- Datatypes, Variables, Input & Output
- Operators, Conditional, Iterative statement
- **Collection Data types:** Strings, List, Set, Tuple, Dictionary
- **Functions:** Lambda Functions, Decorators, Generators

OOPS:

- Class, Object, Attributes, methods
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation
- File Handling, Exceptions, Logging

Datatypes :

Primitive datatypes : integer, string, float, boolean

Collection datatypes : list,tuple,dictionary,set

Variables :

- Variable is a name used to store data that can be changed during program execution.
- It doesn't require data_type declaration and can hold any data_type.

Global Variable :

Global keyword attached to variable in python code enables to access anywhere inside the code and update.

Ex:

```
x = 10                    # Global variable
def update_global ():
   global x               # Declares x as global
   x = 20                 # Updates global variable
update_global()
print(x)                  # Output will be 20
```

Naming Rules:

1. Variable name can only contain alpha-numeric & underscore characters but should not start with number.
2. Variable names are case-sensitive.
3. For MultiWord Variable names we can use Camel case, Pascal case, Snake case most prefered style is snake case
    - camel case : Each word, except the first, starts with a capital letter.

- Pascal case : Each word starts with a capital letter
- Snake Case : Each word is separated by an underscore character

4. Variable name can't be any of the Python keywords.

Assigning values :

- Multiple Values to Multiple variables : a,b,c= 1,"hero","Ram charan"
- Single value to Multiple variables : a=b=c=1

Input :

user_input = input("Enter something: ")
ex: age=int(input("enter your age:"))
name=str(input("enter your name:"))
height=float(input("enter your height:"))

Output :

- Using + for concate : print("Name"+name+"Age is:"+str(age))
- Using , to print : print("Name:", name, ", Age:", age)
- Using f-Strings method : print(f"Name:{name} Age:{age}")
- Using .format method : print("Name:{},Age:{}".format(name,age))

Operators:

Arithmetic Operators :

- + (Addition) - (Subtraction) * (Multiplication) / (float Division)
- // (decimal Division) % (Modulo) ** (Exponential)

Bitwise Operators :

- & (AND) | (OR) ~ (NOT)
  ^ (XOR)
- << (Left Shift) >> (Right Shift)

Comparison Operators :

- == (Equal) != (Not Equal)
- > (Greater Than) < (Less Than) <= (Less or Equal) >= (Greater or Equal)

Logical Operators :

- and (Logical AND) or (Logical OR) not (Logical NOT)

Assignment Operators :

- variable (Arithmetic/Bitwise Operator)= expression

Membership Operators :

- in (Present in) not in (Not Present in)

Conditional Statements:

 # Conditional Statements

```
if condition1:
   print("Statement")
elif condition2:
   print("Statement")
elif condition3:
   print("Statement")
else:
   print("Statement")
```

 # Single-line version

```
if condition1: print("Statement")
elif condition2: print("Statement")
elif condition3: print("Statement")
else: print("Statement")
```

 # Short Hand If ... Else

```
result = "Positive" if num > 0 else "Non-Positive"
```

Iterative Statements :

1.Iterating through collection datatypes :

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

2.Iterating through String :

```
for x in "banana":
   print(x)
```

3.range function : range(start_index,end_index-1)

```
L = [10, 20, 30, 40, 50]
for i in range(len(L)):
   print(L[i])
```

4.else in For loop :

Print all numbers from 0 to 5, and print message when loop ends:

```
for x in range(6):
   print(x)
else:
```

print(“Finally finished!”)

5.continue, break, pass :

Break : Used to exit a loop prematurely when a certain condition is met.

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

Output: 0 1 2

Continue **:** Skips the current iterate and moves to the next iterate of the loop

```
for i in range(5):
  if i == 3:
    continue
  print(i)
```

Output: 0 1 2 4

Pass **:** A placeholder statement that does nothing.

```
for i in range(5):
    if i == 3:
        pass  # No action taken
    print(i)
```

Output: 0 1 2 3 4

Strings :

Strings are immutable, indexed & a powerful data type for text handling.

Strings in python are surround by either single('')/double("")/triple quotation marks.

You can assign a multiline string to a variable using three quotes(''' ''')/(""" """).

1. Access string:

- Square brackets can be used to access elements of the string :

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

- Looping through string : Since strings are arrays, we can loop through the characters in a string, with a for loop.

```
for x in "banana":
    print(x)
```

- We can perform the String Slicing.
- in, not in membership operators can be used to check patterns in a string either present or not.

2. Modify Strings:

- lower case : string.lower()
- Upper case : string.upper()
- Remove Whitespace : string.strip()
- Replace String : string.replace(old,updated,occurences)
- Split String : string.split(seperator)

3. String methods:

isal : Checks if the string consists of letters or not.

isnum : Checks if the string consists of numbers or not.

isalnum : Checks if the string consists of letters and numbers or not.

islower : Checks whether the string is in lower case or not.

isupper : Checks whether the string is in upper case or not.

Lists:[ ]

List is a collection datatype in python where it allows

- **mutable, ordered, allowed duplicates.**

1. Adding elements, Lists
2. Removing elements
3. Changing elements
4. Accesing, Looping, List comprehension
5. Methods in list

1. Adding elements,list:

append method : list.append(element)

insert method : list.insert(index,element)

extend method : list1.extend(list2) (or) list1+list2

2. Removing elements:

remove method : list.remove(element)

pop method : list.pop(index)

clear method : list.clear()

3. Changing elements:

list[index]="updated value"

4. Accessing elements, Looping, List comprehension:

indices : [start:stop+1:step]

[0:length-1] or [-length:-1]

[start_index:] = prints from start_index to last index.

[::-1] = prints the elements in the reverse order with stepsize 1

looping : looping through list

```
for i in list:
    print(i)
```

looping through the index numbers

```
for i in range(len(list)):
    print(list[i])
```

list comprehension:

newlist = [expression for item in list if condition == True]

5.List methods :

Sort() = list.sort()

Reverse() = list.reverse()

Tuple: ( )

Tuple is a collection datatype in python where it allows

- **ordered, Immutable**(can't change, add, or remove items once tuple is created)**, allowed duplicates.**

1. Creating tuple : We create list & then we typecast to tuple

2. Accessing the tuple

Set:{}

Set is a collection datatype in python where it allows

- **unordered, Immutable, duplicates not allowed.**

1. Adding elements, list/tuple/dictionary:

add method : set.add(element)

update method : set.update(list/tuple/dictionary)

2. Removing elements:

Remove/discard method : set.remove(element) /set.discard(element)

pop method : set.pop()

clear method : set.clear()

3. Accessing elements:

looping : looping through set

```
for i in list:
    print(i)
```

4. Set methods :

Union() : Combines all unique elements from two sets.

Set1.union(set2,set3,…..) (or) set3= set1 | set2

Intersection() : Finds common elements between two sets.

Set1.intersection(s2,s3,…..) (or) set3= set1&set2

Difference() : Finds elements that are in 1st set but not in 2nd set.

set1.difference(set2) (or) set3=set2-set1

Symmetric_difference() : Find element that are in either of sets but not in both

set1.symmetric_difference(s2) (or) set3=set1 ^ set2

dictionary:{ }

Dictionary is a collection datatype in python where it allows

- ordered, mutable, duplicates not allowed .

1. **Adding elements:**

   dictionary[key]=value

   dictionary.update({key1:value1},{key2:value2})

2. **Remove elements:**

   dictionary.pop(key)

clear()

3. **update elements:**

   dictionary[key]=updated_value

   dictionary.update({key1:updated_value1},{key2:updated_value2})

4. **Access elements:**

   a. To print all key names in the dictionary:

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

   ```
   for key in dictionary.keys():
           print(key)
   ```

   b. To print all values in the dictionary :

   ```
   for key in dictionary:
           print(dictionary[key])
   ```

   ```
   for value in dictionary.values():
           print(value)
   ```

   c. To print key,value pairs in the dictionary:

   ```
   for key,value in thisdict.items():
       print("Key:",key,"Value:",value)
   ```

Functions :

Syntax :

```
def function_name(arguments):
    return values
```

- When multiple values are separated by commas in the return statement, Python automatically groups them into a tuple.

```
def get_person_details():
name = "Alice"
age = 25
city = "New York"
return name, age, city  # These values will be returned as a tuple


# Unpack the returned tuple into separate variables
```

```
    name, age, city = get_person_details()
```

1. **Required Arguments**

   These are the arguments that must be passed in the correct positional order. If you do not provide the required arguments, Python will raise an error.

```
def add(a, b):
    return a + b
print(add(3, 4))  # Output: 7
```

2. **Keyword Arguments**

   You can specify the arguments by name when calling a function, allowing you to pass them in any order.

```
def introduce(name, age):
    print(f"My name is {name}, and I am {age} years old.")

# Using keyword arguments
introduce(age=25, name="Alice")  # Output: My name is Alice, and I am 25 years old.
```

3. **Default Arguments**

   You can provide default values for arguments. If the caller doesn't provide values for these arguments, the defaults will be used.

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

# Call with only the required argument
greet("Alice")  # Output: Hello, Alice!

# Call with both arguments
greet("Bob", "Good morning")  # Output: Good morning, Bob!
```

4. **Variable-Length Arguments**

   These allow a function to accept an arbitrary number of arguments.

a. *args for Variable-Length Positional Arguments:

You can pass any number of positional arguments to a function using *args.

Example:

```
def add_all(*args):
    return sum(args)
print(add_all(1, 2, 3, 4))  # Output: 10
```

b. **kwargs for Variable-Length Keyword Arguments:

You can pass any number of keyword arguments using **kwargs.

Example:

```
def print_info(**kwargs):
for key, value in kwargs.items():
print(f"{key}: {value}")
# Call with multiple keyword arguments
print_info(name="Charlie", age=30, city="New York")

# Output:
    name: Charlie
    age: 30
    city: New York
```

Lambda function:

A lambda function can take any number of arguments, but can only have one expression.

Syntax : lambda arguments : expression

example :

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

Generators:

- Multiple Values on Demand: Generators produce values one at a time using the yield keyword, returning the next value only when requested.

- Lazy Evaluation with State Preservation: Generators evaluate data only when needed and remember where they left off, resuming execution from the last yield when called again.

- Memory Efficiency: They don't store all values in memory at once, making them ideal for processing large datasets or files.

- Usage Scenario: Perfect for handling large files, streams of data, or infinite sequences without overloading memory.

```
def square_numbers(n):
    for i in range(1, n+1):
        yield i * i  # Yields square of the current number
squares = square_numbers(5)
for square in squares:
    print(square)
```

Decorators:

A **decorator** is a function that:

1. Accepts the original function as a parameter.
2. Defines an inner wrapper that adds additional logic (e.g; modifyingarguments).
3. Calls the original function within the wrapper.
4. Returns the wrapper, allowing the original function to execute with the enhanced behavior when invoked.

Example :

```
def original_function(a,b):
    print(a/b)

def decorator_function(func):
    def wrapper(x,y):
        if x < y:   x, y = y, x
        return func(x,y)
    return wrapper

decorated_function = decorator_function(original_function)
```

decorated_function(2, 4)

Exceptions :

Exceptions are events that occur during execution, disrupting the normal flow of a program.

Types of Exceptions:

1. Built-in Exceptions (ex: SyntaxError, ValueError, TypeError)
2. User-defined Exceptions (custom classes)

Handling Exceptions:

1. Try: Enclose code that may raise an exception.
2. Except: Handle the exception (specific or general).
3. Else: Execute if no exception occurs.
4. Finally: Execute regardless of exception.

Basic Syntax:

```
try:
    # code that may raise exception
except ExceptionType:
    # handle exception
except Exception as e:
        print(e)
else:
    # code if no exception
finally:
    # code always executed
```

Ex:

```
try:
    # code block where exception can occur
    a = int(input("Enter the number 1: "))
    b = int(input("Enter the number 2: "))
    c = a / b
```

```
except NameError:
    print("The user has not defined the variable.")

except ZeroDivisionError:
    print("Please provide a number greater than 0.")

except TypeError:
    print("Try to make the datatype similar.")

except Exception as ex:
    print("An error occurred:", ex)

else:
    # Executes if no exception occurs
    print("Result:", c)

finally:
    # Always executes
    print("The execution is done.")
```

Raising Exceptions:

1. Raise a built-in exception: raise ValueError("Invalid input")
2. Raise a custom exception: raise MyCustomError("Something went wrong")

Custom Exceptions:

Create a class inheriting from Exception: