PYTHON LANGUAGE TRAINING

Mohammad Rafiq

PYTHON PROGRAMMING

- 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. For developing web Applications
2. For developing database Applications
3. Network Programming
4. For developing games
5. For Data Analysis Applications
6. AI/ML – Tensorflow/pytorch

INSTALL PYTHON

Python installer download

- www.python.org

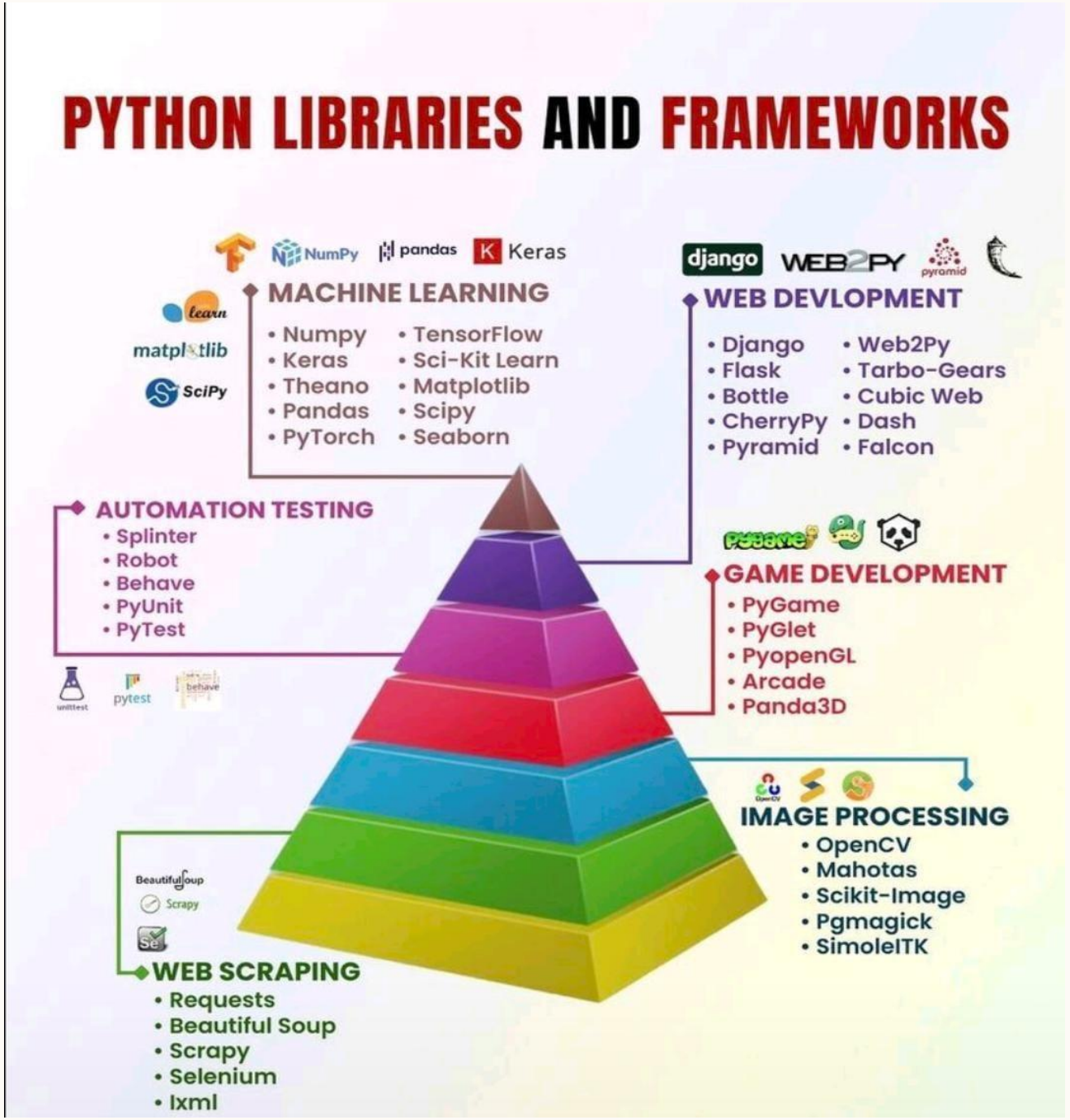
Python libraries

- 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

Static - Data Types are checked before execution.

Dynamic - Data Types are checked during execution.

PYTHON

Pros of Compiled Languages

- Private code.
- Faster execution.
- Fully optimized.

Cons of Compiled Languages

- No portability.
- Extra compilation step.

Pros of Interpreted Languages

- Portable.
- Easy debugging.

Cons of Interpreted Languages

- Requires interpreter.
- Slower.
- Public code.

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
```

Package Manager *pip*

- Install new libraries:
  - ***pip install <library_name>***
- ***pip list***
- ***pip freeze --local > requirements.txt***
- ***pip install -r requirements.txt***

VIARTUALENV

- python -m venv /path/to/new/virtual/environment
  Ex: python -m venv testenv
- Activate virtual environment:
  - .\testenv\Scripts\activate
- Deactivate virtual environment:
  - .\testenv\Scripts\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

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

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

Operators & Strings

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

Python Operators

+ (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 (O101 ^ 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

<= (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**.

Control Flow

- 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

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
pass	Does nothing	Placeholder for future code
continue	Skips the rest of the loop for the current iteration	When you want to skip a specific iteration
break	Terminates the loop	When you want to end the loop prematurely

Python Functions

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

Python Functions

*Definition: A function in Python is a **block of reusable code** designed to perform a specific task.*

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

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 - ***lambda***
- A ***map()*** function

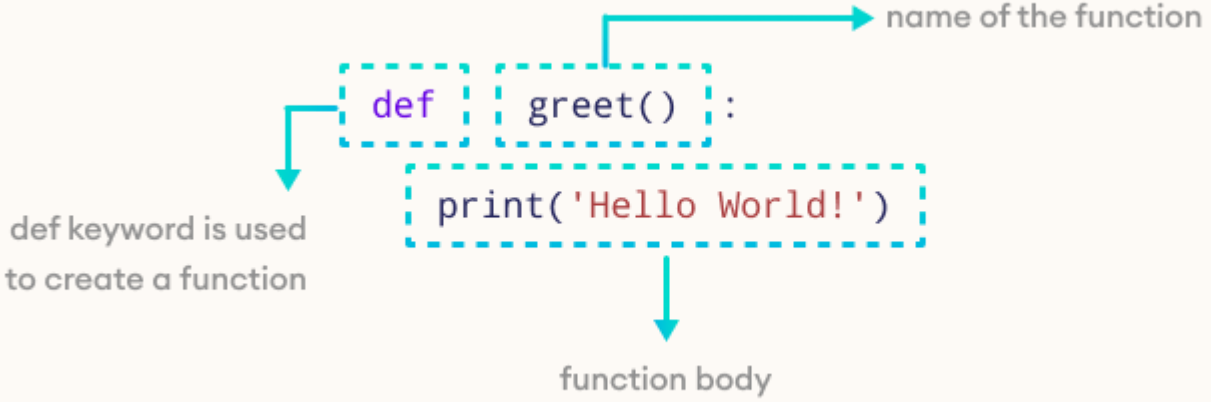
map(function, iterable[iterable1, iterable2,..., iterableN])

- A ***filter()*** function
- A ***reduce()*** function

User-defined Functions

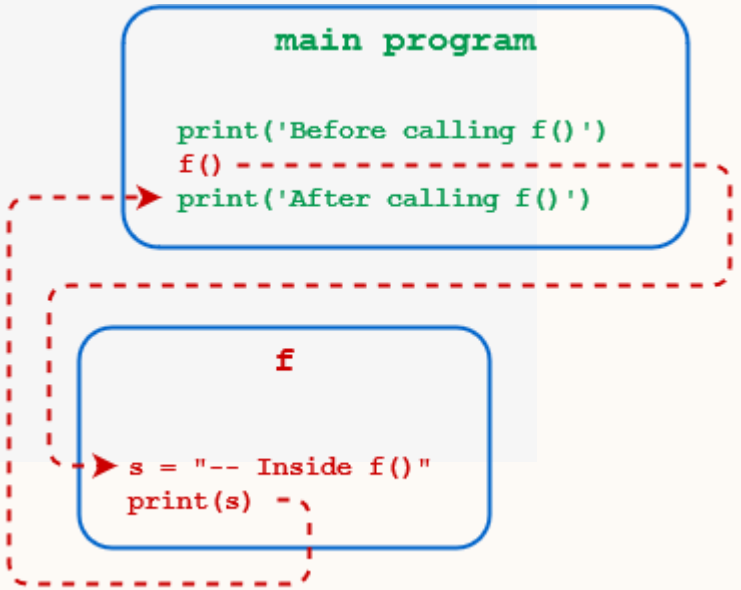

```
def <function_name>([<parameters>]):
    <statement(s)>
```

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

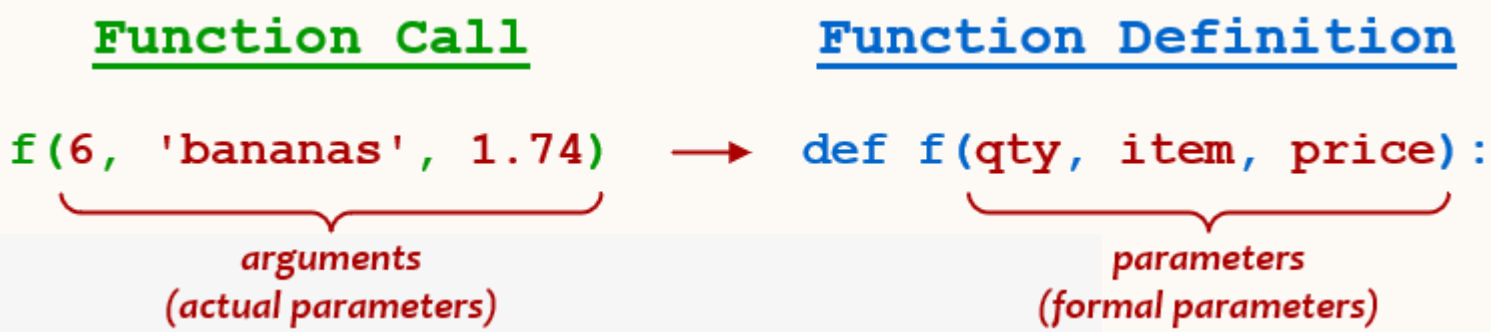


User-defined Functions

- Abstraction
- 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

Recursion

- **Base Case:**

  The function stops calling itself when n

  n is 0 or 1. The factorial of 0 or 1 is defined as 1.

- **Recursive Case: The function calls itself with n – 1 until it reaches the base case. Each call multiplies the result of the recursive call by n.**

Exceptions Handling, File Handling

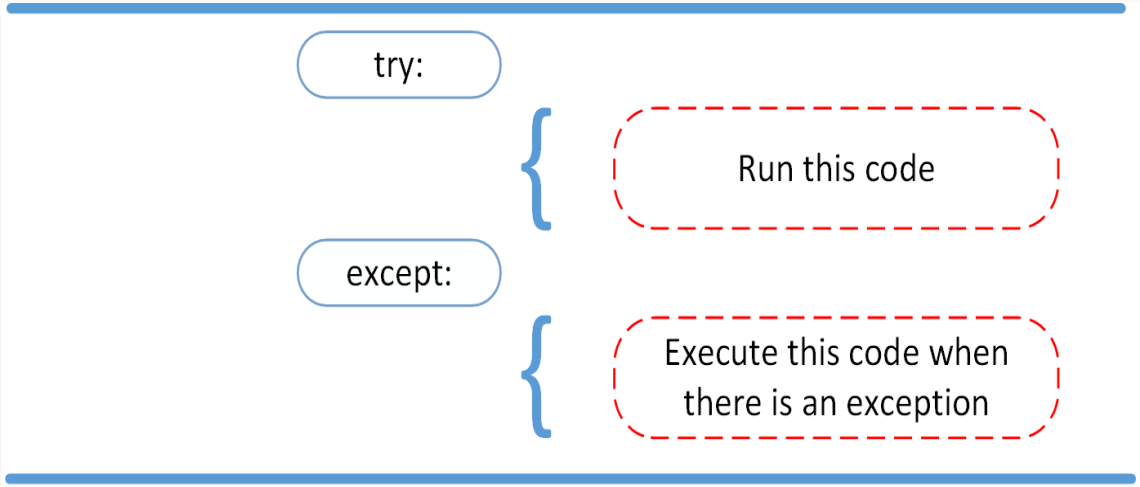
- Exception Handling
- File Handling

Built-in Exceptions

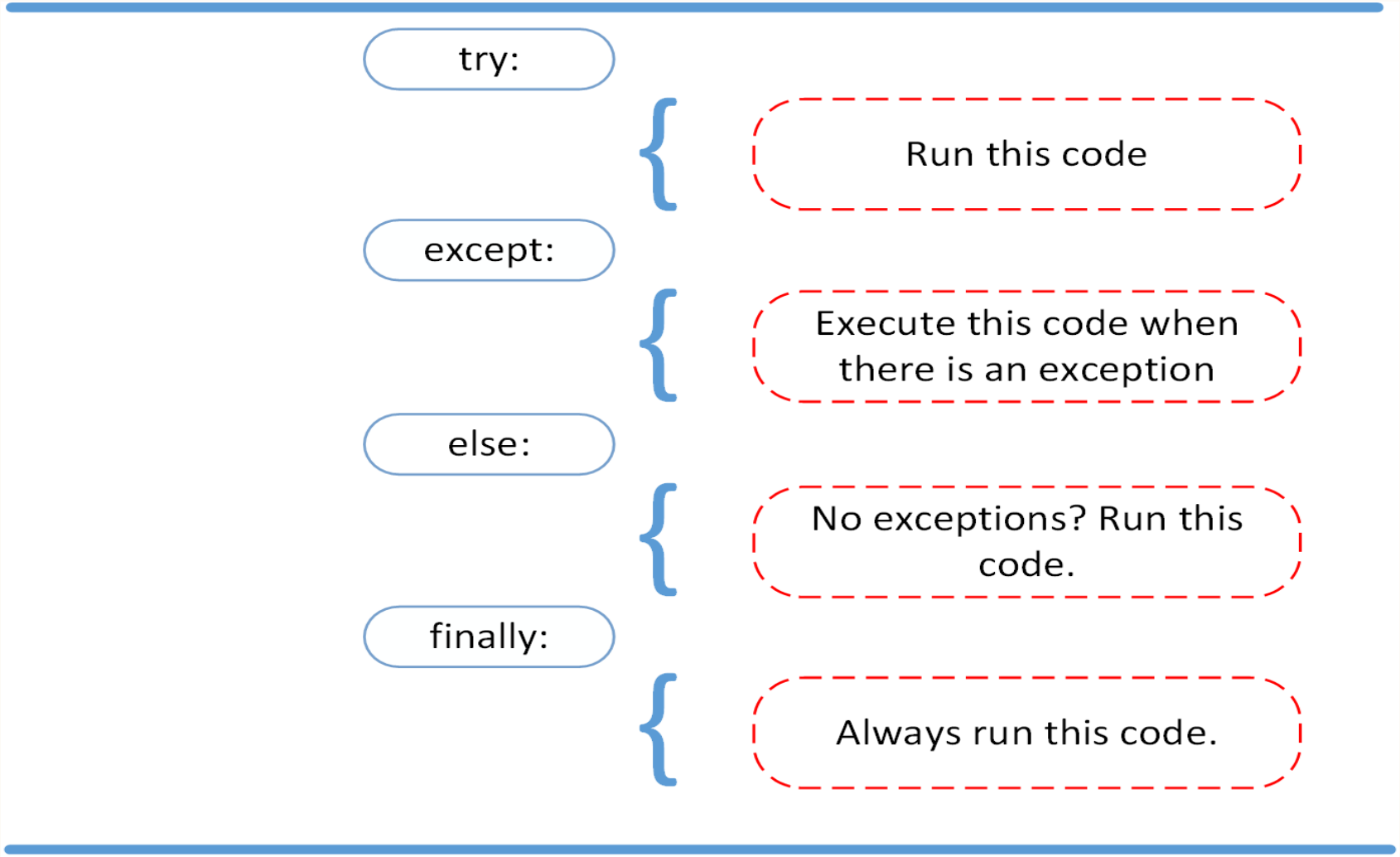
- *SyntaxError*
- *ModuleNotFoundError*
- *ValueError*
- *KeyError*
- *TypeError*
- *NameError*
- *KeyboardInterrupt*
- *ZeroDivisionError*

Exceptions Handling

- Exception handling with try
- Handling Multiple Exceptions
- Built-in Exceptions
- Writing your own Exception



Exceptions Handling



File Handling

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

COMPREHENSIONS

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

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

DECORATORS

Definition: *A function that returns another function.*

- A decorator is a function that modifies the behavior of another function.
- It is used to add functionality to an existing function without modifying its structure.

How to apply:

Use the *@decorator_name* syntax before the function definition.

Usage Examples:

Logging: Automatically log function calls.

Access Control: Enforce permissions or authentication before running a function.

Memoization: Cache results to optimize performance for expensive functions.

DECORATORS

Sample Code:

```
def decorator_name(func):
    def wrapper(*args, **kwargs):
        # Additional code to modify behavior
        return func(*args, **kwargs)
    return wrapper
```

decorator_name: The function name of the decorator.

func: The function to be decorated.

wrapper: The inner function that wraps around the original function to modify its behavior.

```
@decorator_name
def original_function():
    pass
```

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()
or
breakpoint()
```

PYTHON DEBUGGER

Key commands:

- ***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*
- ***continue*** *or* ***c*** *- Resume execution until the next breakpoint*
- ***quit*** *or* ***q*** *- Exit the debugger and stop the program.*
- ***breakpoint*** *or* ***b*** *- Set breakpoints in the code*

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***

Collections Module

deque A sequence-like collection that supports efficient addition and removal of items from either end of the sequence

defaultdict A dictionary subclass for constructing default values for missing keys and automatically adding them to the dictionary

namedtuple() 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

OrderedDict A dictionary subclass that keeps the key-value pairs ordered according to when the keys are inserted

Counter A dictionary subclass that supports convenient counting of unique items in a sequence or iterable

ChainMap A dictionary-like class that allows treating a number of mappings as a single dictionary object

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
```

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))

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'
```

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)
```

Collections Module

Method	Description
`.clear()`	Remove all the elements from a deque
`.copy()`	Create a shallow copy of a deque
`.count(x)`	Count the number of deque elements equal to `x`
`.remove(value)`	Remove the first occurrence of `value`

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

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})
```

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']
```

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
```

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}
```

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})
```

Collections Module

- 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.

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'}]
```

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})
```

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'})
```

datetime

- Today = date.today()
- Timestamp = date.fromtimestamp(1000000000)
- Local_datetime = datetime.now()
- Timezone_Toronto = pytz.timezone('America/Toronto’)
- Local_datetime_Toronto = datetime.now(Timezone_Toronto)

sleep function

```
import time

while True:
    localtime = time.localtime()
    result = time.strftime("%I:%M:%S %p", localtime)
    print(result)
    time.sleep(1)
```

Advanced Python Concepts

- Regular expressions
- Database connection
  - Postgres db connection
  - SELECT, INSERT, DELETE operations
  - Handling Errors

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

match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string

```
#!/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']
```

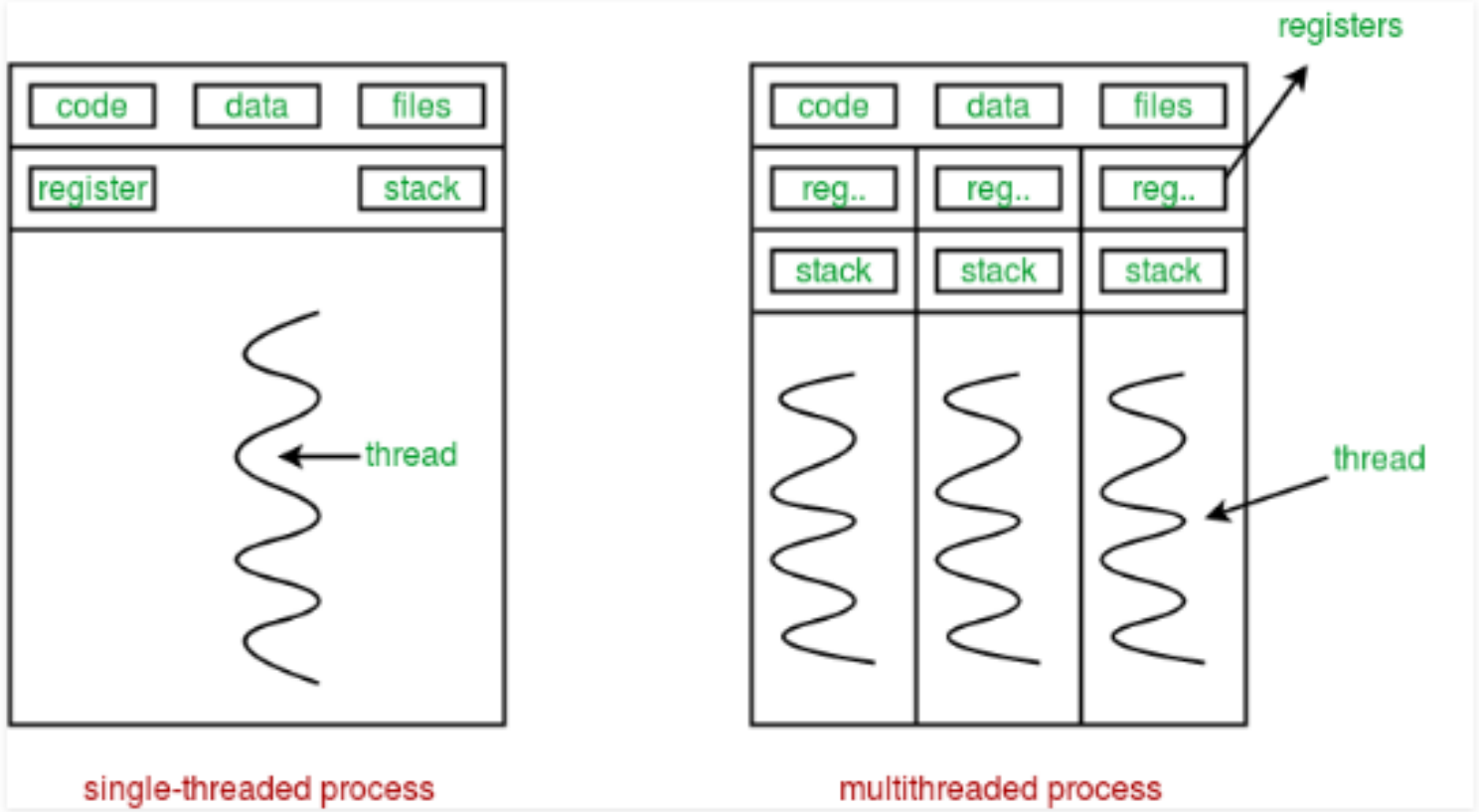
Advanced Python Concepts

- Multi-threading, GIL
- Object Oriented Programming

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()

Python Multithreading

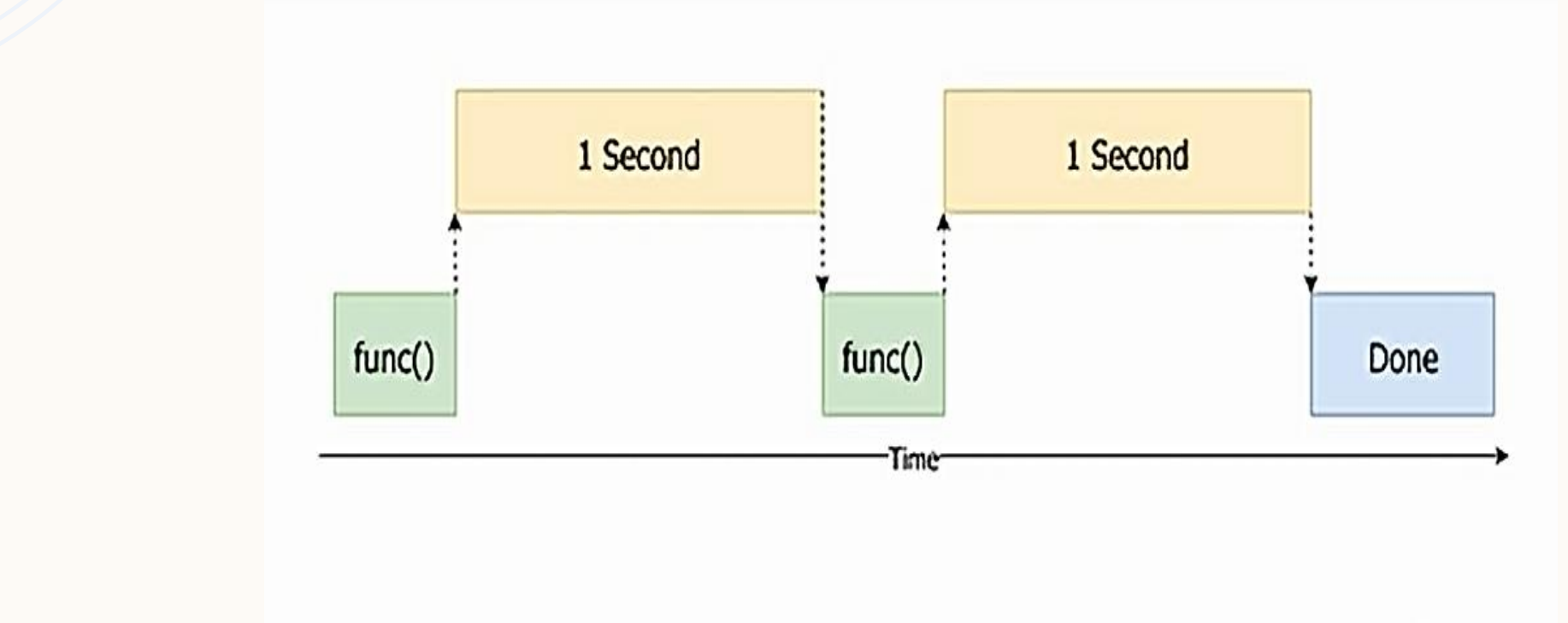


Python Multithreading

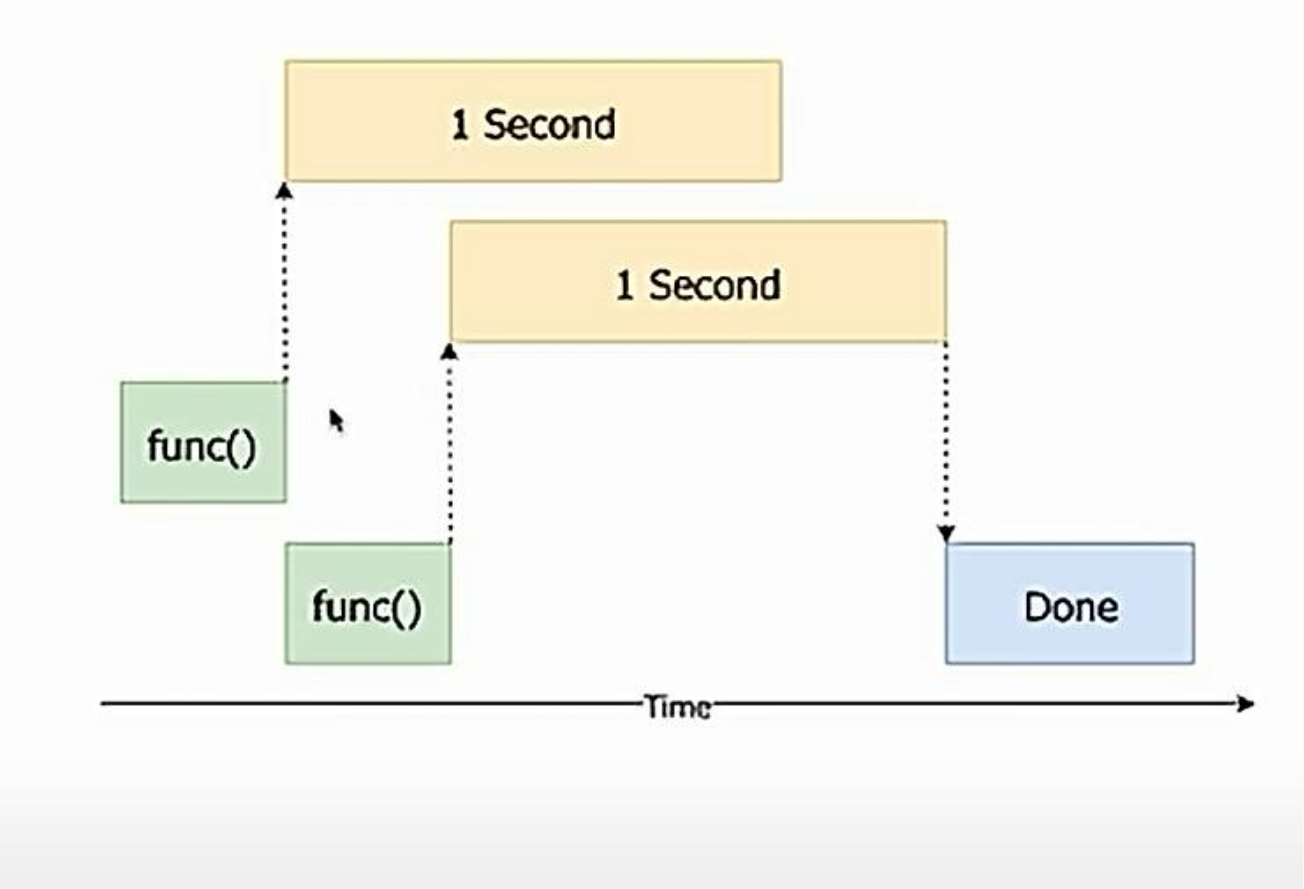
The *Thread* class has following methods.

- *run():* It is the entry point function for any thread.
- *start()*: Triggers a thread when run method is called.
- *join([time])*: Enables a program to wait for threads to terminate.
- *isAlive()*: Verifies an active thread.
- *getName()*: Retrieves the name of a thread.
- *setName()*: Updates the name of a thread.

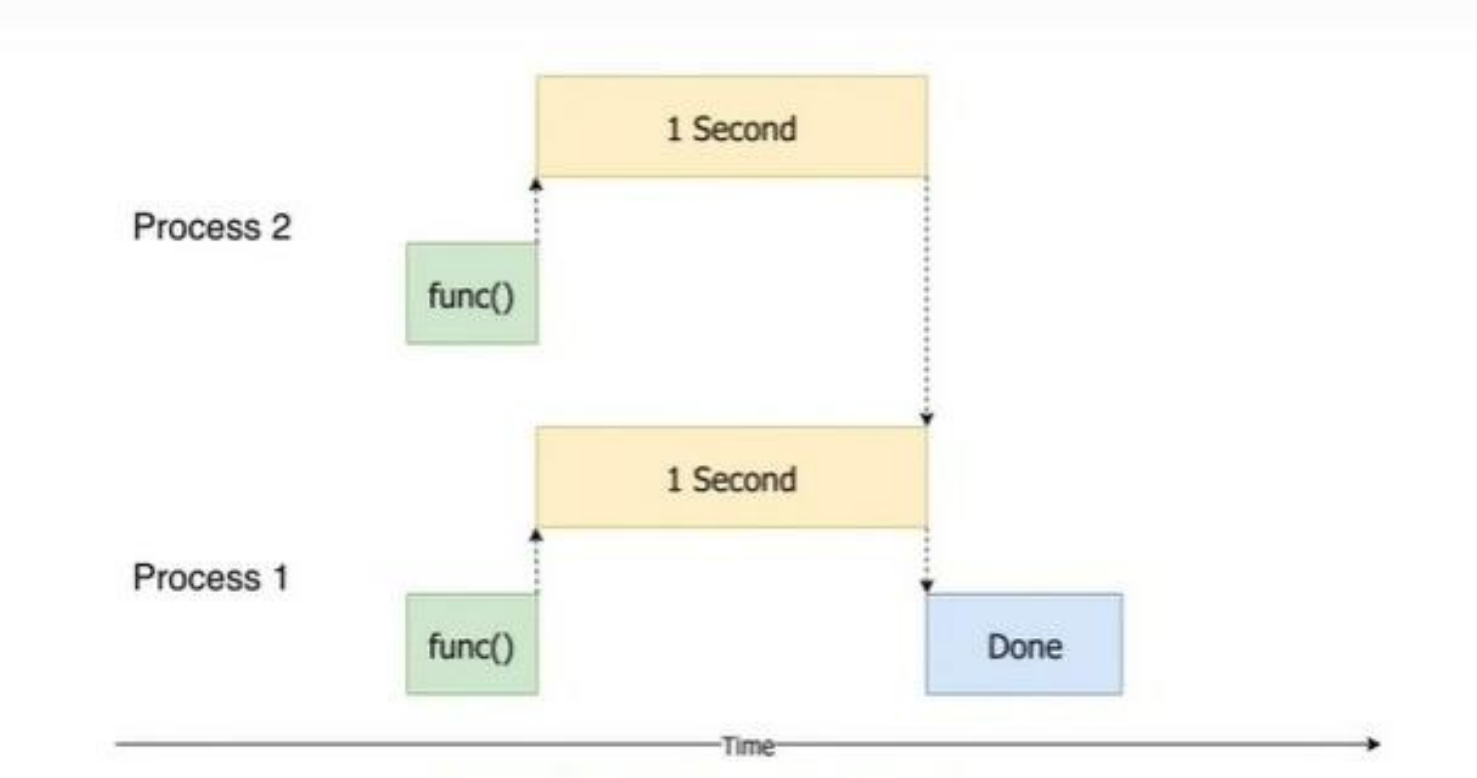
Single Thread



Multithreading



Multiprocessing



GIL

Global Interpreter Lock

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

Object-oriented Programming

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

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

Object-oriented Programming

Types of methods:

- **Instance methods**:
- **Class method**: A class method receives the class as an implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
    @classmethod
    def f(cls, arg1, arg2): ...
```

- **Static method:** A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class S:
    @staticmethod
    def f(arg1, arg2, argN): ...
```

Object-oriented Programming

Types of methods:

Type of Method	Purpose	Access	Parameters
Instance Method	Operates on instance data and attributes.	Through an instance	`self` (instance)
Class Method	Operates on class-level data.	Through the class	`cls` (class)
Static Method	Utility method that doesn't operate on instance or class data.	Through the class or instance	None
Property Method	Defines behavior of an attribute with getter/setter logic.	Through an instance	`self` (instance)
Abstract Method	Defines a method signature that must be implemented by subclasses.	N/A (abstract)	N/A (abstract)

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()
```

PROJECTS

- Project - 1: Python for Data Science
- Project - 2: Website development with Django framework
- Project - 3: City Weather with API
- Project - 4: Captcha generator with Tkinter
- Project - 5: QR Code reader/generator

PROJECT - 1
PYTHON FOR DATA SCIENCE

JUPYTER NOTEBOOKS

Installation:

- Install Anaconda distribution for your platform, it includes jupyter notebooks and other data science tools

Launch Jupyter Notebook:

- Open cmd or terminal and run below command
- ***jupyter notebook***
- Start jupyter notebooks with below command
- It will open in new tab in default web browser with url http://localhost:8888/

About Jupyter notebook:

- Web based notebook
- Combined text and code into one notebook
- Save notebooks as .ipynb files

JUPYTER NOTEBOOKS

Writing and Running Code:

- Type code in a Code Cell
- Execute code by pressing Shift + Enter.

Markdown Cells:

- Switch a cell to Markdown by selecting "Cell" > "Cell Type" > "Markdown" from the menu.
- Use Markdown syntax to format text (e.g., headers, lists, links).

Shortcuts:

a - add a cell above
b - add a cell below
m - convert cell to markdown
y - convert cell to code
dd - delete cell

Python Libraries For Data Science

Popular python libraries:

- ***pandas***
- ***numpy***
- ***matplotlib***
- ***Seaborn***
- ***scikit-learn***

Data Handling With *pandas*

Pandas comes from the econometrics term 'panel data' describing data sets that include observations over multiple time periods.

import pandas as pd

Basic data structures:

1. **Series**: a one-dimensional labeled array holding data of any type, such as integers, strings, Python objects etc.

   Creating a series:

   s = pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])

   Accessing elements:

   print(s['a']) # Access by index label

   print(s[0]) # Access by integer position

2. **DataFrame**: a two-dimensional data structure that holds data like a two-dimension array or a table with rows and columns.

   Creating a DataFrame:

   df = pd.DataFrame({ 'A': [1, 2, 3], 'B': [4, 5, 6] }, index=['x', 'y', 'z'])

   Accessing elements:

   print(df['A']) # Access column A

   print(df.loc['x']) # Access row with label 'x'

   print(df.iloc[0]) # Access row with integer position 0

Data Manipulation using *pandas*

Indexing and Selecting Data:

Label-based Indexing with .loc[]

df.loc['x'] # Selects the row with label 'x'

df.loc[:, 'A'] # Selects all rows in column ‘A’

Filtering Data:

df[df['A'] > 1] # Filters rows where column 'A' values are greater than 1

Adding and Dropping Columns/Rows:

Adding Columns:

df['C'] = [7, 8, 9] # Adds a new column 'C’

Dropping Columns:

df.drop('C', axis=1, inplace=True) # Drops column 'C'

Adding Rows:

df.loc['w'] = [10, 11] # Adds a new row with label 'w'

Dropping Rows :

df.drop('w', axis=0, inplace=True) # Drops row with label ‘w’

Handling missing data:

Detecting Missing Data:

df.isna() # Returns a DataFrame of the same shape with True for NaNs

Filling Missing Data:

df.fillna(0) # Replaces NaN with 0

Dropping Missing Data:

df.dropna() # Drops any rows with NaN values

Numerical computation with *numpy*

NumPy (Numerical Python) provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. Widely used in data science, machine learning.

```
import numpy as np
```

Creating Arrays:

```
arr = np.array([1, 2, 3, 4, 5]) # From a List
arr = np.array((1, 2, 3, 4, 5)) # From a Tuple
```

Creating Arrays with Default Values:

```
zeros = np.zeros((3, 4))  # 3x4 array of zeros
ones = np.ones((2, 3))    # 2x3 array of ones
full_array = np.full((2, 2), 7)  # 2x2 array filled with 7
```

Creating Arrays with a Range of Values:

```
range_array = np.arange(10)  # Array of values from 0 to 9
```

Array Attributes:

```
print(arr.shape)  # Returns the shape of the array
print(arr.size)  # Returns the number of elements
print(arr.ndim)  # Returns the number of dimensions
print(arr.dtype)  # Returns the data type of the array
```

Numerical computation with *numpy*

Reshaping and Resizing:

Reshaping Arrays:

```
reshaped = arr.reshape(3, 1)  # Reshape to 3x1
arr.T  # to transpose array
```

Flattening Arrays:

```
flattened = matrix.flatten()  # Flatten into a 1D array
```

Appending and Inserting:

```
appended = np.append(arr, [6, 7])  # Append values
```

Linear Algebra:

Matrix Multiplication:

```
result = np.dot(matrix, arr)  # Dot product
```

Matrix Inversion:

```
inv_matrix = np.linalg.inv(matrix)  # Inverse of a matrix
```

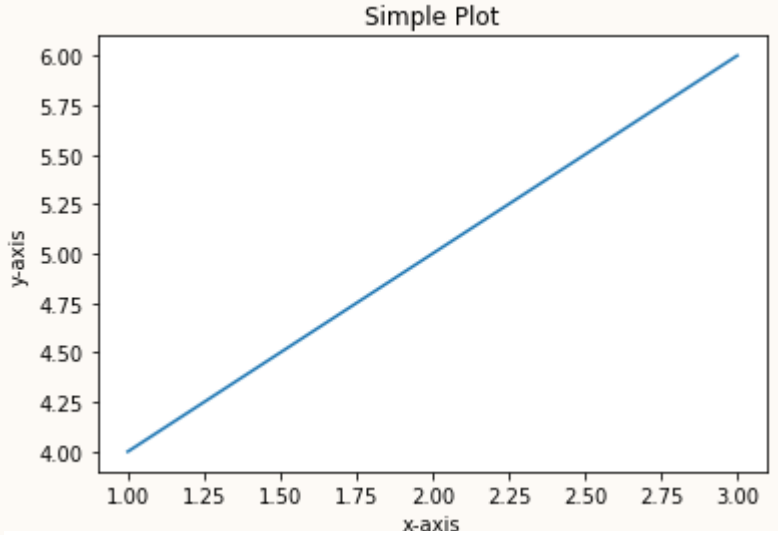
Data Visualization

Data visualization with ***matplotlib*** and ***seaborn***

```
import matplotlib.pyplot as plt
import seaborn as sns
```

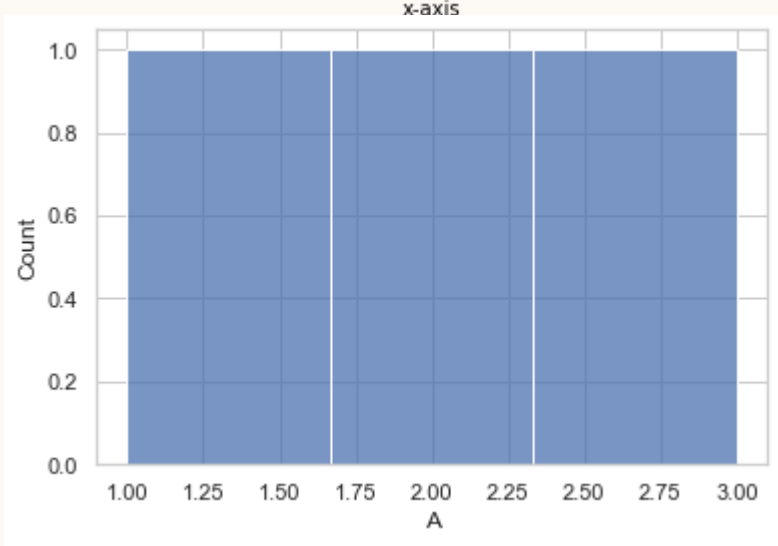
Matplotlib:

```
plt.plot([1, 2, 3], [4, 5, 6])
plt.xlabel('x-axis')
plt.ylabel('y-axis')
plt.title('Simple Plot')
plt.show()
```



Seaborn:

```
sns.set(style='whitegrid')
sns.histplot(df['A'])
plt.show()
```



Machine Learning with *Scikit-Learn*

Loading Data:

```
from sklearn.datasets import load_iris
data = load_iris()
X = data.data
y = data.target
```

Training a Model:

```
 from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
model = LogisticRegression()
model.fit(X_train, y_train)
```

Evaluating a Model:

```
from sklearn.metrics import accuracy_score
y_pred = model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
```

PROJECT - 2
WEBSITE DEVELOPMENT WITH DJANGO FRAMEWORK

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

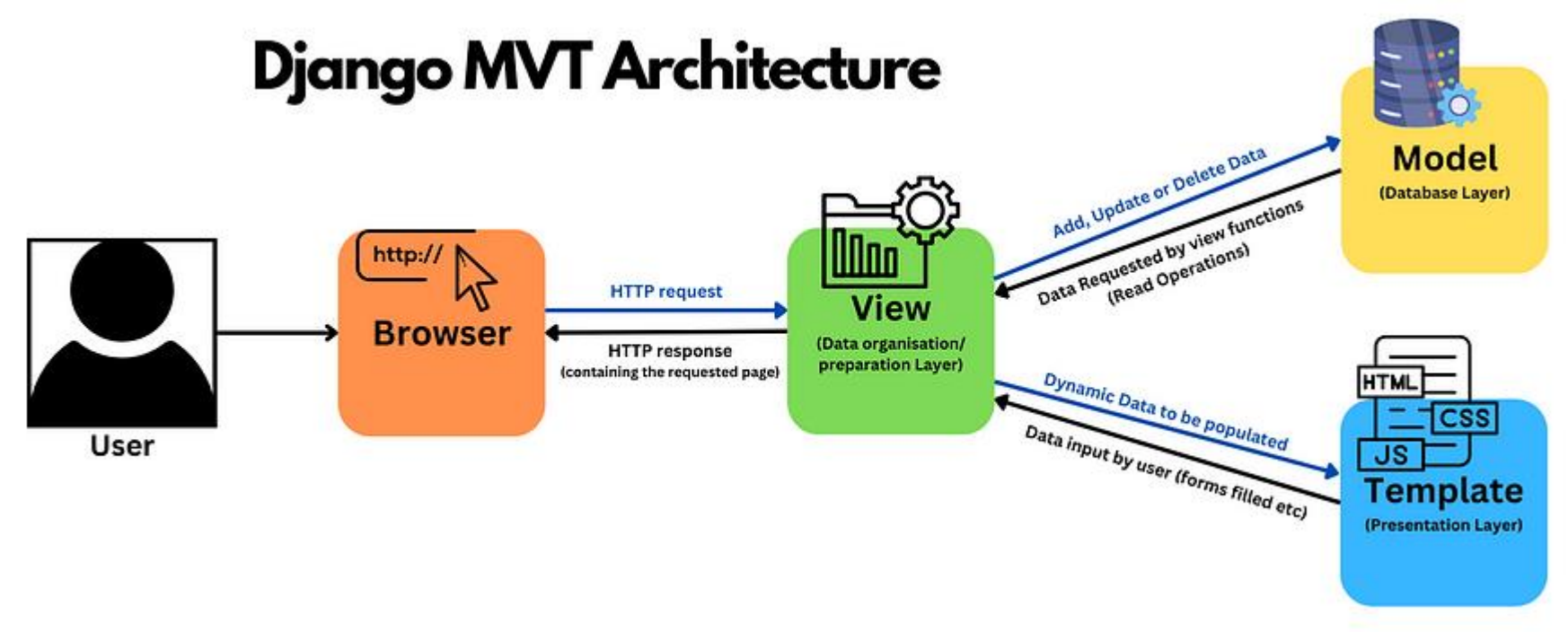
Django Framework

Models (M) – Django ORM
Object relational Mapping

Templates(T) – Django template engine

Views(V) - Python Functions, Request in,
Response out

Django Framework

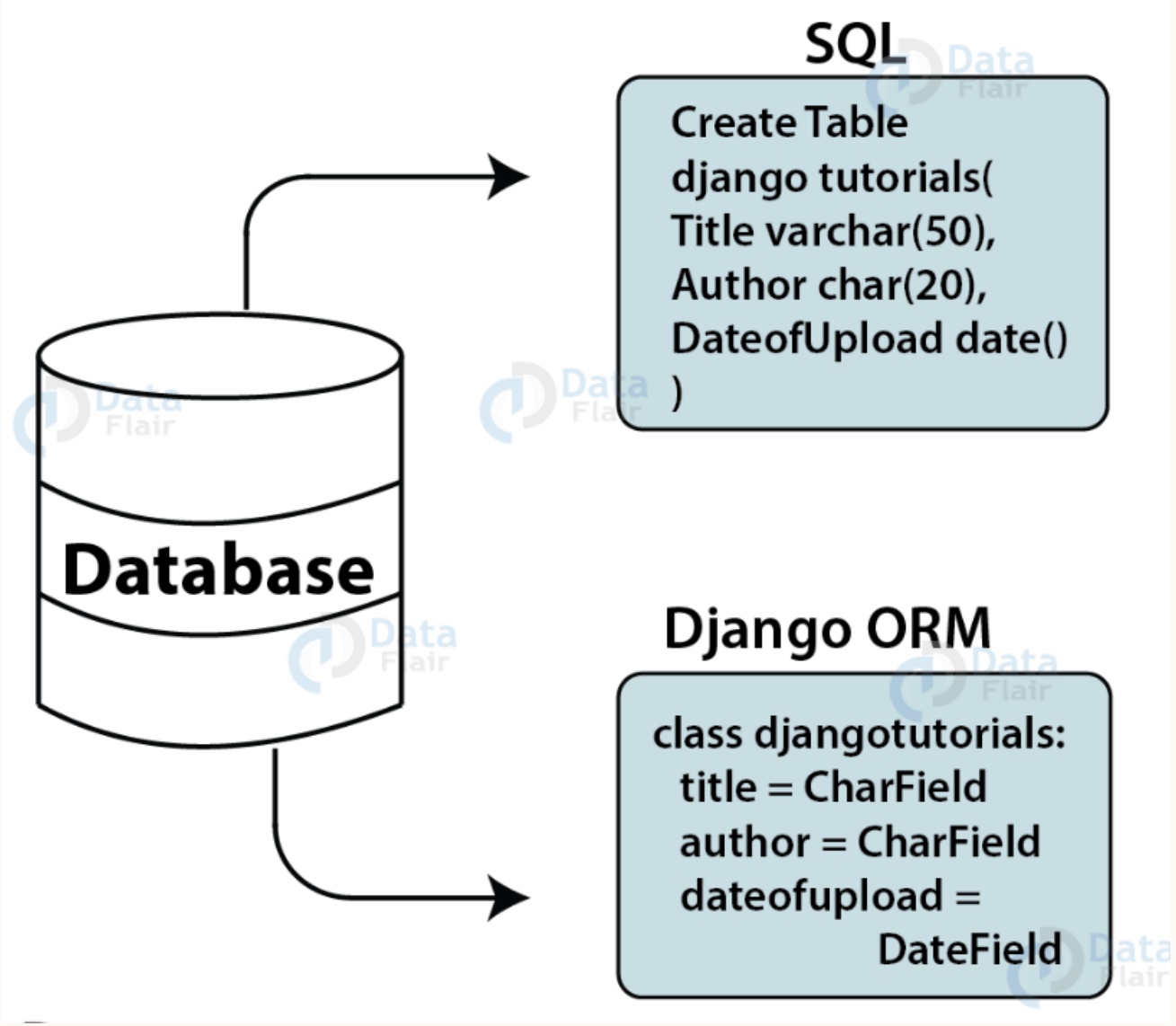


Django Framework

Why Django?

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

Django ORM


SQL
Create Table
django tutorials(
Title varchar(50),
Author char(20),
DateofUpload date()
)
Database
Django ORM
class djangotutorials:
title = CharField
author = CharField
dateofupload =
DateField

PROJECT - 3
CITY WEATHER WITH API

API access with *requests*

requests is a HTTP library designed for making HTTP requests more accessible

import requests

Reading URLs:

We can read any given url with get method

```
response = requests.get('https://api.example.com/data')
print(response.status_code)  # HTTP status code
print(response.text)         # Response body as a string
```

Refer to **fetch_current_weather.py** for working example code

PROJECT - 4
QR CODE READER/GENERATOR

QR Code with *qrcode*

qrcode library helps to create QR code of any given text

import qucode

```
# Create a QR code object
qr = qrcode.QRCode(    version=1,  # controls the size of the QR Code
box_size=10,  # controls how many pixels each “box” of the QR code is
    border=4,  # controls how many boxes thick the border should be)
# Add data to the QR code object
qr.add_data(‘Pay as you GO!’) # text you want create QR code for
qr.make(fit=True)
# Create an image from the QR Code instance
img = qr.make_image(fill='black', back_color='white')
# Save the image
img.save(' myqrcode.jpg’)
```

Refer to **qr_generator.py** for working example code

QR Code reader

We can read text from any QR code with ***pyzbar*** and ***PIL*** libraries

```
from pyzbar.pyzbar import decode
from PIL import Image
img = Image.open(r"C:\path\to\qrcode\image\myqrcode.jpg")
result = decode(img)
print(result)
```

Refer to **qr_reader.py** for working example code

Test Automation Framework - pytest

What is Pytest?
Pytest is a testing framework for Python that makes it easy to write simple and scalable test cases.

Key features:

- **Simple syntax.**
- **Supports fixtures for setup/teardown.**
- **Parameterized testing.**
- **Supports plugins (e.g., pytest-html, pytest-mock).**
- **Integrates with CI/CD pipelines.**

Test Automation Framework - pytest

What is Pytest?
Pytest is a testing framework for Python that makes it easy to write simple and scalable test cases.

Key features:

- **Simple syntax.**
- **Supports fixtures for setup/teardown.**
- **Parameterized testing.**
- **Supports plugins (e.g., pytest-html, pytest-mock).**
- **Integrates with CI/CD pipelines.**

Validating json and xml

Python-based ETL testing example to validate JSON and XML data structures using jsonschema for JSON and lxml for XML.

Popular libraries:
JSON Validation → ***jsonschema***

XML Validation → ***xml.etree.ElementTree***
→ ***lxml***

THANK YOU

Rafiq
mohammadr5@hexaware.com
Mohd.rfq@gmail.com