Python Basics 🐍

1. Data types and their operations like CRUD and Insert New data: list, tuple, dictionary, set
2. slicing in python
3. While and for loop in Python and range
4. Functions and built-in functions, lambda function
5. class, objects, types of inheritance, polymorphism and its types, types of class
6. args, kwargs,

- set doesn't support direct update methods like list
- tuple don't support delete and update. It can be deleted using del as the entire tuple.
- self is used as an arrow function or pointer to the instance of the class in the Python methods. self parameter to the h() method. self refers to the instance of the Hello class that the method is being called on.

Python Data Structures:

Overview:

- Python provides several built-in data structures that are essential for organizing and storing data.
- Understanding these structures is crucial for efficient Python programming.

Key Data Structures:

Lists: Ordered, mutable sequences.
Tuples: Ordered, immutable sequences.
Dictionaries: Unordered collections of key-value pairs.
Sets: Unordered collections of unique elements.

Control Flow:

Definition:

- Control flow statements determine the order in which statements in a program are executed.
- They allow you to create programs that can make decisions and repeat actions.

Key Control Flow Statements:

- **Conditional Statements (if, elif, else):** Used to execute different blocks of code based on conditions.
- **Loops (for, while):** Used to repeat blocks of code.
- **Exception Handling (try, except):** Used to handle errors that occur during program execution.

 # example class

```
class He:
    def __init__(self):
        None

    def hello(self, username):  # Added 'self' as the first parameter
        return f"Hello!, {username} "

ob = He()
r = ob.hello("dilli hang rae")
print(r) #output: Hello!, dilli hang rae
```

 # Single Inheritance class parent —> class child: A class inherits from only one parent class.

```
class Animal:
def speak(self):
    return "animal speaks"

class Dog(Animal):

def s(self):
     return "dog braks"

dog = Dog()
dog.speak()
'animal speaks'
dog.s()
'dog braks'
```

 # Multiple Inheritance class parent1, class parent2..... —> class child: A class inherits from multiple parent classes.

```
class Swimmer:
    def swim(self):
        print("Swimming")

class Walker:
    def walk(self):
        print("Walking")

class Amphibian(Swimmer, Walker):  # Amphibian inherits from Swimmer and Walker
    pass

frog = Amphibian()
frog.swim()  # Output: Swimming
frog.walk()  # Output: Walking
```

```
>>> class Parent1:
...  def speak(self):
...    return "speaks"
...
>>> class Child(Parent1, Parent2):
...   pass
...
>>> obje = Child()
>>> obje.speak()
'speaks'
>>> obje.sings()
'sings'
```

 # Multi-Level inheritance: class grandparent -> class parent -> class child:
A class inherits from another class, which in turn inherits from another class, forming a chain.

```
>>> class grandparent():
...  def gp(self):
...   return "traits of gp"
...
>>> class parent(grandparent):
...   def p(self):
...    return "traits of parents"
...
>>> class child
  File "<stdin>", line 1
    class child
              ^
SyntaxError: expected ':'
>>> class child(parent):
...   pass
...
>>> obj1 = child()
>>> obj1.gp()
'traits of gp'
>>> obj1.p()
'traits of parents'
```

```
class Grandparent:
    def feature1(self):
        print("Grandparent feature")

class Parent(Grandparent):  # Parent inherits from Grandparent
    def feature2(self):
        print("Parent feature")
```

```
class Child(Parent):  # Child inherits from Parent
    def feature3(self):
        print("Child feature")

child = Child()
child.feature1()  # Output: Grandparent feature
child.feature2()  # Output: Parent feature
child.feature3()  # Output: Child feature
```

Hierarchical Inheritance: Multiple classes inherit from a single parent class.

```
class Vehicle:
    def start(self):
        print("Vehicle starting")

class Car(Vehicle):  # Car inherits from Vehicle
    def drive(self):
        print("Driving car")

class Bike(Vehicle):  # Bike inherits from Vehicle
    def ride(self):
        print("Riding bike")

car = Car()
bike = Bike()

car.start()  # Output: Vehicle starting
car.drive()  # Output: Driving car

bike.start()  # Output: Vehicle starting
bike.ride()  # Output: Riding bike
```

*args (Arbitrary Positional Arguments):

In Python, *args and **kwargs are powerful tools that allow you to create functions that can accept a variable number of arguments. Here's a breakdown:

***args (Arbitrary Positional Arguments):**

- *args allows you to pass a variable number of non-keyword arguments to a function.
- Inside the function, args becomes a tuple containing all the positional arguments passed.
- It's useful when you don't know in advance how many positional arguments a function will receive.

**kwargs (Arbitrary Keyword Arguments):

- **kwargs allows you to pass a variable number of keyword arguments to a function.
- Inside the function, kwargs [1] becomes a dictionary containing all the keyword arguments passed

Combining *args and **kwargs:

- You can use both *args and **kwargs in the same function definition.
- If you do, *args must come before **kwargs

- args and kwargs are just conventions. You could use other names, but it's strongly recommended to stick with these.
- The asterisks (* and **) are what's important. They tell Python to pack the arguments into a tuple or dictionary.
- These are very useful for creating flexible functions.

Types of Polymorphism in Python

1. **Duck Typing** (Dynamic Typing)
2. **Method Overriding** (In Inheritance)
3. **Operator Overloading**

1. Duck Typing

If it "quacks like a duck and walks like a duck," it's a duck — Python doesn't care about object types, just behavior.

Python

```
class Dog:
    def speak(self):
        return "Woof!"

class Cat:
    def speak(self):
        return "Meow!"

def animal_sound(animal):
    print(animal.speak())

animal_sound(Dog())  # Output: Woof!
animal_sound(Cat())  # Output: Meow!
```

2. Method Overriding (Inheritance-based Polymorphism)

Python

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

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

b = Bird()
s = Sparrow()

b.fly()   # Output: Bird is flying
s.fly()   # Output: Sparrow flies fast
```

3. Operator Overloading

Python allows redefining the meaning of operators for user-defined classes.

python

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

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

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y)  # Output: 4 6
```

🔍 Summary

Type	Description
Duck Typing	Behavior-based, not type-based
Method Overriding	Subclass redefines parent method
Operator Overloading	Custom meaning to built-in operators

Virtual environments and pip

Virtual environments and **pip** are essential tools in Python development, especially when working on multiple projects or managing dependencies. Here's a breakdown of each:

Virtual Environments:

- **Purpose:**
  - A virtual environment is an isolated Python environment. It allows you to create separate installations of Python and its packages for different projects.

    - This prevents conflicts between project dependencies, ensuring that each project has its own set of required packages.

    - This helps with reproducibility so that your code can be run in other environments with the same dependencies.

- **How it Works:**
    - When you activate a virtual environment, your system's Python interpreter and package installations are temporarily overridden by the environment.

    - Any packages you install while the environment is active are installed within that environment, not globally.
- **Creating and Activating:**
    - You can create virtual environments using the built-in venv module:
        - python -m venv myenv (where myenv is the name of your environment)
    - To activate the environment:
        - On Windows: myenv\Scripts\activate
        - On macOS/Linux: source myenv/bin/activate
    - To deactivate the environment use the command deactivate. _

- **Benefits:**
    - Dependency isolation.
    - Avoidance of conflicts.
    - Project reproducibility.
    - Clean global Python environment.

PIP (Package Installer for Python):

- **Purpose:**
    - pip is the standard package manager for Python.
    - It's used to install, upgrade, and manage Python packages from the Python Package Index (PyPI) and other indexes.

- **Common Commands:**
    - pip install package_name: Installs a package.
    - pip uninstall package_name: Uninstalls a package.
    - pip list: Lists installed packages.
    - pip freeze: Generates a list of installed packages with their versions (useful for creating requirements.txt).
    - pip install -r requirements.txt: Installs packages from a requirements.txt file.
    - pip show package_name: Shows information about an installed package.

    - pip upgrade pip: upgrades pip.
- **requirements.txt:**
    - A requirements.txt file is a text file that lists all the packages and their versions required for a project.
    - It's used to ensure that other developers or systems can easily install the same dependencies.
    - It is created with the command pip freeze > requirements.txt
- **Benefits:**
    - Easy package installation and management.
    - Access to a vast library of Python packages.
    - Easy dependency management with requirements files.

How They Work Together:

- Virtual environments and pip work together seamlessly.
- You typically create a virtual environment for each project and then use pip to install the project's dependencies within that environment.
- This ensures that each project has its own isolated set of packages, preventing conflicts and maintaining reproducibility.

1. NumPy (Numerical Python)

NumPy is the core library for numerical computation in Python. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical [1] functions to operate on these arrays. [2]

- **Arrays:**
    - NumPy's main object is the ndarray (n-dimensional array).
    - Arrays can be created from Python lists or tuples.

Python

```
import numpy as np

# Creating a 1D array
arr1d = np.array([1, 2, 3, 4, 5])
print(arr1d)

# Creating a 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d)
```

- **Array Operations:**
    - NumPy allows element-wise operations, broadcasting, and various mathematical functions.

Python

```
arr = np.array([1, 2, 3])
print(arr + 2)  # Element-wise addition
print(arr * 3)  # Element-wise multiplication
print(np.sin(arr)) # sine function
```

- **Array Indexing and Slicing:**
    - Similar to Python lists, NumPy arrays support indexing and slicing.

Python

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])     # Accessing the first element
print(arr[1:4])   # Slicing from index 1 to 3
```

- **Creating Arrays:**
    - np.zeros(), np.ones(), np.arange(), and np.linspace() are useful for creating arrays.

Python

```
zeros_arr = np.zeros((2, 3)) # array of zeros
ones_arr = np.ones((3, 2)) # array of ones
arange_arr = np.arange(0, 10, 2) # sequence from 0 to 10 with step 2
linspace_arr = np.linspace(0, 1, 5) # 5 evenly spaced numbers from 0 to 1

print(zeros_arr)
print(ones_arr)
print(arange_arr)
print(linspace_arr)
```

2. Pandas (Panel Data)

Pandas is a library for **data manipulation and analysis.** It provides data structures like **Series (1D labeled array) and DataFrame (2D labeled table).**

- **Series:**
    - A Series is like a 1D array with labeled indices.

Python

```
import pandas as pd

s = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])
print(s)
```

- **DataFrame:**
    - A DataFrame is like a table with rows and columns.

Python

```
data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],
        'Age': [25, 30, 22, 35],
        'City': ['New York', 'London', 'Tokyo', 'Paris']}
df = pd.DataFrame(data)
print(df)
```

- **Reading and Writing Data:**
  - Pandas can read and write data from various file formats (CSV, Excel, etc.).

```
# df.to_csv('data.csv', index=False) # writes the dataframe to a csv file.
# df_read = pd.read_csv('data.csv') # reads a csv file into a dataframe.
```

- **Data Selection and Manipulation:**
  - Pandas allows easy selection, filtering, and manipulation of data.

```
print(df['Name'])      # Selecting a column
print(df.loc[0])       # Selecting a row by label
print(df.iloc[0]) # selecting a row by integer location.
print(df[df['Age'] > 25]) # Filtering rows based on a condition
```

- **Basic Operations:**
  - Pandas has functions for statistical operations, such as mean, median, standard deviation, and many more.

```
print(df['Age'].mean())
print(df['Age'].max())
```

3. Matplotlib (Matlab-style plotting)

Matplotlib is a plotting library for creating static, interactive, and animated visualizations in Python.

- **Basic Plotting:**
  - matplotlib.pyplot is commonly used for creating plots.

```
 import matplotlib.pyplot as plt
x = np.array([1, 2, 3, 4, 5])
y = x**2

plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Simple Plot')
plt.show()
```

- **Scatter Plots:**

```
plt.scatter(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')
plt.show()
```

- **Bar Plots:**

```
categories = ['A', 'B', 'C', 'D']
values = [20, 35, 30, 25]
```

```
plt.bar(categories, values)
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Plot')
plt.show()
```

- **Histograms:**

```
data = np.random.randn(1000) # generate random data
plt.hist(data, bins=30)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

- **Customization:**
  - Matplotlib allows extensive customization of plots (colors, line styles, labels, etc.).
  - These are the basics! You can explore the libraries further to learn about more advanced features and functionalities.