Unit 5: Introduction to Server Side Programming (9hrs.)

Compiled By

Lecturer, Sunil Bahadur Bist

NAST, Dhangadhi

Setting up the Python environment

- **Installing Python**

  Download and Install Python from the official website
  https://www.python.org/downloads/
- **Find the Python Version & Installation Directory**

  python --version

  where python
- **To Install python Packages**

  pip install <<packagename>>
- **To list installed python packages**

  pip list

Using a virtual environment

- **To install virtual environment**

  pip install virtualenv

- **To Check the version of virtual environment**

  virtualenv –version

- **To Create your own virtual environment**

  virtualenv <env_name>>

  OR

  python -m venv <<myenv>>

Using a virtual environment…

- **Activate Virtual Environment on Windows**

  cd <env_name>

  Scripts\activate

- **Deactivate Virtual Environment**

  deactivate

(Note: Remember to activate the relevant virtual environment every time you work on the project)

Choose a Code Editor/IDE

- **Install VSCode**

  Download and Install VSCode and Python Plugin

  https://code.visualstudio.com/

- **PyCharm IDE**

  Download and Install PyCharm Community Python IDE

  https://www.jetbrains.com/pycharm/

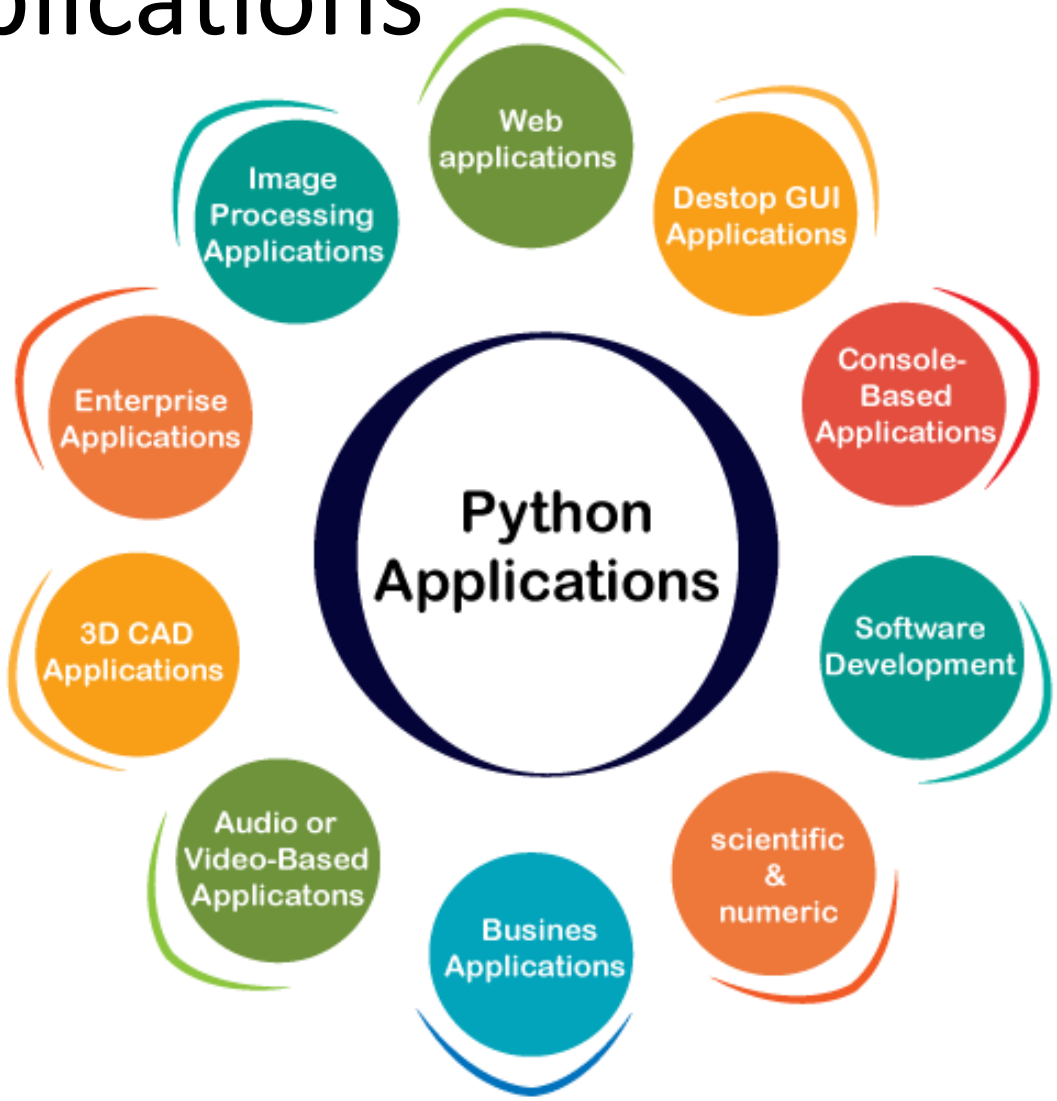
Introduction to Python

- Python is a popular programming language.
- It was created by Guido van Rossum, and released in 1991.
- Python programs are executed by an interpreter.
- There are many different environments in which the Python interpreter might run an IDE, a browser, or a terminal window.
- It is a versatile programming language used for a wide range of applications, including web development, software development, data analysis, machine learning, and automation.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Python Applications



Python syntax

- Python was designed for readability, and has some similarities to the English language with influence from mathematics.
- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.
- Python 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.

Python first program

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

Save as hello.py

To run the program

python hello.py

The Python Command Line

- Type py or python in command prompt
- When the interpreter starts, a prompt appears where you can type programs into a so-called “read-evaluation-print loop” (or REPL).
- For example, in the following output, the interpreter displays its copyright message and presents the user with the >>> prompt, at which the user types a familiar “Hello World” program:

```
>>> print('Hello World')
Hello World
```

Python Comments

- Comments in python can be used to explain Python code.
- It can be used to make the code more readable.
- It can be used to prevent execution when testing code.
- It starts with a #, and Python will ignore them

For example,

```
#This is a simple single line comment
print("Hello, World!")
```

Python Comments

- Python does not really have a syntax for multiline comments.
- To add a multiline comment you could insert a # for each line:

For example,

```
#This is a comment
#written in
#more than just one line
print("Hello, World!")
```

OR

Python Comments

```
"""This is a comment
written in M
more than just one line
"""
print("Hello, World!")
```

Or

```
'‘’ Comment  in
Multiline
multiple
line ‘‘’
```

As long as the string is not assigned to a variable, Python will read the code, but then ignore it, and you have made a multiline comment.

Basic operations

- Python has several types of operators to perform different operations:
- **Arithmetic operators:**
- Used for mathematical calculations.

+ (addition)
- (subtraction)
* (multiplication)
/ (division)
// (floor division - returns the integer part of the division)
% (modulus - returns the remainder of the division)
** (exponentiation)

Basic operations

- **Assignment operators:**
- Used to assign values to variables.

= (assigns the value on the right to the variable on the left)

+=, -=, *=, /=, //=, %=, **=

(compound assignment operators, combine an arithmetic operation with assignment)

Basic operations

- **Comparison operators:**
- Used to compare values.

== (equal to)

!= (not equal to)

> (greater than)

< (less than)

>= (greater than or equal to)

<= (less than or equal to)

Basic operations

- **Logical operators:**
- Used to combine or modify boolean values.

and (returns True if both operands are True)

or (returns True if at least one operand is True)

not (returns the opposite boolean value of the operand)

Basic operations

- **Bitwise operators:**
- Used to perform operations on individual bits.

& (AND)

| (OR)

^ (XOR)

~ (NOT)

<< (left shift)

>> (right shift)

Basic operations

- **Membership operators:**
- Used to check if a value is present in a sequence (e.g., list, tuple, string).

in (returns True if the value is present)

not in (returns True if the value is not present)

Basic operations

- **Identity operators:**
- Used to compare the identity of objects (i.e., if they are the same object in memory).

is (returns True if both operands are the same object)

is not (returns True if both operands are not the same object)

Basic operations

- The order of precedence of operators in Python (from highest to lowest) is:
- Parentheses, Exponentiation, Multiplication, Division, Floor division, Modulus, Addition, Subtraction, Comparison operators, Logical operators, and Assignment operators.

Data Types and Variables

- Python has several built-in data types used to classify data.
- Variables are names assigned to stored data values, acting as references to memory locations.

Data Types

- **Numeric Types:**
- int: Represents whole numbers (e.g., 10, -5, 0).
- float: Represents real numbers with decimal points (e.g., 3.14, -2.5).
- complex: Represents complex numbers in the form a + bj, where a and b are floats and j is the imaginary unit (e.g., 2 + 3j).

Data Types

String Type:

- str: Represents a sequence of characters, used for text (e.g., "hello", 'world').

Sequence Types:

- list: An ordered, mutable (changeable) collection of items (e.g., [1, 2, 3], ['a', 'b', 'c']).
- tuple: An ordered, immutable (unchangeable) collection of items (e.g., (1, 2, 3), ('a', 'b', 'c')).
- range: Generates a sequence of numbers within a specified range.

Data Types

Mapping Type:

- dict: A collection of key-value pairs, where keys are unique and immutable (e.g., {'name': 'Alice', 'age': 30}).

Set Types:

- set: An unordered collection of unique items (e.g., {1, 2, 3}).
- frozenset: An immutable version of a set.

Boolean Type:

- bool: Represents truth values, either True or False.

Data Types

Binary Types:

- bytes: Represents immutable sequences of bytes.
- bytearray: Represents mutable sequences of bytes.
- memoryview: Provides a memory view of an object without copying.

None Type:

- NoneType: Represents the absence of a value (None).

Variables

- Variables are created when a value is assigned to them.
- Python is dynamically typed, meaning the data type of a variable is automatically inferred at runtime and can be changed.
- For example,

```
x = 10 # x is an integer
x = "hello" # x is now a string
```

Variables

- Variable names must follow certain rules:
- They can contain letters, numbers, and underscores, but cannot start with a number.
- They are also case-sensitive.
- For example,

```
my_variable = 10
_my_variable = 20
myVariable = 30
```

Variables

- If you want to specify the data type of a variable, this can be done with casting.

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

- **Checking Data Types**
- The type() function is used to determine the data type of a variable.
- For example,

```
x = 10
print(type(x)) # Output: <class 'int'>
```

Variable Assignment & Naming Conventions

- Variable assignment in Python involves associating a name with a value, allowing the program to store and manipulate data.
- The assignment is done using the equals sign (=).
- For example, x = 10 assigns the value 10 to the variable x.
- Python is dynamically typed, so you don't need to explicitly declare the variable type; it's inferred from the assigned value.

Variable Assignment

Basic Assignment

- For example,

```
name = "Alice“
age = 30
price = 99.99
is_active = True
```

Variable Assignment

Multiple Assignment

- Python allows assigning the same value to multiple variables in a single line or assigning multiple values to multiple variables.
- For example,

```
a = b = c = 100
x, y, z = 1, 2, 3
```

Variable Assignment

Reassignment

- A variable can be reassigned to a new value, even of a different type.
- For example,

```
x = 5
x = "Hello"
```

Variable Naming Rules

- Variable names can contain letters, numbers, and underscores.
- They must start with a letter or an underscore.
- They are case-sensitive (myVar and myvar are different).

Unpacking

- Python allows unpacking values from collections (like lists or tuples) into variables.

```
fruits = ["apple", "banana", "cherry"]
x, y, z = fruits
```

Variable Naming Rules

Deleting variables

- A variable can be deleted using the del keyword

```
x = 10
del x
```

- # trying to print(x) here will raise an error since x does not exist anymore

Naming Conventions

- Python naming conventions, as outlined in PEP 8, aim to improve code readability and consistency.
- PEP stands for Python Enhancement Proposal.
- Here's a summary:

Modules and Packages:

  - Short, lowercase names.
  - Underscores can be used if it improves readability (e.g., my_module, my_package).

Naming Conventions

Classes:

- Use CamelCase (e.g., MyClass, HTTPServer).
- Functions and Methods:
- Lowercase with words separated by underscores (snake_case) (e.g., calculate_area, get_name).

Variables:

- Lowercase with words separated by underscores (snake_case) (e.g., user_name, total_count).

Constants:

- All uppercase with words separated by underscores (e.g., MAX_SIZE, API_KEY).

Naming Conventions

Avoid:

- Single-character names (except for counters or iterators).
- Reserved keywords.
- Names that are too general or too verbose.

Be Descriptive:

- Choose names that clearly indicate the purpose of the variable, function, or class.

Naming Conventions

Case Sensitivity:

- Python is case-sensitive, so myVar and myvar are different variables.

Underscores:

- Leading underscores (e.g., _my_var) suggest internal use.
- Double leading underscores (e.g., __my_var) name mangling for class-specific variables.
- Trailing underscores (e.g., my_var_) are sometimes used to avoid conflicts with Python keywords.

Acronyms:

- When using acronyms in CamelCase names, capitalize all letters of the acronym (e.g., HTTPServer instead of HttpServer).

Type Conversion

Type conversion, also known as type casting, involves changing a variable's data type from one type to another.

Python supports two main types of conversion:

- implicit and
- explicit.

Implicit Type Conversion

- Python automatically converts one data type to another without the need for explicit user intervention.
- This usually happens when performing operations involving different data types.
- For example, when adding an integer and a float, Python will implicitly convert the integer to a float before performing the addition.

Implicit Type Conversion

- For example,

```
a_int = 10
b_float = 5.5
result = a_int + b_float
print(result)
print(type(result))
```

Output:

- 15.5
- <class 'float'>

Explicit Type Conversion

- Users manually convert the data type of an object using built-in functions.
- This is useful when you need to convert a value to a specific type for further operations or to meet certain requirements.

    int(): Converts a value to an integer data type.

    float(): Converts a value to a floating-point number.

    str(): Converts a value to a string.

    bool(): Converts a value to a boolean (True or False).

    list(), tuple(), set(): Convert to list, tuple and set types respectively.

Explicit Type Conversion

- For exaple,

```
num_str = "123"
num_int = int(num_str)
print(num_int)
print(type(num_int))
price_str = "99.99"
price_float = float(price_str)
print(price_float)
print(type(price_float))
age = 30
age_str = str(age)
print(age_str)
print(type(age_str))
```

Output:

```
123
<class 'int'>
99.99
<class 'float'>
30
<class 'str'>
```

Control Structures

- Control structures in Python manage the flow of execution within a program.
- They enable decisions, repetitions, and alterations in the sequence of code execution.
- The primary control structures in Python are conditional statements and loops.
- Conditional statements allow different blocks of code to be executed based on whether a condition is true or false.

Control Structures

- **if statement:** Executes a block of code if a condition is true.

For example,

```
x = 10
if x > 5:
    print("x is greater than 5")
```

Control Structures

- **elif statement:** Checks an additional condition if the preceding if condition is false.

For example,

```
x = 3
if x > 5:
    print("x is greater than 5")
elif x > 2:
    print("x is greater than 2 but not greater than 5")
```

Control Structures

- **else statement:** Executes a block of code if all preceding if and elif conditions are false.

For example,

```
x = 1
if x > 5:
    print("x is greater than 5")
elif x > 2:
    print("x is greater than 2 but not greater than 5")
else:
    print("x is not greater than 2")
```

Loops

- Loops allow a block of code to be executed repeatedly.
- **for loop:** Iterates over a sequence (like a list, tuple, or string) or other iterable objects.

For example,

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

Loops

You can use the range() function with a step argument to achieve this. The range() function takes three arguments: start, stop, and step. The step argument specifies the increment value.

```
for i in range(0, 10, 2):
        print(i)
```

Loops

- **while loop:** Executes a block of code as long as a condition is true.
- For example,

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

Loop Control Statements

- Loop control statements alter the execution of loops.
- **break statement:** Terminates the loop prematurely.

For example,

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Loop Control Statements

- **continue statement:** Skips the rest of the current iteration and proceeds with the next iteration of the loop.
- For example,

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Loop Control Statements

- **pass statement:** It is used as a placeholder for future code.
- When the pass statement is executed, nothing happens, but you avoid getting an error when empty code is not allowed.
- For example,

```
for i in range(5):
    if i == 2:
        pass  # Do nothing for i == 2
    else:
        print(i)
```

List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

Example:

- Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
- Without list comprehension you will have to write a for statement with a conditional test inside:

List Comprehension

for example,

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)
```

Output:

['apple', 'banana', 'mango']

List Comprehension

- With list comprehension you can do all that with only one line of code:
- For example,

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

Output:

['apple', 'banana', 'mango']

List Comprehension

The Syntax

```
newlist = [expression for item in iterable if condition == True]
```

- The return value is a new list, leaving the old list unchanged.

Condition

- The condition is like a filter that only accepts the items that evaluate to True.

Example

- Only accept items that are not "apple":

```
newlist = [x for x in fruits if x != "apple"]
```

List Comprehension

- The condition if x != "apple" will return True for all elements other than "apple", making the new list contain all fruits except "apple".
- The condition is optional and can be omitted:

Example

- With no if statement:

```
newlist = [x for x in fruits]
```

Iterable

- The iterable can be any iterable object, like a list, tuple, set etc.

List Comprehension

Example

- You can use the range() function to create an iterable:

```
newlist = [x for x in range(10)]
 # Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Same example, but with a condition:

Example

- Accept only numbers lower than 5:

```
newlist = [x for x in range(10) if x < 5] # [0, 1, 2, 3, 4]
```

List Comprehension

Expression

- The expression is the current item in the iteration, but it is also the outcome, which you can manipulate before it ends up like a list item in the new list:

Example

- Set the values in the new list to upper case:

```
newlist = [x.upper() for x in fruits]
```

List Comprehension

- You can set the outcome to whatever you like:

Example

- Set all values in the new list to 'hello’:

```
newlist = ['hello' for x in fruits]
```

- The expression can also contain conditions, not like a filter, but as a way to manipulate the outcome:

List Comprehension

Example

- Return "orange" instead of "banana":

```
newlist = [x if x != "banana" else "orange" for x in fruits]
```

- The expression in the example above says:
- "Return the item if it is not banana, if it is banana return orange".

Working with Python Data Structures

- Lists: mylist = ["apple", "banana", "cherry"]
- Dictionaries: thisdict = {"brand": "Ford", "model": "Mustang", "year": 1964}
- Tuples: mytuple = ("apple", "banana", "cherry") and
- Sets: myset = {"apple", "banana", "cherry"}

Lists

- 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.
- When we say that lists are ordered, it means that the items have a defined order, and that order will not change.
- If you add new items to a list, the new items will be placed at the end of the list.
- List items are indexed, the first item has index [0], the second item has index [1] etc.

Lists

- To determine how many items a list has, use the len() function

```
print(len(mylist))
```

- List, tuple,set items can be of any data type:

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

- A list, tuple,set can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
```

Lists

- It is also possible to use the list() constructor when creating a new list.

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

- List items are indexed and you can access them by referring to the index number:

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

Lists

- Negative indexing means start from the end
- -1 refers to the last item, -2 refers to the second last item etc.

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

- List, tuple, set supports negative indexing

Lists

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

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(2, "watermelon")
print(thislist)
```

Lists

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

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

Lists

- The remove() method removes the specified item.

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)
```

Lists

- The pop() method removes the specified index.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)
```

- If you do not specify the index, the pop() method removes the last item.

```
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)
```

Dictionaries

- Dictionaries are used to store data values in key:value pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered.
- Dictionaries are written with curly brackets, and have keys and values: {}
- It is also possible to use the dict() constructor to make a dictionary.

Dictionaries

```
thisdict = dict(name = "John", age = 36, country = "Norway")
print(thisdict)
```

- You can access the items of a dictionary by referring to its key name, inside square brackets:

```
x = thisdict["model"]
```

- There is also a method called get() that will give you the same result:

```
x = thisdict.get("model")
```

Dictionaries

- The keys() method will return a list of all the keys in the dictionary.

```
x = thisdict.keys()
```

- You can change the value of a specific item by referring to its key name:

```
thisdict["year"] = 2018
```

- The update() method will update the dictionary with the items from the given argument.
- The argument must be a dictionary, or an iterable object with key:value pairs.

```
thisdict.update({"year": 2020})
```

Dictionaries

- Adding an item to the dictionary is done by using a new index key and assigning a value to it:

```
thisdict["color"] = "red"
print(thisdict)
```

- You can also use the values() method to return values of a dictionary:

```
for x in thisdict.values():
print(x)
```

- You can use the keys() method to return the keys of a dictionary:

Dictionaries

```
for x in thisdict.keys():
 print(x)
```

- Loop through both keys and values, by using the items() method:

```
for x, y in thisdict.items():
 print(x, y)
```

Tuples

- 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. ()

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

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.

Tuples

- One item tuple, remember the comma:

```
thistuple = ("apple",)
print(type(thistuple))
```

- #NOT a tuple

```
thistuple = ("apple")
print(type(thistuple))
```

Tuples

- It is also possible to use the tuple() constructor to make a tuple.

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

- You can access tuple items by referring to the index number, inside square brackets:

```
print(thistuple[1])
```

Tuples

- Convert the tuple into a list to be able to change it:

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

Tuples

- Convert the tuple into a list, remove "apple", and convert it back into a tuple:

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

Sets

- Sets are used to store multiple items in a single variable.
- A set is a collection which is unordered, unchangeable*, and unindexed.
- * Note: Set items are unchangeable, but you can remove items and add new items.
- Sets are written with curly brackets. {}
- Set items are unordered, unchangeable, and do not allow duplicate values.
- It is also possible to use the set() constructor to make a set.

```
my_set = set((1, 2, 2, 3, 4, 4, 5))
print(my_set)
```

Sets

- To remove an item in a set, use the remove(), or the discard() method.

```
thisset = {"apple", "banana", "cherry"}
thisset.remove("banana")
print(thisset)
```

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

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

Sets

- To add items from another set into the current set, use the update() method.

```
thisset = {"apple", "banana", "cherry"}

tropical = {"pineapple", "mango", "papaya"}

thisset.update(tropical)

print(thisset)
```

Working with Python Data Structures

Assignment

- Differences between lists, tuples, and sets

Introduction to functions

- In Python, a function is a reusable block of code designed to perform a specific task.
- Functions help organize code, making it more readable, maintainable, and efficient by avoiding repetition.

```
def say_hello():
  print("Hello from a function")
```

Types of Functions

- **Built-in functions:** Python provides several built-in functions like print(), len(), max(), etc.
- **User-defined functions:** Functions created by the user to suit their specific needs.
- **Lambda functions:** Anonymous, small, single-expression functions defined using the lambda keyword.

Defining a Function

- A function is defined using the def keyword, followed by the function name, parentheses (), and a colon :
- The code block within the function is indented.

```
def say_hello():
    print("Hello, Programmer!")
```

Calling a Function

- To execute a function, you call it by its name followed by parentheses.

say_hello() # Output: Hello Programmer!

Function Arguments

- Functions can take arguments (inputs) to operate on.
- These are specified within the parentheses in the function definition.

```
def greet(name):
    print(f"Hello, {name} welcome!")

greet(“Sunil")  # Output: Hello, Sunil Welcome!
greet(“NAST")  # Output: Hello, NAST Welcome!
greet(“BCA")  # Output: Hello, BCA Welcome!
```

Return Statement

- The return statement is used to exit a function and return a value.
- If no return statement is present, the function returns None.

```
#Defining
def add(x, y):
    """This function adds two numbers."""
    return x + y
 #Calling
result = add(5, 3)
print(result)  # Output: 8
```

Scope and global variables

- In programming, scope determines where a variable is accessible within a program.
- **Global variables** have a global scope, meaning they can be accessed from anywhere in the program, while **local variables** have a limited scope, typically within a function or block of code.

Scope and global variables

Scope:

- Scope defines the visibility and accessibility of a variable.
- It determines which parts of the code can access and modify a variable.

Global Variables:

- Global variables are declared outside of any function or block of code.
- They are accessible from anywhere within the program.

Local Variables:

- Local variables are declared within a function or a block of code.
- Their scope is limited to that function or block, meaning they cannot be accessed from outside it.

Scope and global variables

```
global_var = 10

def my_function():
    local_var = 5
    print(local_var)  # Output: 5
    print(global_var) # Output: 10


my_function()
# print(local_var)  # This would cause an error, as local_var is not defined outside the function
print(global_var)  # Output: 10
```

Scope and global variables

Shadowing:

- If a local variable has the same name as a global variable, the local variable will "shadow" the global variable within the scope of the function or block, meaning the local variable takes precedence.

Python Lambda Function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.

Syntax:

lambda arguments : expression

- The expression is executed and the result is returned:

Python Lambda Function

Example

- Add 10 to argument a, and return the result:

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

Python Lambda Function

- Lambda functions can take any number of arguments for **Example,**
- Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example

- Summarize argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

File Handling in Python

- File handling is an important part of any web application.
- Python has several functions for creating, reading, updating, and deleting files.
- The key function for working with files in Python is the open() function.
- The open() function takes two parameters; filename, and mode.

File Handling in Python

- There are four different methods (modes) for opening a file:
- **"r" - Read** - Default value. Opens a file for reading, error if the file does not exist
- **"a" - Append** - Opens a file for appending, creates the file if it does not exist
- **"w" - Write** - Opens a file for writing, creates the file if it does not exist
- **"x" - Create** - Creates the specified file, returns an error if the file exists

File Handling in Python

- In addition you can specify if the file should be handled as binary or text mode
- **"t" - Text** - Default value. Text mode
- **"b" - Binary** - Binary mode (e.g. images)

File Handling in Python

- **Syntax**
- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

- The code above is the same as:

```
f = open("demofile.txt", "rt")
```

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.

File Handling in Python

demofile.txt

Hello! Welcome to demofile.txt
This file is for testing purposes.
Good Luck!

```
f = open("demofile.txt")
print(f.read())
```

File Handling in Python

- If the file is located in a different location, you will have to specify the file path, like this:

Example

- Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt")
print(f.read())
```

File Handling in Python

- **Using the with statement**
- You can also use the with statement when opening a file:

```
with open("demofile.txt") as f:
print(f.read())
```

File Handling in Python

- **Close Files**
- It is a good practice to always close the file when you are done with it.
- If you are not using the with statement, you must write a close statement in order to close the file:

Example

- Close the file when you are finished with it:

```
f = open("demofile.txt")
print(f.readline())
f.close()
```

File Handling in Python

- **Read Only Parts of the File**
- By default the read() method returns the whole text, but you can also specify how many characters you want to return:

Example

- Return the 5 first characters of the file:

```
with open("demofile.txt") as f:
 print(f.read(5))
```

File Handling in Python

- **Read Lines**
- You can return one line by using the readline() method:

Example

- Read one line of the file:

```
with open("demofile.txt") as f:
 print(f.readline())
```

File Handling in Python

- By calling readline() two times, you can read the two first lines:

Example

- Read two lines of the file:

```
with open("demofile.txt") as f:
 print(f.readline())
 print(f.readline())
```

File Handling in Python

- By looping through the lines of the file, you can read the whole file, line by line:

Example

- Loop through the file line by line:

```
with open("demofile.txt") as f:
  for x in f:
   print(x)
```

File Handling in Python

- **Write to an Existing File**
- To write to an existing file, you must add a parameter to the open() function:
- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content

File Handling in Python

- Open the file "demofile.txt" and append content to the file:

```
with open("demofile.txt", "a") as f:
 f.write("Now the file has more content!")
```

- #open and read the file after the appending:

```
with open("demofile.txt") as f:
  print(f.read())
```

File Handling in Python

- **Overwrite Existing Content**
- To overwrite the existing content to the file, use the w parameter:

Example

- Open the file "demofile.txt" and overwrite the content:

```
with open("demofile.txt", "w") as f:
 f.write("Woops! I have deleted the content!")
```

File Handling in Python

- **#open and read the file after the overwriting:**

```
with open("demofile.txt") as f:
  print(f.read())
```

Example

- **Create a new file called "myfile.txt":**

```
f = open("myfile.txt", "x")
```

File Handling in Python

- **Delete a File**
- To delete a file, you must import the OS module, and run its os.remove() function:
- **Remove the file "demofile.txt":**

```
import os

os.remove("demofile.txt")
```

File Handling in Python

- **Check if File exist:**
- To avoid getting an error, you might want to check if the file exists before you try to delete it:

Example

- **Check if file exists, then delete it:**

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

File Handling in Python

- **Delete Folder**
- To delete an entire folder, use the os.rmdir() method:

Example

- **Remove the folder "myfolder":**

```
import os
os.rmdir("myfolder")
```

Working with CSV and JSON Files

- Working with **CSV** and **JSON** files in Python involves using the csv and json libraries to read, write, and manipulate these structured data formats.
- **CSV (Comma Separated Values)** is a simple text format for tabular data,
- while **JSON (JavaScript Object Notation)** is a more flexible format for storing structured data.
- Both are commonly used for data storage, transfer, and analysis.

Working with CSV and JSON Files

```
# Example of writing to a CSV file
import csv
with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerow(['name','email','mobile'])  # Write header
    csv_writer.writerow(['sunil','sunil@nast.edu.np','9858585858’]) #data
    csv_writer.writerow(['ravi', 'ravi@nast.edu.np', '9858585859']) #data
    csv_writer.writerow(['pusp', 'pusp@nast.edu.np', '9858585860']) #data
```

Working with CSV and JSON Files

```
# Example of reading a CSV file
import csv
with open('output.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)
```

Working with CSV and JSON Files

```
# Example of writing to a JSON file
import json
data = {'name': 'Sunil', 'age': 35, 'city': 'Dhangadhi'}
with open('output.json', 'w') as file:
    json.dump(data, file, indent=4) # 'indent' for pretty-printing
```

Working with CSV and JSON Files

```
# Example of reading a JSON file
import json
with open('output.json', 'r') as file:
    data = json.load(file)
    print(data)
```

Python OOPs Concepts

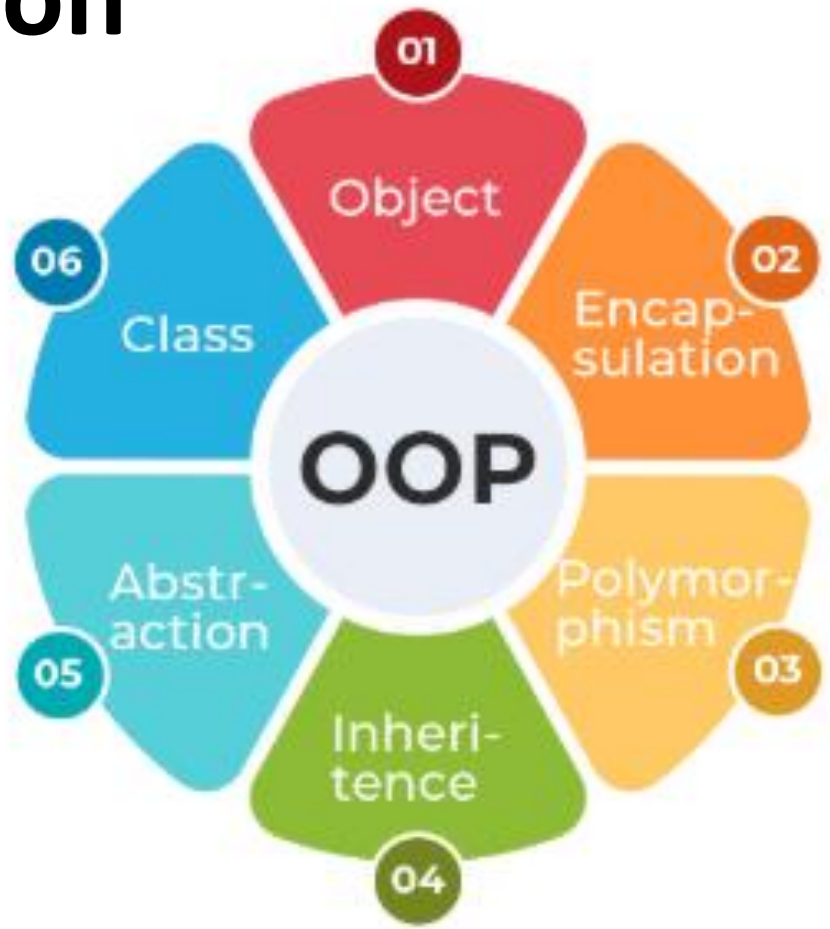
- Object Oriented Programming is a fundamental concept in Python, empowering developers to build modular, maintainable, and scalable applications.
- By understanding the core OOP principles (classes, objects, inheritance, encapsulation, polymorphism, and abstraction), programmers can leverage the full potential of Python OOP capabilities to design elegant and efficient solutions to complex problems.

Python OOPs Concepts

- OOPs is a way of organizing code that uses objects and classes to represent real-world entities and their behavior.
- In OOPs, object has attributes thing that has specific data and can perform certain actions using methods.

OOPs Concepts in Python

- Class in Python
- Objects in Python
- Inheritance in Python
- Polymorphism in Python
- Encapsulation in Python
- Data Abstraction in Python



Python OOPs Concepts

Python Class

- A class is a collection of objects. Classes are blueprints for creating objects. A class defines a set of attributes and methods that the created objects (instances) can have.

- **Some points on Python class:**
  - Classes are created by keyword class.
  - Attributes are the variables that belong to a class.
  - Attributes are always public and can be accessed using the dot (.) operator. Example: Myclass.my_attribute

Creating a Class

- Here, the class keyword indicates that we are creating a class followed by name of the class (Dog in this case).

```
class Dog:
    species = "Canine"  # Class attribute


    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute
```

Creating a Class

Explanation:

- class Dog: Defines a class named Dog.
- species: A class attribute shared by all instances of the class.
- **__init__ method:** Initializes the name and age attributes when a new object is created.

Python Objects

- An Object is an **instance of a Class**.
- It represents a specific implementation of the class and holds its own data.
- **An object consists of:**
  - **State:** It is represented by the attributes and reflects the properties of an object.
  - **Behavior:** It is represented by the methods of an object and reflects the response of an object to other objects.
  - **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Creating Object

- Creating an object in Python involves instantiating a class to create a new instance of that class.
- This process is also referred to as object instantiation.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute
```

Creating Object

- # Creating an object of the Dog class

```
dog1 = Dog("Buddy", 3)


print(dog1.name)
print(dog1.species)
```

Output:

```
Buddy
Canine
```

Creating Object

Explanation:

- dog1 = Dog(“Buddy”, 3): Creates an object of the Dog class with name as “Buddy” and age as 3.
- dog1.name: Accesses the instance attribute name of the dog1 object.
- dog1.species: Accesses the class attribute species of the dog1 object.

Self Parameter

- self parameter is a reference to the current instance of the class.
- It allows us to access the attributes and methods of the object.

```
class Dog:
    species = "Canine"  # Class attribute

    def __init__(self, name, age):
        self.name = name  # Instance attribute
        self.age = age  # Instance attribute
```

Self Parameter

```
dog1 = Dog("Buddy", 3)  # Create an instance of Dog
dog2 = Dog("Charlie", 5)  # Create another instance of Dog

print(dog1.name, dog1.age, dog1.species)  # Access instance and class attributes
print(dog2.name, dog2.age, dog2.species)  # Access instance and class attributes
print(Dog.species)  # Access class attribute directly
```

Output

```
Buddy 3 Canine
Charlie 5 Canine
Canine
```

Self Parameter

Explanation:

- self.name: Refers to the name attribute of the object (dog1) calling the method.
- dog1.bark(): Calls the bark method on dog1.

__init__ Method

- **__init__** method is the constructor in Python, automatically called when a new object is created.
- It initializes the attributes of the class.

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

__init__ Method

```
dog1 = Dog("Buddy", 3)
print(dog1.name)
```

Output:

Buddy

Explanation:

- __init__: Special method used for initialization.
- self.name and self.age: Instance attributes initialized in the constructor.

Class and Instance Variables

- In Python, variables defined in a class can be either class variables or instance variables, and understanding the distinction between them is crucial for object-oriented programming.

Class Variables

- These are the variables that are shared across all instances of a class.
- It is defined at the class level, outside any methods.
- All objects of the class share the same value for a class variable unless explicitly overridden in an object.

Instance Variables

- Variables that are unique to each instance (object) of a class.
- These are defined within the __init__ method or other instance methods.
- Each object maintains its own copy of instance variables, independent of other objects.

Instance Variables

```
class Dog:
    # Class variable
    species = "Canine"
    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

# Create objects
dog1 = Dog("Buddy", 3)
dog2 = Dog("Charlie", 5)
```

Instance Variables

```
# Access class and instance variables
print(dog1.species)  # (Class variable)
print(dog1.name)     # (Instance variable)
print(dog2.name)     # (Instance variable)

# Modify instance variables
dog1.name = "Max"

print(dog1.name)     # (Updated instance variable)
# Modify class variable
Dog.species = "Feline"
```

Instance Variables

```
print(dog1.species)  # (Updated class variable)
print(dog2.species)
```

Output

```
Canine
Buddy
Charlie
Max
Feline
Feline
```

Instance Variables

Explanation:

- Class Variable (species):
  - Shared by all instances of the class.
  - Changing Dog.species affects all objects, as it’s a property of the class itself.
- Instance Variables (name, age):
  - Defined in the __init__ method.
  - Unique to each instance (e.g., dog1.name and dog2.name are different).

Instance Variables

Explanation:

- Accessing Variables:
  - Class variables can be accessed via the class name (Dog.species) or an object (dog1.species).
  - Instance variables are accessed via the object (dog1.name).
- Updating Variables:
  - Changing Dog.species affects all instances.
  - Changing dog1.name only affects dog1 and does not impact dog2.

Python Inheritance

- Inheritance allows a class (child class) to acquire properties and methods of another class (parent class).
- It supports hierarchical classification and promotes code reuse.

Types of Inheritance

1. **Single Inheritance:** A child class inherits from a single parent class.
2. **Multiple Inheritance:** A child class inherits from more than one parent class.
3. **Multilevel Inheritance:** A child class inherits from a parent class, which in turn inherits from another class.
4. **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
5. **Hybrid Inheritance:** A combination of two or more types of inheritance.

Single Inheritance

```
# Single Inheritance
class Dog:
    def __init__(self, name):
        self.name = name

    def display_name(self):
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Single Inheritance
    def sound(self):
        print("Labrador woofs")
```

Multilevel Inheritance

```
# Multilevel Inheritance
class GuideDog(Labrador):  # Multilevel Inheritance
    def guide(self):
        print(f"{self.name}Guides the way!")
```

Multiple Inheritance

```
# Multiple Inheritance
class Friendly:
    def greet(self):
        print("Friendly!")

class GoldenRetriever(Dog, Friendly):  # Multiple Inheritance
    def sound(self):
        print("Golden Retriever Barks")
```

Example Usage

```
# Example Usage
lab = Labrador("Buddy")
lab.display_name()
lab.sound()

guide_dog = GuideDog("Max")
guide_dog.display_name()
guide_dog.guide()
```

```
retriever = GoldenRetriever("Charlie")
retriever.display_name()
retriever.greet()
retriever.sound()
```

Example Usage

Explanation:

- Single Inheritance: Labrador inherits Dog’s attributes and methods.
- Multilevel Inheritance: GuideDog extends Labrador, inheriting both Dog and Labrador functionalities.
- Multiple Inheritance: GoldenRetriever inherits from both Dog and Friendly.

Python Polymorphism

- Polymorphism allows methods to have the same name but behave differently based on the object’s context.
- It can be achieved through method overriding or overloading.

Types of Polymorphism

Compile-Time Polymorphism:

- This type of polymorphism is determined during the compilation of the program.
- It allows methods or operators with the same name to behave differently based on their input parameters or usage.
- It is commonly referred to as method or operator overloading.

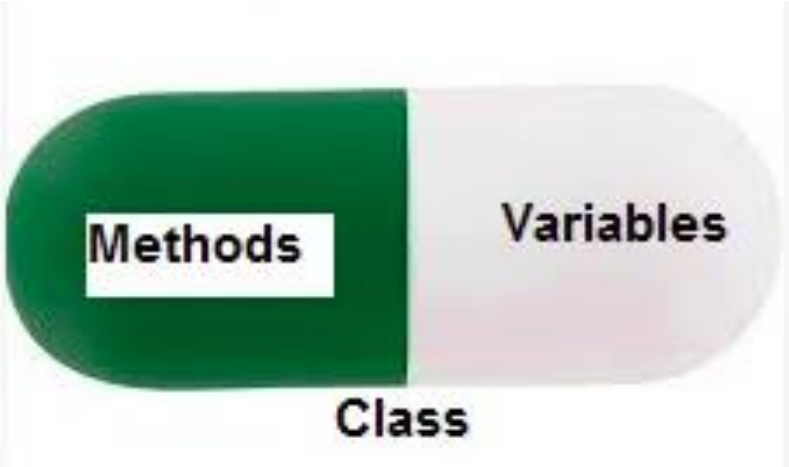
Types of Polymorphism

Run-Time Polymorphism:

- This type of polymorphism is determined during the execution of the program.
- It occurs when a subclass provides a specific implementation for a method already defined in its parent class, commonly known as method overriding.

Python Encapsulation

- Encapsulation is the bundling of data (attributes) and methods (functions) within a class, restricting access to some components to control interactions.
- A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc.



Python Encapsulation

Types of Encapsulation:

- Public Members: Accessible from anywhere.
- Protected Members: Accessible within the class and its subclasses.
- Private Members: Accessible only within the class.

Python Encapsulation

Code Example:

```
class Dog:
    def __init__(self, name, breed, age):
        self.name = name  # Public attribute
        self._breed = breed  # Protected attribute
        self.__age = age  # Private attribute

    # Public method
    def get_info(self):
        return f"Name: {self.name}, Breed: {self._breed}, Age: {self.__age}"
```

Python Encapsulation

```
# Getter and Setter for private attribute
def get_age(self):
        return self.__age

def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Invalid age!")
```

Python Encapsulation

 # Example Usage

dog = Dog("Buddy", "Labrador", 3)

 # Accessing public member

print(dog.name) # Accessible

 # Accessing protected member

print(dog._breed) # Accessible but discouraged outside the class

Python Encapsulation

 # Accessing private member using getter

print(dog.get_age())

 # Modifying private member using setter

dog.set_age(5)

print(dog.get_info())

Python Encapsulation

- **Explanation:**
- Public Members: Easily accessible, such as name.
- Protected Members: Used with a single _, such as _breed. Access is discouraged but allowed in subclasses.
- Private Members: Used with __, such as __age. Access requires getter and setter methods.

Data Abstraction

- Abstraction hides the internal implementation details while exposing only the necessary functionality.
- It helps focus on “what to do” rather than “how to do it.”

Types of Abstraction:

- Partial Abstraction: Abstract class contains both abstract and concrete methods.
- Full Abstraction: Abstract class contains only abstract methods (like interfaces).

Data Abstraction

```
from abc import ABC, abstractmethod

class Dog(ABC):  # Abstract Class
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def sound(self):  # Abstract Method
        pass
```

Data Abstraction

```
def display_name(self):  # Concrete Method
        print(f"Dog's Name: {self.name}")

class Labrador(Dog):  # Partial Abstraction
    def sound(self):
        print("Labrador Woof!")

class Beagle(Dog):  # Partial Abstraction
    def sound(self):
        print("Beagle Bark!")
```

Data Abstraction

```
# Example Usage
dogs = [Labrador("Buddy"), Beagle("Charlie")]
for dog in dogs:
    dog.display_name()  # Calls concrete method
    dog.sound()  # Calls implemented abstract method
```

Explanation:

- Partial Abstraction: The Dog class has both abstract (sound) and concrete (display_name) methods.
- Why Use It: Abstraction ensures consistency in derived classes by enforcing the implementation of abstract methods.

Chapter Assignment