python

Agenda

01 Python Introduction

02 Data Structures

03 Control Structures

04 Functions

05 Advanced Functions

06 File Handling & Exception Handling

07 Raise Error Built in Errors

08 Object Oriented Programming

09 Multithreading and Multiprocessing

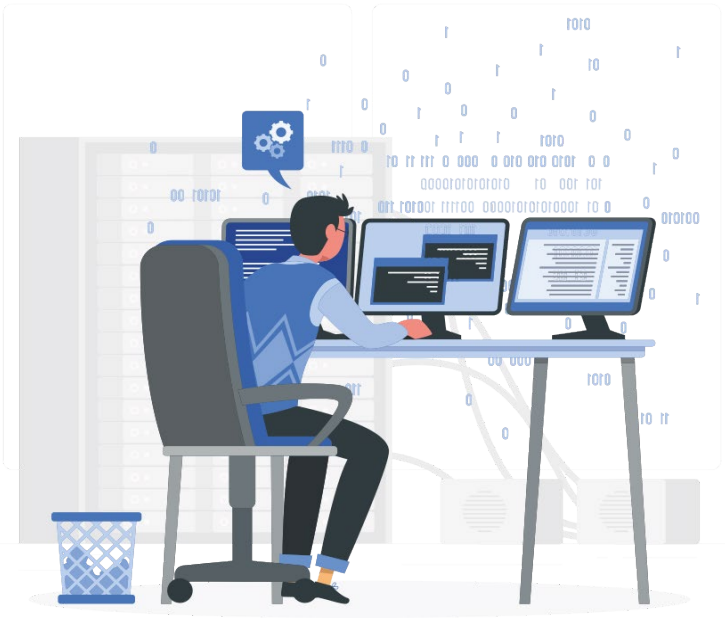
10 Flask

Python Introduction

Python Introduction

What is Python?

Python is an object-oriented, high-level programming language with integrated dynamic semantics primarily for web and app development



The World's #1 Programming Language

Python Introduction

keeping it smile

Simplicity and Readability

```
Class helloWorld {
      Public static void main( string []arge ) {
    System.out.println( “Hello World!” );
     }
}
```

```
print(“Hello World!”)
```

Python Introduction

Convenience

Windows 10

Mac OS

Use your favorite Operating System

Python Introduction

Cross-Language Operations

“The #1 language for a reason!”

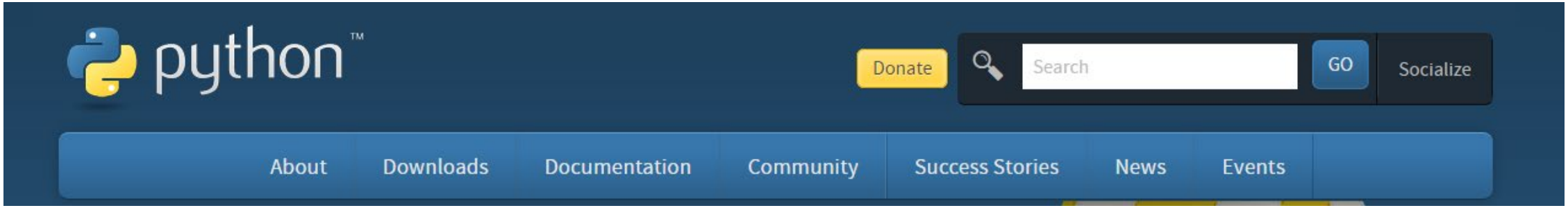
Python Introduction

Installing Python on Windows

Step 1:

Go to python.org and head to the downloads page

Download Python



Python Introduction

Installing Python on Windows

Step 2:

Click on “Download Python 3.10.4”

Python Introduction

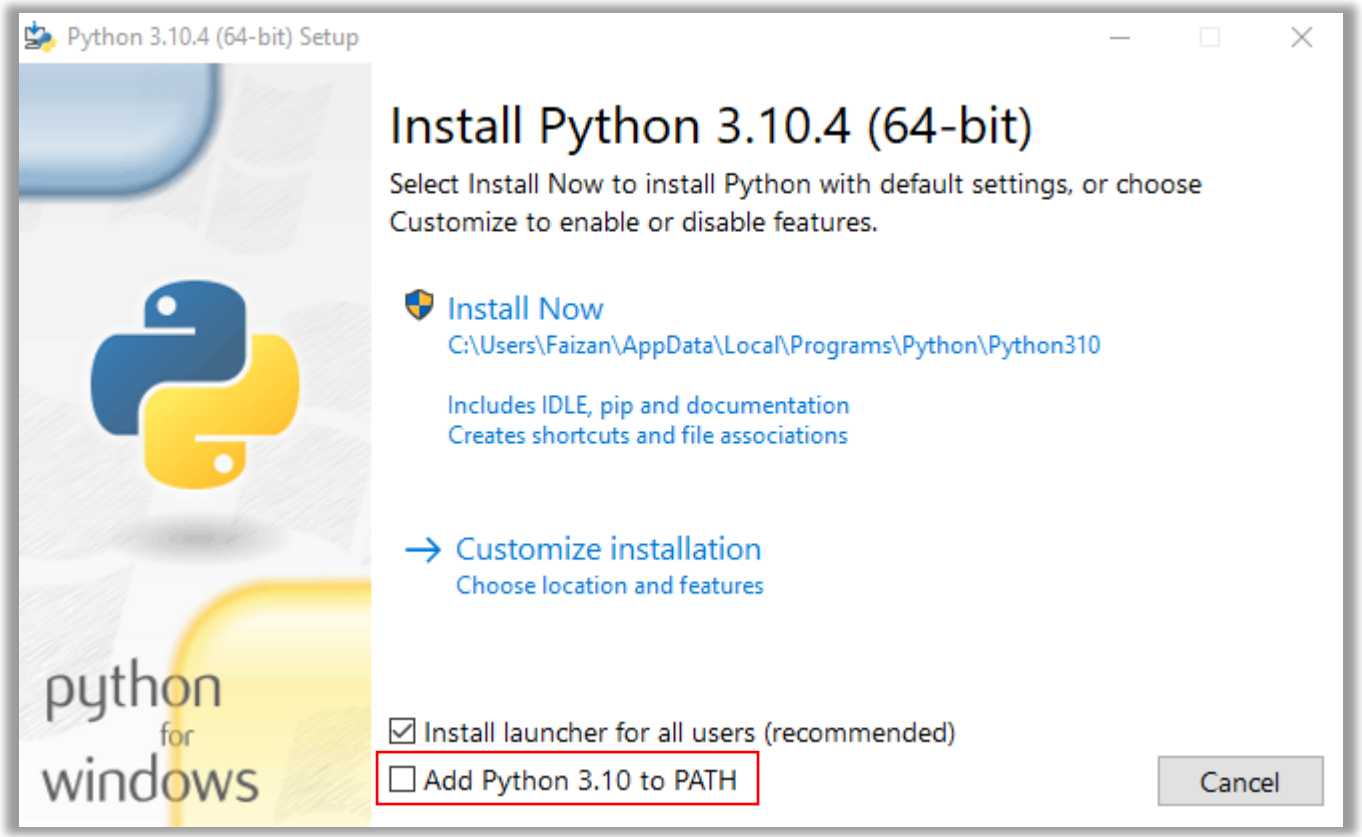
Installing Python on Windows

Step 3: **Open the installer and you should be presented with this screen!**

Python Introduction

Installing Python on Windows

Step 4: **Check the last option to add Python 3.8 to the PATH and install**

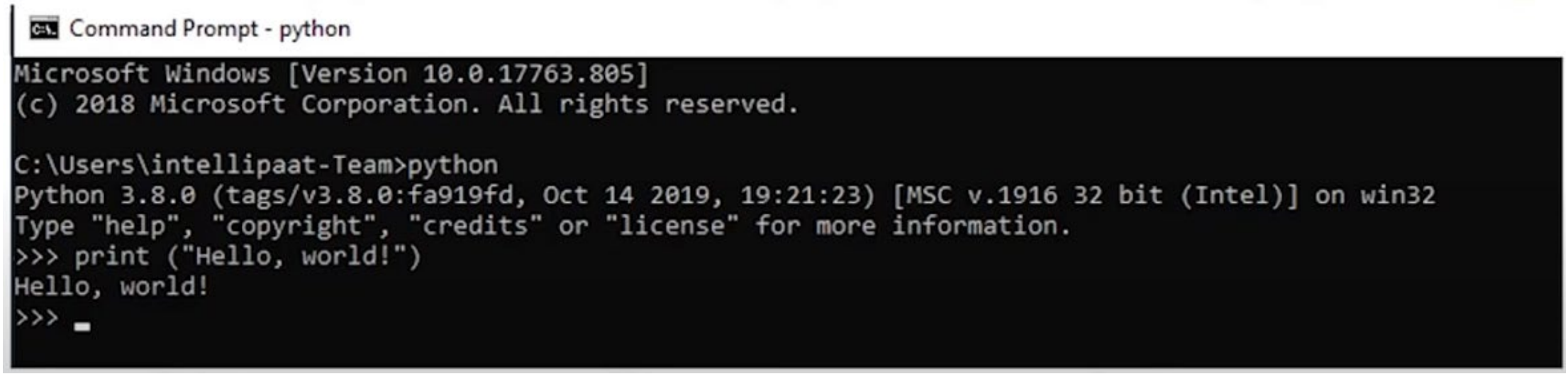


Python Introduction

Installing Python on Windows

Step 5:

Verify your installation

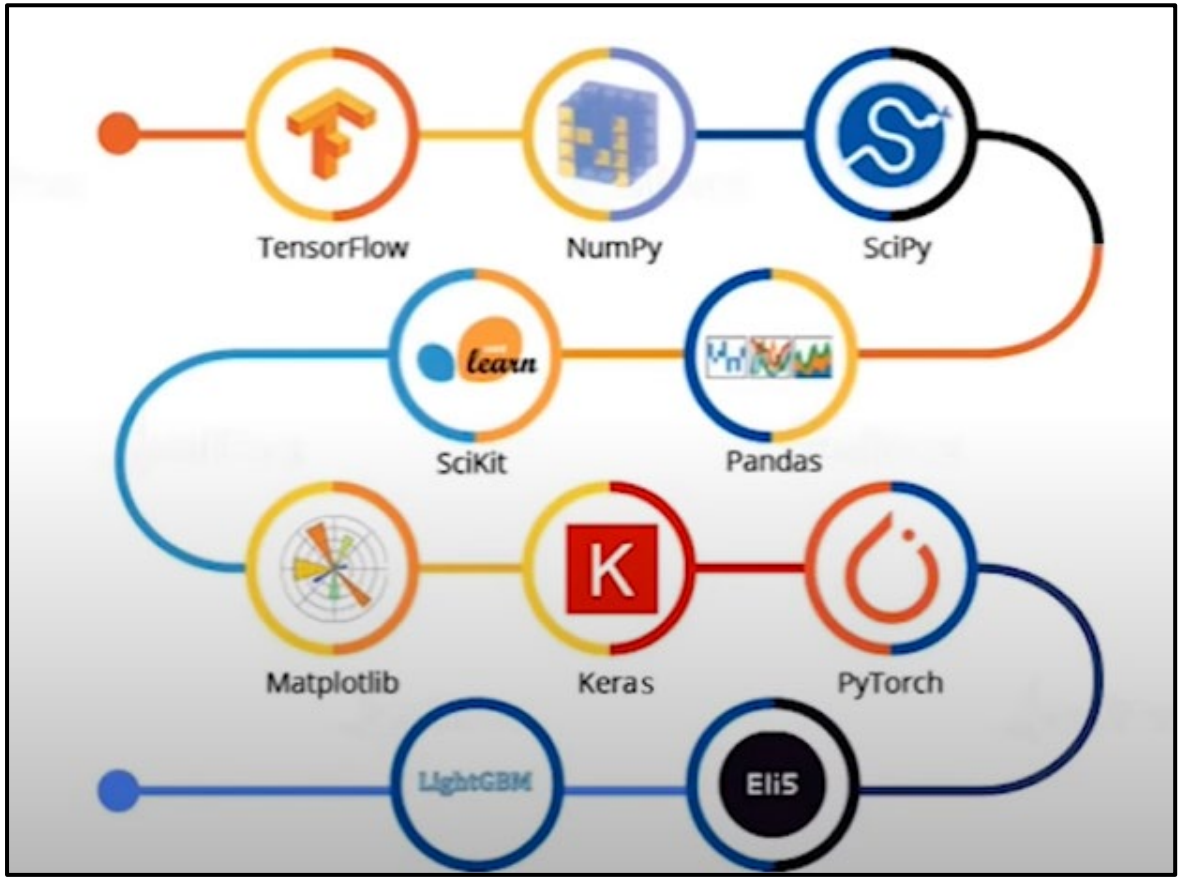


Python Introduction

Installing Python on Windows

Step 6:

Install all the libraries you want!



Python Introduction

Python Tokens

In Python, every logical line of code is broken down into components known as **Tokens**

Normal Token Types

- Keywords
- Identifiers
- Literals
- Operators

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

What are Keywords?

01	Python keywords are special reserved words
02	They convey a special meaning to the Compiler/Interpreter
03	Each keyword has a special meaning and a specific operation
04	**NEVER** use it as a variable

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Python Keywords

true	false	none	and	as
asset	def	class	continue	break
else	finally	ELIF	del	expect
global	for	if	from,	import
raise	try	or	returns	pass
nonlocal	in	not	is	lambda

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

An identifier is the name used to identify a variable, function, class, or object

Rules for naming an identifier:

1. No special character, except underscore (_). Can be used as an identifier
2. Keywords should not be used as an identifier
3. Python is case sensitive, i.e., ‘Var’ and ‘var’ are two different identifiers
4. The first character of an identifier can be a alphabet or underscore (_) but no a digit

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

A literal is the raw data given to a variable

Various Types of Literals

- String literals
- Numeric literals
- Boolean literals
- Special literals

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

What are string literals?

Formed by enclosing a text within quotes. Both single and double quotes can be used

Input

```
name1 = "john"
name2 = "james"
print(name1)
print(name2)
text1 = "hello world"
print(text1)
multiline = '''
str1
str2
str3
'''
```

Output

```
john
james
hello world
st1
st2
st3
```

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

What are numeric literals?

Formed by a character string of digits from 0 to 9, decimal point, and a plus/minus sign

Numeric Literal Formats

Int	Long	Float	Complex
+ve and –ve numbers (integers) with no fractional part E.g.: 100,-234	An unlimited string of integers followed by upper or lowercase L E.g.: 233424243L	Real numbers with both integer and fractional parts E.g.:-213.3	Strings in the from of a +bj, where ‘a’ is the real part & ‘b’ is the imaginary part E.g.: 3.14j

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

What are numeric literals?

1. In Python, the value of an integer is not restricted by the number of bits, and it can expand to the limit of the available memory
2. No special arrangement is required for storing large numbers

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

What are numeric literals?

It can either be True or False

```
var1 = True
var2 = False
var1 == var2
```

False

```
var1 = True
var2 = True
var1 == var2
```

True

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Operators are special symbols that are used to carry out arithmetic and logical operations

Various Types of Operators

- Arithmetic
- Assignment
- Comparison
- Logical
- Bitwise
- Identity
- Membership

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Arithmetic Operators

Used for common mathematical operations

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponentiation

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Assignment Operations

Used to assign values to variables

Operator	Operation
=	X=10
+=	X=X+2
-=	X=X-29
*=	X=X*12
/=	X=X/3
//=	X=X // 6

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Comparison Operators

Compares values and return either True or False

Operator	Operation
==	Equal
!=	Not equal
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Logical Operators

Used to combine conditional statements

Operator	Description
and	True if both statements are True
or	True if one of the statements is True
not	If True, then returns False

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Bitwise Operators

Used to compare binary numbers

Operator	Operation
&	AND
|	OR
^	XOR
~	NOT
<<	LEFT SHIFT
>>	RIGHT SHIFT

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Identity Operators

Used to check if the objects are the same or not

Operator	Operation
Is	Returns True if both variables are the same object
Is not	Returns True if both variables are not the same object

Python Introduction

- Keywords
- Identifiers
- Literals
- Operators

Python Tokens

Membership Operators

Used to test if a sequence is present in an object

Operator	Operation
In	Returns True if the specified value is present in the object
not in	Returns True if the specified value is not present in the object

Python Introduction

Python variables

A variable is a memory location where we can store values. In Python, the data type will be identified according to the data we provide

A variable should start with a letter or an underscore and cannot start with number. **There are two ways of assigning values to a variable:**

1. Assigning a Single Value
2. Multiple Assignment

Python Introduction

String Methods

len () function

For Ex. print(len("Training Basket"))
it counts the characters of strings.

lower () method.

For Ex. Name = Training Basket
print (Name.lower ())
it coverts all alphabet into lower case.

upper () method.

For Ex. Name = Training Basekt
print (Name.upper())
it coverts all alphabet in upper case latter

Python Introduction

String Methods

title () method.
For ex. name = Training Basket
print (name.title())
It coverts first letter of word into capital letter

Count () method
For Ex. print (name.count("T")

- It counts the same character in a word or string
- It is case sensitive

Replace and find method

- Replace is used for replace a word character space with anything.
- Find is used for find the word, character in our string

For Ex. string = "She is very beautiful"
print (string.find("is")

Python Introduction

User input
Input () function
For ex:

```
name = input (“type your name”)
print (“hello”+name)
```

Note: it can always take input from user in the from of strings.

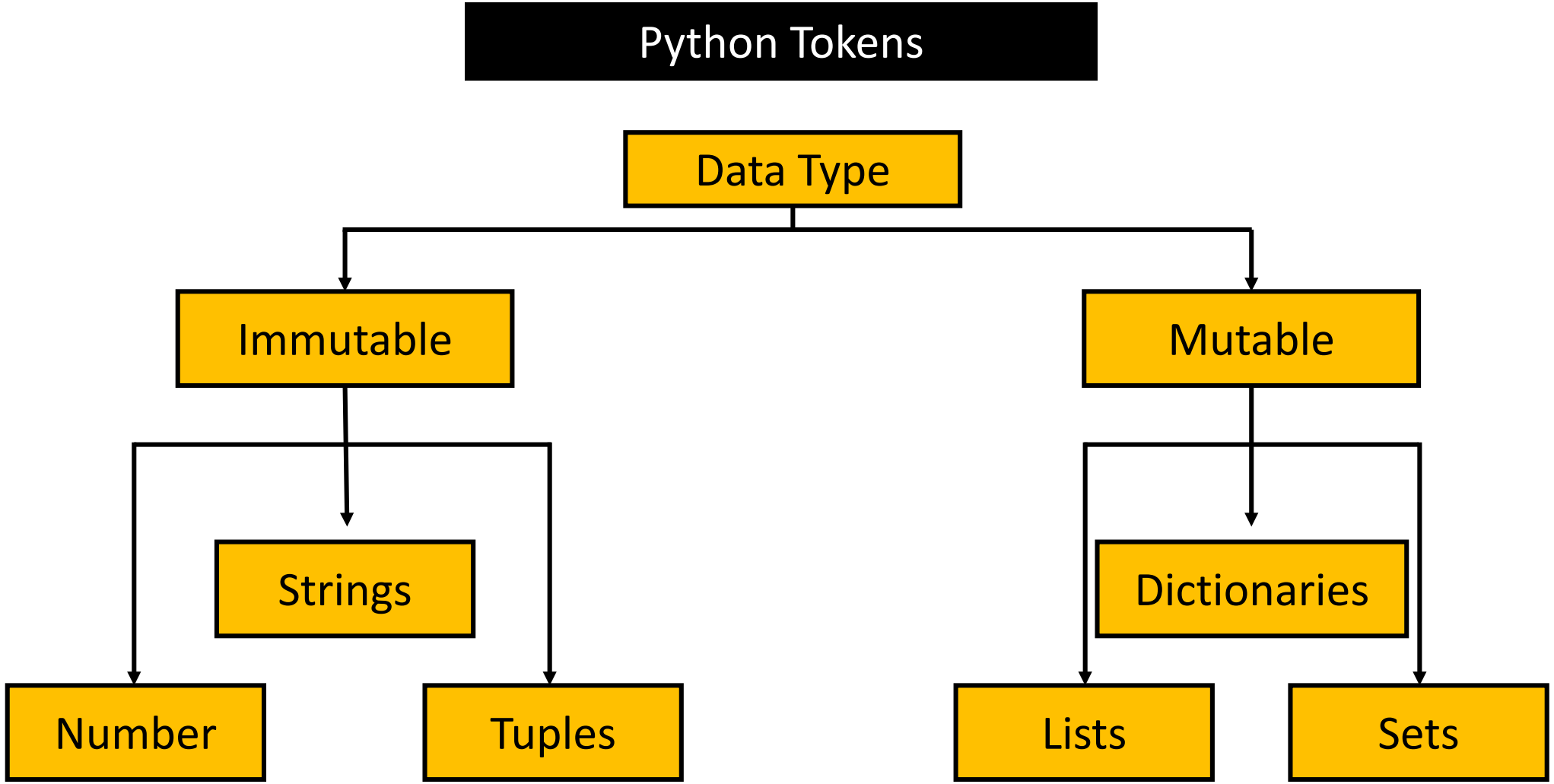
Two or more input in one lone
For ex:

```
name, age = input (“enter your name & age”).split()
```

Note: user can input two values in same line.

Data Structures

Data Structures



Data Structures

- Numbers
- Strings
- Tuples

Python Tokens

There are three numeric data types:

Data Types	Description
Int	Integer type (whole number, no decimals)
float	Floating point type (one or more decimals)
complex	Written with a imaginary number j

```
num = 12334
rate = 13.6
com = 6j
print (type(num))
print(type(num))
print(type(com))
```

```
<class 'int'>
<class 'float'>
<class 'complex'>
```

Data Structures

- Numbers
- Strings
- Tuples

Python Tokens

Strings

The value of the string data type is a string literal

```
var1 = "okey"
var2 = "pythonTutorial"
print(type(var1))
print(var1[0])
print(var2[1:5])
```

```
<class 'str'>
o
ytho
```

```
st = "Replacement"
st.replace("R", "P")
```

```
'Replacement'
```

Data Structures

- Numbers
- Strings
- Tuples

Python Tokens

Tuples

A sequence of immutable Python objects

```
mygroup = ('a','b','c','d')
```

concatenation

```
##Concatenation -- Add two string/ characters
mygroup = ('a','b','c','d')

mygroup += ('f',)
print(mygroup)
```

('a', 'b', 'c', 'd', 'f')

Data Structures

- Lists
- Dictionaries
- Sets

Data Types in Python

Lists

A sequence of mutable objects

```
mygroup = ('a', 10, 7.2 , 'data')
```

concatenation

```
## Concatenation -- Add elements to list
mylist = ['a' , 1 , 3.14 , 'python']
mylist  += ['d' , ]
print(mylist)
```

['a', 1, 3.14, 'python', 'd']

Data Structures

- Lists
- Dictionaries
- Sets

Data Types in Python

Dictionaries

An unordered collection of items

```
mydict = {1:'rahul' , 2: 'salman' , 3: 'Alice'}
print(mydict)
```

{1: 'rahul', 2: 'salman', 3: 'Alice'}

Empty dictionary

```
## empty dictionary
my = {}
```

Data Structures

Lists

Dictionaries

Sets

Data Types in Python

Sets

An unordered collection of immutable data which has no duplicate elements

```
myset = {1,2,3}
```

```
#creating set
myset = {1, 2 ,3 ,3 }
print(myset)
```

{1, 2, 3}

```
#union
mys1 = {1 , 2 , 'c'}
mys2= {1 , 'b' , 'c'}
mys1 | mys2
```

{'c', 1, 2, 'b'}

Data Structures

Python Arrays

01 Accessing an element

```
x = cars [0]
x
```

‘Honda’

02 Modifying an element

```
cars [0] = “Honda”
print(cars[0])
```

Honda

03 Getting the length of an array

```
x = len(cars)
x
```

3

04 Looping an array

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

Honda
BMW
Ope1

Data Structures

Python Arrays

05 Adding an element

```
cars.append("Opel")
print(len(cars))
```

```
4
```

06 Removing an element from a position

```
cars
```

```
['Ford', 'Volvo', 'BMW']
```

```
cars.pop(1)
cars
```

```
['Ford', 'B
```

07 Removing a specific element

```
cars
```

```
['Ford', 'Volvo', 'BMW']
```

```
cars.remove("BMW")
cars
```

Data Structures

Python Arrays

Used to store multiple values in a single variable

Python does not have built-in support for arrays, but Python lists can be used instead

```
#storing in multiple
variable
car1 = "Ford"
car2 = "Volvo"
car3 = "BMW"
#using Array
cars = ["Ford" , "Volvo",
"BMW"]
print(car1)
print(cars)
```

Ford
['Ford', 'Volvo', 'BMW']

Control Structures

Control Structures

Conditional Statements

These statements are used to change the flow of execution when a provided condition is True or False

You are sick

True

False

Go to the doctor

Go and play football

Control Structures

If-else

Nested if-else

Conditional Statements

Nested if-else

```
##syntax
'''
if (condition 1):
    statement 1 ...
elif:
    statement 2...
else:
    statement 3...
'''
```

Control Structures

If-else

Nested if-else

Conditional Statements

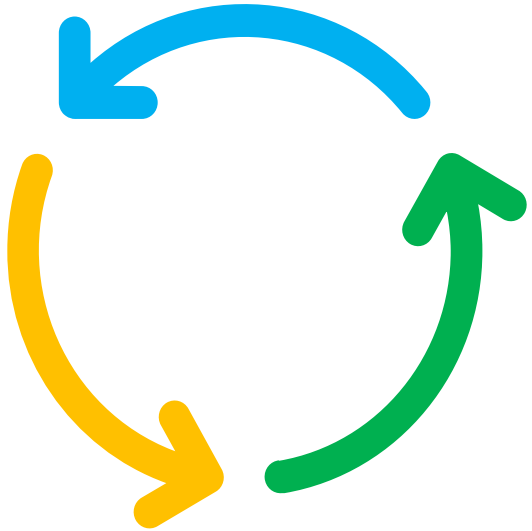
Nested if-else

```
##syntax
'''
if (condition 1):
    statement 1 ...
    if:
        statement 2...
    else:
        statement 3...
'''
```

Control Structures

Looping Statements

Looping is the process in which we have a list of statements that executes repeatedly until it satisfies a condition



Control Structures

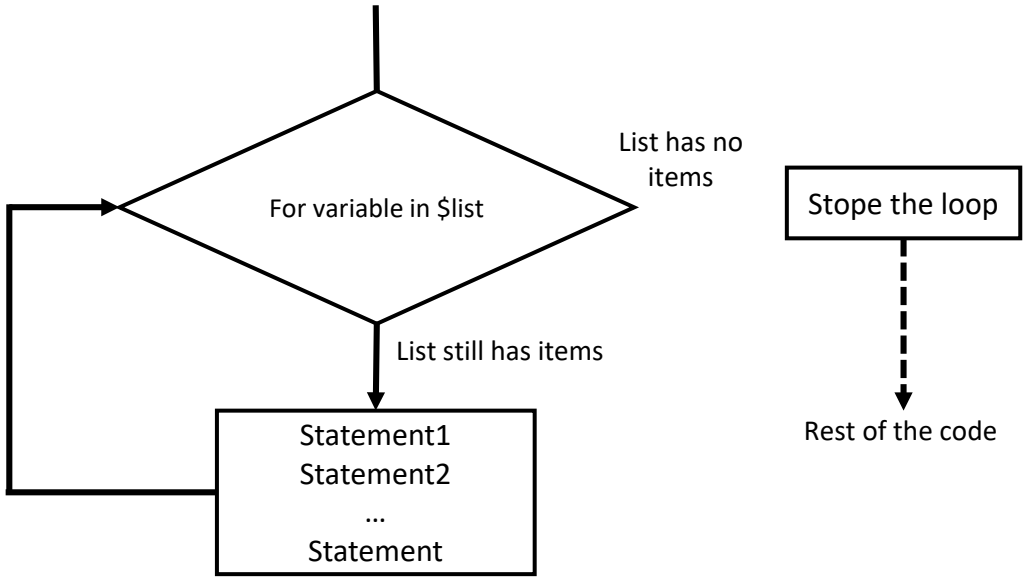
- for
- while
- break
- continue

Looping Statements

Syntax

```
for iterate_var in list:
    Statements
...
```

Flowchart




```
fruits = ["apple" , "banana"
, "cherry"]
for i in fruits:
    print(i)

apple
banana
cherry
```

Control Structures

- for
- while
- break
- continue

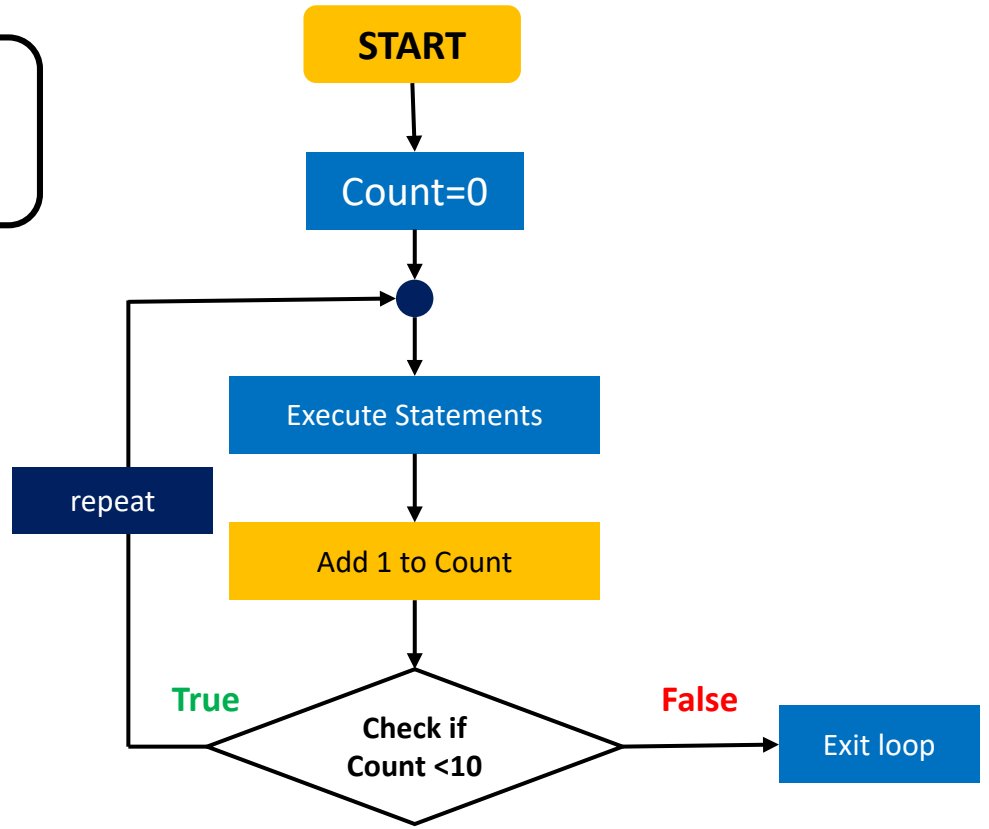
Looping Statements

Syntax

```
while (condition is True):
    Statements
    ...
```

```
a = 1
while a<5:
    print(a)
    a+=2

1
3
```

Flowchart



Control Structures

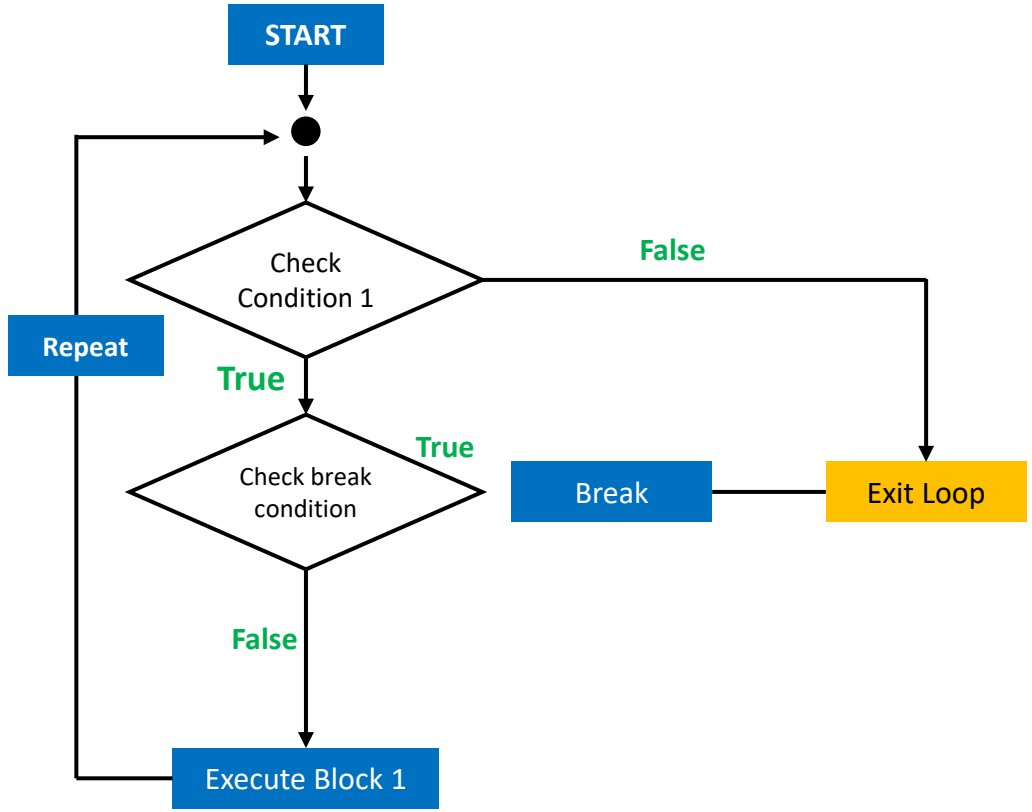
- for
- while
- break
- continue

Looping Statements

Syntax

```
while [command]

do
    If [command]

    then

            break
    fi
done
```

Flowchart



Control Structures

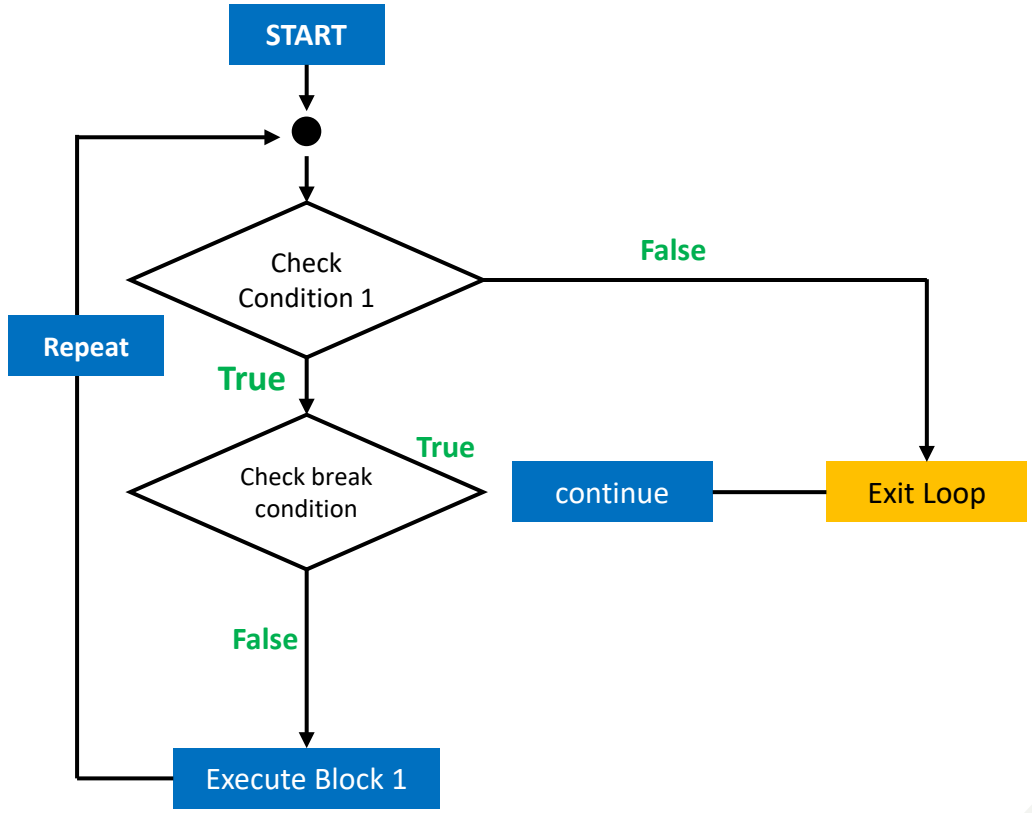
- for
- while
- break
- continue

Looping Statements

Syntax

```
while [command]

do
    If [command]

    then

            continue
     fi
done
```

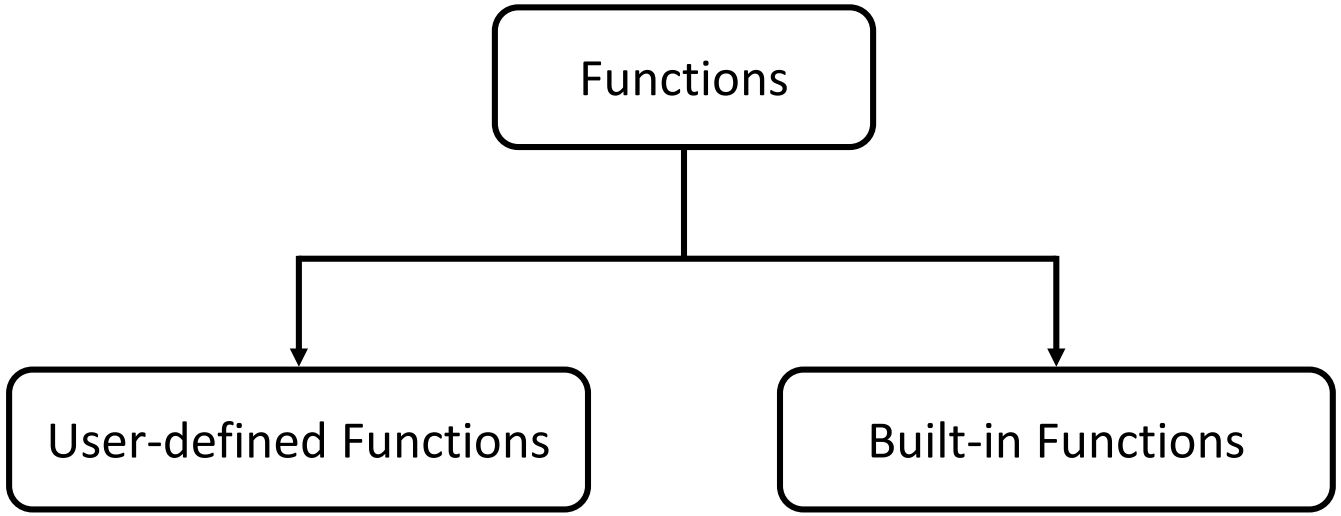
Flowchart



Functions

Functions

A function is a block of organized, reusable set of instructions that is used to platform some related actions



Functions

User-defined Functions

Built-in Functions

These functions are created by users

```
#syntax
'''def func_name (arg1 ,arg2 , arg3 , ...)
        statements...
        return [expression]’‘’

Example

#Example
def add(a , b):
    s = a+b
    return s
```

Functions

User-defined Functions

Built-in Functions

A function already available in a language that we can directly use in our code

Abs(): returns the absolute value of a number

All(): Returns True if all items in an iterable object are true

Any(): Returns True if any item in an iterable object is true

Ascii(): Returns a readable version of an object and replaces non-ASCII characters with a ’escape’ character

Bin(): Returns the binary version of a number

Bool(): Returns the boolean value of a specified object

Functions

Lambda Function

An anonymous function, i.e., a function having no name. A Lambda function cannot contain more than one expression

```
#syntax
'''
lambda arguments : expression
'''


x = lambda a : a+10
print(x(5))
```

15

Functions

Functions vs Lambda Functions

```
#function
def multiply(x , y):
    return x*y
```

```
#lambda Function
r = lambda x , y : x*y
print(r(5 , 8))
```

36

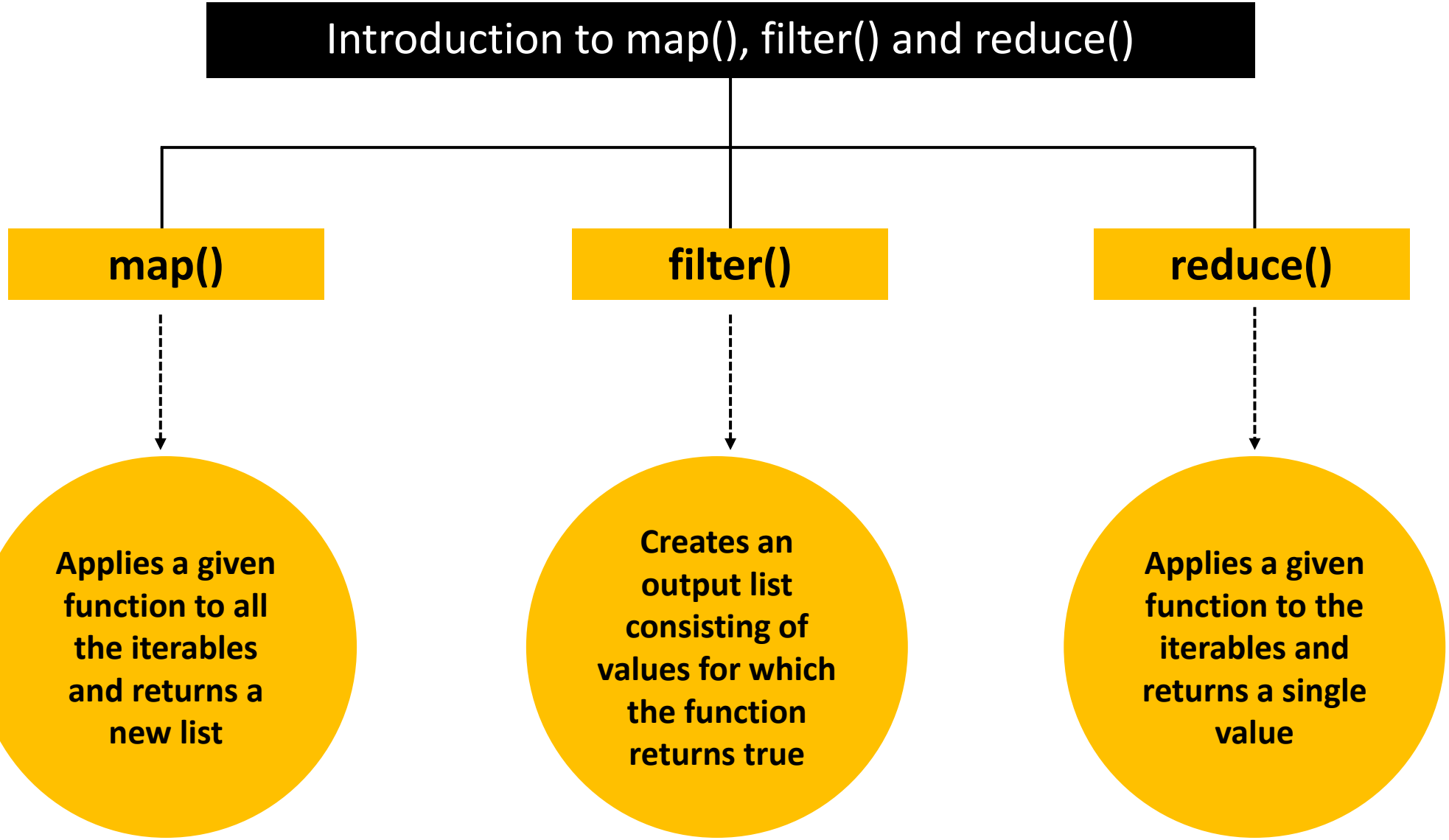
Power of Lambda: An anonymous function inside another function

```
def my_func(n):
    return lambda a : a+n
my_sum = my_func(3)
print(my_sum(10))
```

13

Advanced Functions

Advanced Functions

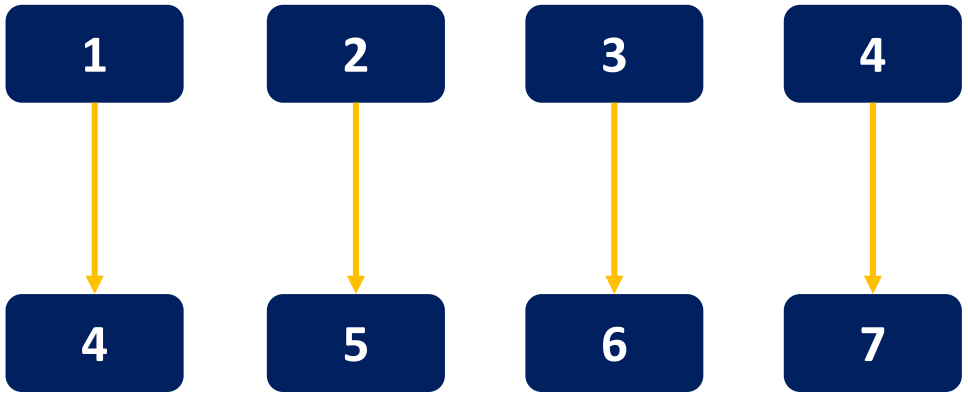


Advanced Functions

map () function

Applies a given function to all the iterables and returns a new list

```
output = map (lambda x: x+3 , [1,2,3,4])
```

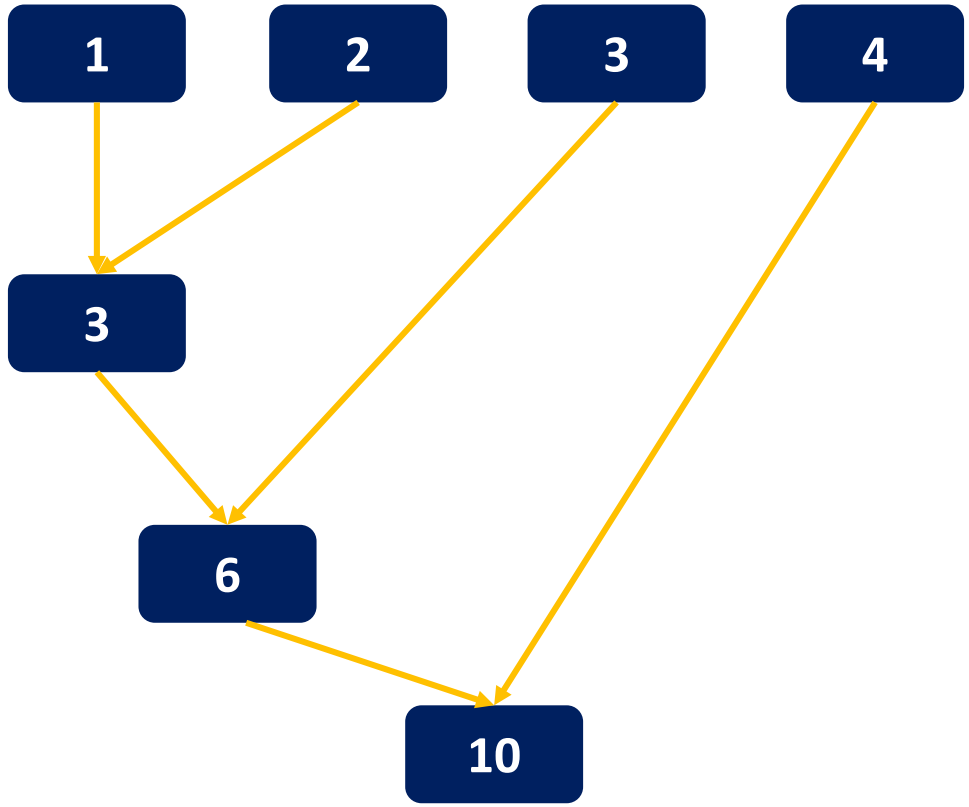


Advanced Functions

reduce () function

Applies some other function to a list of elements that are passed as a parameter to it and finally returns a single value.

```
output = map (lambda x: y: x+y,  [1,2,3,4])
```

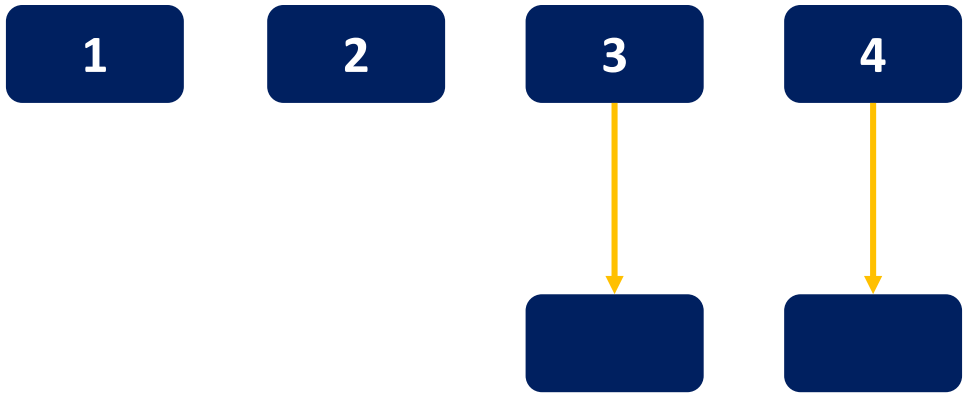


Advanced Functions

Filter() within map()

Filters out integers>=3 resulting in [3.4]. then maps this using (x+x) condition. You will get [6.8]. Which is the output.

```
finlter(lambada x: (x>=3), (1,2,3,4))
```




```
c = map (lambda x: x+x , [3,4])
```

Advanced Functions

Closures

First class function

First class functions allows us to treat functions as any other objects, so that we can use them to:

- Pass as an argument to another function
- Return the function from another function
- Assign the function to another variable

Nested function

Python supports the concept of a “nested function” or “inner function”, which is simply a function defined inside another function.

Note: Not to be confused with recursive functions.

Advanced Functions

Closures

Closures

Closure is an inner function that remembers and has access to the variables in the local scope in which it was created even after the outer function has finished executing.

```
def outer_func(msg):
    Message = msg

    def inner_func():
        print(message)
        return inner_func

My_func = outer_func("hello")
My_func()
```

Advanced Functions

Decorators

A decorator in Python is a function that takes another function as its argument, adds some functionality to it and returns yet another function.

Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated.

Advanced Functions

Yield

The yield is a keyword used in Python to return some value from the function without finishing the states of a local variable.

Iterators

Iterator in Python is simply an object that can be iterated upon. An object which will return data one element at a time.

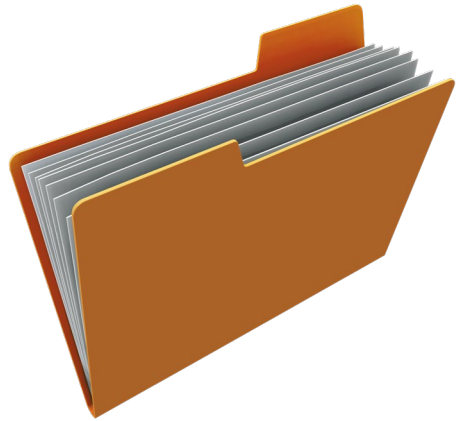
File Handling and Exception Handling

File Handling and Exception Handling

File Handling

Why do we need file handling?

File handling is important in any application that handles permanent data. We will need file handling if we have to read from or write to files



File Handling and Exception Handling

File Handling

Important File Handling Operations in Python

- Open
- Read
- Write/Create
- Delete

File Handling and Exception Handling

- Open
- Read
- Write/Create
- Delete

File Handling

The open() function takes two parameters:
Filename and mode

```
#symtax
f = open("path of file")
```

Mode Options

- ‘r’-Read: the default value; opens a file for reading; returns an 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 a file with the same name already exists

File Handling and Exception Handling

- Open
- Read
- Write/Create
- Delete

File Handling

The read() function is used to read n bytes from the mentioned file

```
#Example
f = open('demofile.txt' , 'r')
print(f.read())
```

Reading the first 5 lines

```
#Reading parts of file
f = open('demofile.txt' , 'r')
print(f.read(5))
```

Reading line by line

```
## Loop through the file
# Read the file line by line
f = open('demofile.txt' , 'r')
for i in f:
    print(i)
```

File Handling and Exception Handling

- Open
- Read
- Write/Create
- Delete

File Handling

To write to an existing file, we must add a parameter to the open () function

- ‘a’ – Append: Appends at the end of the file
- ‘w’ – Write: Overwrites the existing content of the file

```
# Example : Append
f = ("demofile.txt", 'a')
f.write("now the file has one more line!")
```

```
#Example : Overwrite
f = open("demofile.txt" , 'w')
f.write("Woops! I have deleted the Content!")
```

File Handling and Exception Handling

- Open
- Read
- Write/Create
- Delete

File Handling

To import the OS module

Use the remove() function to delete the mentioned file

```
#Deleting the File
import os
os.remove("demofile.txt")
```

File Handling and Exception Handling

Exception Handling

An exception is a runtime error caused by things that are outside the developers control, e.g.: FileNotFoundException etc

```
Exception                                 Traceback (most recent call last)
<ipython-input-2-4653874ad6d5> in <module>
----> 1 throw_exception(0)

<ipython-input-1-917ec045a2a7> in throw_exception(num)
      1 def throw_exception(num):
----> 2     if num == 0: raise Exception("Argument should not be zero")
      3     else: print(num)

Exception: Argument should not be zero
```

Exception Handling is the process of writing code which can expect things to go wrong and raise exceptions and handle them accordingly

File Handling and Exception Handling

try

except

finally

raise

Exception Handling

You can create your exceptions and raise them when appropriate

Try allows you to define a code block that could throw an exception

```
try:
    throw_exception(1)
except Exception as e:
    print(e)
else:
    print("did not throw exception")
```

1
Did not throw exception

File Handling and Exception Handling

try

except

finally

raise

Exception Handling

You can create your exceptions and raise them when appropriate

Allows you define a code block when an exception is

```
try:
    raise_custom_exception()
except MyException as e:
    print(e)
```

Raised Custom Exception

File Handling and Exception Handling

try

except

finally

raise

Exception Handling

You can create your exceptions and raise them when appropriate

Code block that gets run even if an exception is thrown or not

```
try:
    k = 5//0
    print(k)
except ZeroDivisionError:
    print("number diveded by zero")
finally:
    print("whatever happens this
will excecute")
```

No divide by zero
Whatever happens this will execute

File Handling and Exception Handling

try

except

finally

raise

Exception Handling

You can create your exceptions and raise them when appropriate

Raise allows you to raise your own custom exceptions

```
class Myexception(Exception):
    def __init__(self):
        super().__init__(Exception , self ).__init__()
        self.args =("Raised Custom Exception" , )
```

```
def raise_custom_exception():
    raise Myexception()
```

Object Oriented Programming

Object Oriented Programming

Python: Classes and objects

What is a class and what is an object in Python?

1. Python is an object-oriented programming language
2. Almost everything in Python is an object, with its properties and methods
3. A class is like a ‘blueprint’ for creating objects

Class

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

Object

```
obj1 = MyClass()
   print(obj1.x)
```

5

Object Oriented Programming

Introduction to OOPs

Object-Oriented Programming is a programming paradigm where you can use a real world entity which is called an **Object.**

Let us consider an example

- Attribute: Name, Age Color
- Behaviour: Singing, Dancing

Parrot

Object Oriented Programming

Basic Principle of OOPs

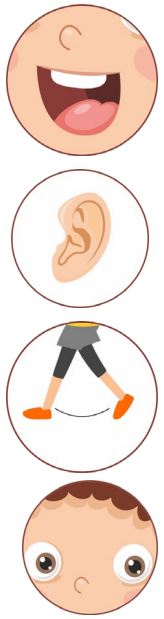
- Polymorphism
- Encapsulation
- Inheritance
- Object
- Class

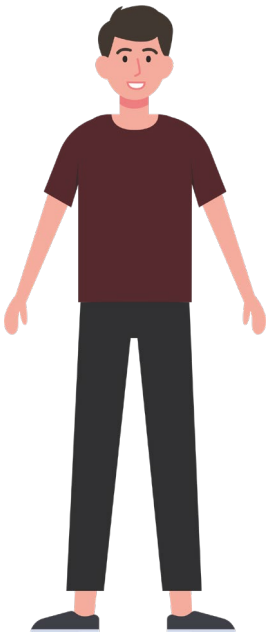
Object Oriented Programming

Every Human Being is Classified into:

FEMALE

Same Functions

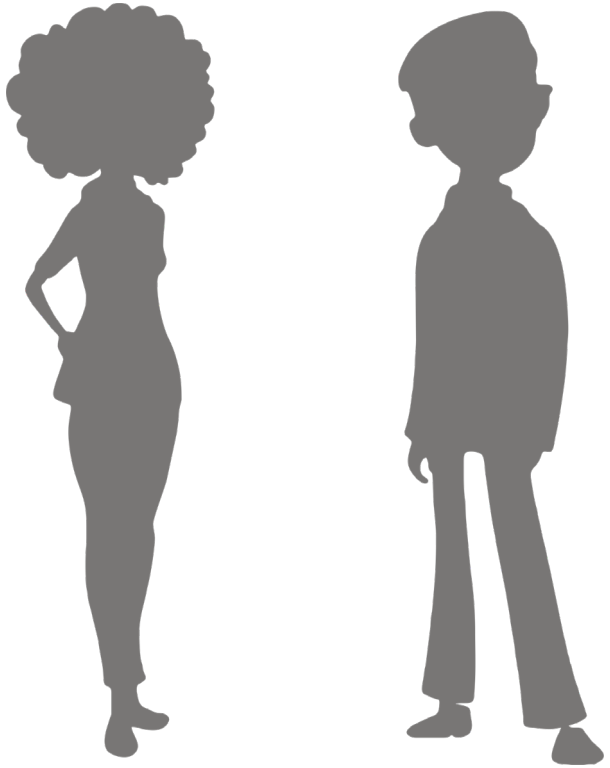




MALE

Object Oriented Programming

Considering Human Being is a **class**



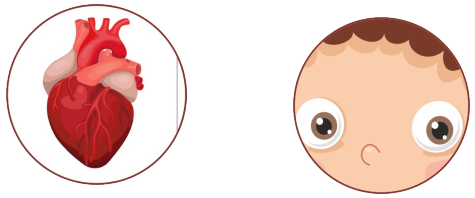
Object Oriented Programming

Common body features and functions are **Class Attributes**

Common Body Parts:

Every Human has:

NOSE HAND LEGS



HEART EYES

Common Body Function:

Every Human has:

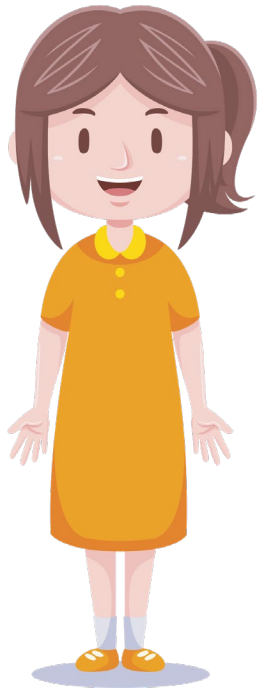
WALK LISTEN SPEAK

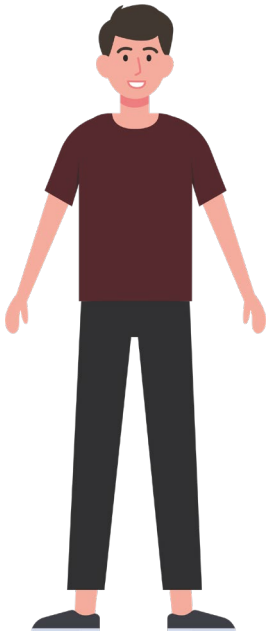
SEE SMELL

Object Oriented Programming

Male and Female are inherited from Class Human Being

FEMALE

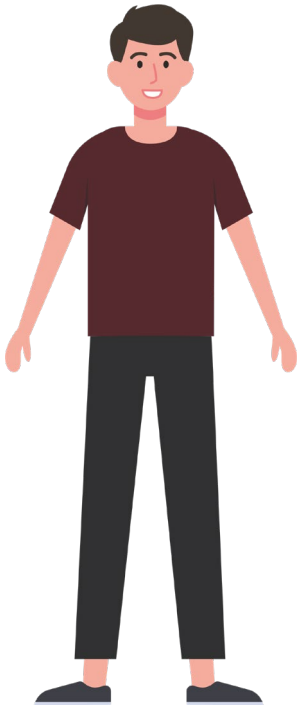




MALE

Object Oriented Programming

‘Name’ and **‘Age’** are object of class Male

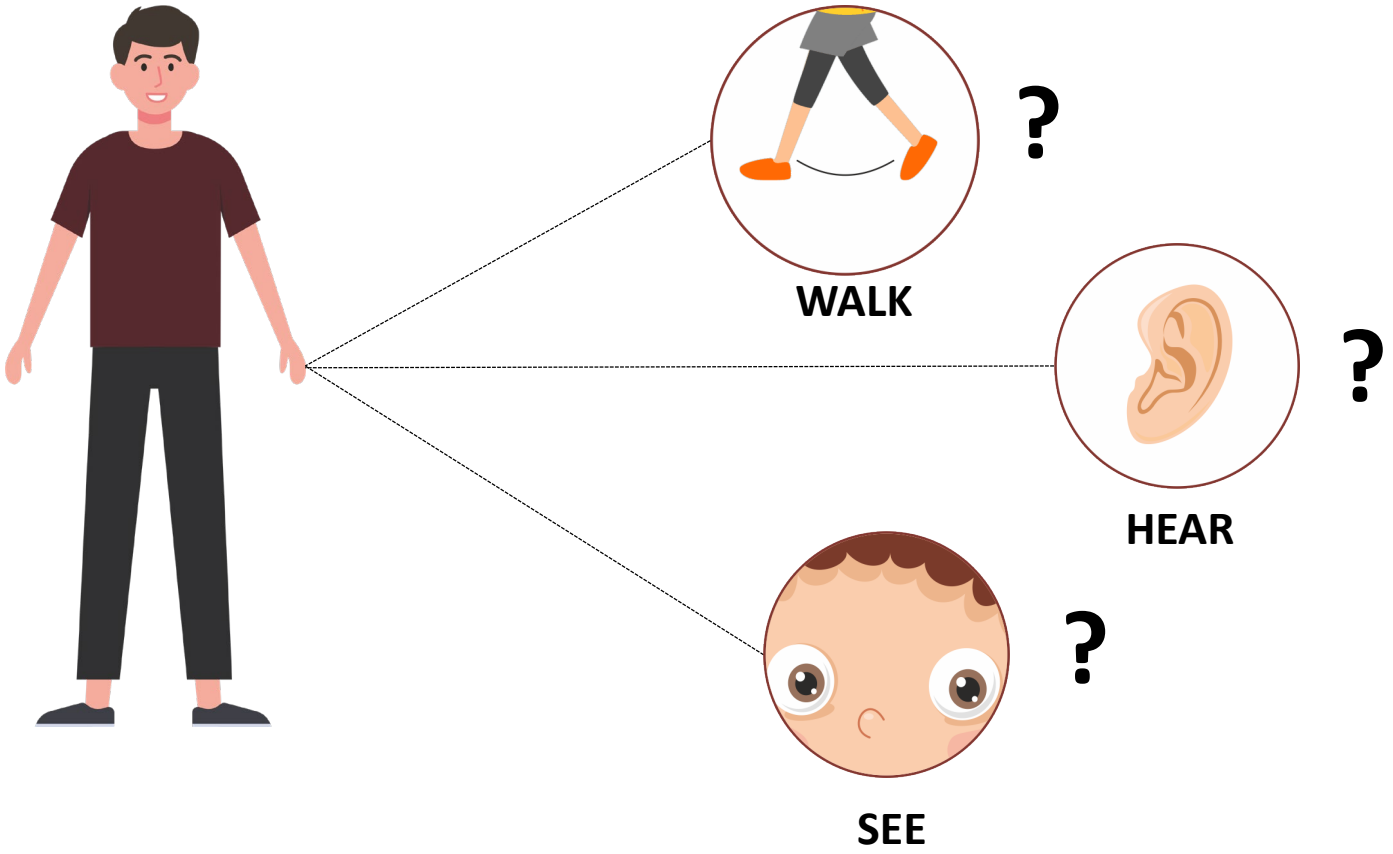


Class: Male
Name: Victor
Age: 24

- Objects have a physical existence
- Class is just a logical definition

Object Oriented Programming

You don’t know the detail of how you walk, listen or see.
i.e. its hidden or Encapsulated

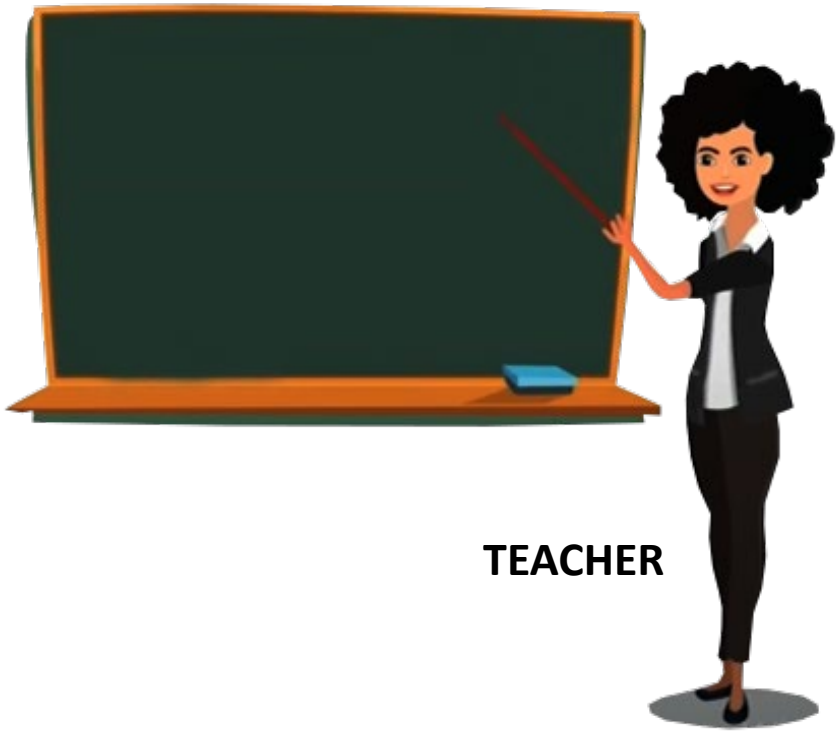


Object Oriented Programming

‘She’ can be a woman, wife, mother and a teacher at the same time which is many forms or polymorphism

WOMAN

WIFE OF MOTHER

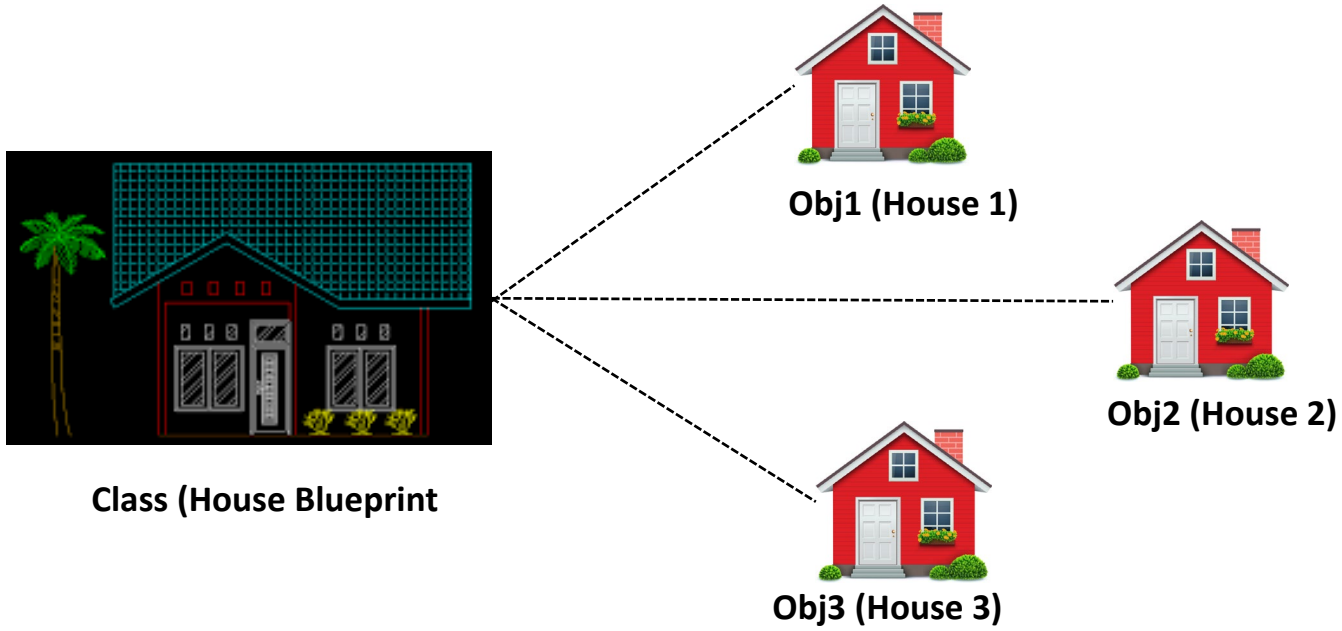


TEACHER

Object Oriented Programming

What are Objects and Classes?

Class is a blueprint for an object and the objects are defined and created from classes (blueprint)



Object Oriented Programming

What are objects and Classes?

01	Object is the basic unit of object oriented programming
02	An object represents a particular instance off a class
03	There can be more than one instance of an object
04	Each instance off an object can holt its own relevant data
05	Objects with similar properties and methods are grouped together to from a class

Object Oriented Programming

How to create a class in Python?

Syntax

```
Class NameOfclass:
<statement-1>
.
.
<statement-2>
```

Example

```
class ClassName:
   variable = “I am a class Attribute”
   def function (self):
          print(“I am from inside the class”)
```

```
ClassName.function
```

```
<function__main__.ClassName.function(self)>
```

Object Oriented Programming

How to create a class in Python?

Syntax

<obj-name> = NameOfClass()

Example

```
obj1 = ClassName()
obj1
```

```
<__main__.ClassName at 0x1a0eb1d3d48>
```

Object Oriented Programming

How to access Class Members?

Example

```
obj1 = ClassName()
Obj1 = ClassName()
#Creating new instance attribute for obj2
Obj2.variable = “I was just created”
print(obj1.variable)
print(obj2.variable)
Print(ClassName.variable)
Obj1.function()
```

```
I am a class Attribute
I was just created
I am a class Attribute
I am from inside the class
```

- Here obj1 and obj2 are object of class ClassName
- To assess the members of a Python class, we use the dot operator

Object Oriented Programming

init() method in Python

Example

```
class Student(object)

   def__init__(self, name, branch, year):
            self.name = name
            self.branch = branch
            self.year = year
            print("A student object is created.")

   def print_details(self):
            print("Name:", self.name
            print("Branch:", self branch)
            print("Year:", self.year)
```

```
ob1= Student( "Paul","CSE", 2019)
ob1.print_details()
```

```
A student object is created.
Name: Paul
Branch: CSE
```

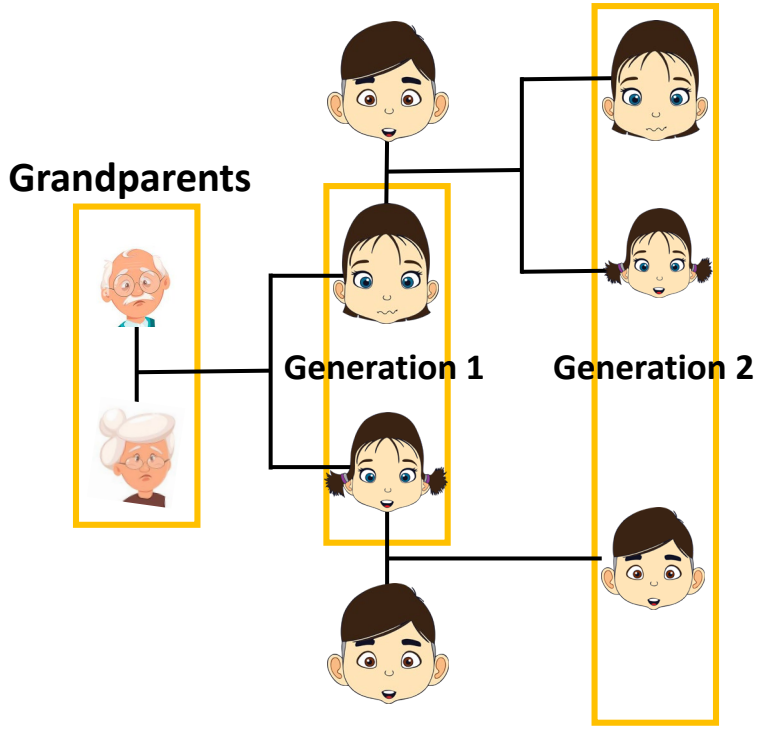
- _init_is a special method in Python classes is constructor method for c class
- _inti_is called when ever an object of the class is constructed

Object Oriented Programming

Inheritance

One class acquiring the property of another class. For example, you would have inherited few qualities from your parents.

In a family tree, traits such as hair color and poor eyesight are passed from generation to generation.



Object Oriented Programming

Inheritance

Different Types of inheritance in Python

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

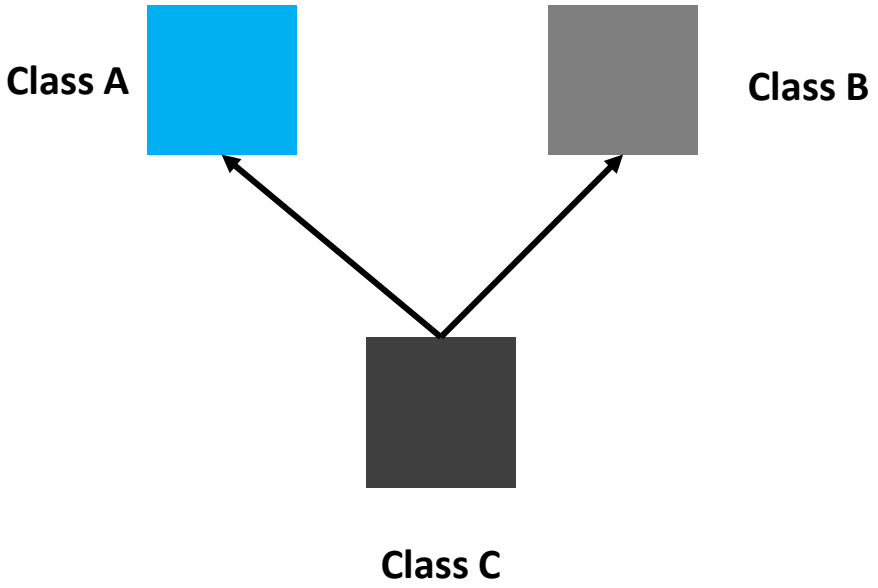
Single class inherits from a class

```
class fruit:
        def__init__(self):
                print('i'm a fruit"
class citrus (fruit):
        def__inti__(self):
                super().__inti__()
                print("i'm citrus"
lemon = citrus()
```

Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

A class inherits from multiple classes



Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

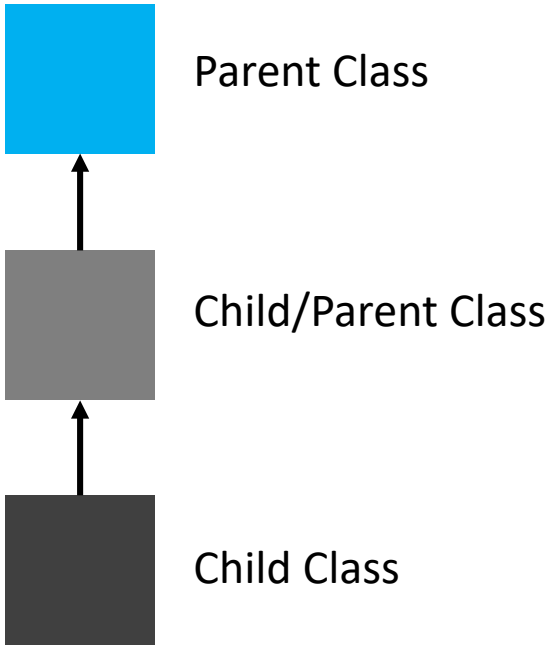
A class inherits from multiple classes

```
class A:
    pass
class B:
    pass
class C(A,B):
    pass
Issubclass(C,A) and issubclass(C,B)
```

Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- **Multilevel Inheritance**
- Hierarchical Inheritance
- Hybrid Inheritance

One class inherits from a class, which will inherit from another class



Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

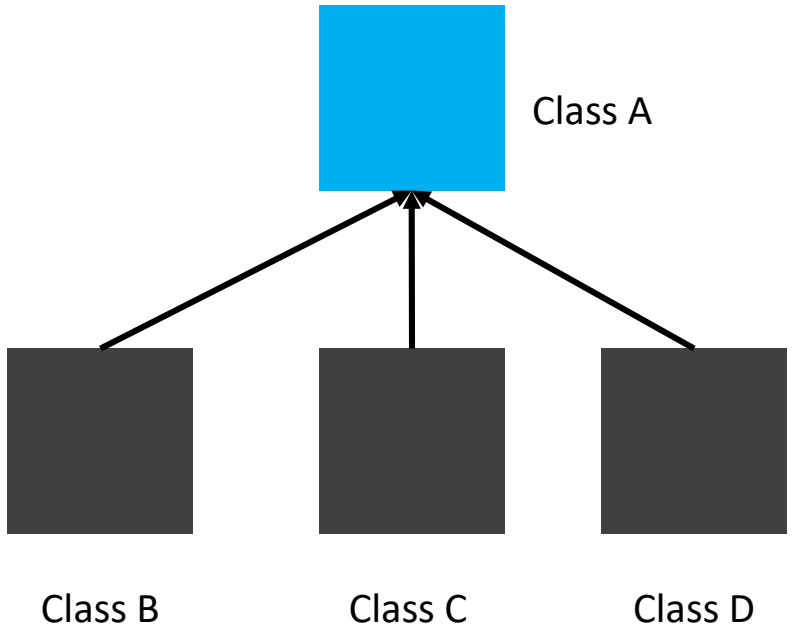
One class inherits from a class, which will inherit from another class

```
class A:
        x=1
class B(A):
        pass
class C (B):
        pass
cobj=C()
cobj.x
```

Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- **Hierarchical Inheritance**
- Hybrid Inheritance

More than one class inherits from a class



Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- **Hierarchical Inheritance**
- Hybrid Inheritance

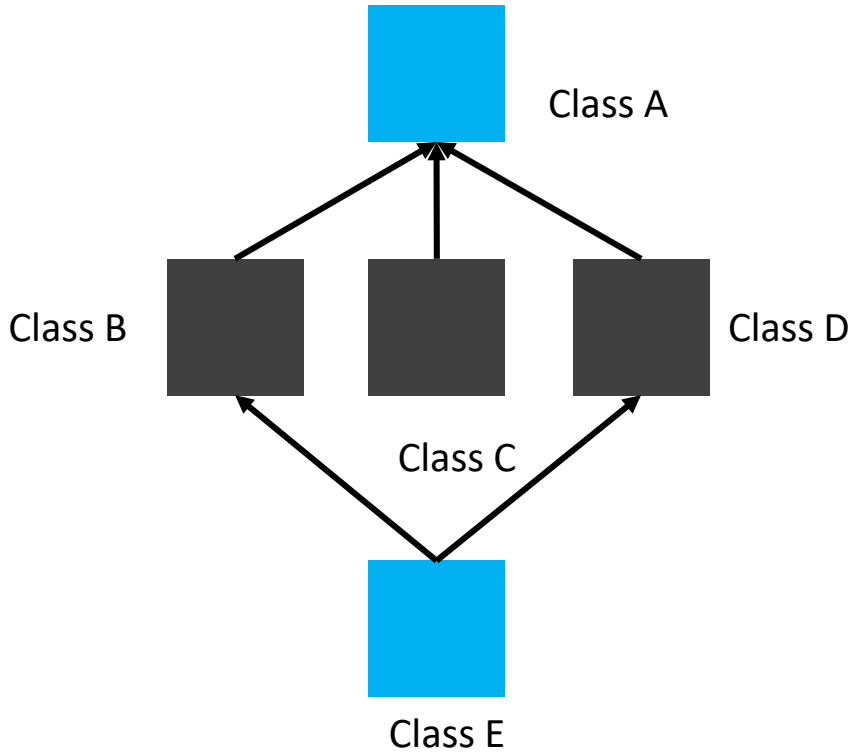
More than one class inherits from a class

```
class A:
        pass
class B(A):
        pass
class C(A):
        pass
Issubclass(B,A) and issubclass(C,A)
```

Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Combination of any two kinds of inheritance



Object Oriented Programming

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Combination of any two kinds of inheritance

```
>>> class A:
        x=1
>>> class B(A):
        pass
>>> class C(A):
        pass
>>> class D(B,C):
        pass
>>> dobj=D()
>>> dobj.x
```

Object Oriented Programming

Inheritance Super Function

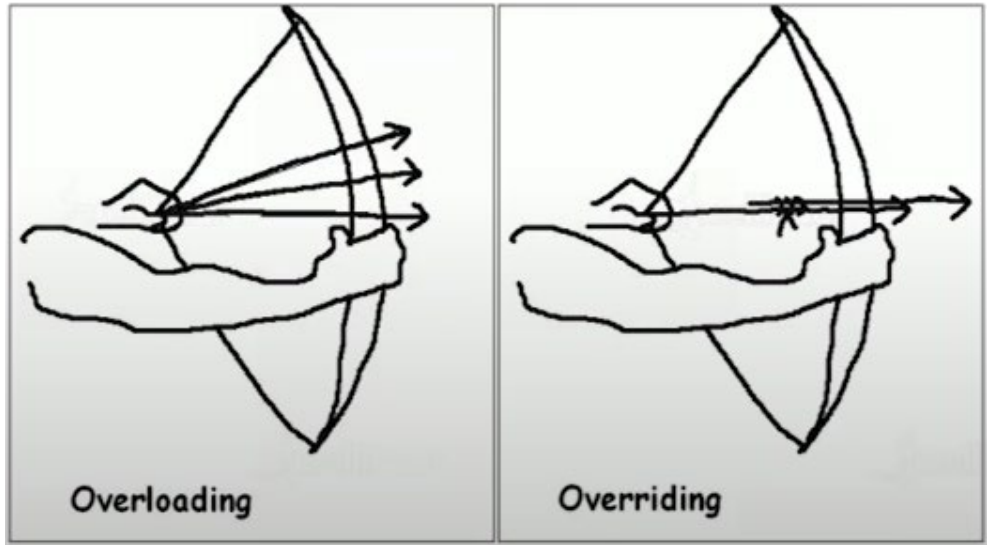
Used to call a method from the parent class

```
class Vehicle:
        def start (self):
                print("Starting engine")
        def stop(self):
                print("Stopping engine")
class TwoWheeler(Vechle):
        def say(self):
                super() start()
                print("I have tow wheels")
                super().stop()
Harley=TwoWheeler()
Harley.say()
```

Object Oriented Programming

Overriding vs Overloading

Developers sometimes get confused between them



Object Oriented Programming

Overloading

Overriding

Same function with different parameters

Way overloading a function?

```
def add(a,b):
    return a+b
def add(a,b,c):
    return a+b+c
add(2,3)
```

```
TypeError: add()missing 1
Required positional argument: 'c'
```

Object Oriented Programming

Overloading

Overriding

Same function with different parameters

How to overload a function?

```
def
add(instanceOf,*args):
  if instanceOf==‘int’:
            result=0
If instanceOf==‘str’:
            result=“
For I in args:
            result+=I
return result
add(‘int’,3,4,5)
```

Object Oriented Programming

Overloading

Overriding

Subclass may change the functionality of a Python method in the superclass

Overriding a function

```
class A:
        def sayhi(self):
                print("I'm in A")
class B(A):
        def sayhi(self):
                print("I'm in B")
Bobj=B()
Bobj.sayhi()
```

Object Oriented Programming

Encapsulation

Encapsulation = Abstraction + Data Hiding. Abstraction is showing essential features and hiding non-essential features to the user.

While writing a mail you don’t know how things are actually happening in the backend

Object Oriented Programming

Encapsulation

How to access a Private Method?

```
class Car:
        def_init_(self):
          self._updateSofware()

Def drive(self):
  print('driving')

 def _ updateSofware(self):
   print ('updating software')

redcar = Car()
redcar.drive()
redcar._car_updateSoftware()
```

Private method can be called using
redcar._car_updateSoftware

Object Oriented Programming

Encapsulation

How to access a Private Method?

To change the value of a private variable, a setter method is used

```
def setMaxSpeed(self,speed):
        self._maxspeed = speed

redcar = Car()
redcar.drive()
redcar._maxspeed = 10 # will not
change variable because its private
redcar.setMaxSpeed(320)
redcar.drive()
```

Object Oriented Programming

Polymorphism

Functions with same name, but functioning in different ways

You behave differently in from of elders, and friends. A single person behaves differently at different time

Object Oriented Programming

Polymorphism

Polymorphism with a function

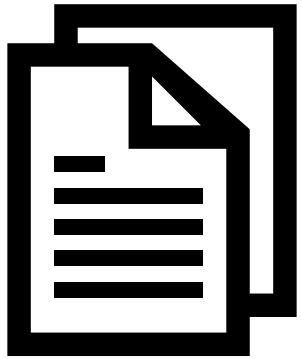
Example

```
def in_the_pacific(fish):
        fish.swim()

sammy = Shark()

casey = Clownfish()

in_the_pacific(Sammy)
in_the_pacific(casey)
```

Object Oriented Programming

Modules

To put it simply, Module is a file containing python code



A module can define functions, classes and variables and can also include runnable code. There are pre-defined modules in python standard library

Object Oriented Programming

Modules

from

Allows you to specify the things that you want to import from a module

From utils import add, subtract

import

Import command allows you to import the complete module

import os

Flask

Introduction to Flask

What is Flask?

Flask is a web application framework written in Python.

Introduction to Flask

What is a Web Framework?

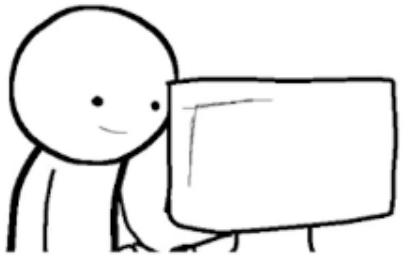
Libraries

Modules

Web Developer

Application

Life without Flask!



Using Flask!

Introduction to Flask

Flask!!

Enthusiasts named Pocco!

Werkzeug WSGI Toolkit

Jinga2 Template Engine

Installing Flask

Installation – Prerequisite

Prerequisite

virtualenv

Virtual Python Environment builder

Pip install virtualenv

Sudo apt – get install virtualenv

Installation – Flask

Installation

Once installed, new virtual environment is created in a folder

```
mkdir newproj
Cd newproj
Virtualenv venv
```

To activate corresponding environment, use the following:


```
venv\scripts\activate
```

```
Pip install Flask
```

Flask Application

Flask – Application

Test Installation

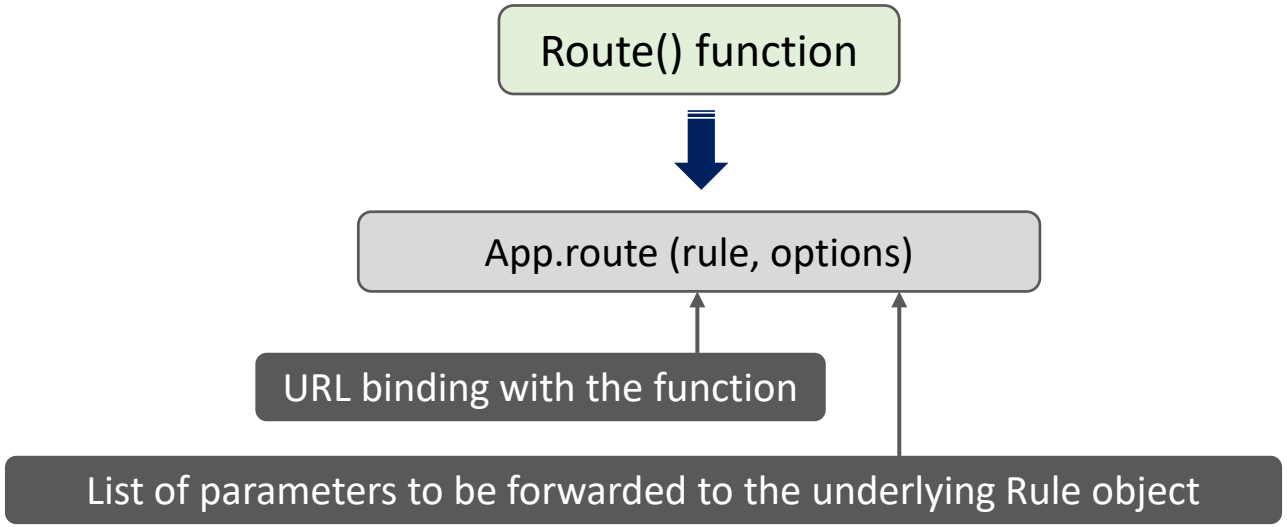
Use this simple code, save it as Hello.py

```
From flask import Flask
App = Flask (__name__)

@app.route(‘ / ‘)
Def hello_world():
        return ‘Hello World’

If __name__ == ‘__main__’ :
app.run()
```

Flask – Application

Importing flask module in the project is mandatory!

Flask constructor takes name of current module (__name__) as argument



Flask – Application

App.run(host, port, options

All these parameters are optional

Sl. No	Parameter	Description
1	host	Hostname to listen on. Defaults to 127.0.0.1 (localhost). Set to '0.0.0.0' to have server available externally
2	port	Defaults to 5000
3	debug	Defaults to false. If set to true, provides a debug information
4	Options	To be forwarded to underlying Werkzeug server.

Python hello.py

* Running on http://127.0.0.1:5000/ (Press CTRL + C to quit)

Flask – Application

Flask – Application

Debug mode

Flask application is started by calling run() method

How to enable Debug mode?

app.debug = True

app.run()

app.run(debug = True)

Flask Routing

Route() decorator in Flask is used to bind URL to a function

ample

```
@app.route('/hello')
def hello_world():
    return 'hello world'
```

Add_url_rule() function is also used to bind URL with function

Check out the following representation

```
Def hello_world():
    return 'hello world'
App.add_url_(' / ' , 'hello' , hello_world
```

Flask – Variable Rules

Flask – Variable Rules

It is possible to build a URL dynamically!

How? By adding variable parts to the rule parameter

Consider the example

```
From flask import Flask
app = Flask(__name__)

@app.route('/hello/<name>')
Def hello_name (name):
   return ' Hello %s! ' % name

If __name__ == '__main__':
app.run(debug =True)
```

http://localhost:5000/hello/trainingbasket

Flask – Variable Rules

More rules can be constructed using these converters

Sl. No	Parameter	Description
1	Int	Accepts integer
2	Float	For Floating point value
3	Path	Accepts slashes used as directory separator character

Visit the URL: → http://localhost:5000/blog/11

Browser Output → Blog number 11

http://localhost:5000/rev/1.1 → Revision Number 1.100000

```
From flask import Flask
app = Flask(__name__)

@app.route('/blog/<int:postID>')
def show_Blog(postID):
  return ' Blog Number %d'  % post ID

@app.route('/rev/<float:revNo>')
def revision(revNo):
   return 'Revision Number %f' % revNo

If __name__ == '__main__':
  app.run
```

Flask – Variable Rules

Consider the following code:

```
from flask import Flask
app = Flask (__name__)

@app.route('/flask')
def hello_flask():
   return 'Hello Flask'

@app.route('/python/')
def hello_python():
   return 'Hello Python'

If __name__ == '__main__':
App.run()
```

Run the code

/python = /python/

/flask ≠ /flask/

Flask – URL Binding

Flask – URL Building

url_for() function is used for dynamically building a URL for a specific function

```
From flask import Flask, redirect, url_for
app = Flask (__name__)

@app.route( ‘ /admin ‘ )
def hello_admin():
   return ‘Hello Admin’

@app.route( ‘ /guest / <guest> ‘ )
def hello_guest (guest):
   return ‘Hello %s as Guest ‘ % guest

@app.route ( ‘ /user/<name> ‘ )
Def hello_user (name):
   if name == ‘admin’:
        return redirect(url_for (‘hello_admin’))
   else:
       return
Redirect (url_for(‘hello_guest’,guest = name))

If __name__ == ‘__main__’:
   app.run debug =True)
```

http://localhost:5000/user/admin

Flask – HTTP Methods

Flask – HTTP Methods

HTTP Protocols are the foundation for data communication in www

Sl.No	Method	Description
1	GET	Sends data in unencrypted form to server
2	HEAD	Same as GET, but without response body
3	POST	Used to send HTML from data to server
4	PUT	Replaces all current representations of target resource with uploaded content
5	DELETE	Removes all current representations of target resource given by URL

Let’s look at an example

Flask – HTTP Methods

First we look at the HTML file

```
<html>
   <body>

            <from action = 'http://localhost:5000/login" method = "post">
               <p> Enter Name:</P>
               <p><input type = "text" name = "nm" /></p>
               <p><input type = "submit value = "submit" /></p>
         </form>

      </body>
 </html>
```

Flask – HTTP Methods

Next is Python Script

```
from flask import Flask, redirect, url_for, request
app = Flask (__name__)

@app.route( ' / success/<name> ' )
def success (name):
   return 'welcome %s' % name

@app.route( ' /login ' ,methods = ['POST' , 'GET'])
def login():
   if request .method == 'POST' :
        user = request.form['nm']
        return redirect (uri_for ( ' success' , name = user))

If __name__ == '__main__':
app.run(debug = True)
```

Let's check out the output!

Flask – Templates

Flask – Templates

Can we return the output of a function bound to a UR : in form of HTML?

```
from flask import Flask
App = Flask (__name__)

@app.route('/')
def index():
   return'<html><body><h1>'Hello
World'</h1></body></html>'

If __name__ == '__main__' :
    app.run(debug = True)
```

But this is cumbersome

```
from flask import Flask
app = Flask (__name__)

@app.route('/')
def index():
   return'<html><body><h1>'Hello
World'</h1></body></html>'

If __name__ == '__main__' :
    app.run(debug = True)
```

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

Flask – Templates

Flask uses jinga 2 template engine

```
<!doctype html>
<html>
    <body>

            <h1>Hello {{ name }}!</h1>

    </body>
</html>
```

```
from flask import Flask, render_template
app = Flask (__name__)

@app.route( ' /hello/<user> '
Def hello-name (user):
   return render_template ('hello.heml', name = user)

If __name__ == '__main__':
   app.run(debug = True)
```

Flask will try to find the HTML file in the templates folder, in the same folder in which this script is present.

The jinga2 template engine uses the following delimiters for escaping from HTML

- {% … %}for Statements
- {{…}} for Expressions to print to the template output
- {#...#} for Comments not included in the template output
- #...## for line Statements

Flask – Templates

Conditional statements in templates

```
from flask import Flask, render_template
app = Flask (__name__)

@app.route( ‘ /hello/<int : score> ‘ )
Def hello_name( score ) :
   return render_template( ‘ hello.html ‘ , marks = score)

If __name__ == ‘__main__ ‘:
  app.run(debug = True)
```

```
<!doctype html>
<html>
   <body>
        {% if marks>50 %}
        <h1> Your result is pass!</h1>
        {% else %}
        <h1> Your result is fail </h1>
        {% endif %}

   </body>
</html>
```

HTML Template script

Flask – Templates

```
from flask import Flask, render_template
app = Flask (__name__)

@app.route( ‘ result ‘ )
def result() :
   dict = {phy’ : 50, ‘che’ :60, ‘maths’ : 70}
   return render_template(‘result.html’ ,  result = dict)

If __name__ == ‘__main__ ‘:
  app.run(debug = True)
```

Another example

```
<!doctype html>
<html>
   <body>

            <table border = 1>
               {% for key, value in result.iteritems() %}

                  <tr>
                        <th> {{ key }} </th>
                        <td> {{ value }} </td>
                  </tr>

            {% endfor %}
         </table>

      </body>
 </html>
```

Flask – Static Files

Flask – Static Files

Web application will require a static file such as JS or CSS file

```
from flask import Flask, render_template
App = Flask (__name__)

@app.route ( ‘ /result ‘)
Def result ()
   dict = { ‘ phy ‘ :50, ‘ che’ : 60,  ‘ maths ‘ : 70}
   retyrb rebder _tenokate ( ‘result.html ‘ , result = dict)

If __name__ == ‘__main__’  :
   app.run(debug = True)
```

Python

JS File

```
Function sayHello() }{
    alert (“Hello World”)
}
```

```
<!doctype html>
<html>
   <body>

            <table border = 1>
               {% for key, value in result.iteritems() %}

                  <tr>
                        <th> {{ key }} </th>
                        <td> {{ value }} </td>
                  </tr>

            {% endfor %}
         </table>

      </body>
 </html>
```

HTML

Flask – Request Object

Flask – Request Object

Data from client's webpage is end to server as a global request object

Form	Dictionary object containing key-value pairs of from parameters and values
Args	Parsed contains of query string which is part of URL after question mark (?)
Cookies	Dictionary object holing Cookie names and values
Files	Data pertaining to uploaded file
Method	Current request method

Flask – Cookies

Flask – Cookies

Cookie is stored on client's machine. And helps with data tracking

```
@app.route ( ' / ' )
Def index ():
      return render_template ('index.html')
```

```
<html>
   <body>

            <from action = "/setcookie" method = "post">
               <p><h3 Enter userID</h3></P>
               <p><input type = "text" name = "nm" /></p>
               <p><input type = "submit value = "Login" /></p>
         </form>

      </body>
</html>
```

```
@app.rote( ' /setcookie', methods = ['POST', 'GET'])
def setcookie():
    if request.method == 'POST':
    user = request.form ['nm']

    resp = make_response(render_template('readcookie.html'))
    resp.set_cookie(userID' , user)

    Return resp
```

```
@app.route( ' /getcookie')
Def getcookie():
   name = request.cookies.get('userID')
   return '<h1>welcome ' +name+'</h1>'
```

Flask – Redirect & Errors

Flask – Redirect & Errors

Flask Class has a redirect() function which returns a response object

Prototype → Flask.redirect(location, statuscode, response)

- URL where response should be redirected
- Statuscode send to browser's header
- Response parameter used to instantiate response

Flask – Redirect & Errors

Standardized status codes

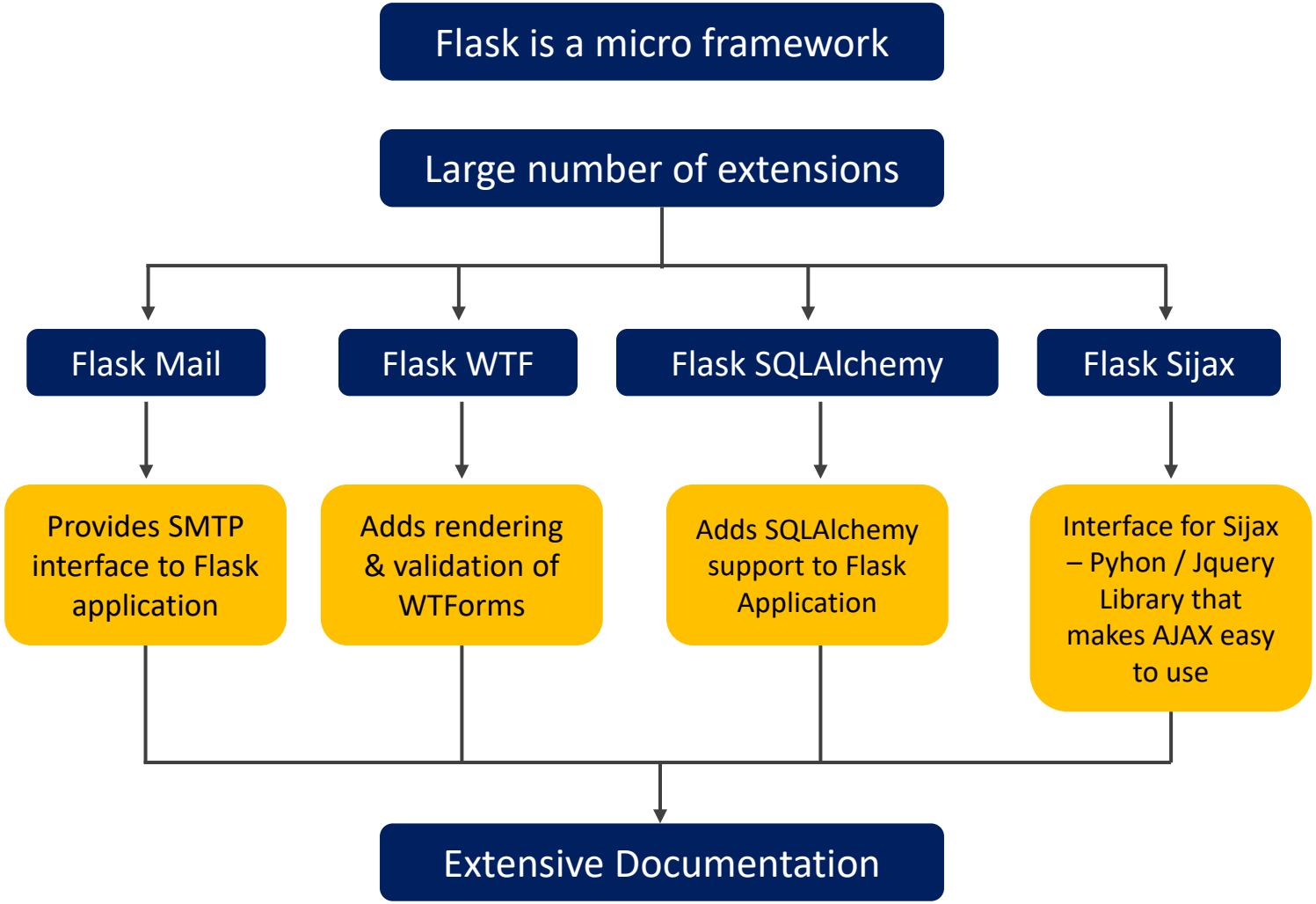
Prototype → Flask.abort(code)

Sl. No	Status Code
1	HTTP_300_MULTIPLE_CHOICES
2	HTTP_301_MOVED_PERMANENTLY
3	HTTP_302_FOUND
4	HTTP_303_SEE_OTHER
5	HTTP_304_NOT_MODIFIED
6	HTTP_305_USE_PROXY
7	HTTP_306_RESERVED

Sl. No	Code	Description
1	400	Bad Request
2	401	Unauthenticated
3	403	Forbidden
4	404	Not Found
5	406	Not Acceptable
6	415	Unsupported Media Type
7	429	Too Many Requests

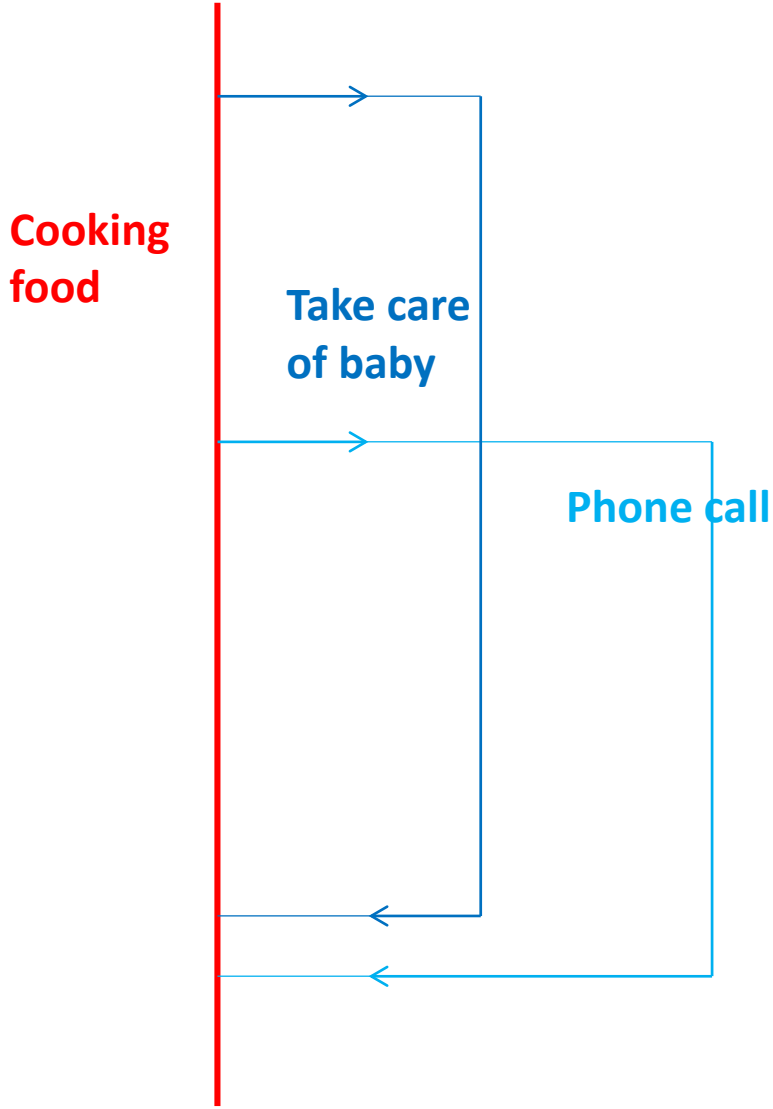
Flask – Extensions

Flask – Extensions

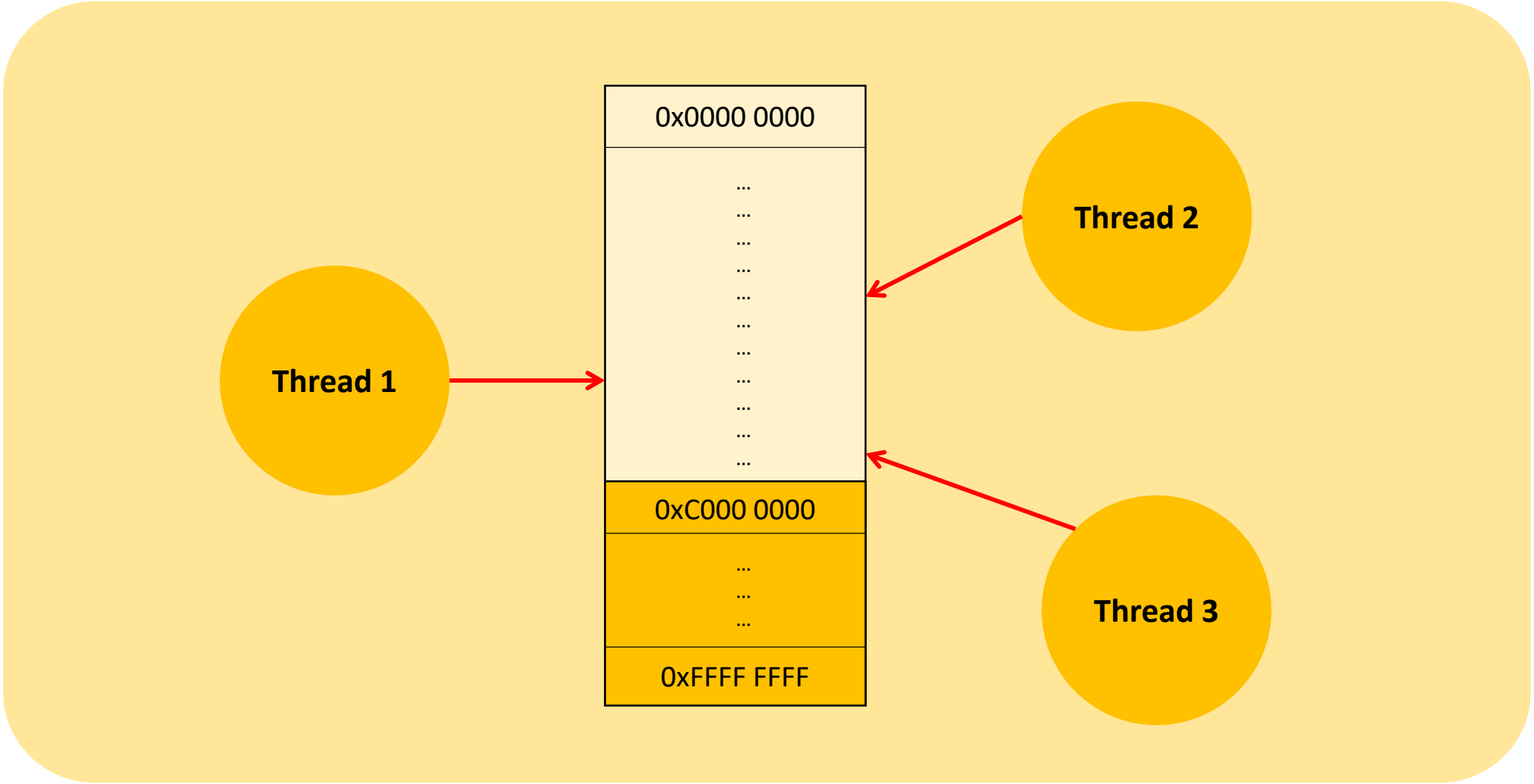


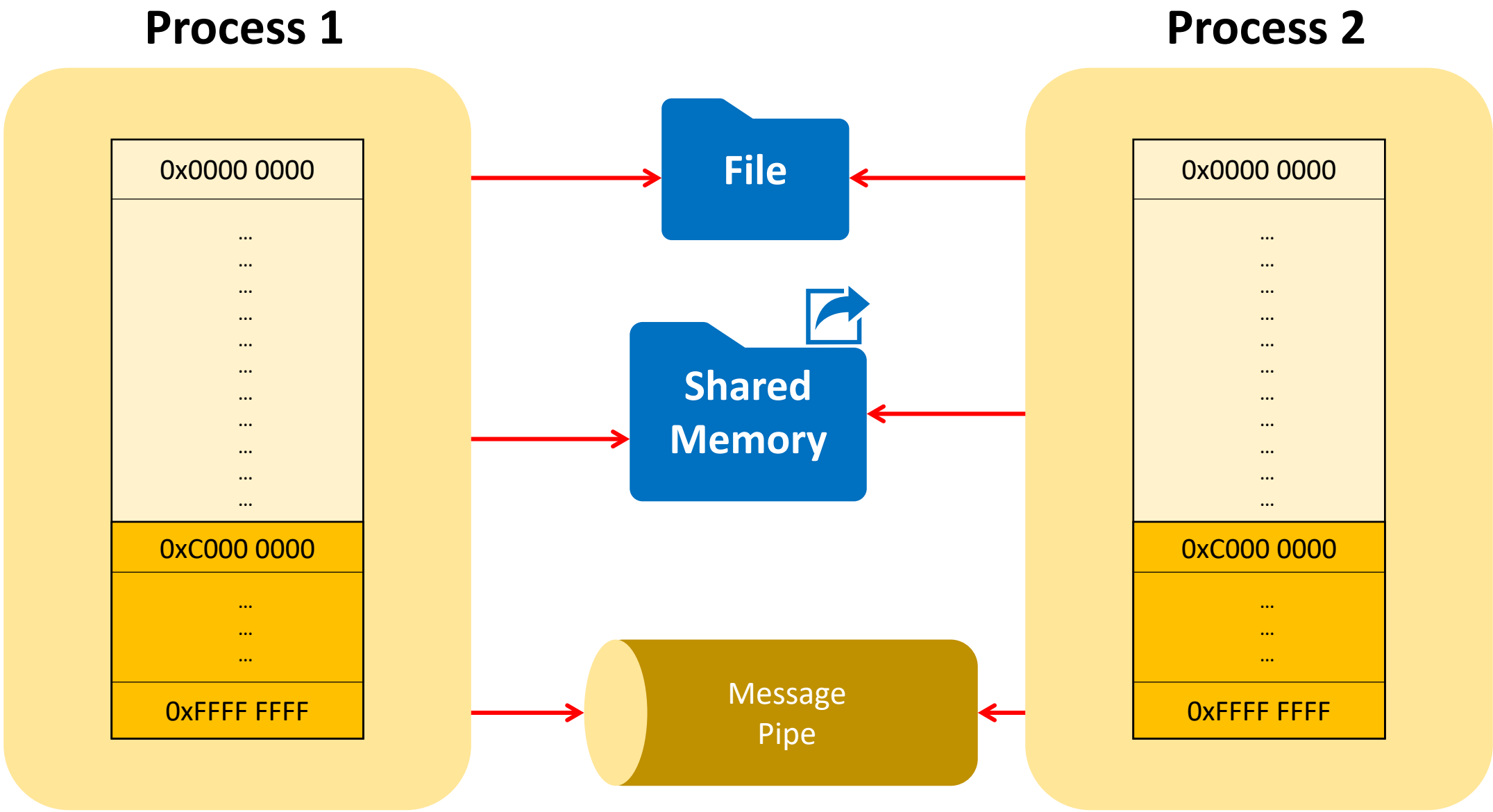
Multiprocessing and Multithreading

Multiprocessing and Multithreading



Process


0x0000 0000
...
...
...
...
...
...
...
...
...
...
...
0xC000 0000
...
...
...
0xFFFF FFFF
Thread 1
Thread 2
Thread 3


Process 1
Process 2
0x0000 0000
...
0xC000 0000
...
0xFFFF FFFF
File
Shared Memory
Message Pipe
0x0000 0000
...
0xC000 0000
...
0xFFFF FFFF

Multiprocessing and Multithreading

Difference between thread and process

Multiprocessing and Multithreading

Difference between thread and process

The benefit of multiprocessing is that error or memory leak in one process won’t hurt execution of another process