Kristu Jayanti College
AUTONOMOUS Bengaluru
Reaccredited A++ Grade by NAAC | Affiliated to Bengaluru North University

Python Programming

Unit I

Dr. Stephen A
Assistant Professor
Department of Computer Science
Kristu Jayanti College, Bengaluru

Introduction to Python

INTRODUCTION

- Invented In 1989 by Guido van Rossum
- Named after Monty Python's flying circus
- It is a programming language
- Free s/w (open source)
- A scripting language

  Scalable, object oriented
- Python is an interpreted language
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

INTRODUCTION

- Designed to be easy to learn and master
  - Clean, clear syntax
  - Very few keywords
- Extensible
  - Designed to be extensible using C/C++, allowing access to many external libraries
  - Python comes standard with a set of modules, known as the “standard library”

FEATURES

- **Python is Interpreted:** Python is processed at runtime by the interpreter. You do not need to compile your program before executing it.
- **Python is Interactive:** You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- **Python is Object-Oriented:** Python supports Object-Oriented style or technique of programming that encapsulates code within objects

FEATURES

- **Python is a Beginner's Language**: Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games
- Python is derived from many other languages, including ABC, C, C++, and Unix shell and other scripting languages.

- **Easy-to-learn:**
- Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.

FEATURES

- **Easy-to-read:**
- Python code is more clearly defined and visible to the eyes.
- **Easy-to-maintain:**
- Python's source code is fairly easy-to-maintain.

Why do people use Python...?

The following primary factors cited by Python users seem to be these:

Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance.

Indentation

Indentation is one of the greatest future in Python.

It's free (open source)

Downloading and installing Python is free and easy Source code is easily accessible

Why do people use Python...?

It's powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, SciPy)
- Automatic memory management

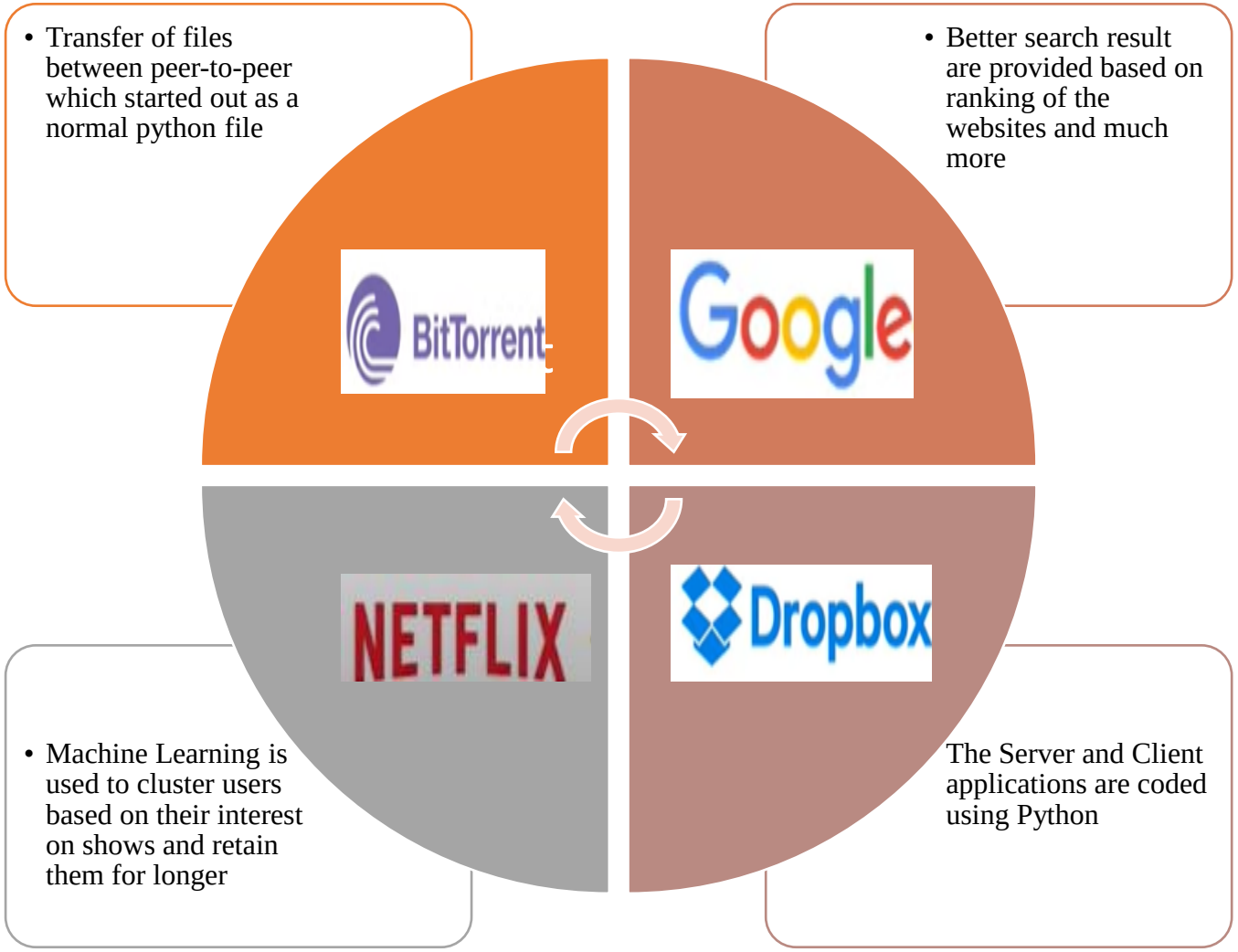
It's portable

- Python runs virtually every major platform used today
- As long as you have a compatible Python interpreter installed, Python program will run in exactly the same manner, irrespective of platform.

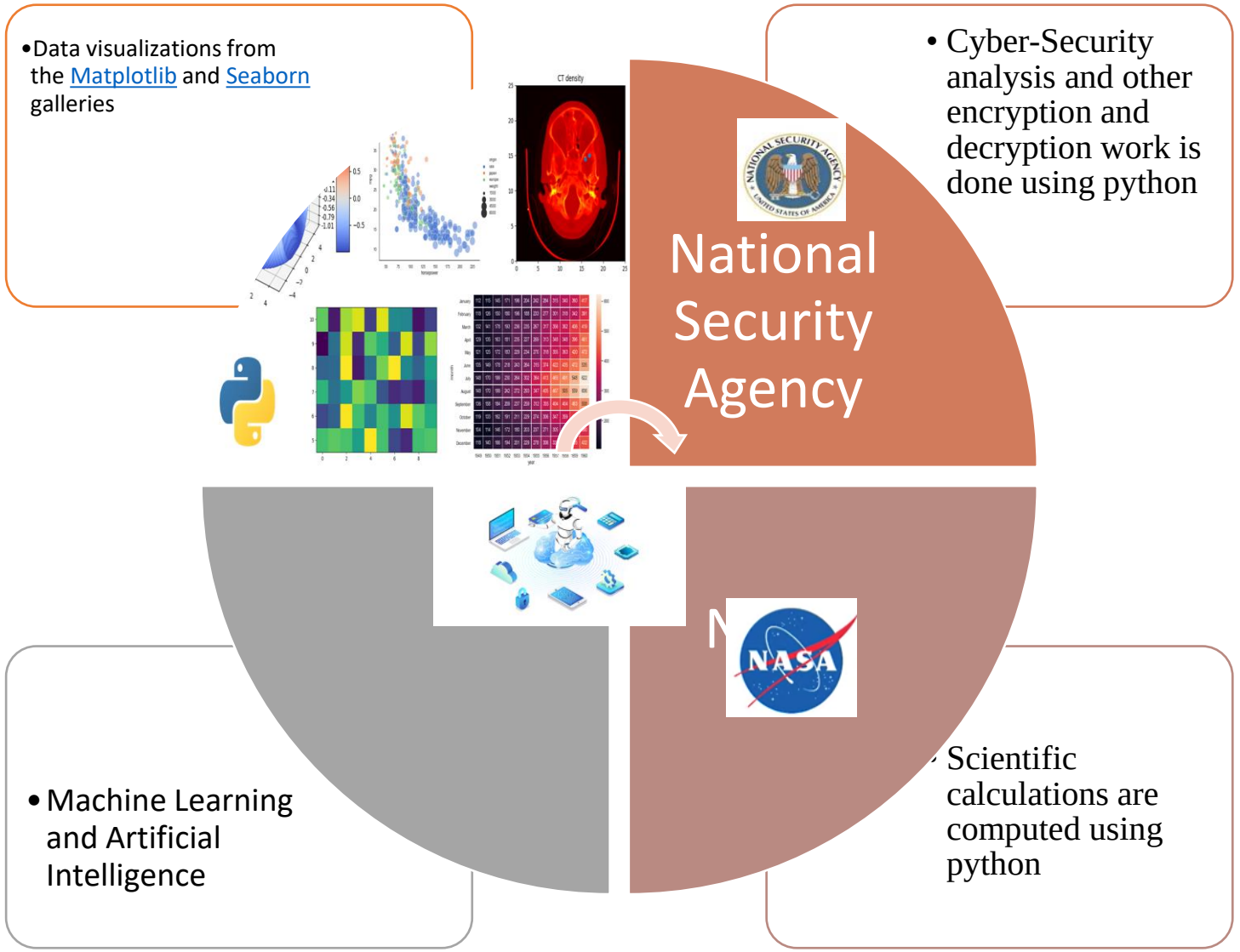
Why do people use Python...?

- **It's mixable**
- Python can be linked to components written in other languages easily
- Linking to fast, compiled code is useful to computationally intensive
- problems
- - Python/C integration is quite common
- **It's easy to use**
- No intermediate compile and link steps as in C/ C++
- Python programs are compiled automatically to an intermediate form called *bytecode*, which the interpreter then reads
- This gives Python the development speed of an interpreter without
- the performance loss inherent in purely interpreted languages
- **It's easy to learn**
- Structure and syntax are pretty intuitive and easy to grasp

Where is python used in the Industry



Where is python used in the Industry




ROADMAP TO LEARN
PYTHON PROGRAMMING
Basics
Basic Syntax
Variables
Data types
Conditionals
Typecasting
Exceptions
Functions
Lists, Tuples, Sets
Dictionaries
Advanced
List
Comprehensions
Generators
Expressions
Paradigms
regex
Decorators
Iterators
Lambdas
OOP
Classes
Inheritance
Methods (including Dunder Methods)
Data Science
NumPy
Pandas
Matplotlib
Seaborn
Scikit-learn
TensorFlow
PyTorch
Data Structures and Algorithms
Arrays and Linked Lists
Heaps, Stacks, Queues
Hash Tables
Binary Search Trees
Recursion
Sorting Algorithms
Web Frameworks
Django
Flask
FastAPI
Tornado
Package Manager
PyPI
pip
conda
Automation
File manipulation (os, shutil, pathlib)
Web scraping (BeautifulSoup, Scrapy)
GUI automation (PyAutoGUI)
Network automation

Career Opportunities

Difference between Interactive and Script mode in Python

Interactive mode: Instructions are given in front of Python prompt

(eg., >>>) in Python Shell. Python carries out the given instruction and shows the result there itself.

Script mode: Python instructions are stored in a file generally with **.py extension** and are executed together in one go as a unit. The saved instructions are known as **Python script or Python program**.

Working with Python

- Default installation from **www.python.org** is called Cpython installation and comes with Python interpreter, Python IDLE(Python GUI) and Pip(package installer).

- Other Python distributions – Anaconda Python distribution that comes preloaded with many packages and libraries(eg. Numpy,SciPy,Panda libraries,etc.)

- Popular IDEs – Spyder IDE,PyCharm IDE etc.
- Spyder IDE is already available as a part of Anaconda Python Distribution.

Major Packages, libraries, frameworks

- Numpy (NUMeric Python) : matrices and linear algebra
- Scipy (SCIentific Python) : many numerical routines
- Pandas : data analysis and modeling library
- Pytest (Python TESTing) : a code testing framework

print() function

print() function

- The print() function in Python is used to output data to the console or terminal.
- It can print strings, numbers, and other data types, and it supports various formatting options.

```
print("Hello, world!")  # Prints a string
print(123)              # Prints an integer
print(3.14)             # Prints a float
```

Multiple Arguments print() function

- You can pass multiple arguments to print(), and they will be separated by spaces by default

```
print("The answer is", 42)  # Prints: The answer is 42
```

End and Separator Parameters print() function

- **sep:** Defines the separator between multiple arguments (default is a space).

- **end:** Defines what is printed at the end of the line (default is a newline).

```
print("Hello", "world", sep="-")  # Prints: Hello-world
print("Hello", end=", ")
print("world!")  # Prints: Hello, world!
```

input() function

input() function

- The input() function in Python is used to read user input from the console.
- It allows you to prompt the user to enter data, which is then returned as a string.

```
input(prompt)
```

Basic Input

```
user_input = input("Enter your name: ")
print(f"Hello, {user_input}!")
```

- This will display the prompt **Enter your name:,** and whatever the user types in will be stored in the user_input variable

Using the Prompt

```
age = input("How old are you? ")
print(f"You are {age} years old.")
```

- Here, How old are you? is the prompt displayed to the user.

Converting Input

- By default, input() returns a string. To use the input as another data type, such as an integer or float, you need to convert it:

```
age = int(input("Enter your age: "))  # Convert input to integer
height = float(input("Enter your height in meters: "))  # Convert input to float
```

Handling Invalid Input

- You can use try-except blocks to handle cases where the user input cannot be converted to the expected type:

```
try:
    number = int(input("Enter a number: "))
    print(f"You entered the number {number}.")
except ValueError:
    print("That's not a valid number.")
```

Input Without Prompt

- If you don't provide a prompt, input() will simply wait for the user to enter something:

```
user_input = input()  # No prompt displayed
print(f"You entered: {user_input}")
```

formatted string literal(f-string)

formatted string literal(f-string)

- An f-string, short for formatted string literal, is a feature introduced in Python 3.6
- It provides a convenient and readable way to embed expressions inside string literals.
- F-strings are prefixed with an f or F and allow expressions to be included within curly braces {}.
- These expressions are evaluated at runtime and formatted using the specified format.

format specifiers

- In Python, format specifiers are used to control the formatting of strings when using various string formatting methods. These include the **%** operator,

- Common Format Specifiers:
- %d: Integer
- %f: Floating-point number
- %.2f: Floating-point number with 2 decimal places
- %s: String
- %x: Hexadecimal integer (lowercase)
- %X: Hexadecimal integer (uppercase)

format specifiers

```
name = "Alice"

age = 30

height = 5.7


print("Name: %s, Age: %d, Height: %.1f" % (name, age, height))
```

Output:

```
Name: Alice, Age: 30, Height: 5.7
```

formatted string literal(f-string)

Syntax

f"string text {expression} string text"

Variable Embedding – f-string

Example:

```
user = "Bob"

print(f"Hello, {user}!")
```

Output:

```
Hello, Bob!
```

Arithmetic Expressions – f-string

Example:

```
a = 5

b = 3

print(f"{a} + {b} = {a + b}")
```

Output:

```
5 + 3 = 8
```

Function Calls– f-string

Example:

```
def greet(name):
    return f"Hello, {name}!"
name = "Carol"
print(f"{greet(name)}“
```

Output:

```
Hello, Carol!
```

Width and Alignment

Example:

```
number = 42

print(f"{number:5}")  # Right-aligned by default

print(f"{number:<5}")  # Left-aligned

print(f"{number:^5}")  # Center-aligned
```

Output:

```
42

42

 42
```

Coding style in Python

Coding style in Python

- Coding style in Python is an important aspect of writing clean, readable, and maintainable code.

- The Python community has established guidelines and best practices, primarily documented in PEP 8, the Python Enhancement Proposal that provides conventions for writing code in Python.

Coding style in Python

1. Indentation
2. Line Length
3. Blank Lines
4. Imports
5. Naming Conventions
6. Docstrings
7. Comments
8. Whitespace in Expressions and Statements
9. Error Handling

Coding style in Python

1. Indentation

- Use 4 spaces per indentation level.
- Avoid using tabs.
- Consistent indentation is crucial in Python, as it defines the structure of the code.

```
1. def my_function(x):
2.     if x > 0:
3.         print("Positive")
4.     else:
5.         print("Non-positive")
```

Coding style in Python

2. Line Length

- Limit all lines to a maximum of 79 characters.
- For long blocks of text (e.g., docstrings or comments), lines should be limited to 72 characters.
- Use parentheses to wrap long lines instead of using a backslash (\) for line continuation.

```
# Example of wrapping lines
result = some_function_that_does_something(
    argument1, argument2, argument3,
    argument4, argument5
)
```

Coding style in Python

3. Blank Lines

- Use blank lines to separate top-level functions and class definitions.
- Inside functions, use blank lines to separate logical sections of the code.

```
# Example class MyClass:

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

    def reset(self):
        self.value = 0
```

Coding style in Python

4. Imports

- Imports should be on separate lines and typically at the top of the file.
- Follow the order: standard library imports, related third-party imports, and then local application/library-specific imports.

Example

```
# Correct way to do imports
import os
import sys


from datetime import datetime
from collections import defaultdict


import my_local_module
```

Coding style in Python

5. Naming Conventions

Variables, Functions, and Attributes: Use lowercase words separated by underscores.

- **Example**

```
def calculate_area(radius):
    pi_value = 3.14159
    return pi_value * (radius ** 2)
```

Coding style in Python

6. Docstrings

- Use docstrings to document all public modules, classes, functions, and methods.
- Follow the convention for multiline docstrings:
- The first line is a short summary.
- The second line is blank

Example

```
def calculate_area(radius):
    """
    Calculate the area of a circle given its radius.

    :param radius: The radius of the circle.
    :return: The area of the circle.
    """
    return 3.14159 * (radius ** 2)
```

Coding style in Python

7. Comments

- Use comments to explain why something is done, not what is done.
- If the code isn't self-explanatory, it may need to be refactored.
- Comments should be complete sentences, with the first word capitalized and ending with a period.

Example

```
# Initialize the count to zero
count = 0
```

Coding style in Python

8. Whitespace in Expressions and Statements

- Avoid extraneous whitespace.
  - Immediately inside parentheses, brackets, or braces: list[1], not list[1].
  - Before a comma, semicolon, or colon: if x == 1:, not if x == 1 :.
  - Around the assignment operator when assigning a default value: def func(param=None):, not def func(param = None):.

Example

```
# Good
a = (1, 2, 3)


# Avoid
b = ( 1, 2, 3 )
```

Coding style in Python

9. Error Handling

- Use exceptions to handle errors. Use try/except blocks to manage exceptions gracefully, and always catch specific exceptions.

Example

```
try:
    result = some_function()
except ValueError as e:
    print(f"ValueError encountered: {e}")
```

Variables and Expressions

Variables in Python

- Variables in Python are used to store data values.
- They don't have a type themselves; the type is associated with the value they hold.
- You assign a value to a variable using the = operator.

```
x = 5        # An integer
name = "Alice"  # A string
pi = 3.14    # A float
```

Expressions

- An expression is a combination of variables, operators, and values that evaluates to a result

- **Example:**

result = (5 + 3) * 2 # *The expression (5 + 3) * 2 evaluates to 16*

Python Datatypes

Python Datatypes

Classification	Datatypes
Text Type	str
Numeric Types	int, float, complex
Sequence Types	list, tuple, range
Mapping Type	Dictionary
Set Types	set, frozenset
Boolean Type	bool
Binary Types	bytes, bytearray, memoryview
None Type	NoneType

List (Sequence type)

- It represents an ordered collection of items.
- Different types of data can be stored under single variable.
- A list is created using square brackets []
- Pairs of Sigle or double quotes are used to add string items in the list

Syntax:

list_name=[item1, item2,.....itemn]

List (Sequence type)

Example

```
mylist=['V bcae', 5]
print(mylist)
print(mylist[1])
```

Output

```
['V bcae', 5]
5
```

Tuple (Sequence type)

- Tuples are used to store multiple items in single variable.
- Different types of data can be stored in the tuple.
- A tuple is created using round brackets () (parentheses)
- Pairs of Sigle or double quotes are used to add string items in the tuple

Syntax:

Tuple_name=(item1, item2,.....itemn)

Tuple (Sequence type)

Example

```
tuple_list=('V bcae', 5)
print(tuple_list)
print(tuple_list[1])
```

Output

```
['V bcae', 5]
5
```

List vs Tuple

Feature	List	Tuple
Mutability	Lists are mutable, meaning you can change, add, or remove elements after the list is created. *my_list = [1, 2, 3]* *my_list[1] = 4 # Allowed* *my_list.append(5) # Allowed* *print(my_list) # Output: [1, 4, 3, 5]*	Tuples are immutable, meaning once a tuple is created, you cannot change its elements. *my_tuple = (1, 2, 3)* *my_tuple[1] = 4 # Not allowed, raises TypeError*
Syntax:	Lists are defined using square brackets []. my_list = [1, 2, 3]	Tuples are defined using parentheses (). my_tuple = (1, 2, 3)
Methods:	Lists have a variety of built-in methods for modifying the content, such as append(), remove(), extend(), pop(), clear(), sort(), and reverse(). *my_list = [1, 2, 3] my_list.append(4) print(my_list) # Output: [1, 2, 3, 4]*	Tuples have fewer methods, primarily for accessing data, such as count() and index() *my_tuple = (1, 2, 2, 3) print(my_tuple.count(2)) # Output: 2 print(my_tuple.index(3)) # Output: 3*

List vs Tuple

Feature	List	Tuple
Performance:	Lists have a larger memory overhead and are generally slower than tuples due to their dynamic nature and ability to be modified.	Tuples are more memory-efficient and faster than lists due to their immutability.
Use Cases:	Use lists when you need a collection of items that can change, such as when items need to be added, removed, or modified. *shopping_list = ['milk', 'eggs', 'bread']* *shopping_list.append('butter') print(shopping_list)* *# Output: ['milk', 'eggs', 'bread', 'butter']*	Use tuples for fixed collections of items, such as coordinates or when you want to ensure the data remains constant. *coordinates = (10.0, 20.0)*
Heterogeneity:	can store heterogeneous data types (different types of items)	can store heterogeneous data types (different types of items)

List vs Tuple

Feature	List	Tuple
Iteration:	can be iterated over in the same way. *# List iteration* *my_list = [1, "apple", 3.14]* *for item in my_list:* *print(item)* *Output* *1* *apple* *3.14*	can be iterated over in the same way. *# Tuple iteration* *my_tuple = (1, "apple", 3.14)* *for item in my_tuple:* *print(item)* *Output* *1* *apple* *3.14*

Range (Sequence type)

- It represents an immutable sequence of numbers
- Used for looping a specific number of times in **'for'** loops.
- It generates a sequence of numbers that start from a given start point and stops at a given end point
- **list() function** is used to list(print) the values given to **range**

Syntax:

```
variable_name=range(start_value)
variable_name=range(start_value, end_value)
variable_name=range(start_value, end_value, step_value)
```

Range (Sequence type)

range(stop) - Generates numbers from 0 to stop - 1.

Example:

```
r = range(5)
print(list(r))
# Output: [0, 1, 2, 3, 4]
```

Range (Sequence type)

range(start, stop) - Generates numbers from start to stop - 1.

Example:

```
r = range(1, 5)
print(list(r))
# Output: [1, 2, 3, 4]
```

Range (Sequence type)

range(start, stop, step) - Generates numbers from start to stop - 1, incrementing by step.

Example:

```
r=range(1, 10, 2)
print(list(r))
# Output: [1, 3, 5, 7, 9]
```

Dictionaries (Mapping Type)

- Python dictionaries are essential for efficient data mapping and manipulation in programming.
- Dictionary is an unordered collection of key-value pairs.
- We create a dictionary by placing **key: value** pairs inside **curly brackets {}, separated by commas**.
- Dictionaries are optimized to retrieve values when the **key is known**.

Syntax:

```
dict_var = {key1 : value1, key2 : value2, …..}
```

Dictionaries (Mapping Type)

- Dict = {1: 'Python', 2: 'Cloud', 3: 'AI'}
- print(Dict)

- **{1: 'Python', 2: 'Cloud', 3: 'AI'}**

- Dict = {
- 1: 'Python',
- 2: 'Cloud',
- 3: 'AI'}
- print(Dict)

Dictionaries (Mapping Type)

- Indexing is not possible
- Dict = {
- 1: 'Python',
- 2: 'Cloud',
- 3: 'AI'}
- print(Dict[3])

- **Output**
- AI

Set (Set Types)

- sets are a built-in data type that represent an unordered collection of unique items.
- They are useful when you need to perform operations that involve membership testing, removing duplicates from a collection, or performing set operations like union, intersection, and difference.
- **Syntax:**
  - ***Using curly braces {}***

    set_name = {item1, item2, item3, …..itemn}
  - ***Using the set()***

    set_name = (item1, item2, item3, …..itemn)

Set (Set Types)

Key Characteristics:

- ***Unordered:*** Sets do not maintain any specific order of elements.

- **Unique Elements**: Each element in a set must be unique. If you try to add a duplicate element, it will be ignored.

- **Mutable:** You can add or remove elements after the set is created.

- **Unindexed:** Sets do not support indexing, slicing, or other sequence-like behaviors.

Set (Set Types)

Example:

```
my_set = {"ZAINAB KARIMI", "AADITH RAJ R", "ABIGAIL ANN ANOOP", "ADITYANAYAK"}
print("V BCA E:",my_set)
```

Output:

```
V BCA E: {'AADITH RAJ R', 'ABIGAIL ANN ANOOP', 'ZAINAB KARIMI', 'ADITYANAYAK'}
```

Set (Set Types)

Example: (Duplicate value)

```
my_set = {"ZAINAB KARIMI", "AADITH RAJ R", "ZAINAB KARIMI", "ABIGAIL ANN ANOOP",
"ADITYANAYAK"}
print("V BCA E:",my_set)
```

Output:

```
V BCA E: {'ADITYANAYAK', 'AADITH RAJ R', 'ZAINAB KARIMI', 'ABIGAIL ANN ANOOP'}
```

frozenset()

Freeze the list, and make it unchangeable

Syntax:

frozenset(iterable)

Parameter	Description
iterable	An iterable object, like list, set, tuple etc.

frozenset()

Example

```
mylist = ['apple', 'banana', 'cherry']

x = frozenset(mylist)

x[1] = "strawberry“
```

Output:

```
File "demo_ref_frozenset2.py", line 3, in <module>
    x[1] = "strawberry"
TypeError: 'frozenset' object does not support item assignment
```

Python Collections (Arrays)

There are four collection data types in the Python programming language:

1. **List** is a collection which is ordered and changeable. Allows duplicate members.
2. **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
3. **Dictionary** is a collection which is ordered** and changeable. No duplicate members
4. **Set** is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.

Booleans

Booleans represent one of two values: True or False.

In programming , often need to know **if an expression is True or False**.

You can evaluate any expression in Python, and get one of two answers, True or False.

When you compare two values, the expression is evaluated and Python returns the Boolean answer:

Booleans

Example:

```
print(10 > 9)
print(10 == 9)
print(10 < 9)
```

Output:

```
True
False
False
```

Operators

Operators

1. Arithmetic Operators
2. Comparison Operators
3. Logical Operators
4. Bitwise Operators
5. Assignment Operators
6. Identity Operators
7. Membership Operators

Arithmetic Operators

Operator	Name		Example	
+	Addition	Adds two operands.	x + y	3 + 2 # 5
-	Subtraction	Subtracts the second operand from the first	x - y	3 - 2 # 1
*	Multiplication	Multiplies two operands.	x * y	3 * 2 # 6
/	Division	Divides the first operand by the second	x / y	3 / 2 # 1.5
%	Modulus	Returns the remainder when the first operand is divided by the second.	x % y	3 % 2 # 1
**	Exponentiation	Raises the first operand to the power of the second	x ** y	3 ** 2 # 9
//	Floor division	Divides the first operand by the second and returns the largest integer less than or equal to the result.	x // y	3 // 2 # 1

Comparison Operators

Operator	Name	Example	
==	Equal	x == y	3 == 2 # False
!=	Not equal	x != y	3 != 2 # True
>	Greater than	x > y	3 > 2 # True
<	Less than	x < y	3 < 2 # False
>=	Greater than or equal to	x >= y	3 >= 2 # True
<=	Less than or equal to	x <= y	3 <= 2 # False

Logical Operators

Operator	Description	Example	
and	Returns True if both statements are true	x < 5 and x < 10	True and False # False
or	Returns True if one of the statements is true	x < 5 or x < 4	True or False # True
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)	not True # False

Bitwise Operators

Operator	Name	Description	Example	
&	AND	Sets each bit to 1 if both bits are 1	x & y	3 & 2 # 2
|	OR	Sets each bit to 1 if one of two bits is 1	x | y	3 | 2 # 3
^	XOR	Sets each bit to 1 if only one of two bits is 1	x ^ y	3 ^ 2 # 1
~	NOT	Inverts all the bits	~x	~3 # -4
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off	x << 2	3 << 1 # 6
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off	x >> 2	3 >> 1 # 1

Assignment Operators

1. = (Assign)
2. += (Add and assign)
3. -= (Subtract and assign)
4. *= (Multiply and assign)
5. /= (Divide and assign)
6. //= (Floor divide and assign)
7. %= (Modulus and assign)
8. **= (Exponentiate and assign)
9. &= (Bitwise AND and assign)
10. |= (Bitwise OR and assign)
11. ^= (Bitwise XOR and assign)
12. <<= (Bitwise Left Shift and assign)
13. >>= (Bitwise Right Shift and assign)

Identity Operators

Operator	Description	Example	
is	Returns True if both variables are the same object	x is y	2 in [1, 2, 3]
is not	Returns True if both variables are not the same object	x is not y	4 not in [1, 2, 3]

Membership Operators

Operator	Description	Example	
in	Returns True if a sequence with the specified value is present in the object	x in y	2 in [1, 2, 3] # True
not in	Returns True if a sequence with the specified value is not present in the object	x not in y	4 not in [1, 2, 3] # True

Types of Numbers in Python

Types of Numbers in Python

- Integers (int): Whole numbers, positive or negative, without a decimal point.

Example:

```
x = 5

y = -10

z = 123456789
```

Types of Numbers in Python

- Floating-Point Numbers (float): Numbers with a decimal point.

- **Example**

  a = 3.14

  b = -0.001

  c = 2.0

Types of Numbers in Python

- Complex Numbers (complex): Numbers with a real and imaginary part.
- **Example**

```
c = 3 + 4j

d = complex(2, -3)
```

Mathematical Functions in Python

Mathematical Functions in Python

- Python's standard library includes the math module, which provides a range of mathematical functions.

  import math

Absolute Value (abs()): result = abs(-10) # 10

round(x, n): Rounds a floating-point number to n decimal places.

Example: rounded = round(3.14159, 2) # 3.14

Mathematical Functions in Python

max(iterable, *args): Returns the largest item in an iterable or among multiple arguments.

Example: max_val = max(1, 5, 3) # 5

min(iterable, *args): Returns the smallest item in an iterable or among multiple arguments.

Example: min_val = min(1, 5, 3) # 1

Mathematical Functions in Python

sum(iterable, start=0): Returns the sum of all items in an iterable, starting with start value.

Example: total = sum([1, 2, 3, 4]) # 10

pow(x, y): returns x ** y.

Example: result = pow(2, 3) # 8

Conditional Statements

Conditional Statements

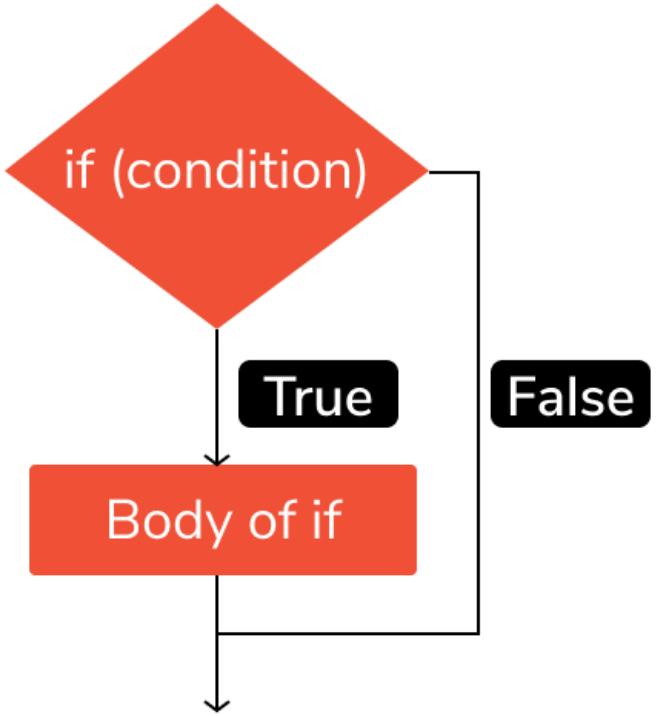
- In python, the Conditional Statements are the statements that controls the execution of the program on the basis of the specified condition.
- It is useful for determining whether a condition is true or not.
- If the condition is true, a single or block of statement is executed.
- Examples
  - **if statement**
  - **elif statement**
  - **else statement**

if statement

- Executes a block of code if a specified condition is true.

Syntax:

```
if condition:
    # code block
```



if statement

Example

```
age = 18
if age >= 18:
    print("You are an adult.")
```

Output

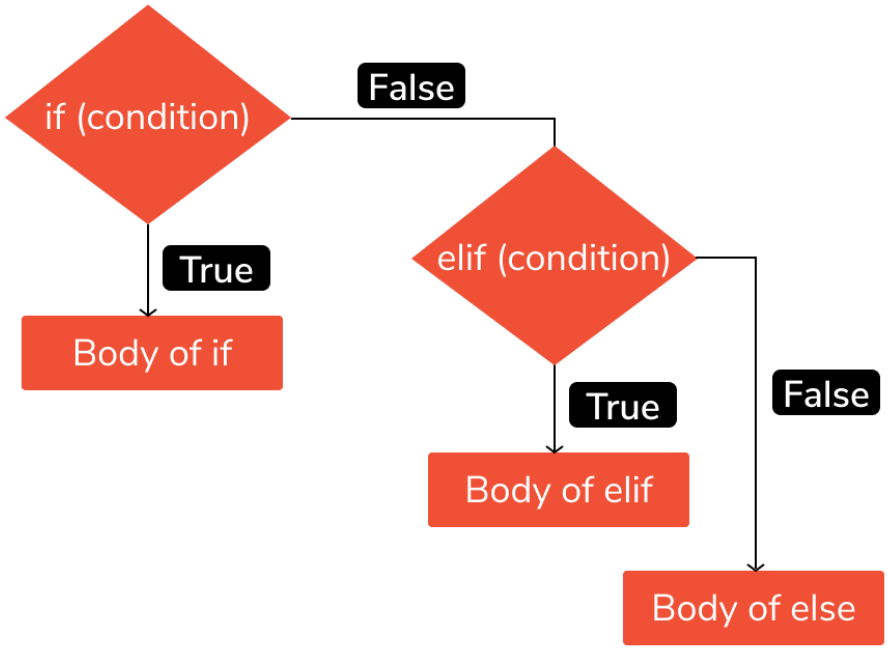
```
You are an adult.
```

elif Statement

- *elif* Short for "else if", it checks another condition if the previous if or elif condition is false.

Syntax:

```
if condition:
    # code block
elif another_condition:
    # another code block
```



elif Statement

Example

```
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
```

Output

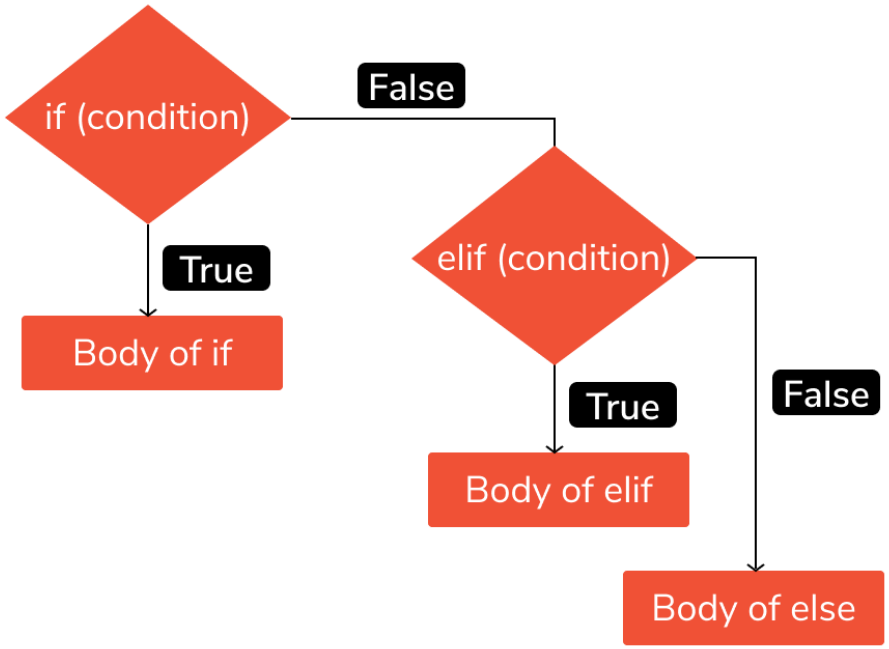
```
Grade: B
```

else statement

- Executes a block of code if none of the preceding conditions are true.

Syntax:

```
if condition:
    # code block
elif another_condition:
    # another code block
else:
    # code block if no conditions
are true
```



else Statement

Example

```
score = 70
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
else:
    print("Grade: C")
```

Output

```
Grade: C
```

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example:

```
a = 200
b = 33
if a > b: print("a is greater than b")
```

Output

a is greater than b

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

Output

B

The pass Statement

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

Example:

```
a = 33
b = 200
if b > a:
  pass
```

Output

Python Loops

Python Loops

Python has two primitive loop commands:

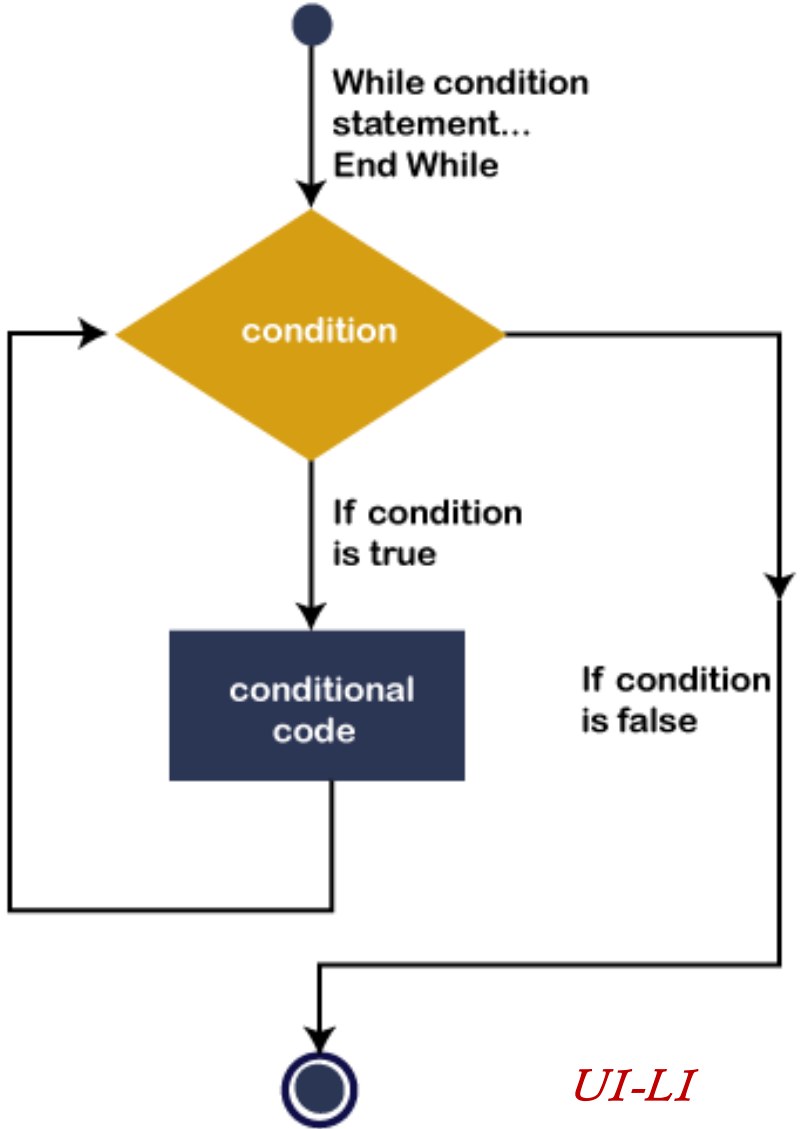
1. while loops
2. for loops

while loops

Executes a block of code as long as a specified condition is true.

Syntax:

```
while condition:
    # code block
```



while loops

Example

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output

```
0
1
2
3
4
```

The break Statement

With the break statement we can stop the loop even if the while condition is true:

Example:

```
i = 1
while i < 6:
  print(i)
  if (i == 3):
    break
  i += 1
```

Output

```
1
2
3
```

The continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

Example:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

Output

```
1
2
3
4
5
6
```

for loops

- The for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string) and execute a block of code for each item in the sequence.

- It is particularly useful when you need to perform an operation on each element of a collection.

for loops

Syntax:

```
for item in sequence:
    # code block
```

for: The keyword to start the loop.

item: A variable that takes the value of the next item in the sequence on each iteration.

in: The keyword used to specify the sequence to iterate over.

sequence: The collection (list, tuple, string, etc.) to iterate over.

code block: The block of code to execute for each item in the sequence.

for loops

Example 1: Iterating Over a List

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


for fruit in fruits:
    print(fruit)
```

Output

```
apple
banana
cherry
```

for loops

Example 2: Iterating Over a String

```
for letter in "Python":
    print(letter)
```

Output

```
P
y
t
h
o
n
```

for loops

Example 3: Using range() Function

```
for number in range(5):
    print(number)
```

Output

```
0
1
2
3
4
```

for loops

Example 4: Iterating Over a Dictionary

```
person = {"name": "Alice", "age": 25}

for key in person:
    print(key)
```

Output

```
Name
age
```

for loops

Example 4: Iterating Over a Dictionary

```
person = {"name": "Alice", "age": 25}

for key in person:
    print(value)
```

Output

```
Alice
25
```

Example 5: Nested for Loop

Example

```
colors = ["red", "green", "blue"]
objects = ["ball", "pen"]

for color in colors:
    for obj in objects:
        print(color, obj)
```

Output:

```
red ball
red pen
green ball
green pen
blue ball
blue pen
```

Functional programming

Key Concepts in Functional Programming:

Immutability:

- Data structures are often immutable, meaning they cannot be changed after they are created.
- Instead of modifying data, you create new instances with the desired changes.

Pure Functions:

- Pure functions are functions that always produce the same output for the same input and have no side effects (such as modifying global variables or changing the input data).
- They do not depend on or alter the state of the program.

Key Concepts in Functional Programming:

Python provides several built-in functions and modules that support functional programming:

map(): Applies a function to all items in an input list.

filter(): Constructs a list of elements from an input list for which a function returns True.

reduce(): Repeatedly applies a function to the elements of a sequence, reducing it to a single value (available in the functools module).

lambda functions: Anonymous functions defined with the lambda keyword, often used for short, simple functions.

Key Concepts in Functional Programming example:

```
double = lambda x: x * 2
add = lambda x, y: x + y
# Take input from the user
num = int(input("Enter a number: "))
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
# Apply the lambda functions
result1 = double(num)
result2 = add(num1, num2)
# Display the results
print("Double of", num, "is", result1)

print("Sum of", num1, "and", num2, "is", result2)
```

Unpacking Methods in python

Unpacking Methods in python

- Tuple
- List
- Dictionary

Unpacking a tuple

- Unpacking a tuple in Python is a way to assign the values of the tuple to multiple variables in a single statement.

- This is done by placing the variables on the left side of the assignment operator (=), matching the number of elements in the tuple on the right side.

Unpacking a tuple 1

```
# Define a tuple
my_tuple = (1, 2, 3)


# Unpack the tuple into variables
a, b, c = my_tuple


# Now, a = 1, b = 2, c = 3
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Unpacking a tuple 2

- Using * for Extended Unpacking

- Python allows extended unpacking using the * operator to capture multiple elements in a tuple:

```
# Define a tuple
my_tuple = (1, 2, 3, 4, 5)

# Unpack the tuple with extended unpacking
a, *b, c = my_tuple

# Now, a = 1, b = [2, 3, 4], c = 5
print(a)  # Output: 1
print(b)  # Output: [2, 3, 4]
print(c)  # Output: 5
```

Unpacking a list 1

```
# Define a list
my_list = [10, 20, 30]


# Unpack the list into variables
x, y, z = my_list


# Now, x = 10, y = 20, z = 30
print(x)  # Output: 10
print(y)  # Output: 20
print(z)  # Output: 30
```

Unpacking a list 2

- Using * for Extended Unpacking

- Python allows extended unpacking using the * operator to capture multiple elements in a tlist:

```
# Define a list
my_list = [10, 20, 30, 40, 50]

# Unpack the list with extended unpacking
a, *b, c = my_list

# Now, a = 10, b = [20, 30, 40], c = 50
print(a)  # Output: 10
print(b)  # Output: [20, 30, 40]
print(c)  # Output: 50
```

Unpacking a dictionary

- Unpacking dictionaries in Python is different from unpacking lists or tuples.

- Instead of unpacking the values directly into variables, you typically work with the keys and values separately.

Unpacking Keys, Values, and Items

Unpacking Keys: You can directly unpack dictionary keys into variables.

```
# Define a dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Unpack the keys into variables
x, y, z = my_dict

# Now, x = 'a', y = 'b', z = 'c'
print(x)  # Output: 'a'
print(y)  # Output: 'b'
print(z)  # Output: 'c'
```

Unpacking Values:

To unpack the values, use the .values() method.

```
# Define a dictionary
my_dict = {'a': 1, 'b': 2, 'c': 3}

# Unpack the values into variables
x, y, z = my_dict.values()

# Now, x = 1, y = 2, z = 3
print(x)  # Output: 1
print(y)  # Output: 2
print(z)  # Output: 3
```

Stack and Queue

Stack

- A stack is a collection of elements that follows the Last In, First Out (LIFO) principle.
- The most recent element added (pushed) to the stack will be the first one removed (popped).

Operations:

- Push: Add an element to the top of the stack.
- Pop: Remove the element from the top of the stack.
- Peek: Get the element at the top of the stack without removing it.
- is_empty: Check if the stack is empty.

Stack

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            return None  # or raise an exception

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            return None  # or raise an exception

    def is_empty(self):
        return len(self.stack) == 0

    def size(self):
        return len(self.stack)

# Example usage:
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop())  # Output: 3
print(s.peek())  # Output: 2
```

Output

```
3
2
```

Queue

- A queue is a collection of elements that follows the First In, First Out (FIFO) principle.
- The first element added to the queue will be the first one removed.

Operations:

- Enqueue: Add an element to the end of the queue.
- Dequeue: Remove the element from the front of the queue.
- Front: Get the element at the front of the queue without removing it.
- is_empty: Check if the queue is empty.

Queue:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            return None  # or raise an exception

    def front(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            return None  # or raise an exception

    def is_empty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage:
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # Output: 1
print(q.front())  # Output: 2
```

Output

```
1
2
```

Fundamentals of OOPS in Python

Fundamentals of OOPS in Python

1. Objects
2. Classes
3. Data abstraction
4. Data encapsulation
5. Inheritance
6. Polymorphism

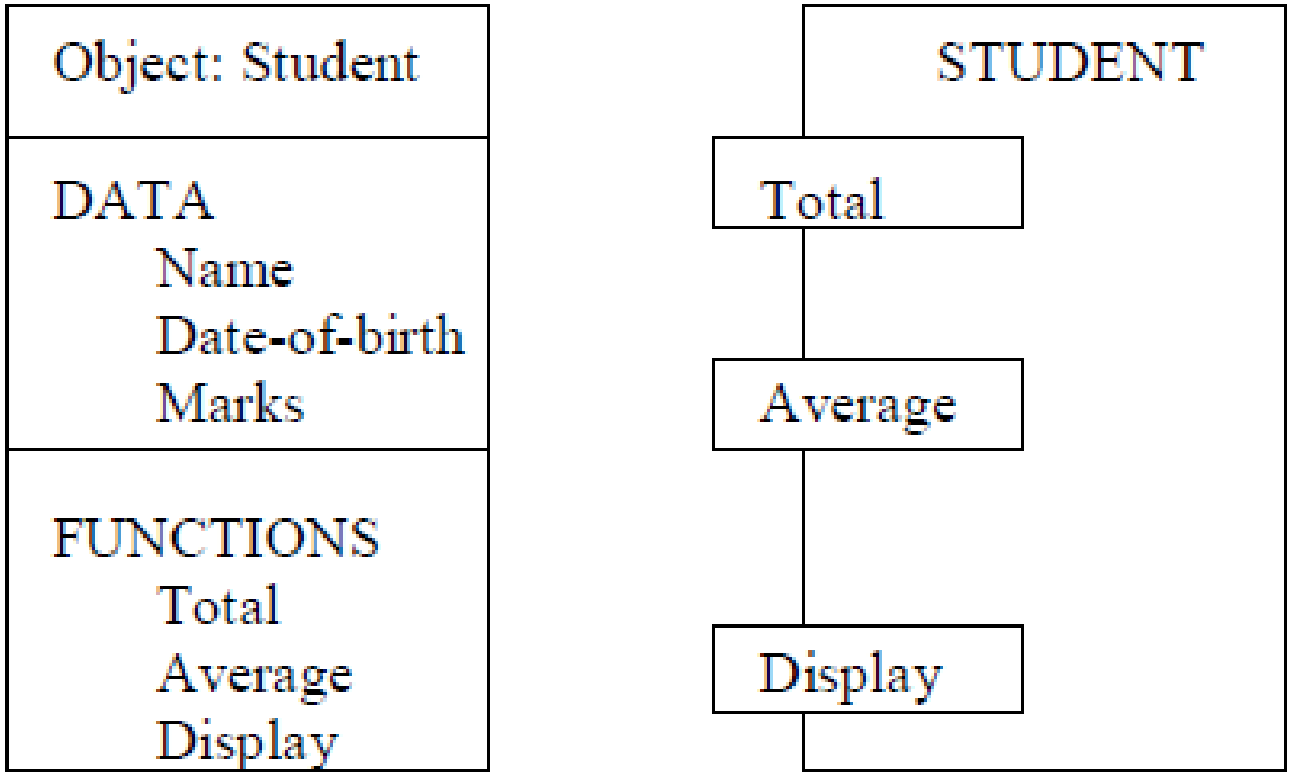
1. OBJECTS

- Objects are the basic **run-time entities** in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program must handle.

- The fundamental idea behind object oriented approach is to **combine both data and function** into a **single unit** and these units are called objects.

- The term objects means **a combination of data and program** that represent some real word entity. For example: consider an example named Amit; Amit is 25 years old and his salary is 2500. The Amit may be represented in a computer program as an object. The data part of the object would be (name: Amit, age: 25, salary: 2500)

Example


Object: Student
DATA
Name
Date-of-birth
Marks
FUNCTIONS
Total
Average
Display
STUDENT
Total
Average
Display

Queue:

```
# Define a class
class Dog:
    # Initialize the object with name and age attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Method to make the dog bark
    def bark(self):
        return f"{self.name} says woof!"

    # Method to get the dog's age
    def get_age(self):
        return self.age
```

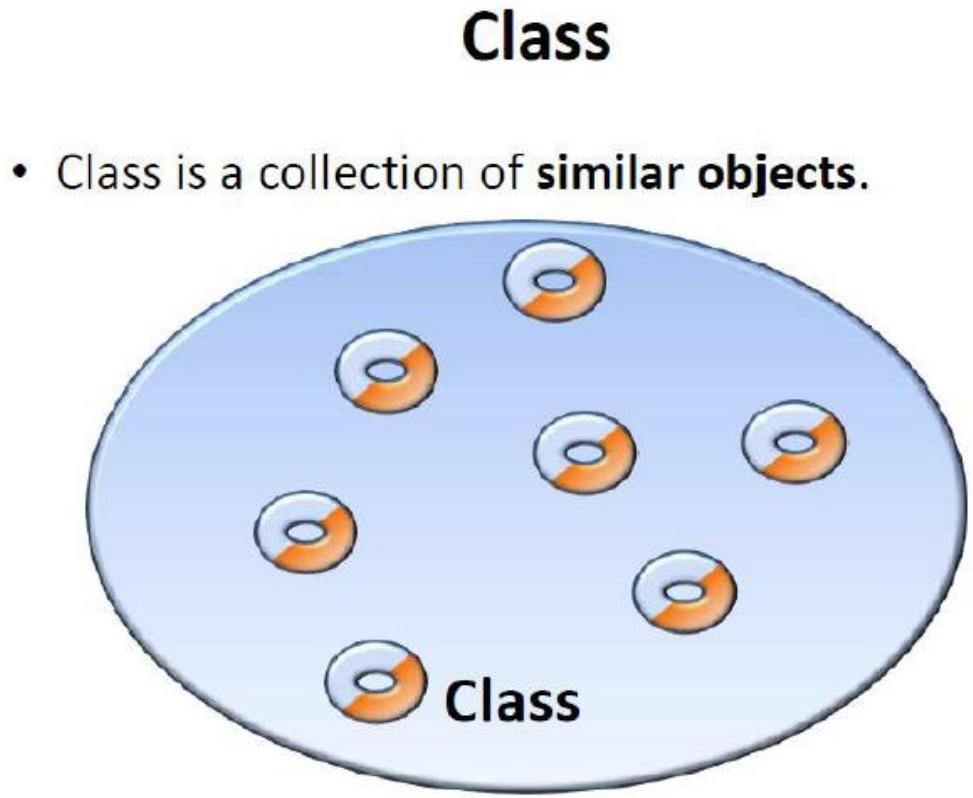
```
# Create an object (instance of the class)
my_dog = Dog("Buddy", 3)

# Access attributes
print(f"My dog's name is {my_dog.name}.")  # Output: My dog's name is Buddy.
print(f"My dog is {my_dog.age} years old.")  # Output: My dog is 3 years old.

# Call methods
print(my_dog.bark())  # Output: Buddy says woof!
print(f"My dog is {my_dog.get_age()} years old.")  # Output: My dog is 3 years old.
```

Output

```
My dog's name is Buddy.
My dog is 3 years old.
Buddy says woof!
My dog is 3 years old.
```

2. CLASS

- A group of objects that share common properties for data part and some program part are collectively called as class.



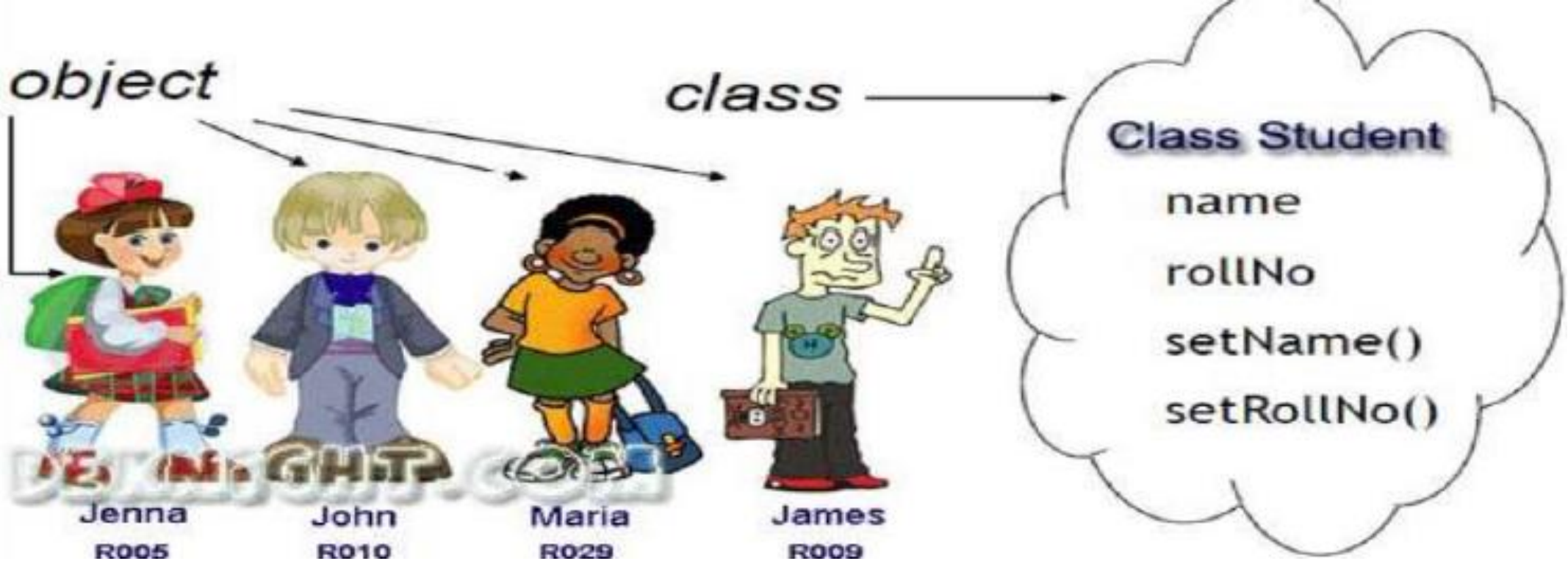
Queue:

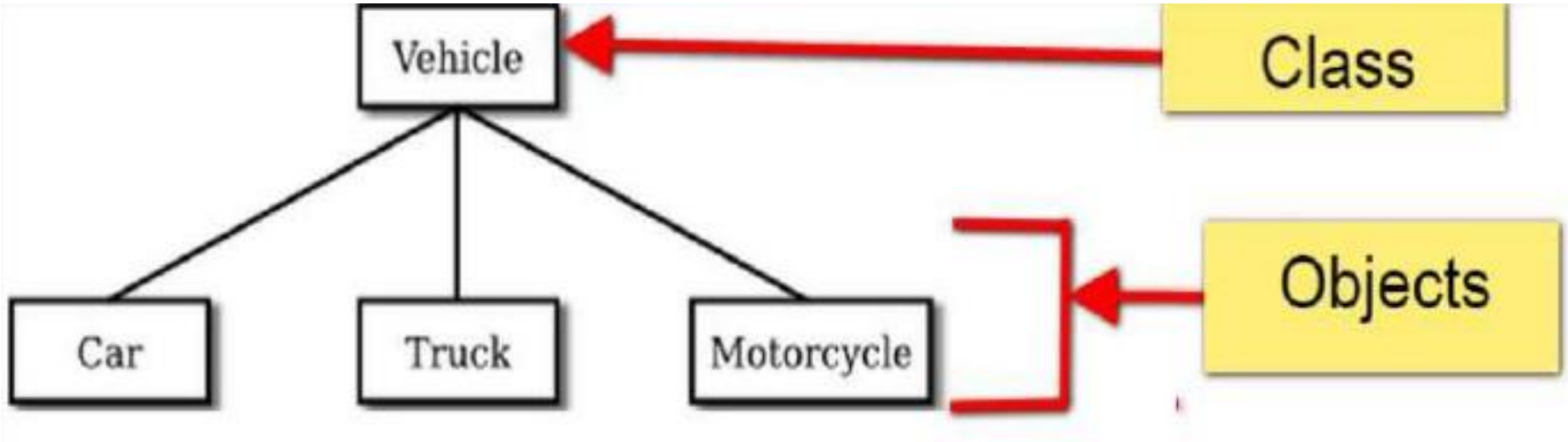
```
# Define a class
class Car:
    # Initialize the object with make, model, and year attributes
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    # Method to get a description of the car
    def description(self):
        return f"{self.year} {self.make} {self.model}"

    # Method to simulate starting the car
    def start(self):
        return f"{self.description()} is starting."
```

```
# Create an object (instance of the class)
my_car = Car("Toyota", "Corolla", 2020)

# Access attributes
print(f"My car's make is {my_car.make}.")  # Output: My car's make is Toyota.
print(f"My car's model is {my_car.model}.")  # Output: My car's model is Corolla.
print(f"My car's year is {my_car.year}.")  # Output: My car's year is 2020.

# Call methods
print(my_car.description())  # Output: 2020 Toyota Corolla
print(my_car.start())        # Output: 2020 Toyota Corolla is starting.
```

Output

```
My car's make is Toyota.
My car's model is Corolla.
My car's year is 2020.
2020 Toyota Corolla
2020 Toyota Corolla is starting.
```


object
class
Class Student
name
rollNo
setName()
setRollNo()
Jenna
R005
John
R010
Maria
R029
James
R009


Vehicle
Class
Car
Truck
Motorcycle
Objects

Person
(class)

Attributes

- Name
- Age
- Gender
- Occupation

Functionality

- Walk ( )
- Eat ( )
- Sleep ( )
- Work ( )

OBJECT	CLASS
Object is an instance ofa class.	Class is a blue print from which objects are created
Object is a real world entity such as chair, pen. table, laptop etc.	Class is a group of similar objects.
Object is a physical entity.	Class is a logical entity.
Object is created many times as per requirement.	Class is declared once.
Object allocates memory when it is created.	Class doesn't allocated memory when it is created.

3. DATA ABSTRACTION

- Abstraction refers to the act of **representing essential features** without including the back ground details or explanations.

- Classes use the concept of abstraction and are defined as size, width and cost and functions to operate on the attributes.

Real life example of Abstraction

Abstraction shows only important things to the user and hides the internal details for example when we ride a bick, we only know about how to ride bick but can not know about how it works and also we do not know internal functionality of bick.

Queue:

```
# Define a class
class Account:
    # Private attribute (indicated by the underscore)
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute

    # Public method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            return f"Deposited ${amount}. New balance: ${self.__balance}"
        return "Deposit amount must be positive.“
# Public method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            return f"Withdrew ${amount}. New balance: ${self.__balance}"
        return "Insufficient balance or invalid amount."
 # Public method to check the balance (abstracted method)
    def check_balance(self):
        return f"Account balance: ${self.__balance}"

# Create an object (instance of the class)
my_account = Account("Alice", 1000)

# Accessing public methods
print(my_account.deposit(500))       # Output: Deposited $500. New balance: $1500
print(my_account.withdraw(200))      # Output: Withdrew $200. New balance: $1300
print(my_account.check_balance())    # Output: Account balance: $1300

# Attempting to access private attribute directly (not recommended)
# print(my_account.__balance)  # This will raise an AttributeError
```

Output

Deposited $500. New balance: $1500

Withdrew $200. New balance: $1300

Account balance: $1300.

4. DATA ENCAPSALATION

- The **wrapping up of data and function into a single unit** (called class) is known as encapsulation.
- The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
- These functions provide the interface between the objects data and the program..

Queue:

```
# Define a class
class Employee:
    # Initialize the object with name and salary attributes
    def __init__(self, name, salary):
        self.name = name
        self.__salary = salary  # Private attribute

    # Public method to set the salary
    def set_salary(self, amount):
        if amount > 0:
            self.__salary = amount
        else:
            print("Salary must be positive!")

    # Public method to get the salary
    def get_salary(self):
        return self.__salary
```

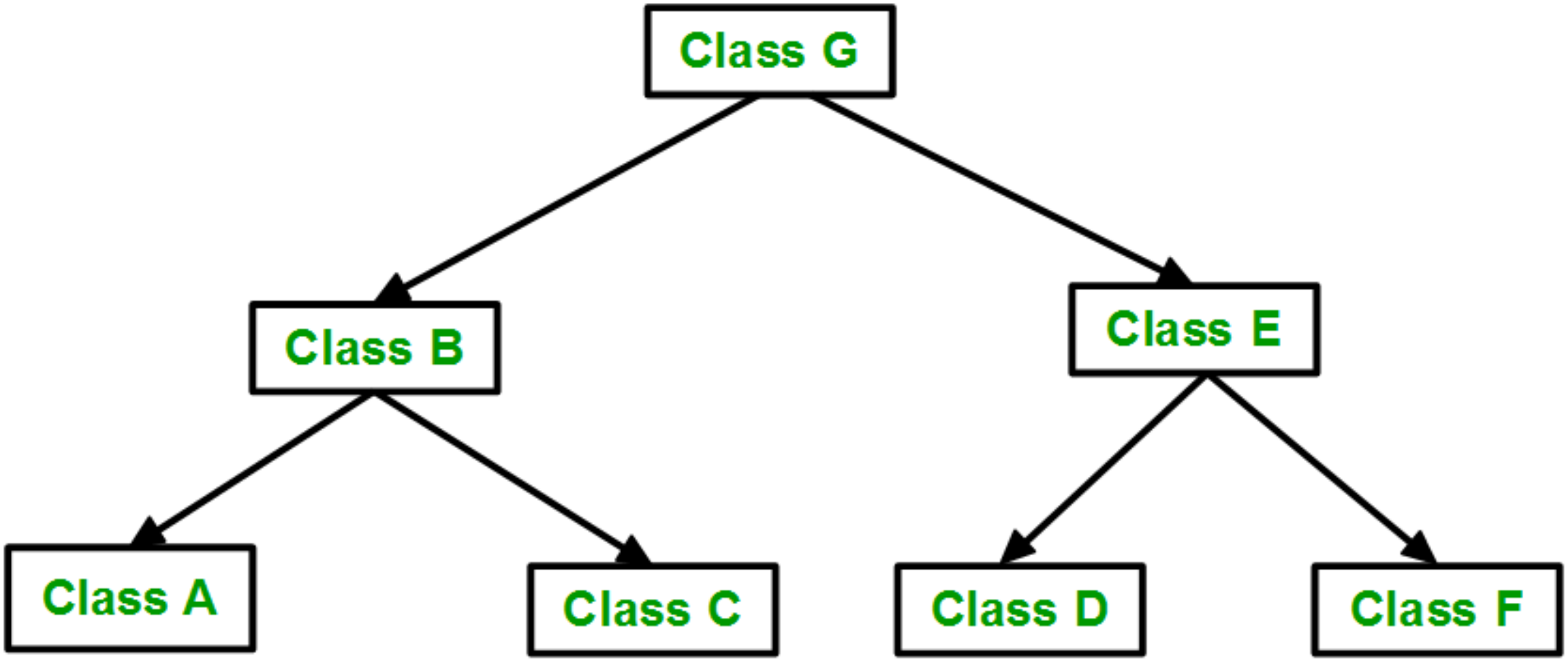
```
# Create an object (instance of the class)
emp = Employee("John", 5000)

# Accessing public attributes
print(emp.name)  # Output: John

# Accessing private attribute directly (not recommended)
# print(emp.__salary)  # This will raise an AttributeError

# Using public methods to interact with private attribute
emp.set_salary(6000)  # Correctly updates the salary
print(emp.get_salary())  # Output: 6000

emp.set_salary(-1000)  # This will not change the salary due to the check in set_salary
print(emp.get_salary())  # Output: 6000
```

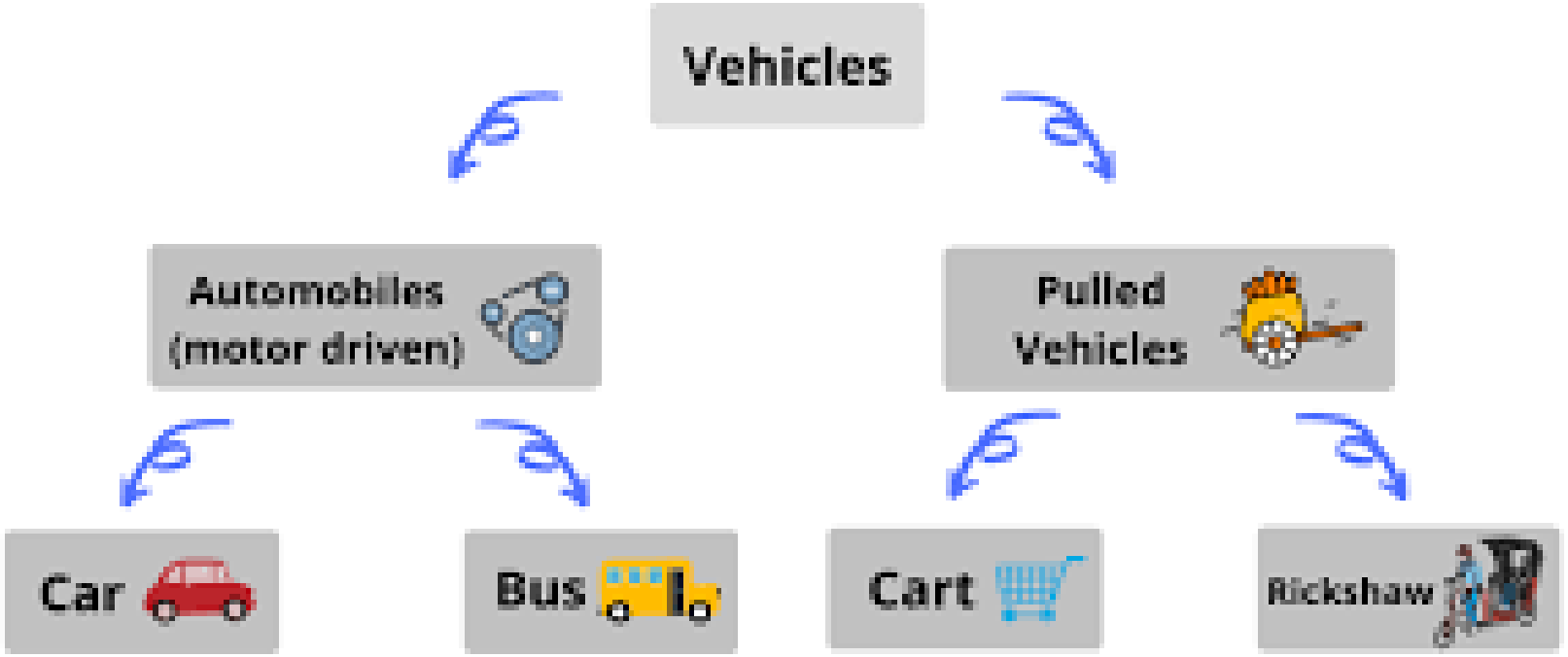
Output

```
John
6000
Salary must be positive!
6000
```

5. INHERITENCE

- Inheritance is the process by which objects of one **class acquire the properties of another class.**
- In the concept of inheritance provides the idea of reusability.
- This mean that we can add additional features to an existing class with out modifying it.
- This is possible by designing a new class will have the combined features of both the classes.


Class G
Class B
Class E
Class A
Class C
Class D
Class F


Vehicles
Automobiles (motor driven)
Pulled Vehicles
Car
Bus
Cart
Rickshaw

Queue:

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

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

# Child class that inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call the constructor of the parent class
        self.breed = breed

    def speak(self):
        return f"{self.name} barks."

# Another child class that inherits from Animal
class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name)  # Call the constructor of the parent class
        self.color = color

    def speak(self):
        return f"{self.name} meows."
```

```
# Create instances of the child classes
dog = Dog("Buddy", "Golden Retriever")
cat = Cat("Whiskers", "Tabby")

# Access attributes and methods
print(dog.name)       # Output: Buddy
print(dog.breed)      # Output: Golden Retriever
print(dog.speak())    # Output: Buddy barks.

print(cat.name)       # Output: Whiskers
print(cat.color)      # Output: Tabby
print(cat.speak())    # Output: Whiskers meows.
```

Output

```
Buddy
Golden Retriever
Buddy barks.
Whiskers
Tabby
Whiskers meows
```

6. POLYMORPHISIM

- Polymorphism means the ability to take more than one form. An operation may exhibit different instance. The behaviour depends upon the type of data used in the operation.

  Meaning of Polymorphism

  Poly - Multiple

  Morphing - Actions

- A language feature that allows a function or operator to be given more than one definition. The types of the arguments with which the function or operator is called determines which definition will be used.

POLYMORPHISIM

- Overloading may be operator overloading or function overloading.

- It is able to express the operation of addition by a single operator say '+'. When this is possible you use the expression x + y to denote the sum of x and y, for many different types of x and y; integers , float and complex no. You can even define the + operation for two strings to mean the concatenation of the strings.

Real life example of polymorphism

Suppose if you are in class room that time you behave like a student, when you are in market at that time you behave like a customer, when you at your home at that time you behave like a son or daughter, Here one person have different-different behaviors.

In Shopping malls behave like Customer

In Bus behave like Passenger

In School behave like Student

At Home behave like Son

POLYMORPHISIM

```
# Base class
class Account:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return f"Deposited ${amount}. New balance: ${self.balance}"
        return "Deposit amount must be positive."

    def withdraw(self, amount):
        if 0 < amount <= self.balance:
            self.balance -= amount
            return f"Withdrew ${amount}. New balance: ${self.balance}"
        return "Insufficient funds or invalid amount."

    def get_balance(self):
        return f"Account balance: ${self.balance}“
# Derived class with additional functionality
class SavingsAccount(Account):
    def __init__(self, account_number, balance, interest_rate):
        super().__init__(account_number, balance)
        self.interest_rate = interest_rate

    def apply_interest(self):
        interest = self.balance * (self.interest_rate / 100)
        self.balance += interest
        return f"Applied interest. New balance: ${self.balance}"
```

```
# Create instances of the classes
checking_account = Account("123456", 1000)
savings_account = SavingsAccount("654321", 2000, 2.5)

# Use the methods of the classes
print(checking_account.deposit(500))  # Output: Deposited $500. New balance: $1500
print(checking_account.withdraw(200))  # Output: Withdrew $200. New balance: $1300
print(checking_account.get_balance())  # Output: Account balance: $1300

print(savings_account.deposit(1000))  # Output: Deposited $1000. New balance: $3000
print(savings_account.apply_interest())  # Output: Applied interest. New balance: $3075.0
print(savings_account.get_balance())  # Output: Account balance: $3075.0
```

Output

```
Deposited $500. New balance: $1500
Withdrew $200. New balance: $1300
Account balance: $1300
Deposited $1000. New balance: $3000
Applied interest. New balance: $3075.0
Account balance: $3075.0
```

Functional programming

- Functional programming (FP) in Python is a programming paradigm where computation is treated as the evaluation of **mathematical functions** and avoids changing state or mutable data.

- In this paradigm, functions are first-class citizens, meaning they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables.

Thank You