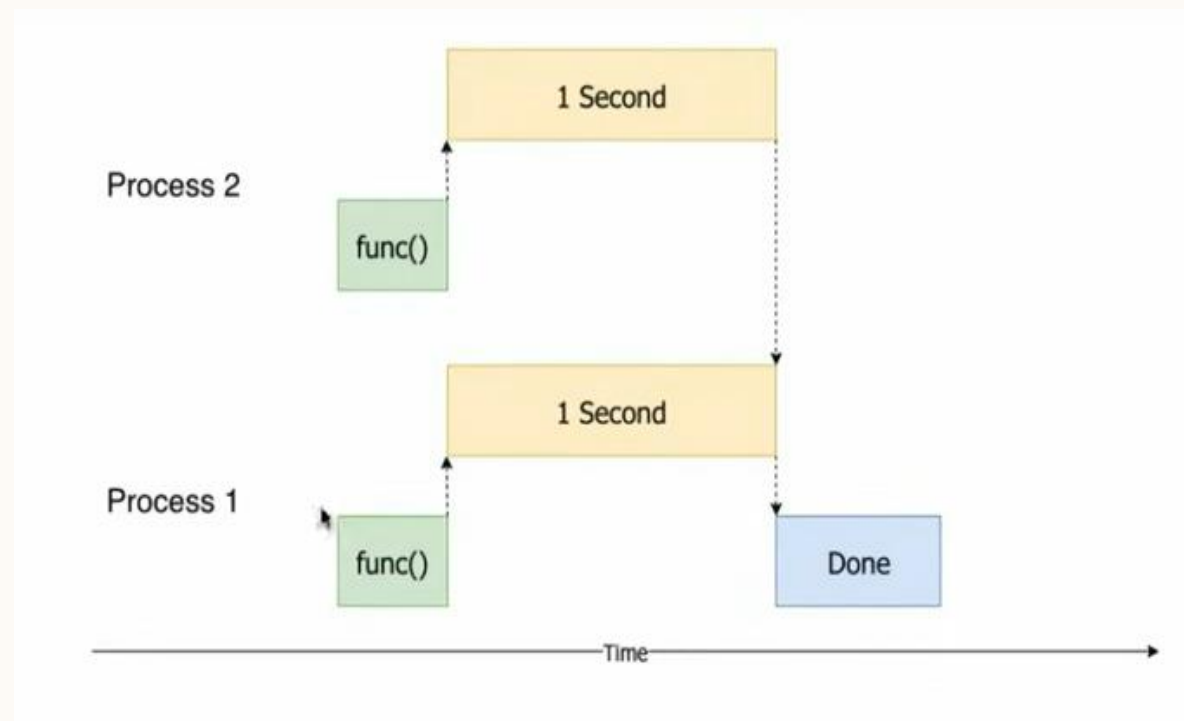
# Multi-threading

80





# Multiprocessing



# Data science Libs

**pandas**: for data manipulation and analysis

**numpy**: multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

**matplotlib**: plotting library for the Python programming language and its numerical mathematics extension NumPy

# Django Framework

## What is Django?



- It is an open source web application framework, written in Python
- Easier to build better web apps with less code

Follows MVT standards

# Django Framework

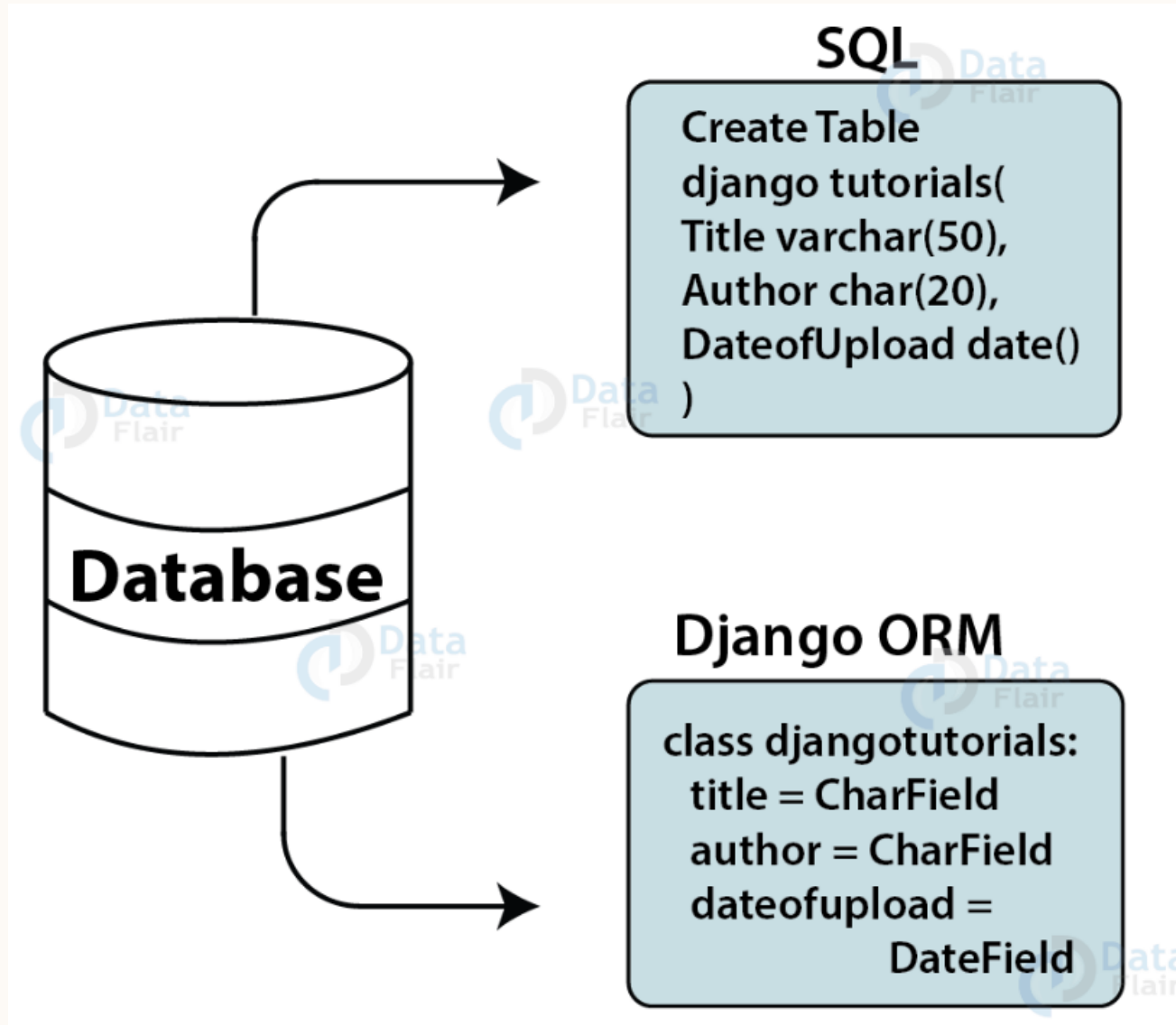
- Models (M)** – Django ORM  
Object relational Mapping
- Templates(T)** – Django template engine
- Views(V)** - Python Functions, Request in,  
Response out

# Django Framework

## Why Django?

- Administration Interface
- User Authentication
- Sessions
- Forms Handling
- Internationalization and Localization
- Templates
- Testing
- Supports Multiple Databases

# Django ORM



# Website with Django

*pip install Django*  
*django-admin startproject myWebsite*  
*cd myWebsite*  
*python manage.py runserver*

*Access 127.0.0.1:8000 in browser to see website*



# THANK YOU

Rafiq

[mohammadr5@hexaware.com](mailto:mohammadr5@hexaware.com)

[mohd.rfq@gmail.com](mailto:mohd.rfq@gmail.com)