Introduction to Python

Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

Python

- A popular programming language; source code file saved with **.py extension**
- Created by Guido van Rossum, and first released in 1991
- S**erver-side scripting language** used for
  - Creating web applications
  - Creating workflows alongside software
  - File Handling (Reading and modifying files)
  - Database Handling (Connecting to database systems; Querying & Updating Information)
  - Handling big data and perform complex mathematics
  - Rapid prototyping, or for production-ready software development.
- Latest stable version Latest Python 3 Release - Python 3.12.6

(Python Software Foundation and community groups)

Python: Advantages

- **Easy to learn, Open Source**
  - **Simple syntax** similar to the English language.
  - Fewer lines of code compared to other programing languages
- **Platform Independent**
  - Available for different platforms - Windows/ Linux/ Solaris (Unix) / Mac / Raspberry PI and others : can be developed in one OS & executed in another
  - Binary platform independent (avoid specific modules for compatibility issues)
- Runs on an **interpreter** system, so code can be executed as soon as it is written (Quick prototyping)
- Supports **multiple programming paradigms**: procedural, object-oriented
- **Feature Rich:** Dedicated modules/libraries for data science, NLP, Graphics, Web Development
- Large Online Community of Programmers who contribute to forums, posts, resources

A Quick Comparison with other languages

- Designed for readability, and has some similarities to the English language with influence from mathematics
  - **Indentations in code, Less punctuations**
- Uses new lines to complete a command
  - **Other programming languages often use semicolons or parentheses**
- Relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes
  - **Other programming languages often use curly-brackets for this purpose**

```
if 5 > 2:
        print("Five is greater than two!")
```

```
if 5 > 2:
print("Five is greater than two!")

--SYNTAX ERROR (indentation)
```

Popular IDEs with Python

- Integrated Development Environment (IDE)
  - Software application for building other applications- combines common developer tools into a single GUI, so that programmers can code efficiently
  - Editor, Interpreter/Compiler, Debugger, Build Automation (compiling computer source code into binary code, packaging binary code, and running automated tests)

- **IDLE** (Integrated Development and Learning Environment) is a default editor that accompanies Python

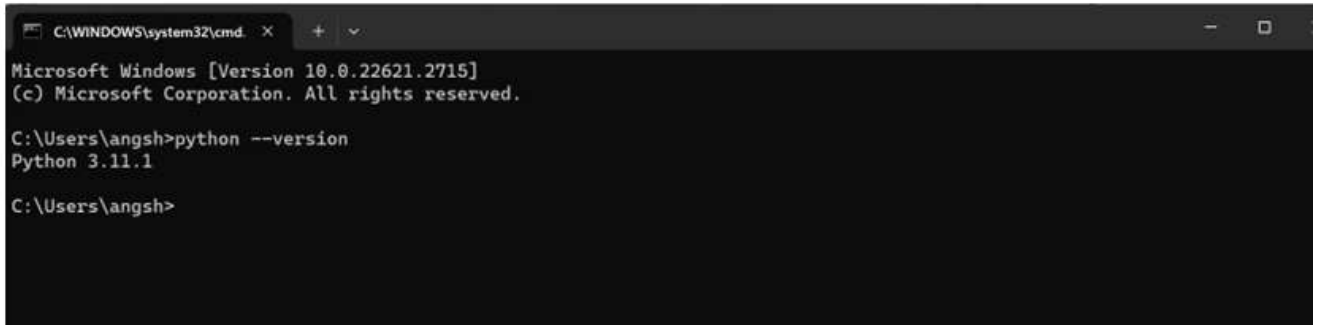
- Spyder, Pycharm, Jupyter, Thonny, Atom, PyDev (included in Eclipse), Netbeans etc. are popular IDEs that are particularly useful when managing large collections of Python files

Pre-Installation (Verify if existing)

First check if Python is already installed on the machine

i) Open Command Line Interface on Windows

ii) Python -- version and press Enter



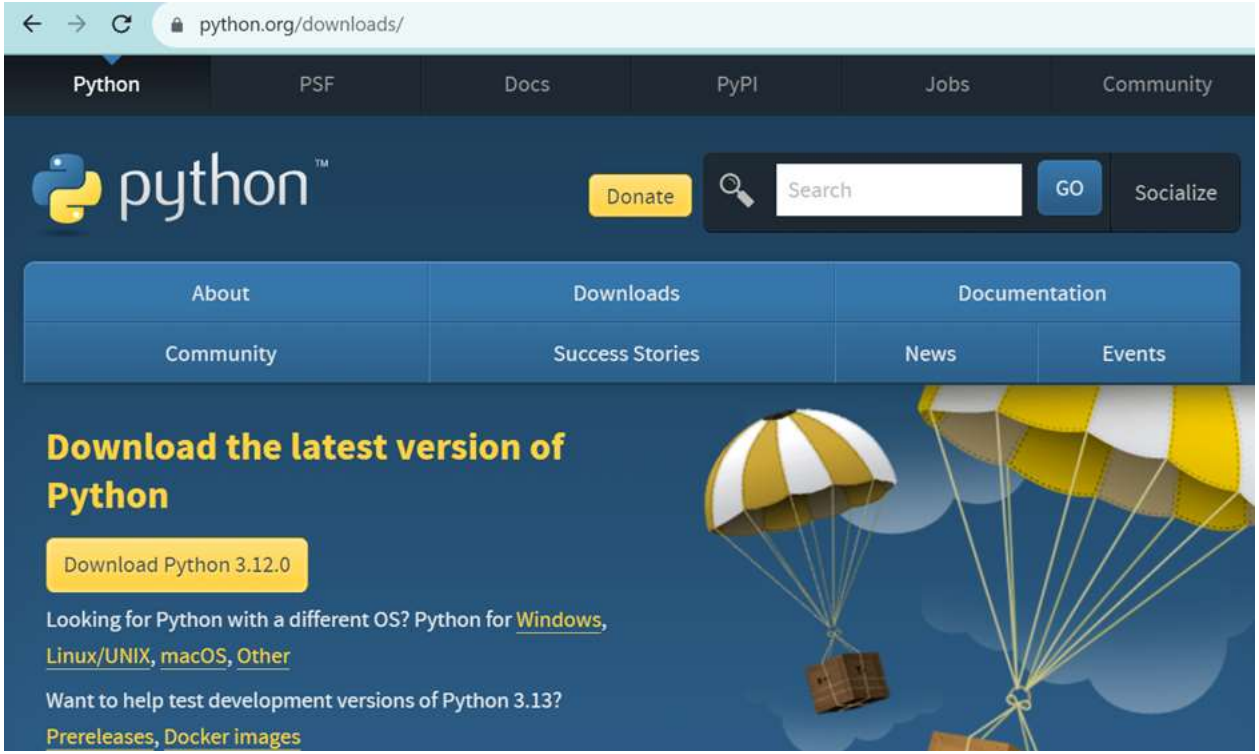
OR Type Python in the Search box – Installed versions will be displayed

Python: Simple Installation

Search for “Download python for windows”

You will be led to **python.org/download**

Download Python 3.12.0



Python Shell (interactive mode)

- IDLE, the default IDE for Python contains Python Shell (interactive interpreter) and Python Editor(to work in script mode)
- Type “python” or just “py” at the Windows command to open the Python shell

```
>>> print("Hello, World!")
Hello World

>>> 18 + 5
23

>>> 27 /5
5.4

>>> 27 // 5
5

>>> 27.0 // 5
5.0

>>> 27.0 % 5
2
```

To run a python program from Windows Command Line Interface:

- Python HelloWorld.py

```
#Program to print
print("Hello, World!")
```

Use the function exit() to exit from Python Shell

Operator Precedence

()	Brackets
******	Exponentiation
+x, -x, ~x (Unary)	Unary
/, //, *, %	Arithmetic
+, -	
<<, >>	
&	Bitwise
^	
|	
==, <, >, <=, >=, !=	Relational
not	
and	
or	Logical

Decreasing Order

Evaluate:

i) 7**2 // 9%3

ii) 10!=9 and 29>=29 and 'bye'<'Bye'

Variables

- Naming Rules:
  - Must start with a letter or the underscore character
  - Can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
  - Cannot be any keyword (true, false, def, break, if, else, not etc.)

- Case-sensitive (age and Age are two different variables)
- No command for declaring a variable
  - **It is created the moment you first assign a value to it**

Dynamic loose typing → associates a data type to a variable based on its value (Variables do not need to be declared with any particular type, and can even change type after they have been set)

```
y = 5          #y  is of type int
y="Sam"        #y  is of type string
```

```
x = 17
y = "Rani"
print(type(x))
print(type(y))
```

Note: Multiline comments written with three quotes at beginning & end of block

Assignment of Values to Variables

Variable = Expression

- Python allows assign values to multiple variables in one line:

  x, y, z = 10, 20, 30

  msg, day, time = 'Meeting', 'Mon', '9'

  (Make sure number of variables match with number of values)

- Same value can also be assigned to multiple variables

  x = y = z = 10

- While initializing string variables, a string literal can be enclosed within single, double or triple quotes

```
print (""" Hello
what's
Happening""")
```

Local & Global Variables

When declared inside a function, a variable is local in scope.

'global' keyword is used to make it global

```
x = "lovely"

def myfunc():
  global x
  x = "wonderful"

myfunc()

print("Python is " + x)
```

Output in Python

print function

```
print('Welcome to', end=' ')
print('Python', end=' ')
print('Programming')

 Welcome to Python Programming
```

```
var1,var2,var3=100, 200, 300
print(var1,var2,var, sep=':')
print(var1,var2,var, sep='---')

100:200:300
100---200---300
```

Input in Python

Input function number1=input("Enter the first number: ")

```
salary=int(input("Enter the salary:  "))
side= float(input("Enter the side of the square: "))
```

Note:

i) Casting functions like int, float, str may be used to specify the data type of the variable being assigned the input

ii) eval() function may be used while taking input, which considers data type based on input provided

```
val1=eval(input('Enter the value of the variable'))
print('val1 =',val1, ' type= ', type(val1))
```

Script Mode

- In interactive mode of IDLE (Python Shell), all computations are lost once we exit from the shell
- Not convenient for most tasks
- Another way – Store all instructions necessary for a task in a .py file
- On clicking 'New Window' option of IDLE, the Python Editor opens

```
number1=input("Enter the first number: ")
number2=input("Enter the second number: ")
print("numbers are ", number1, number2)
```

- After typing the instructions, save the file with .py extension using the 'Save As' option from the file menu

Object Oriented Programming using Python

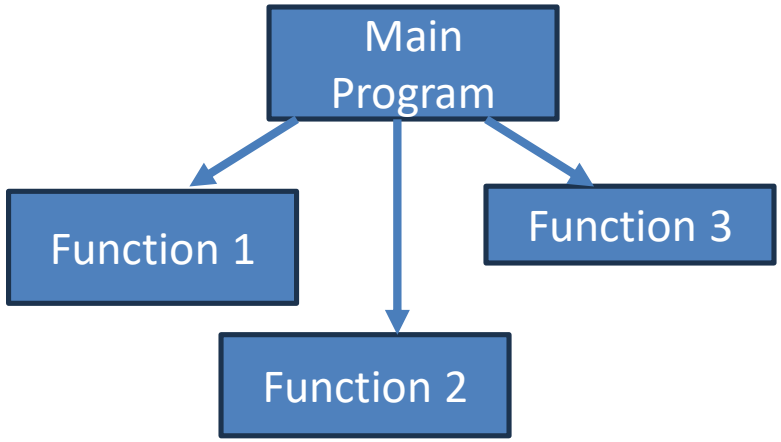
Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

- Procedural vs Object Oriented Programming
- Classes and Objects
- Principles of Object Oriented Programming
  - Inheritance
  - Polymorphism
  - Encapsulation
  - Abstraction

Procedural vs. Object-Oriented Programming

Procedural Programming

- Traditional way of programming
- Consists of writing a list of instructions or actions for the computer to execute sequentially
- Instructions is organized into groups known as **functions**



Features of Procedural Programming

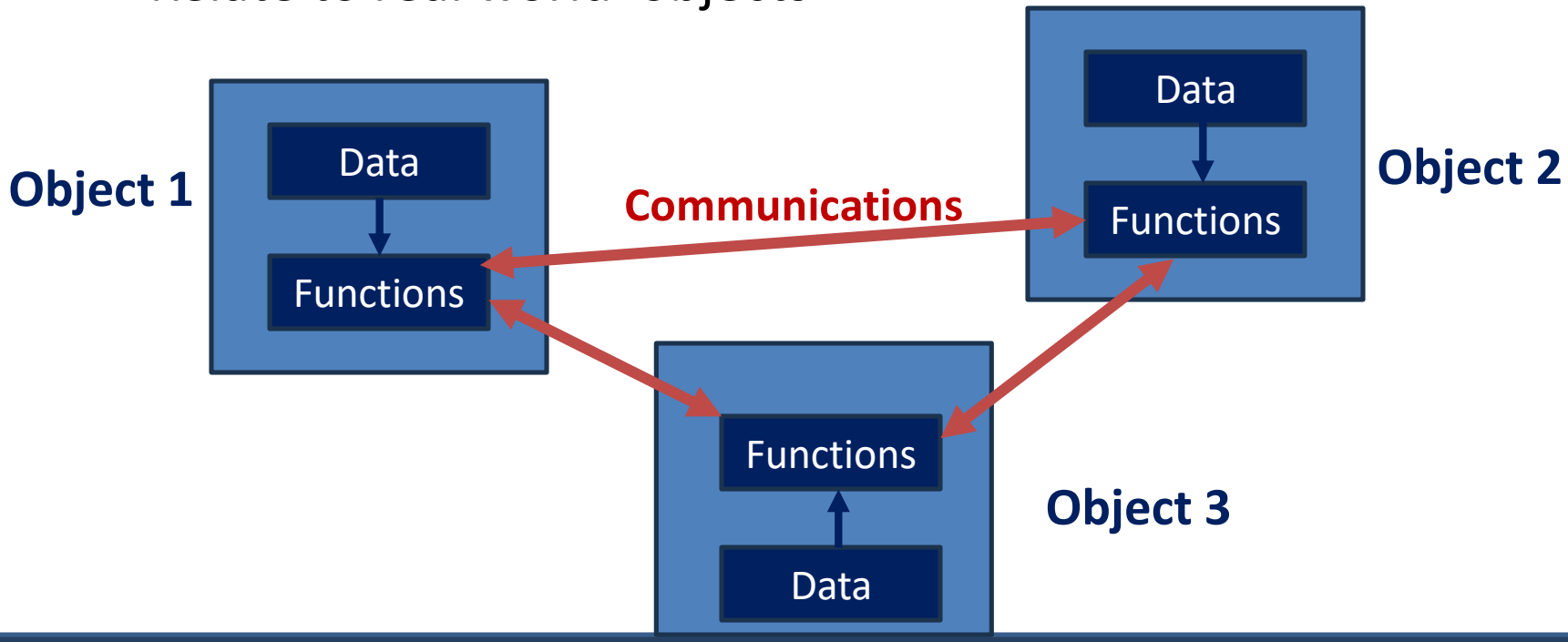
- Emphasis is on doing things(algorithms)
- Employs **top-down approach** in program design
  - Program (larger entity) are divided into smaller **functions**
- Most of these functions **share global data**
- Function transforms data from one form to another
- Data move openly around the system from function to function

Disadvantage

- Global Data- accessed by multiple functions without any data hiding
- Does not mimic the real world very well

Object Oriented Programming

- A programming model that organizes software design around **objects** rather than functions
- Programs are made of **objects** that interact with each other
- Objects consist of both data and methods
- Relate to real world objects



Features of OOP

- Ability to simulate the real world and its events more effectively using objects
- **Data structures** are designed such that they **characterize the objects**
- **Functions** that operate on the data of an object are **tied together in the data structure**
- **Data is hidden** and cannot be accessed directly by external functions
- Objects may communicate with each other through function calls
- New data and functions can be easily added
- Follows **bottom-up approach** in program design
- Code is reusable – less code to be written
- Programmers are able to produce faster, more accurate and better written applications

Procedural vs Object Oriented Programming

Procedural

- Program is divided into small parts called **functions**
- Importance not given to data, but to functions and sequence of actions
- Top Down Approach
- Global Data - Data moves freely from function to function
- It does not have any access specifier
- No data hiding; less secure
- Adding new data and functions is not easy

Object Oriented

- Program is divided into parts called **objects**
- Importance is given to data, rather than procedures or functions
- Bottom-Up Approach
- Data cannot move easily from function to function, it can be kept public or private inside an object (controlled data access)
- Has access specifiers named Public, Private, Protected
- Provides data hiding; more secure
- Adding new data and functions is easy

Classes & Objects

What are Classes & Objects?

- Class is one of the building blocks of OOP
- Acts as the template or "blueprint" that defines the characteristics and behaviour of a collection of objects
- Can also be considered as a prototype from which objects are created
- Class is defined using the **class** keyword

Example

```
class Class1:  //class1 is the name of the class
   x=6         //x is a property of the class (characteristic)
```

- Object Instance: a concrete object that is created from the class "blueprint"

```
p1 = Class1()
print(p1.x)
```

Object Methods

- Methods in objects are functions that belong to the object

Example

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

  def func1(self):
    print("Hello my name is " + self.name)

p1 = Person("Raj", 40)
p1.func1()        //Output → Hello my name is Raj
```

Self parameter

- The **self** parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class (Refer to previous slide)

- Does not have to be named self , but has to be the first parameter of any function in the class (abc, xyz etc.)

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

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

p1 = Person("Raj", 40)
p1.myfunc()          //Output → Hello my name is Raj
```

__init__() function

- All classes have a function called **__init__(),** which is always executed when the class is being initiated

- Used to assign values to object properties, or other operations that are necessary to do when the object is being created

```
class employee:
  def __init__(self,name,age,id,salary):
      self.name = name
      self.age= age
      self.salary= salary
      self.id = id

emp1 = employee("raja",28,123,30000) //creating objects
emp2 = employee("arun",23,223,25000)
print(emp1.name) //accessing properties of the object
```

Modification & Deletion of Object

Modify Object Properties

```
emp1.age  = 35  //Value of age gets modified to 35
```

Delete Object Properties

```
del emp1.age   //Delete the "age" property
```

Delete Object

```
del emp1       //Delete emp1 object
```

Pass statement

- class definitions cannot be empty
- If for some reason there is a class definition with no content, put in the pass statement to avoid getting an error

```
class employee:
      pass
```

Control flow: if

```
x = 5
if x < 0:
   x = 0
   print 'Negative changed to zero'
elif x == 0:
   print 'Zero'
elif x == 1:
   print 'Single'
else:
   print 'More'
```

Control flow: for

```
a = ['cat', ‘donkey', ‘sheep']
for x in a:
  print x, len(x)
```

Loops: break, continue, else

- `break` and `continue` like C
- `else` after loop exhaustion

```
for n in range(2,10):
  for x in range(2,n):
    if n % x == 0:
      print (n, 'equals', x, '*', n/x)
      break
  else:
    # loop fell through without finding a
  factor
    print (n, 'is prime‘)
```

Control flow: while

```
while 1:
    pass
```

```
while condition:
    statements
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series up to
  n."""
  a, b = 0, 1
  while b < n:
    print b,
    a, b = b, a+b

>>> fib(2000)
```

Defining functions

```
def fib(n):
  """Print a Fibonacci series of n ."""
        a, b = 0, 1
        while n > 0:
                print (b)
                a, b = b, a+b
                n = n-1
print(fib(5))
```

Defining functions (Contd.)

```
def gcd(a, b):
    "greatest common divisor"
    while a != 0:
        a, b = b%a, a    # parallel assignment
    return b
>>> gcd(12, 20)
4
```

Features of Object-Oriented Programming

Features of OOP

- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

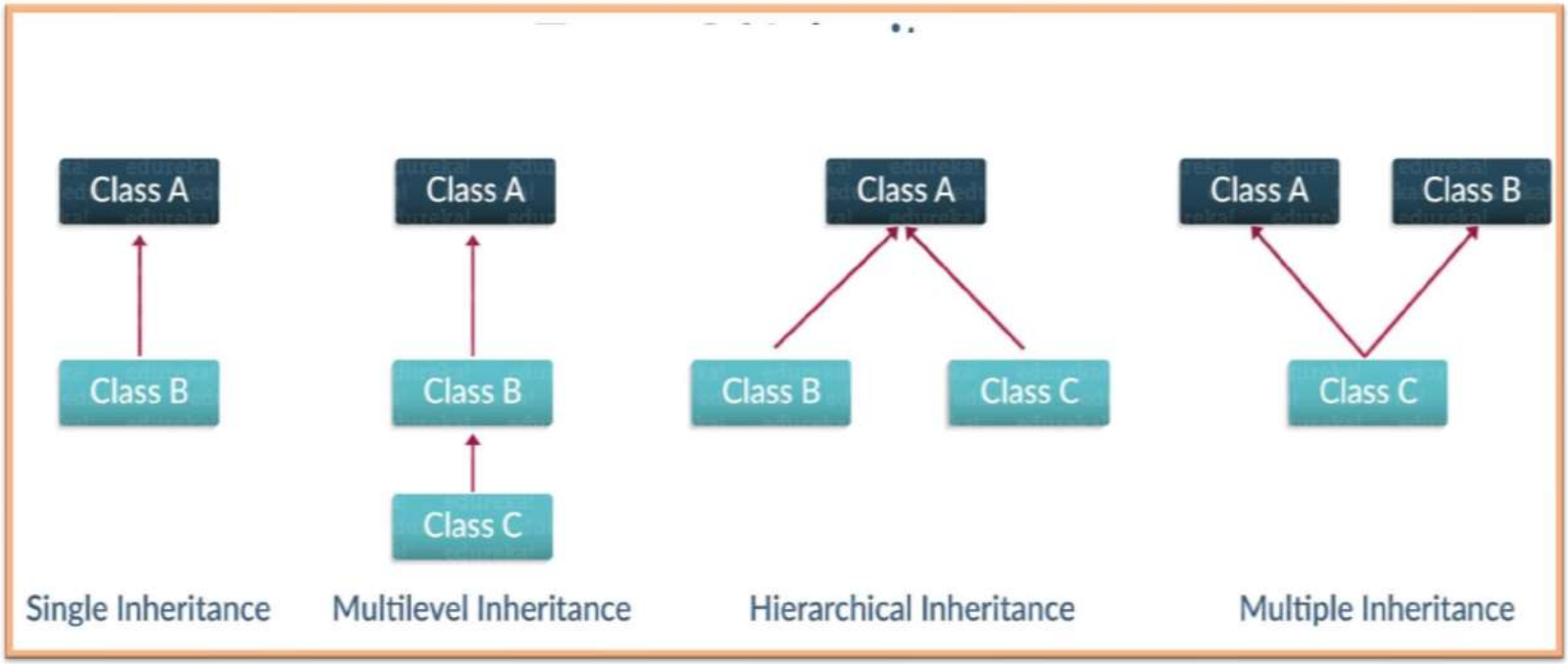
Inheritance

- Process by which one object can acquire the properties of another
- When the child class inherits from the parent class,
  - the **child** is referred to as a **derived class (sub-class)**
  - the **parent** is the **base class (superclass)**
- Child class has two parts:
  - **Derived part** – inherited from parent
  - **Incremental part** – new code written specifically for the child

Syntax:

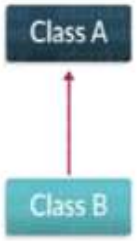
```
class BaseClass:
    Body

class DerivedClass(BaseClass):
    Body
```

Types of Inheritance



Single Inheritance

- Enables a derived class to inherit characteristics from a single parent class




```
# Base class
class Parent:
    def f1(self):
        print("This function is in parent class.")

# Derived class
class Child(Parent):
    def f2(self):
        print("This function is in child class.")


# Driver's code
object = Child()
object.f1()                    → output?
object.f2()                    → output?
```

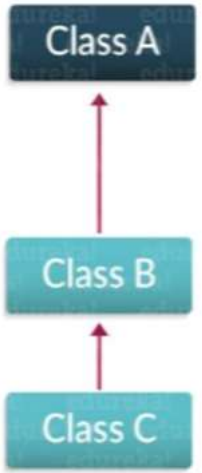
Single Inheritance (contd.)

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def eat(self):
         print("The animal is eating...")

 class Cat(Animal):
    def meow(self):
        print("Meow Meow!")

c =Cat("Pussy", 3)
print(c.name)   # Output: Pussy
c.eat()         # Output: The animal is eating...
c.meow()        # Output: Meow Meow!
```

Multi-level Inheritance

- A type of inheritance where a derived class is created from another derived class.

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

class Cat(Animal): # Intermediate Class: Inherits the Base C
    def __init__(self, name, age, sound):
        super().__init__(name, age)
        self.sound = sound

class Persian(Cat): # Derived Class: Inherits the Intermediate Class
    def __init__ (self, name, age, sound, ears):
        super().__init__(name, age, sound)
        self.ears = ears

c =Persian("Pussy", 3, "Meow", "Small")
print(c.name)  # Output: Pussy
print(c.sound)  # Output: Meow
print(c.ears)  # Output: Small
```



super() function in Python

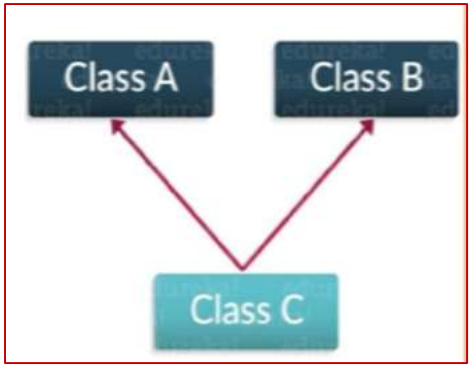
- Note the use of super() function in the previous example
- Provides us the facility to refer to the parent class explicitly
- Returns the proxy object that allows us to refer to the parent class using **'super'**.
- Makes work easier when working with derived classes / inheritance

```
super().__init__(name, age)
```

Multiple Inheritance

- A type of inheritance where a derived class inherits properties from two or more base classes
- In this example, the class “Bat” is derived from two base classes “Bird” and “Mammal”

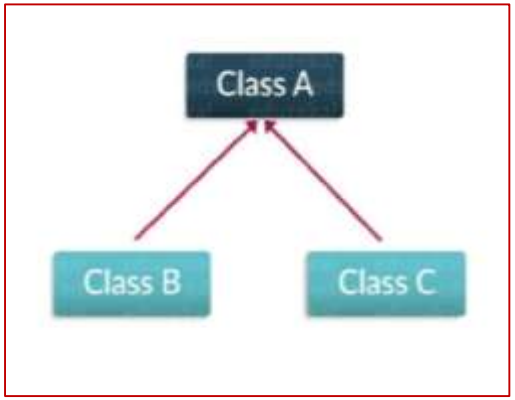
```
class Bird:
    def fly(self):
        print("The bird is flying...")

class Mammal:
    def run(self):
        print("The mammal is running...")

class Bat(Bird, Mammal):
        pass

b = Bat()
b.fly()       # Output: The bird is flying...
b.run()       # Output: The mammal is running…
```



Hierarchical Inheritance

- A type of inheritance where multiple derived classes inherits properties from a base class
- The class “Dog” and “Cat” are derived from the class “Animal”
- Both inherit the properties “name” from Animal

```
class Animal:
    def __init__(self, name):
        self.name = name
    def eat(self):
        print("The animal is eating...")
```




```
class Cat(Animal):
    def meow(self):
        print(“Meow Meow!")

c = Cat(“Pussy")
```

```
class Dog(Animal):
    def bark(self):
        print(“Woof woof!")

d = Dog(“Roger”)
```

Benefits of Inheritance

- Represents real-world relationships well
- Provides **reusability** of a code and also directly facilitates **extensibility** within a system: allows us to add more features to a class without modifying it
- Avoids code duplication
- Transitive in nature: If class B inherits from another class A, then all the subclasses of B would automatically inherit from class A
- Offers a simple, understandable model structure
- Less development and maintenance expenses result from an inheritance

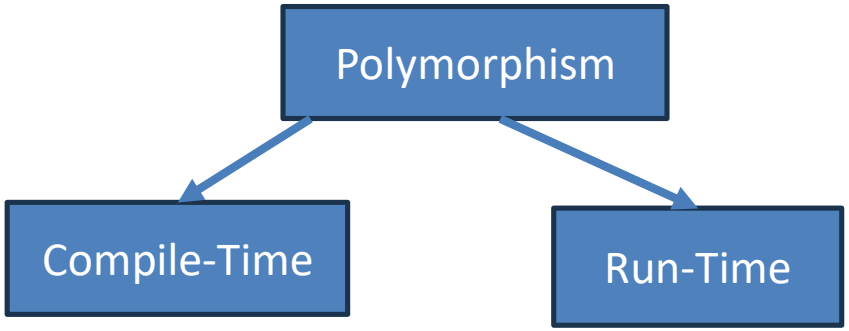
Disadvantages of Inheritance

- **Tight coupling**: Inheritance makes a firmly coupled relationship among classes, making it troublesome to adjust existing code without breaking other program parts

- **Increased complexity:** Overuse may lead to overcomplicated designs; implementations may become difficult

Polymorphism

- Polymorphism – Many forms
- Objects can alter their appearance or behavior depending on the circumstance in which they are used
- One task can be performed in different ways
- Thus “polymorphism” is that property of an object which allows it to take multiple forms



Compile-Time Polymorphism

- Static polymorphism which gets resolved during the compilation time of the program
- Primarily achieved through **method/function overloading**, although Python does not support true function overloading

```
def add(x, y, z = 0):
        return x + y + z

# Driver code
print(add(1, 4))
print(add(1, 4, 5))

Output
5
10
```

```
def add(x, y):
        return x + y

def add(x, y, z):
        return x + y + z

print(add(2, 3))

# Error: Only the latest
 defined function is available
```

Compile-Time Polymorphism

- Inbuilt polymorphic functions available e.g., **len()** function
- For strings len() returns the number of characters

```
x = "Hello!"
print(len(x))
```

- For lists or tuples, it returns the number of items in the list/tuple

```
print(len([20, 40, 60]))
```

```
mytuple = ("apple", "banana", "cherry")

print(len(mytuple))
```

- For dictionaries, len() returns the number of key/value pairs in the dictionary

Operator Overloading

- Another example of compile time polymorphism
- Python supports operator overloading by defining special methods with double underscores (e.g., __add__, __sub__, __eq__, etc.)
- These methods allow you to define how operators should behave for objects of your class.

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

v1 = Vector(1, 2)
v2 = Vector(3, 4)
v3 = v1 + v2
# Calls the __add__ method, resulting in v3 = Vector(4, 6)
```

Run-Time Polymorphism

- Dynamic polymorphism which gets resolved during the run time of the program
- Typically achieved through **method overriding**
- Methods in a subclass provide specific implementations – the method that gets called during run-time depends on the actual object's type

```
class Animal:
    def make_sound(self):
        print("Animal makes a sound")

class Cat(Animal):
    def make_sound(self):
        print("Cat purrs")

my_animal = Cat()
my_animal.make_sound()  # Calls the overridden method in Cat class
```

Interface Polymorphism

- If an object behaves like a particular interface (has the required methods and attributes), it can be treated as an instance of that interface

```
class Bird:
    def fly(self):
        pass

class Sparrow(Bird):
    def fly(self):
        print("Sparrow flies")

class Airplane:
    def fly(self):
        print("Airplane flies")
```

Interface Polymorphism

```
def perform_flight(flying_object):
    flying_object.fly()

sparrow = Sparrow()
airplane = Airplane()

perform_flight(sparrow)  # Output: Sparrow flies
perform_flight(airplane)  # Output: Airplane flies
```

Benefits of Polymorphism

- Reduces the number of lines of code and thus makes it simpler to maintain
- Permits for the execution of more generic algorithms.
- Permits the execution of more adaptable programs
- Permits for the execution of dynamic dispatch and the implementation of interfaces

Disadvantages of Polymorphism

- **Trouble investigating:** Can make it challenging to investigate code; Can be hard to track the stream of execution when the same code is executing in several ways

- **Execution issues:** Can lead to execution issues, as the framework must check each object to decide which strategy to execute

- **Superfluous complexity:** Can lead to pointless complexity; polymorphism may have been needlessly utilized when a less complex approach would suffice

Abstract Class

```
#Abstract Class
class Vehicle:
    def start(self, name=""):
        print(name, "is Started")
    def acclerate(self, name=""):
        pass
    def park(self, name=""):
        pass
    def stop(self, name=""):
        print(name, "is stopped")

class Bike(Vehicle):
    def acclerate(self, name=""):
        print(name, "is accelerating @ 60kmph")
    def park(self, name=""):
        print(name, " is parked at two wheeler
parking")

class Car(Vehicle):
    def acclerate(self, name=""):
        print(name, "is accelerating @ 90kmph")
    def park(self, name=""):
        print(name, "is parked at four wheeler
parking")
```

```
def main():
    print("Bike Object")
    b=Bike()
    b.start("Bike")
    b.acclerate("Bike")
    b.park("Bike")
    b.stop("Bike")

    print("\n Car Object")
    c = Car()
    c.start("Car")
    c.acclerate("Car")
    c.park("Car")
    c.stop("Car")
if __name__=="__main__":
     main()
```

Output

Bike Object

Bike is Started

Bike is accelerating @ 60kmph

Bike is parked at two wheeler parking

Bike is stopped

Car Object

Car is Started

Car is accelerating @ 90kmph

Car is parked at four wheeler parking

Car is stopped

Interface

```
import math

#Interface
class Shape:
    def input(self):
        pass
    def process(self):
        pass
    def output(self):
        pass

class Circle(Shape):
    def __init__(self,rad=0.0):
        self.__radius=rad
        self.__area = 0.0
    def input(self):
        self.__radius=float(input("Enter radius:"))
    def process(self):

 self.__area=math.pi*math.pow(self.__radius,2)
    def output(self):
        print("Area :",self.__area)
```

```
class Rectangle(Shape):
    def __init__(self,len=0,br=0):
        self.__length=len
        self.__breadth=br
        self.__area = 0
    def input(self):
        self.__length=int(input("Enter Length:"))
        self.__breadth = int(input("Enter Breadth:"))
    def process(self):
        self.__area=self.__length*self.__breadth
    def output(self):
        print("Area :",self.__area)

def main():
    print("Circle Object:")
    c=Circle()
    c.input()
    c.process()
    c.output()
    print("\nRectangle Object:")
    r=Rectangle()
    r.input()
    r.process()
    r.output()
if __name__=="__main__":    main()
```

__name__ variable

- The Python interpreter sets the __name__ variable to the name of the module if it is imported and to the string "__main__" if the module is the main entry point to the program.

```
if __name__=="__main__":
        main()
```

- checks if the current script is being run directly as the main program, or if it's being imported as a module into another program.

Encapsulation

- One of the fundamental concepts of Object Oriented Programming (OOP)
- Describes the idea of wrapping **data(attributes)** and the **methods (functions)** that operate on data **within one unit**
- Helps **hide internal state of an object** from outside world, providing a controlled interface for interacting with the object's data and methods

i.e., puts restrictions on accessing variables and methods directly

Encapsulation

- Can prevent accidental modification of data by allowing an object's variable to be changed only by an object's method (**private** variables)
- A class is an example of encapsulation: Encapsulates all the member functions, variables, etc.
- **Goal: information hiding; controlled access to data**

Using Access Modifiers

Encapsulation in Python is implemented using access specifiers to control access to class members:

- **Public Members:** By default, attributes and methods are public and can be accessed from outside the class
- **Protected Members:** Use a single underscore (_) prefix to indicate that an attribute or method is intended for internal use within the class and its subclasses
- **Private Members:** Use double underscores (__) prefix to make an attribute or method private. This leads to name mangling, making it more challenging to access from outside the class.

Example (Using protected member variables)

```
# Python program to demonstrate protected members
# Creating a base class
class Base:
  def __init__(self):
      # Protected member
      self._x = 2

# Creating a derived class
class Derived(Base):
  def __init__(self):
      # Calling constructor of # Base class
      Base.__init__(self)
      print("Access protected member of base class: ", self._x)
       # Modifying the protected member outside base class
       self._x = 5
      print("Accessing modified protected member outside base
             class: ", self._x)
```

Example contd..

```
obj1= Derived()
obj2=Base()
# Accessing protected member
# Can be accessed but should not be done due to convention
print("Accessing protected member of obj1: ", obj1._x)

# Accessing the protected variable outside
print("Accessing protected member of obj2: ", obj2._x)
```

Output

```
Access protected member of base class: 2
Accessing modified protected member outside class: 5

Accessing protected member of obj1: 5
Accessing protected member of obj2: 2
```

Exercise

What will happen if the member variable, x is defined as a private variable in the base class, and you try to modify or access it from a derived class (as shown below)?

```
class Base:
  def __init__(self):
      # Protected member
      self.__x = 2

# Creating a derived class
class Derived(Base):
  def __init__(self):
      # Calling constructor of # Base class
      Base.__init__(self)
      self.__x = 5
      print(self.__x)

#Driver code
 obj1=Derived()
```

Name mangling

- Name mangling is a process by which the interpreter changes the name of a variable in a way that makes it harder to create accidental collisions in subclasses.
- In Python, this is achieved by prefixing the variable name with _ClassName , where ClassName is the name of the class where the variable is defined.
- Accessing private members of a class outside the class
  - We cannot directly access obj.__name, obj.__age, obj.__branch, and call obj.__displayDetails() because they throw errors.
  - Note that in the list of callable fields and methods, __name is saved as _Student__name, __age is saved as _Student__age, __branch is saved as _Student__branch and __displayDetails() is saved as _Student__displayDetails().
  - This conversion is called name mangling, where the python interpreter automatically converts any member preceded with two underscores to _<class name>__<member name>. Hence, we can still access all the supposedly private data members of a class using the above convention.

dir() function

- dir() function returns all properties and methods of the specified object, without the values.
- This function also returns all the properties and methods, even built-in properties which are default for all object.

Abstraction

- One of the fundamental concepts of Object Oriented Programming (OOP)
- Used to hide irrelevant details from the user
- For example, we know the functionalities of a brake, accelerator or clutch in a car, but we do not know their implementation details

- In Python, data abstraction is achieved by using abstract classes

Abstract Classes & Abstract Methods in Python

Abstract Class

- a class in which one or more abstract methods are defined

Abstract Methods

- a method is declared inside the class without its implementation

- To create abstract method and abstract classes, the "**ABC**" and "**abstractmethod**" classes need to be **imported from abc (Abstract Base Class) library**

- Abstract method of base class forces its child class to write the implementation of the all abstract methods defined in base class

Abstract Classes & Abstract Methods in Python

```
from abc import ABC, abstractmethod
class BaseClass(ABC):
        @abstractmethod
        def method_1(self):
                #empty body
                pass
```

- In the below above method_1 is a abstract method created using @abstractmethod decorator
- Any subclass derived from BaseClass must implement this method_1 method.

Note: Any method containing implementation details is called a **concrete method**

Example: Using Abstract Classes & Abstract Methods

```
from abc import ABC, abstractmethod

#Create Base Class
class Car(ABC):
        def __init__(self, brand, model, year):
                self.brand = brand
                self.model = model

        # Create abstract method
        @abstractmethod
        def printDetails(self):
                pass

        # Create concrete method
        def apply_brake(self):
                print ("Car Stopped")
```

Example: Using Abstract Classes & Abstract Methods

```
#Create Child Class
class Hatchback(Car):
        def  printDetails(self):
                print("Brand: ", self.brand)
                print("Model: ", self.model)


# Create an instance of Hatchback Class
car1 = Hatchback("Renault", "Kwid")

# Call methods
car1.printDetails()
car1.apply_brake()
```

Output

```
Brand: Renault
Model: Kwid
Car stpped
```

Importance of Abstraction

- Enables programmers to hide complex implementation details while exposing only essential information and functionalities to users
- Makes it easier to design modular and well-organized code’
- Makes it simpler to understand and maintain
- Promotes code reuse
- Improves developer collaboration

Encapsulation vs Abstraction

Encapsulation

i. Definition

The bundling of data and methods that operate on the data into a single unit, with controlled access to the internal state

ii. Purpose

To protect an object's internal state and expose a controlled interface

iii. Implementation

Achieved through private and protected members

Abstraction

i. Definition

Hiding complex implementation details and showing only the essential features of an object

ii. Purpose

To simplify interaction with objects by focusing on high-level operations rather than implementation details

iii. Implementation

Achieved through abstract classes and methods, interfaces, and high-level class design

Thank You!!

Python Data Types

Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

Built-in Data Types

- Numeric
- Sequence Type
- Boolean
- Set
- Dictionary

Numeric Data Types

- The numeric data type represents data that has a numeric value.
  - Integer (represented by int class)
    - It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
  - Floating value (represented by float class)
    - It is a real number with a floating-point representation. It is specified by a decimal point.
  - Complex number (represented by complex class)
    - It is specified as *(real part) + (imaginary part)j* .

      For example: 2+3j

Example

```
a = 5
print("Type of a: ", type(a))
b = 5.0
print("\nType of b: ", type(b))
c = 2 + 4j
print("\nType of c: ", type(c))
```

Output:

```
Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>
```

Sequence Data Types

- The sequence Data Type in Python is the ordered collection of similar or different Python data types. Sequences allow storing of multiple values in an organized and efficient fashion.
  - String
  - List
  - Tuple
- String
  - A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote.
  - In Python, there is no character data type, a character is a string of length one. It is represented by str class.

String: Example

```
String1 = 'Welcome to the Geeks World'
print("String with the use of Single Quotes: ")
print(String1)
String1 = "I'm a Geek"
print("\nString with the use of Double Quotes: ")
print(String1)
print(type(String1))
String1 = '''I'm a Geek and I live in a world of "Geeks"'''
print("\nString with the use of Triple Quotes: ")
print(String1)
print(type(String1))
String1 = '''Geeks
                For
                Life'''
print("\nCreating a multiline String: ")
print(String1)
```

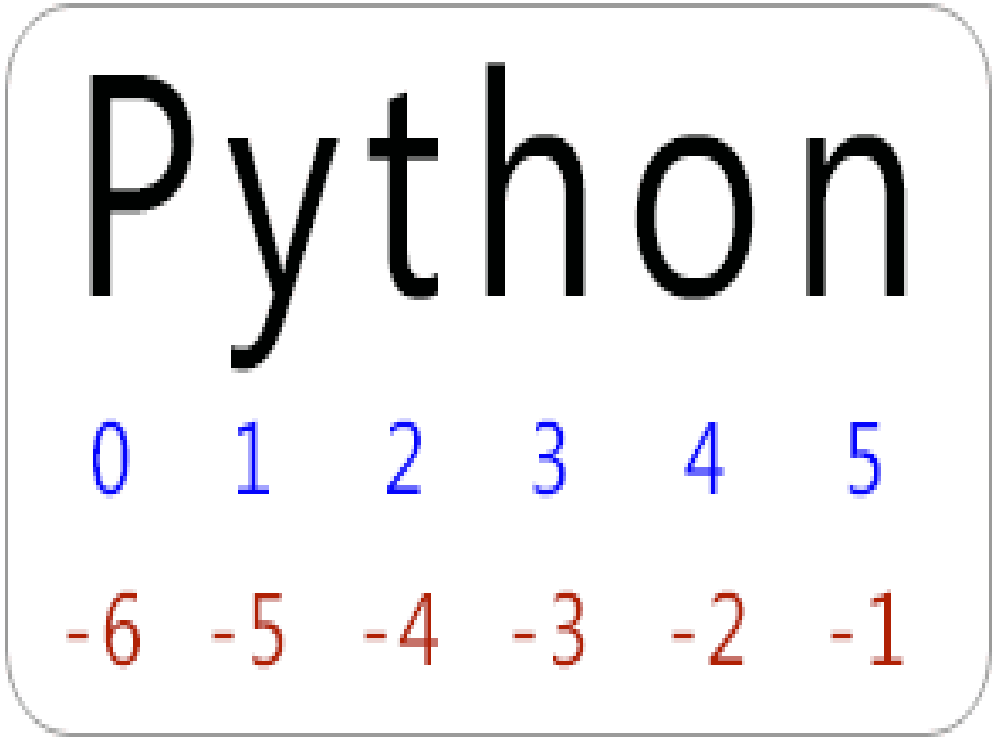
Output

- String with the use of Single Quotes:
  Welcome to the Geeks World
  String with the use of Double Quotes:
  I'm a Geek
  <class 'str'>
  String with the use of Triple Quotes:
  I'm a Geek and I live in a world of "Geeks"
  <class 'str'>
  Creating a multiline String:
  Geeks
  For
  Life

String Length

```
a = "Hello, World"
print(len(a)) # 12
```

String Indexing



Strings are Arrays

- Strings in Python are arrays of bytes representing unicode characters.
- Square brackets can be used to access elements of the string.
- Example:

```
a = "Hello, World!"
print(a[1])
```

Output: ?

Indexing

- Individual characters of a String can be accessed by using the method of Indexing.
- Negative Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

```
String1 = "GeeksForGeeks"
print("Initial String: ")
print(String1)
print("\nFirst character of String is: ")
print(String1[0])
print("\nLast character of String is: ")
print(String1[-1])
```

Output

- Initial String:
  GeeksForGeeks
  First character of String is:
  G
  Last character of String is:
  s

Looping Through a String

- Strings are character arrays. Using a for loop, we can loop through the characters in a string.
- Example:

```
for x in “Hello":
        print(x)
```

Output:

```
H
e
l
l
o
```

Check String

- To check if a certain phrase or character is present in a string, we can use the keyword ”in”.
- Example:

Print only if "Python" is present.

```
txt = “Welcome to Python World!"
if “Python" in txt:
  print("Yes, ‘Python’ is present.")
```

Output: Yes, ‘Python’ is present.

```
print(“Python" in txt)
```

Output: True

Check if NOT

- To check if a certain phrase or character is NOT present in a string, we can use the keyword ” not in”.
- Example:

Print only if “Java" is NOT present.

```
txt = “Welcome to Python World!"
if “Java" not in txt:
  print(“No, ‘Java’ is NOT present.")
```

Output: No, ‘Java’ is NOT present.

```
print(“Java" not in txt)
```

Output: True

String Slicing [1/2]

- String slicing allows you to extract a portion of a string by specifying the start index (included) and the end index (not included), separated by a colon.

  Example: s = “Department"
  print(s[2:5]) # par

- Slice From the Start

  s = “Department"
  print(s[:5]) # Depar

- Slice To the End

  s = “Department"
  print(s[5:]) # tment

String Slicing [2/2]

- Use negative indexes to start the slice from the end of the string
- Example:

```
b = "Hello, World!"
print(b[-5:-2])
```

Basic Operations on String

- Concatenation using '+' operator

  >>> 'abc' + 'def'

  'abcdef'

  >>> 'abc' + " " + 'def'

  'abc def'

- Repetition using '*' operator

  >>> 'Nil' * 3

  'NilNilNil'

- To print a line of 80 dashes

  >>> print ('------....more.....-----')

  >>> print('-' * 80)

Modify strings [1/3]

- upper() method returns the string in upper case

Example:

```
a = "Hello, World!"
print(a.upper()) # HELLO, WORLD!
```

- lower() method returns the string in lower case

Example”

```
print(a.lower()) # hello, world!
```

Modify strings [2/3]

- strip() method removes any whitespace from the beginning or the end

Example:

```
a = " Hello, World! "
print(a.strip()) # Hello, World!
```

- replace() method replaces a string with another string

Example:

```
a = "Hello, World!"
print(a.replace("H", "J")) # Jello, World!
```

Modify strings [3/3]

- The split() method returns a list where the text between the specified separator becomes the list items.

Example:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

String Methods [1/3]

- All string methods return new values. They do not change the original string because it is **immutable** in nature.

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count(value, start, end)	Returns the number of times a specified value occurs in a string
find()	Searches the string for a specified value and returns the position of where it was found

String Methods [2/3]

Method	Description
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdigit()	Returns True if all characters in the string are digits
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isupper()	Returns True if all characters in the string are upper case

String Methods [3/3]

Method	Description
lower()	Converts a string into lower case
replace()	Returns a string where a specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
strip()	Returns a trimmed version of the string
title()	Converts the first character of each word to upper case
upper()	Converts a string into upper case

List

- List is an ordered collection of data.
- It is very flexible as the items in a list do not need to be of the same type.
- Lists can be created by placing the data inside the square brackets[].

```
List = []
print("Initial blank List: ")
print(List)
List = ['GeeksForGeeks']
print("\nList with the use of String: ")
print(List)
List = ["Geeks", "For", "Geeks"]
print("\nList containing multiple values: ")
print(List[0])
print(List[2])
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output

- Initial blank List:
  []
  List with the use of String:
  ['GeeksForGeeks']
  List containing multiple values:
  Geeks
  Geeks
  Multi-Dimensional List:
  [['Geeks', 'For'], ['Geeks']]

Create a List

- Lists are used to store multiple items in a single variable
- Lists are created using square brackets
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.

```
myList = [“Ram", “Shyam", “Ranit"]
print(myList)
```

Allow Duplicates

```
myList = [“Ram", “Shyam", “Ranit“, “Shyam" ]
print(myList)
```

Length of List

```
myList = [“Ram", “Shyam", “Ranit"]
print(myList)

Print(len(myList)) # 3
```

Counting Appearance of an Item

- Return the number of times the value "cherry" appears in myList

```
myList = ['apple', 'cherry', 'banana', 'cherry']
print(myList.count("cherry"))
```

Data Types in List

- List items can be of any data type
- A single list may contain different types of data

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
list1 = ["abc", 34, True, 40, "male"]
print(type(list1)) # <class 'list'>
```

Accessing List Items

```
myList = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(myList[1]) # "banana"
print(myList[1:3]) # "banana", "cherry“
print(myList[4:]) # "kiwi", "melon", "mango“
print(myList[:2]) # "apple", "banana“
print(myList[-1]) # "mango“
print(myList[-3:-1]) # "kiwi", "melon"
```

Change Item Value

myList = ["apple", "banana", "cherry”]

- Change a particular item

myList[2]=“coconut” # ["apple", "banana", “coconut”]

- If you insert more items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly

myList[1:2]=[“mango”, “orange”] # ["apple", “mango”, “orange”, “coconut”]

- If you insert less items than you replace, the new items will be inserted where you specified, and the remaining items will move accordingly

myList[1:3]=[“guava”] # ["apple", “guava”, “coconut”]

Insert Items

- To insert a new list item, without replacing any of the existing values, we can use the insert() method
- insert() method inserts an item at the specified index

```
myList = ["apple", "banana", "cherry"]
myList.insert(2, "melon")
print(myList)
#["apple", "banana", “melon”, "cherry"]
```

Append Items

- To add an item to the end of the list, use the append() method

```
myList = ["apple", "banana", "cherry"]
myList.append("orange")
print(myList)

# ["apple", "banana", "cherry“, "orange"]
```

Extend List

```
L1=["A", "B"]
L2=["C", "D"]
L1.extend(L2)
print(L1)
print(L2)

# L1=["A", "B", "C", "D"]
# L2=["C", "D"]
```

Remove Item from List

- Remove Specified Item

```
myList = ["apple", "banana", "cherry“, “orange”]
myList.remove("banana")
print(myList) # ["apple", "cherry“, “orange”]
```

- Remove Specified Index

```
myList.pop(1)
print(myList) # ["apple“, “orange”]
```

- Remove Last Item

```
myList.pop()
print(myList) # ["apple”]
```

- Clear the List

```
myList.clear()
print(myList) # []
```

- Delete list completely

```
del myList
```

Copy a List

```
L1 = ["apple", "banana", "cherry"]
L2 = L1.copy()
Or
L2 = list(L1)
print(L2) # L2= ["apple", "banana", "cherry"]
```

Concatenate, two or more lists

```
L1 = ["a", "b", "c"]
L2 = [1, 2, 3]

L3= L1 + L2
print(L3) # ["a", "b", "c“, 1, 2, 3]
```

Sort Lists

- Sort the list alphabetically in ascending order

```
myList = [“Cat”, “Rat”, “Ant”, “Bat”]
myList.sort()
print(myList) # [“Ant”, “Bat”, “Cat”, “Rat”]
```

- Sort the list numerically in ascending/descending order

```
myList = [90, 55, 65, 87, 25]
myList.sort()
print(myList) # [25, 55, 65, 87, 90]
myList.sort(reverse = True) # Sort in descending order
print(myList) # [90, 87, 65, 55, 25]
```

Loop Through a List

```
# print all list items one by one directly
myList = ["apple", "banana", "cherry"]
for x in myList:
  print(x)
```

Output:

apple

banana

cherry

Loop Through the Index Numbers

```
# print each items of the through the corresponding  index
myList = ["apple", "banana", "cherry"]
for i in range(len(myList)):
  print(myList[i])
```

Output:

apple

banana

cherry

Using while loop

- thislist = ["apple", "banana", "cherry"]
  i = 0
  while i < len(thislist):
    print(thislist[i])
    i = i + 1

Output:

apple

banana

cherry

Tuple

- Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and **unchangeable**.
- Tuples are written with round brackets.

Example:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple)
```

Output:

('apple', 'banana', 'cherry')

Tuple Items

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Ordered
  - Items have a defined order; that order will not change.
- Unchangeable
  - We cannot change, add or remove items after the tuple has been created.
- Allow duplicates
  - Tuples allow duplicate values

Example:

```
thistuple = ("apple", "banana", "cherry", "apple", "cherry")
print(thistuple)
```

Output:

('apple', 'banana', 'cherry', 'apple', 'cherry')

Tuple Length

- len() function is used to determine how many items a tuple has.

Example:

```
thistuple = ("apple", "banana", "cherry")
print(len(thistuple))
```

Output: 3

Create Tuple With One Item

- To create a tuple with only one item, add a comma after the item, otherwise Python will not recognize it as a tuple.

Example:

```
thistuple = ("apple",)
print(type(thistuple)) # <class, 'tuple'>

#NOT a tuple
thistuple = ("apple")
print(type(thistuple))
```

Tuple Items - Data Types

- Tuple items can be of any data type.

Example:

```
tuple1 = ("apple", "banana", "cherry")    #string
tuple2 = (1, 5, 7, 9, 3)                  #int
tuple3 = (True, False, False)             #boolean
```

- A tuple can contain different data types.

Example:

```
tuple1 = ("abc", 34, True, 40, "male")
print(tuple1) #('abc', 34, True, 40, 'male')
Print(type(tuple1)) # <class 'tuple'>
```

tuple() Constructor

- Use the tuple() constructor to make a tuple.

Example:

```
thistuple = tuple(("apple", "banana", "cherry"))
print(thistuple)
```

Output: ('apple', 'banana', 'cherry')

Access Tuple Items

- Tuple items are accessed by using the index number inside square brackets. The first item has index 0.

Example:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[1])    # banana
```

- -1 refers to the last item, -2 refers to the second last item etc. (negative indexing)

Example:

```
thistuple = ("apple", "banana", "cherry")
print(thistuple[-1])   # cherry
```

Range of Indexes

- If you specify a range, the return value will be a new tuple with the specified items.

Example:

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:5])
```

Output: ('cherry', 'orange', 'kiwi')

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[:4])        # ('apple', 'banana', 'cherry', 'orange')
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[2:])        # ('cherry', 'orange', 'kiwi', 'melon', 'mango')
```

```
thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")
print(thistuple[-4:-1])     # ('orange', 'kiwi', 'melon')
```

Check if Item Exists

- use the ”in” keyword

```
thistuple = ("apple", "banana", "cherry")
if "apple" in thistuple:
  print("Yes, 'apple' is in the fruits tuple")

if "apple" not in thistuple:
  print(“No, 'apple' is not in the fruits tuple")
```

Update Tuples

- Change Tuple Values (Tuples are **unchangeable**, or **immutable.)**
  - You can convert the tuple into a list, change the list, and convert the list back into a tuple.

Example:

```
x = ("apple", "banana", "cherry")
y = list(x)
y[1] = "kiwi"
x = tuple(y)

print(x)
```

Output:

('apple', 'kiwi', 'cherry')

Add Items

- Tuples are immutable, they do not have a built-in append() method.
  - Convert it into a list, add item(s), and convert it back into a tuple.

Example:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.append("orange")
thistuple = tuple(y)
print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry', 'orange')
```

Add tuple to a tuple

- If you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple.

Example:

```
thistuple = ("apple", "banana", "cherry")
y = ("orange",)
thistuple += y

print(thistuple)
```

Output:

```
('apple', 'banana', 'cherry', 'orange')
```

Remove Items

- Tuples are **unchangeable**, so you cannot remove items from it.
  - Convert the tuple into a list, remove "apple", and convert it back into a tuple.

Example:

```
thistuple = ("apple", "banana", "cherry")
y = list(thistuple)
y.remove("apple")
thistuple = tuple(y)
```

Output:

```
('banana', 'cherry')
```

  - The del keyword can delete the tuple completely.

```
thistuple = ("apple", "banana", "cherry")
del thistuple
print(thistuple) #this will raise an error because the tuple no longer exists
```

Loop Through a Tuple

Example:

```
thistuple = ("apple", "banana", "cherry")
for x in thistuple:
    print(x)
```

Output:

```
apple
banana
cherry
```

- Print all items by referring to their index number.

Example:

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

Output: ?

Using a While Loop

Example:

```
thistuple = ("apple", "banana", "cherry")
i = 0
while i < len(thistuple):
  print(thistuple[i])
  i = i + 1
```

Output:

```
apple
banana
cherry
```

Join Two Tuples

- Use the + operator to join two or more tuples.

Example:

```
tuple1 = ("a", "b" , "c")
tuple2 = (1, 2, 3)

tuple3 = tuple1 + tuple2
print(tuple3)      # ('a', 'b', 'c', 1, 2, 3)
```

- Multiply Tuples
  - If you want to multiply the content of a tuple a given number of times, you can use the * operator

Example:

```
fruits = ("apple", "banana", "cherry")
mytuple = fruits * 2

print(mytuple) # ('apple', 'banana', 'cherry', 'apple', 'banana', 'cherry')
```

Tuple Methods

- count() method - Returns the number of times a specified value occurs in a tuple

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.count(5)
print(x)    # 2
```

- index() method - Search for the first occurrence of the value, and return its position

```
thistuple = (1, 3, 7, 8, 7, 5, 4, 6, 8, 5)
x = thistuple.index(8)
print(x)      # 3
```

Boolean

- Booleans represent one of two values: True or False.
- Evaluate any expression and get one of two answers, True or False.

Example 1:

```
print(10 > 9)    # True
print(10 == 9)   # False
print(10 < 9)    # False
```

Example 2:

```
x = 10
y = 4

if x > y:
  print("x is greater than y")
else:
  print("x is not greater than y")
```

bool() function [1/2]

- bool() function allows you to evaluate any value, and return True or False.

```
print(bool("Hello"))      # True
print(bool(15))           # True


x = "Hello"
y = 15
print(bool(x))            # True
print(bool(y))            # True
```

- Almost any value is evaluated to True if it has some sort of content.
- Any string is True, except empty strings.
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.

bool() function [2/2]

- There are not many values that evaluate to False, except empty values
  - (), [], {}, "", the number 0, and the value None.
  - the value False evaluates to False.

Example:

```
bool(False)
bool(None)
bool(0)
bool("")
bool(())
bool([])
bool({})
```

Set

- Sets are used to store multiple items in a single variable.
- A set is a collection which is *unordered*, *unchangeable*, and *unindexed*.
- Set *items* are unchangeable (once a set is created, you cannot change its items), but you can remove items and add new items.
- Sets are written with curly brackets.

Example: Create a set.

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

- Sets are unordered, so you cannot be sure in which order the items will appear.

Duplicates Not Allowed

- Sets cannot have two items with the same value.
- Duplicate values will be ignored.

Example:

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

- True and 1 is considered the same value.

```
thisset = {"apple", "banana", "cherry", True, 1, 2}
print(thisset)          # {True, 2, 'banana', 'cherry', 'apple'}
```

- False and 0 is considered the same value

```
thisset = {"apple", "banana", "cherry", False, True, 0}
print(thisset) # {'apple', False, True, 'cherry', 'banana'}
```

Set Items [1/2]

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))              # 3
```

- Set items can be of any data type.

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}

print(set1)

print(set2)

print(set3)
```

Output:

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

{1, 3, 5, 7, 9}

{False, True}
```

Set Items [2/2]

- A set can contain different data types.

```
set1 = {"abc", 34, True, 40, "male"}
print(set1)     # {True, 34, 40, 'male', 'abc'}
```

- Data type of a set

```
print(type(set1))      # <class, ‘set’>
```

set() Constructor

- Use the set() constructor to make a set.

Example:

```
thisset = set(("apple", "banana", "cherry")) # note the double
                                             # round-brackets
print(thisset)
```

Access Set Items

- Items in a set cannot be accessed using an index or a key.
- We can loop through the set items using a for loop, or ask if a specified value is present in a set by using the in keyword.

Example:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Output:

Cherry

Apple

banana

- Check if "banana" is present in the set.

```
thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)   # Output: ??
```

- Check if "banana" is NOT present in the set:

```
thisset = {"apple", "banana", "cherry"}
print("banana" not in thisset)
```

Add Set Items

- Once a set is created, you cannot change its items, but you can add new items using add() method.

Example:

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)                  # {'banana', 'cherry', 'orange', 'apple'}
```

- To add items from another set into the current set, use the update() method. update() method can update a set using objects like tuples, lists, dictionaries etc.

Example:

```
thisset = {"apple", "banana", "cherry"}
tropical = {"pineapple", "mango", "papaya"}
thisset.update(tropical)
print(thisset)
```