# Project – 9

# Python Basics

Submitted By:

Nishigandha Patil

# INDEX

| Sr. No | Topics | Page. No |
|---|---|---|
| 1 | Introduction to Python | 3 |
| 2 | Variables | 4 |
| 3 | Data Types | 5 |
| 4 | Operators | 9 |
| 5 | Conditional Statements | 11 |
| 6 | Loops and Functions | 12 |
| 7 | Object Oriented Programming | 15 |
| 8 | Modules | 18 |

# INTRODUCTION TO PYTHON

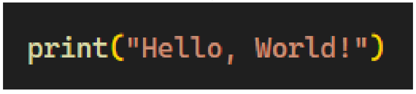
Python is a high-level programming language. It was created by Guido van Rossum, and released in 1991. It has a simple syntax similar to the English language. It is an interpreted and versatile language.

- **What is Python used for:**
  - **Web Development:** Python is used with frameworks like Django and Flask to build dynamic and scalable web applications.
  - **Data Science:** Python is a popular choice for data analysis, visualization, and machine learning tasks. Libraries like NumPy, Pandas, Matplotlib, and scikit-learn support these activities.
  - **Machine Learning and AI:** Python is extensively used in developing machine learning models and artificial intelligence applications. TensorFlow and PyTorch are prominent libraries in this field.
  - **Automation**: Python is commonly employed for scripting and automation tasks, simplifying repetitive operations and workflows.
  - **Game Development:** Python is used for developing simple games and prototypes, with libraries like Pygame providing game development functionalities.
  - **Desktop GUI Applications:** Python, with libraries like Tkinter and PyQt, is used for creating desktop graphical user interface (GUI) applications.
  - **Backend Development:** Python is often used for server-side development in conjunction with frameworks like Flask and Django, powering the backend of web applications.

- **Installing Python:**
  Go to https://www.python.org/downloads/ → Download latest Version → To verify the installations, open cmd and type: python --version.

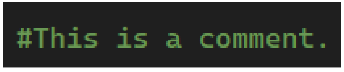
- **Python in VS Code:**
  Open folder → Create a new python file (file > new file)→ Write python code → Run.
  **VS Code Extension for Python:** Python, Python Debugger by Microsoft.

- **Python QuickStart:** Hello, World! Program.
  Use **print ()** – Python’s built-in function to display output to the console.



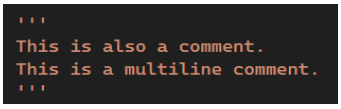
```
print("Hello, World!")
```

- **Comments:**
  - **Single-line comment:** use #



```
#This is a comment.
```

  - **Multiline Comment:** use ‘‘‘  ’’’

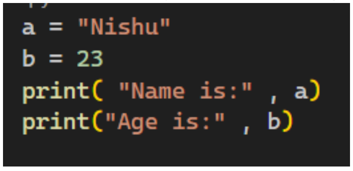


```
'''
This is also a comment.
This is a multiline comment.
'''
```

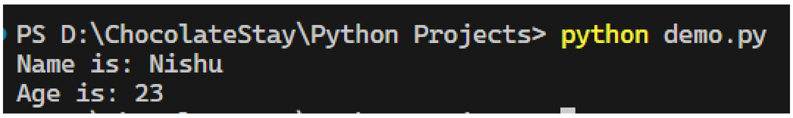
# VARIABLES

Variables are containers for storing data values.

- **Creating Variable:** Python has no command for declaring a variable. A variable is created the moment you first assign a value to it.

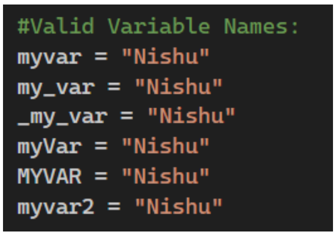


```
a = "Nishu"
b = 23
print( "Name is:" , a)
print("Age is:" , b)
```

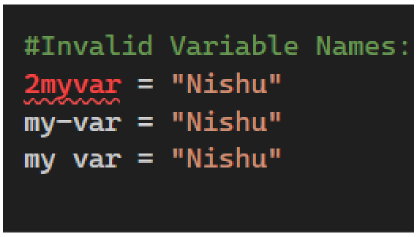


```
PS D:\ChocolateStay\Python Projects> python demo.py
Name is: Nishu
Age is: 23
```

- **Rules for declaring Variables:**
  - A variable name must start with a letter or the underscore character.
  - A variable name cannot start with a number.
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _ ).
  - Variable names are case-sensitive (age, Age and AGE are three different variables).
  - A variable name cannot be any of the Python keywords.

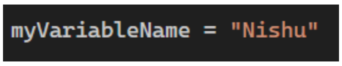


```
#Valid Variable Names:
myvar = "Nishu"
my_var = "Nishu"
_my_var = "Nishu"
myVar = "Nishu"
MYVAR = "Nishu"
myvar2 = "Nishu"
```



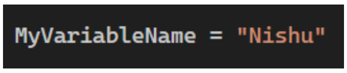
```
#Invalid Variable Names:
2myvar = "Nishu"
my-var = "Nishu"
my var = "Nishu"
```

- **Multi Words Variable Names:**
  1. **Camel Case:** Each word, except the first, starts with a capital letter.



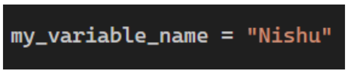
```
myVariableName = "Nishu"
```

  2. **Pascal Case:** Each word starts with a capital letter.



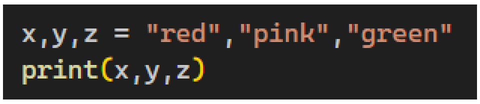
```
MyVariableName = "Nishu"
```

  3. **Snake Case:** Each word is separated by an underscore character.



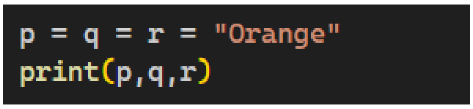
```
my_variable_name = "Nishu"
```

- **Many Values to Multiple Variables:** Python allows you to assign values to multiple variables in one line.



```
x,y,z = "red","pink","green"
print(x,y,z)
```

- **One Value to Multiple Variables:** You can assign the *same* value to multiple variables in one line.

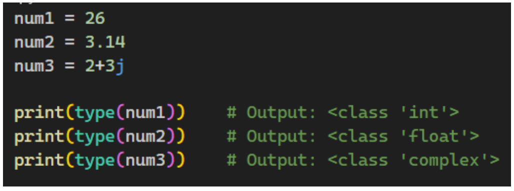


```
p = q = r = "Orange"
print(p,q,r)
```

# DATA TYPES

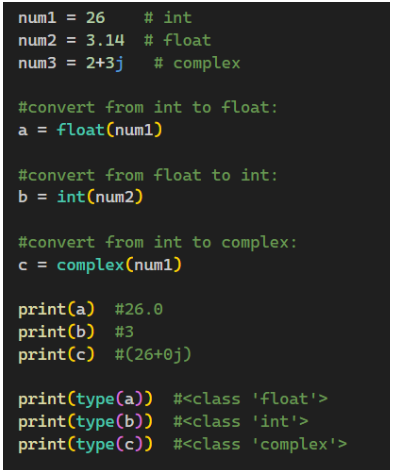
Python supports several built-in data types that allow you to represent and manipulate different kinds of data.

1. **Numbers:**
   - **int:** Represents whole numbers (e.g., 26, -10).
   - **float:** Represents decimal numbers (e.g., 3.14, -0.5).
   - **complex:** Complex type represents complex numbers (e.g., 2 + 3j).



```
num1 = 26
num2 = 3.14
num3 = 2+3j

print(type(num1))    # Output: <class 'int'>
print(type(num2))    # Output: <class 'float'>
print(type(num3))    # Output: <class 'complex'>
```

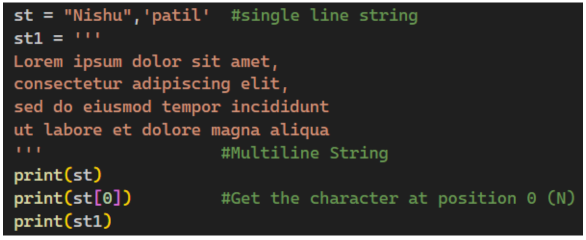
   - **Type Conversion:** You can convert from one type to another with the int(), float(), and complex() methods.



```
num1 = 26    # int
num2 = 3.14  # float
num3 = 2+3j   # complex

#convert from int to float:
a = float(num1)

#convert from float to int:
b = int(num2)

#convert from int to complex:
c = complex(num1)

print(a)  #26.0
print(b)  #3
print(c)  #(26+0j)

print(type(a))  #<class 'float'>
print(type(b))  #<class 'int'>
print(type(c))  #<class 'complex'>
```

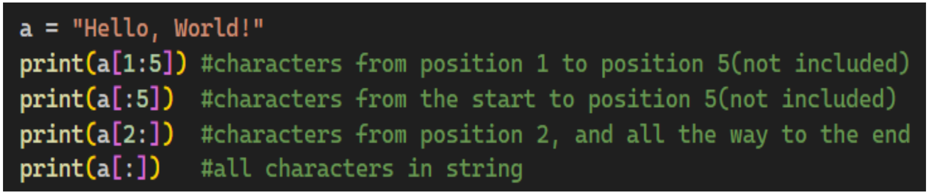
**type ():** used to get the type of variable.

2. **String:** Represents text (e.g., "hello", 'Python').



```
st = "Nishu",'patil'  #single line string
st1 = '''
Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua
'''                    #Multiline String
print(st)
print(st[0])           #Get the character at position 0 (N)
print(st1)
```

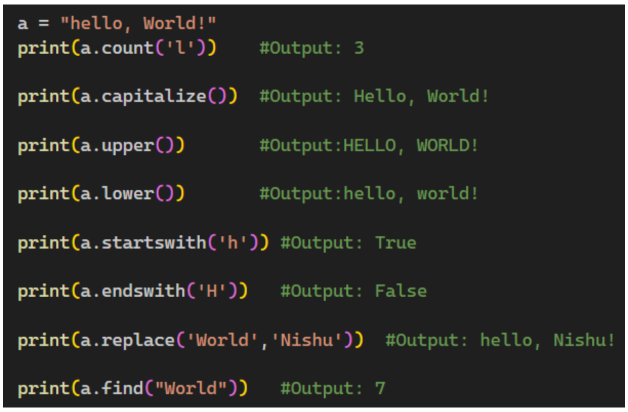
   - **String slicing:** You can return a range of characters by using the slice syntax. Specify the **start index** and the **end index**, separated by a **colon**, to return a part of the string.



```
a = "Hello, World!"
print(a[1:5]) #characters from position 1 to position 5(not included)
print(a[:5])  #characters from the start to position 5(not included)
print(a[2:])  #characters from position 2, and all the way to the end
print(a[:])   #all characters in string
```

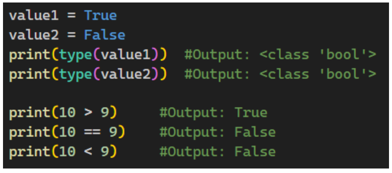
- **String methods:**

| Methods | Description |
|---|---|
| **.count()** | Count the occurrences of letters. |
| **.capitalize()** | Converts only the first character to uppercase. |
| **.upper()** | Converts all characters to uppercase. |
| **.lower()** | Converts all characters to lowercase. |
| **.startswith() & .endswith()** | Checks if the string starts or ends with a specified prefix or suffix. |
| **.replace()** | Replaces occurrences of a substring with another substring |
| **.find()** | Returns index of specified word |



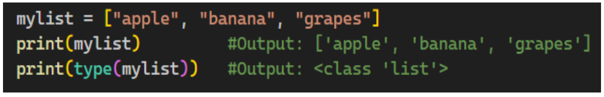
```
a = "hello, World!"
print(a.count('l'))    #Output: 3

print(a.capitalize())  #Output: Hello, World!

print(a.upper())       #Output:HELLO, WORLD!

print(a.lower())       #Output:hello, world!

print(a.startswith('h')) #Output: True

print(a.endswith('H'))   #Output: False

print(a.replace('World','Nishu'))  #Output: hello, Nishu!

print(a.find("World"))   #Output: 7
```

3. **Boolean:** Represent one of two values: True or False.



```
value1 = True
value2 = False
print(type(value1))  #Output: <class 'bool'>
print(type(value2))  #Output: <class 'bool'>

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

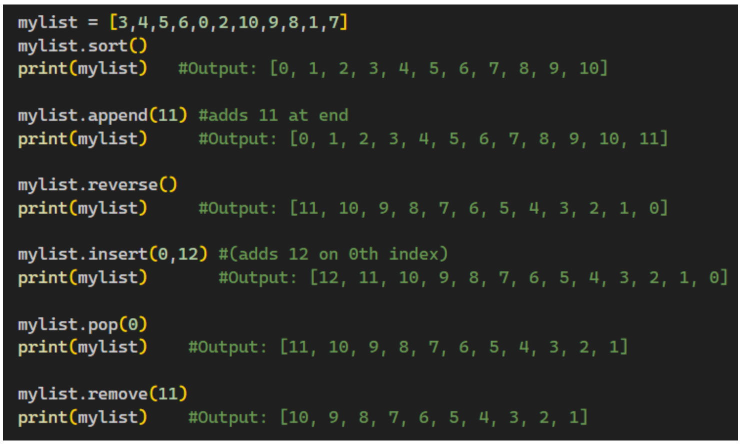
4. **List:** It is an ordered and mutable (changeable) collection of items. Created using square brackets [] Example: [1, 2, 3].



```
mylist = ["apple", "banana", "grapes"]
print(mylist)          #Output: ['apple', 'banana', 'grapes']
print(type(mylist))   #Output: <class 'list'>
```

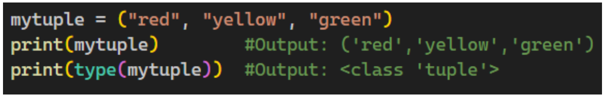
- **List Methods:**

| Methods | Description |
|---|---|
| **sort()** | Sort the list items in ascending order. |
| **append()** | Adds an item to the end of the list. |
| **reverse()** | Reverse the list items. |
| **insert(i,x)** | Inserts an item(x) at a specific index (i). |
| **pop()** | Removes and returns the item at index i. If no index is specified, it removes and returns the last item. |
| **remove()** | Removes specified item from the list |



```
mylist = [3,4,5,6,0,2,10,9,8,1,7]
mylist.sort()
print(mylist)   #Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

mylist.append(11) #adds 11 at end
print(mylist)     #Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

mylist.reverse()
print(mylist)     #Output: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

mylist.insert(0,12) #(adds 12 on 0th index)
print(mylist)       #Output: [12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

mylist.pop(0)
print(mylist)    #Output: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

mylist.remove(11)
print(mylist)    #Output: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

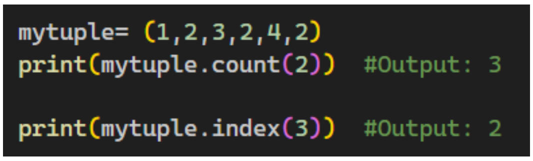
5. **Tuple:** It is an ordered and immutable (unchangeable) collection of items. Created using round brackets (). Example: (1,2,3)



```
mytuple = ("red", "yellow", "green")
print(mytuple)        #Output: ('red','yellow','green')
print(type(mytuple))  #Output: <class 'tuple'>
```

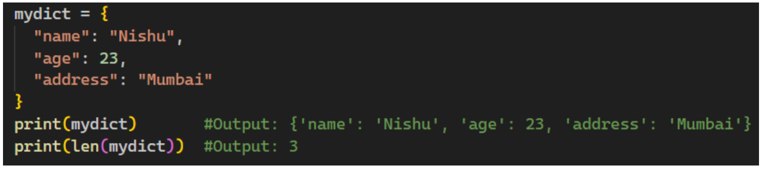
- **Tuple Methods:**

| **Methods** | **Description** |
|---|---|
| **.count()** | Count the total no of occurrence of specified item |
| **.index()** | Returns the index of specified item |



```
mytuple= (1,2,3,2,4,2)
print(mytuple.count(2))  #Output: 3

print(mytuple.index(3))  #Output: 2
```

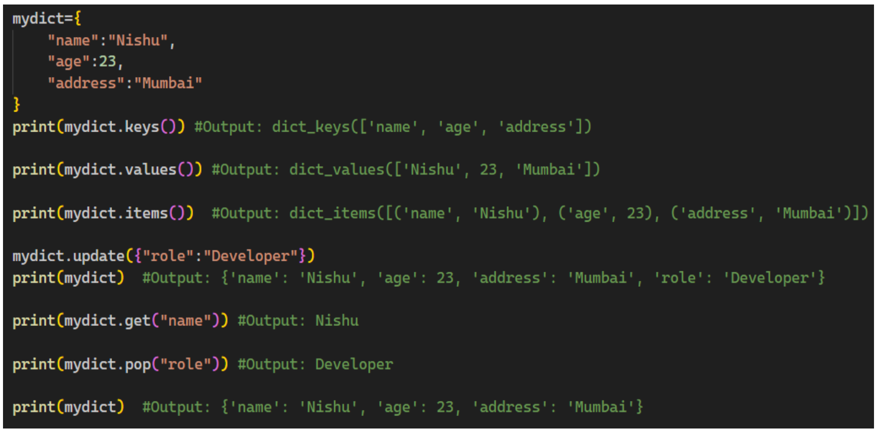
6. **Dictionary:** It is an unordered collection of key-value pairs. It is changeable and do not allow duplicates. Created using curly braces {}. Example: {'name': 'Nishu', 'age': 23}



```
mydict = {
  "name": "Nishu",
  "age": 23,
  "address": "Mumbai"
}
print(mydict)       #Output: {'name': 'Nishu', 'age': 23, 'address': 'Mumbai'}
print(len(mydict))  #Output: 3
```

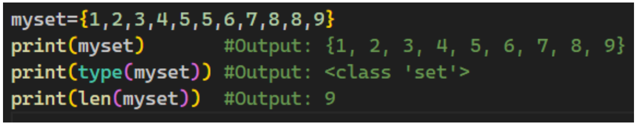
- **Dictionary Methods:**

| **Methods** | **Description** |
|---|---|
| **.keys()** | Returns keys |
| **.values()** | Returns values |
| **.items()** | Returns key:value pair |
| **.update()** | Update dict with supplied key:value pair |
| **.get()** | Returns key. If specified key is not present then returns **None**. |
| **.pop()** | Remove item using key. |



```
mydict={
    "name":"Nishu",
    "age":23,
    "address":"Mumbai"
}
print(mydict.keys()) #Output: dict_keys(['name', 'age', 'address'])

print(mydict.values()) #Output: dict_values(['Nishu', 23, 'Mumbai'])

print(mydict.items())  #Output: dict_items([('name', 'Nishu'), ('age', 23), ('address', 'Mumbai')])

mydict.update({"role":"Developer"})
print(mydict)  #Output: {'name': 'Nishu', 'age': 23, 'address': 'Mumbai', 'role': 'Developer'}

print(mydict.get("name")) #Output: Nishu

print(mydict.pop("role")) #Output: Developer

print(mydict)  #Output: {'name': 'Nishu', 'age': 23, 'address': 'Mumbai'}
```

7. **Sets:** It is an unordered collection of unique items. Example: {1, 2, 3}).

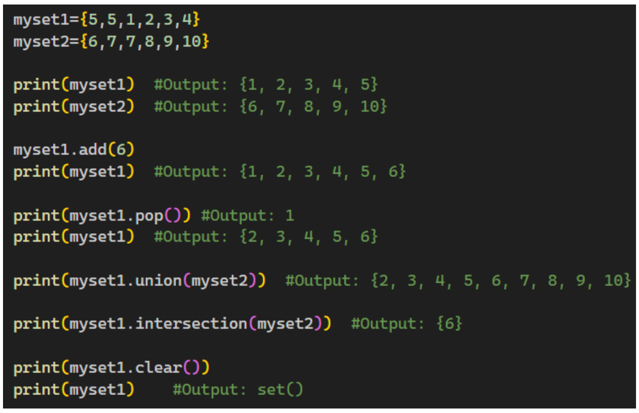


```
myset={1,2,3,4,5,5,6,7,8,8,9}
print(myset)       #Output: {1, 2, 3, 4, 5, 6, 7, 8, 9}
print(type(myset)) #Output: <class 'set'>
print(len(myset))  #Output: 9
```

**len():** Returns the length.

- **Set Methods:**

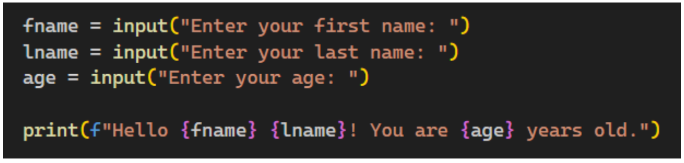
| **Methods** | **Description** |
|---|---|
| **.add()** | Adds an item to the set |
| **.pop()** | Removes and returns an arbitrary element from the set. |
| **.union()** | Returns a new set containing all unique elements from both sets |
| **.intersection** | Returns a new set containing common elements between two sets. |
| **.clear()** | Removes all elements from the set |



```
myset1={5,5,1,2,3,4}
myset2={6,7,7,8,9,10}

print(myset1)  #Output: {1, 2, 3, 4, 5}
print(myset2)  #Output: {6, 7, 8, 9, 10}

myset1.add(6)
print(myset1)  #Output: {1, 2, 3, 4, 5, 6}

print(myset1.pop()) #Output: 1
print(myset1)  #Output: {2, 3, 4, 5, 6}

print(myset1.union(myset2))  #Output: {2, 3, 4, 5, 6, 7, 8, 9, 10}

print(myset1.intersection(myset2))  #Output: {6}

print(myset1.clear())
print(myset1)     #Output: set()
```

➢ **User Input:**

**input ()** method is used to take the user input



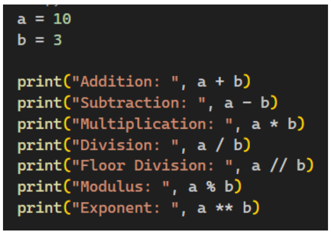
```
fname = input("Enter your first name: ")
lname = input("Enter your last name: ")
age = input("Enter your age: ")

print(f"Hello {fname} {lname}! You are {age} years old.")
```

# OPERATORS

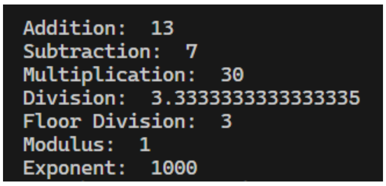
Python supports various types of operators that allow you to perform operations on variables and values.

1. **Arithmetic Operators:** Performs mathematical operations.

| Operators | Description |
|---|---|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| / | Division |
| // | Floor Division |
| % | Modulus (Return division remainder) |
| ** | Exponentiation |



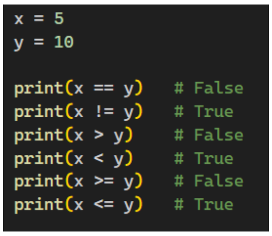
```
a = 10
b = 3

print("Addition: ", a + b)
print("Subtraction: ", a - b)
print("Multiplication: ", a * b)
print("Division: ", a / b)
print("Floor Division: ", a // b)
print("Modulus: ", a % b)
print("Exponent: ", a ** b)
```



```
Addition:  13
Subtraction:  7
Multiplication:  30
Division:  3.3333333333333335
Floor Division:  3
Modulus:  1
Exponent:  1000
```

2. **Comparison Operators:** Compare two values and return True or False based on the condition.

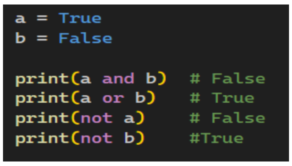
| Operators | Description |
|---|---|
| = = | equal to |
| != | not equal to |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |



```
x = 5
y = 10

print(x == y)   # False
print(x != y)   # True
print(x > y)    # False
print(x < y)    # True
print(x >= y)   # False
print(x <= y)   # True
```

3. **Logical Operators:** Used to perform logical operations on Boolean values.

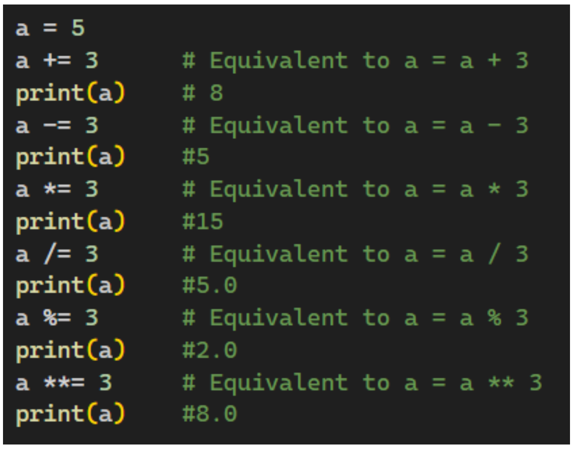
| Operator | Description |
|---|---|
| **and** | Logical AND |
| **or** | Logical OR |
| **not** | Logical NOT |



```
a = True
b = False

print(a and b)  # False
print(a or b)   # True
print(not a)    # False
print(not b)    #True
```

4. **Assignment Operators:** Used to assign values to variables. They are a shorthand way of combining an operation with assignment.

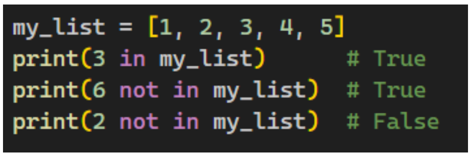
| **Operator** | **Description** |
|---|---|
| = | Assignment |
| += | Addition assignment |
| -= | Subtraction assignment |
| *= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulus assignment |
| **= | Exponentiation assignment |



```
a = 5
a += 3       # Equivalent to a = a + 3
print(a)     # 8
a -= 3       # Equivalent to a = a - 3
print(a)     #5
a *= 3       # Equivalent to a = a * 3
print(a)     #15
a /= 3       # Equivalent to a = a / 3
print(a)     #5.0
a %= 3       # Equivalent to a = a % 3
print(a)     #2.0
a **= 3      # Equivalent to a = a ** 3
print(a)     #8.0
```

5. **Membership Operators:** Used to test whether a value exists within a sequence (like a list, tuple, string, or set) or not.

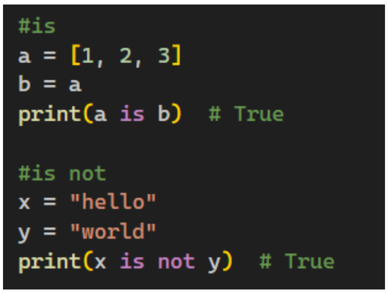
- **in:** True if a value is found in the sequence.
- **not in:** True if a value is not found in the sequence.



```
my_list = [1, 2, 3, 4, 5]
print(3 in my_list)      # True
print(6 not in my_list)  # True
print(2 not in my_list)  # False
```

6. **Identity Operators:** Used to check whether two variables reference the same object in memory.

- **is:** True if both variables are the same object.
- **is not:** True if both variables are not the same object.

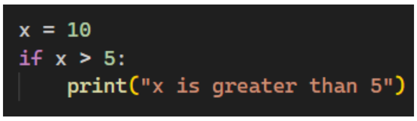


```
#is
a = [1, 2, 3]
b = a
print(a is b)  # True

#is not
x = "hello"
y = "world"
print(x is not y)  # True
```

# CONDITIONAL STATEMENTS

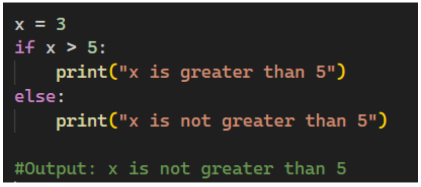
Conditional statements allow you to control the flow of your program based on certain conditions.

1. **if Statement:** Used to execute a block of code only if a specified condition is true.



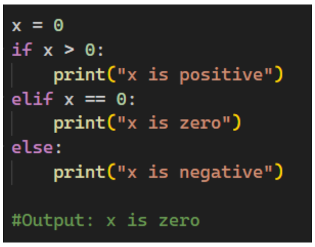
```
x = 10
if x > 5:
    print("x is greater than 5")
```

2. **if-else Statement:** The 'else' statement used in conjunction with 'if' to execute a block of code when the specified condition is false.



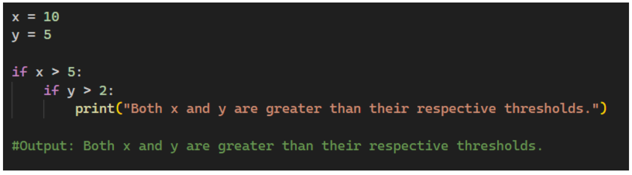
```
x = 3
if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")

#Output: x is not greater than 5
```

3. **if-elif-else Statement:** The 'elif' statement allows you to check multiple conditions sequentially after the initial 'if'. If the 'if' condition is false, it checks the 'elif' conditions one by one until a true condition is found, or it executes the 'else' block if none of the conditions is true.



```
x = 0
if x > 0:
    print("x is positive")
elif x == 0:
    print("x is zero")
else:
    print("x is negative")

#Output: x is zero
```

4. **Nested if Statements:** You can nest if statements inside other if statements to handle more complex conditions.

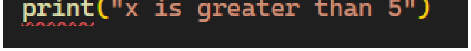


```
x = 10
y = 5

if x > 5:
    if y > 2:
        print("Both x and y are greater than their respective thresholds.")

#Output: Both x and y are greater than their respective thresholds.
```

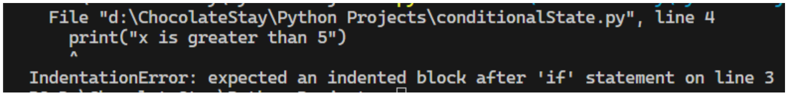
➢ **Indentation:**

Python relies on indentation (whitespace at the beginning of a line) to define scope in the code. Other programming languages often use curly-brackets for this purpose.

**Example:** if statement, without indentation (will raise an IndentationError):



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

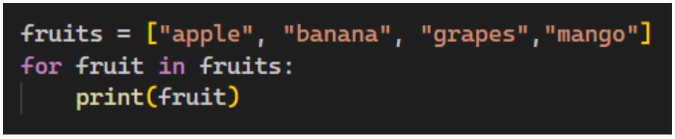


```
  File "d:\ChocolateStay\Python Projects\conditionalState.py", line 4
    print("x is greater than 5")
    ^
IndentationError: expected an indented block after 'if' statement on line 3
```

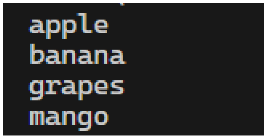
# LOOPS AND FUNCTIONS

- **LOOPS:** Python has two primitive loops:

1. **for loop:** Used for iterating over a sequence (such as a list, tuple, string, or range) or other iterable objects. The loop iterates over each item in the sequence.

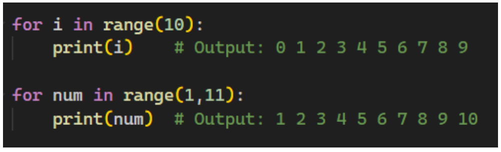


```
fruits = ["apple", "banana", "grapes","mango"]
for fruit in fruits:
    print(fruit)
```



```
apple
banana
grapes
mango
```

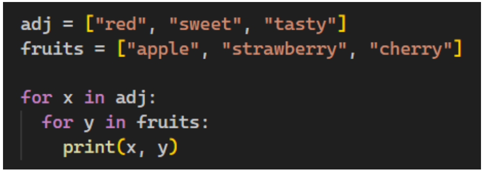
- **range () function:** used to generate a sequence of numbers.



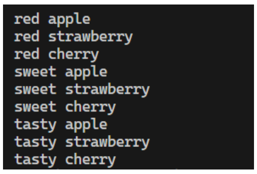
```
for i in range(10):
    print(i)    # Output: 0 1 2 3 4 5 6 7 8 9

for num in range(1,11):
    print(num)  # Output: 1 2 3 4 5 6 7 8 9 10
```

The range () function defaults to **0** as a starting value, we can specify the starting value by adding a parameter: **range (1, 11)**, which means values from 1 to 11 (but not including 11).

- **Nested for Loops:** One or more loops are placed inside another loop.



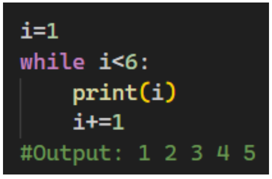
```
adj = ["red", "sweet", "tasty"]
fruits = ["apple", "strawberry", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```



```
red apple
red strawberry
red cherry
sweet apple
sweet strawberry
sweet cherry
tasty apple
tasty strawberry
tasty cherry
```

2. **while loop:** The while loop continues to execute a block of code as long as a specified condition is true.

**Example:** Print **i** as long as **i** is less than 6:

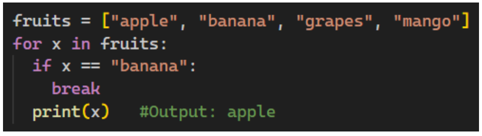


```
i=1
while i<6:
    print(i)
    i+=1
#Output: 1 2 3 4 5
```

Note: Remember to increment i (i+=1), or else the loop will continue forever.

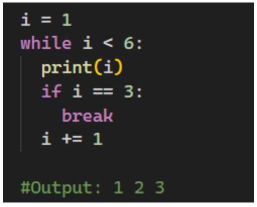
- **break, continue, pass statements in loops:**
  - **break:** Used to exit the loop prematurely.

**In for loop:**



```
fruits = ["apple", "banana", "grapes", "mango"]
for x in fruits:
  if x == "banana":
    break
  print(x)   #Output: apple
```

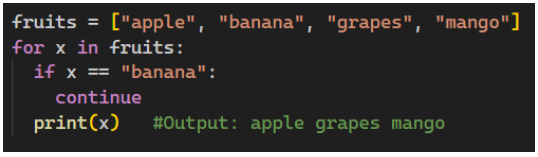
**In while Loop:**



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

#Output: 1 2 3
```

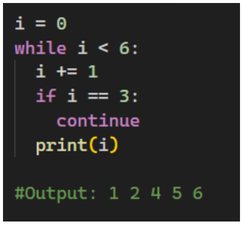
- **continue:** used to stop the current iteration of the loop, and continue with the next.

  **In for loop:**



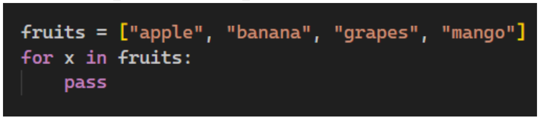
```
fruits = ["apple", "banana", "grapes", "mango"]
for x in fruits:
  if x == "banana":
    continue
  print(x)   #Output: apple grapes mango
```

  **In while loop:**



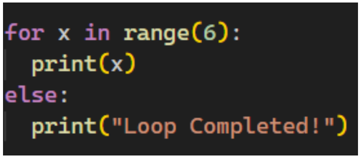
```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)

#Output: 1 2 4 5 6
```

- **pass:** for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

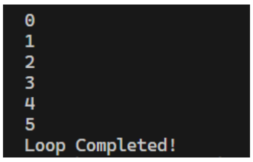


```
fruits = ["apple", "banana", "grapes", "mango"]
for x in fruits:
    pass
```

➢ **Else in for loop:** The 'else' keyword in a 'for' loop specifies a block of code to be executed when the loop is finished

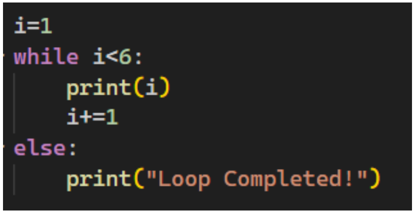


```
for x in range(6):
  print(x)
else:
  print("Loop Completed!")
```

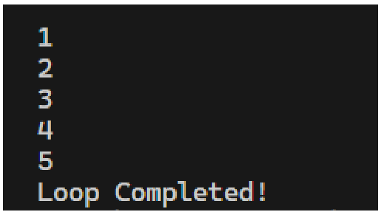


```
0
1
2
3
4
5
Loop Completed!
```

➢ **Else in While loop:**



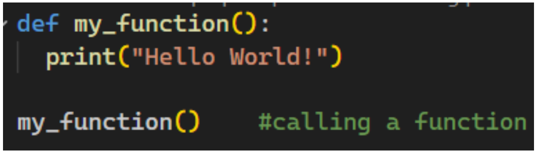
```
i=1
while i<6:
    print(i)
    i+=1
else:
    print("Loop Completed!")
```



```
1
2
3
4
5
Loop Completed!
```

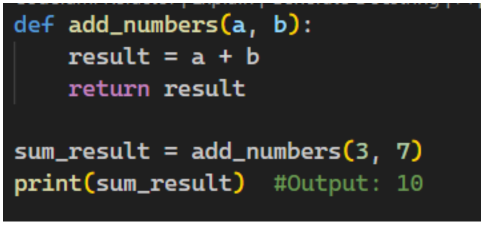
## ➢ FUNCTIONS:

A function is a block of code which only runs when it is called. In Python a function is defined using the **def** keyword:



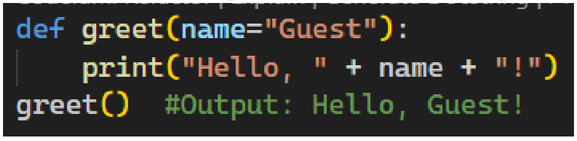
```python
def my_function():
  print("Hello World!")

my_function()    #calling a function
```

- **Parameters and Arguments:**
  **Parameters:** These are variables that are used in the function definition to represent the data that the function will operate on. (e.g. a,b)
  **Arguments:** Actual values passed to the function when it is called. (e.g. 3,7)



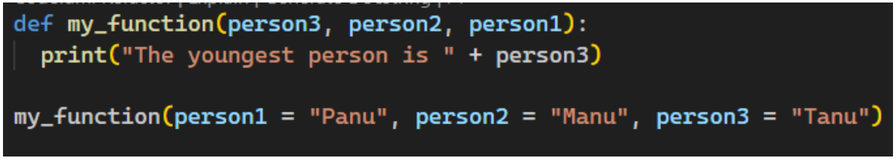
```python
def add_numbers(a, b):
    result = a + b
    return result

sum_result = add_numbers(3, 7)
print(sum_result)  #Output: 10
```

- **Default Arguments:** You can provide default values for parameters in a function. If the caller doesn't provide a value for a parameter, the default value is used.



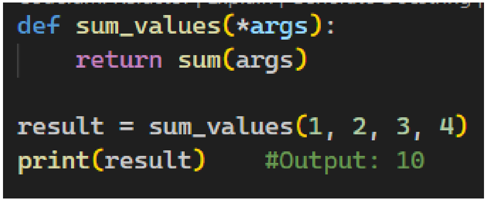
```python
def greet(name="Guest"):
    print("Hello, " + name + "!")
greet()  #Output: Hello, Guest!
```

- **Keyword Arguments:** You can also send arguments with the ‘key = value’ syntax. This way the order of the arguments does not matter.



```python
def my_function(person3, person2, person1):
  print("The youngest person is " + person3)

my_function(person1 = "Panu", person2 = "Manu", person3 = "Tanu")
```

- **Arbitrary Arguments, *args:** Allows a function to accept any number of arguments.



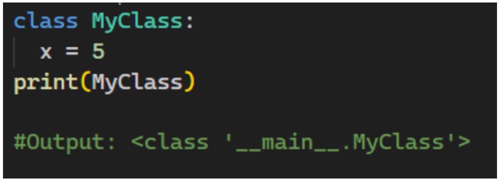
```python
def sum_values(*args):
    return sum(args)

result = sum_values(1, 2, 3, 4)
print(result)    #Output: 10
```

# OBJECT ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a programming paradigm that uses objects - collections of data and methods - to design and organize code. Python is an object-oriented programming language that supports the creation and manipulation of objects.

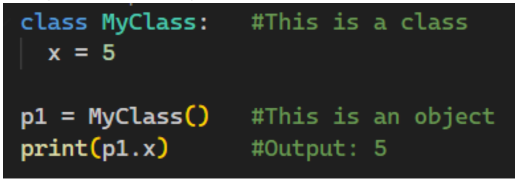
- **Classes and Objects:**

  **Class:** A class is a blueprint or a template for creating objects. It defines the attributes (data) and methods (functions) that the objects of the class will have.



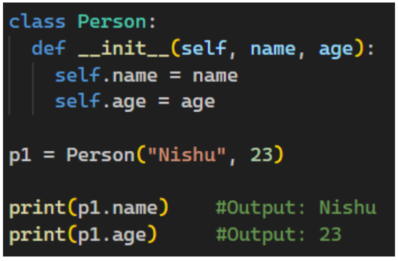
```
class MyClass:
  x = 5
print(MyClass)

#Output: <class '__main__.MyClass'>
```

**Object:** An object is an instance of a class. It performs actions through methods.



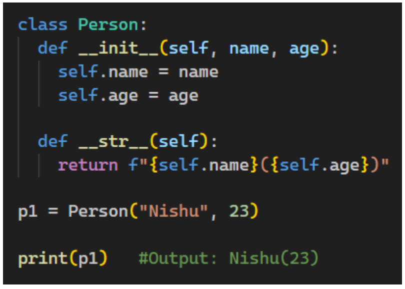
```
class MyClass:    #This is a class
  x = 5

p1 = MyClass()    #This is an object
print(p1.x)       #Output: 5
```

- **__init__():** It is a special method of classes that is called when an object is created. It initializes the object's attributes (e.g. self.name =name and self.age=age ).



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

p1 = Person("Nishu", 23)

print(p1.name)    #Output: Nishu
print(p1.age)     #Output: 23
```

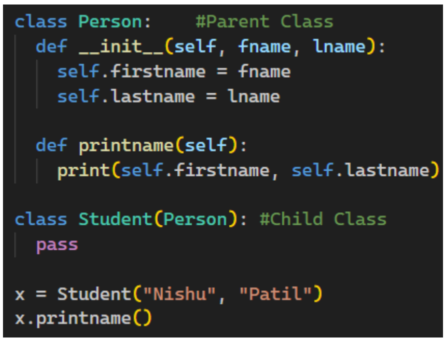
- **__str__():** It is a special method used to define the "informal" or "user-friendly" string representation of an object.



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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("Nishu", 23)

print(p1)    #Output: Nishu(23)
```

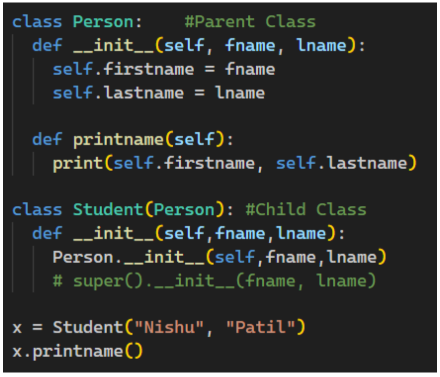
- **self** keyword is used as the conventional name for the first parameter in the method of a class. It refers to the instance of the class itself and is passed automatically when calling a method on an object. you could technically use any other name for the first parameter of a method, but using **self** is recommended for readability.

- **Inheritance:** Allows us to define a class that inherits all the methods and properties from another class.
  **Parent class:** class being inherited from, also called ‘base class’.
  **Child class:** class that inherits from another class, also called ‘derived class’.



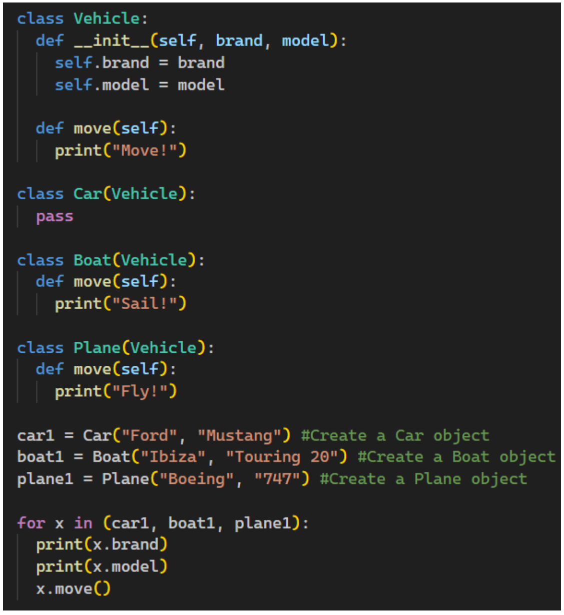
```python
class Person:    #Parent Class
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person): #Child Class
  pass

x = Student("Nishu", "Patil")
x.printname()
```

To keep the inheritance of the parent's **__init__()** function, add a call to the parent's **__init__()** function.
Python also has a **super()** function that will make the child class inherit all the methods and properties from its parent:



```python
class Person:    #Parent Class
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

class Student(Person): #Child Class
  def __init__(self,fname,lname):
    Person.__init__(self,fname,lname)
    # super().__init__(fname, lname)

x = Student("Nishu", "Patil")
x.printname()
```

- **Polymorphism:** The word "polymorphism" means "many forms", and in programming it refers to methods/functions/operators with the same name that can be executed on many objects or classes.

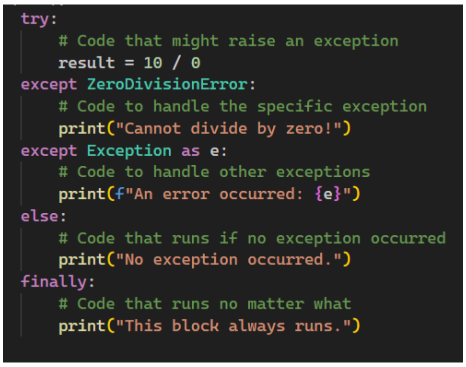


```python
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

Child classes inherits the properties and methods from the parent class.
In the example above you can see that the **Car** class is empty, but it inherits **brand**, **model**, and **move()** from Vehicle.
The Boat and Plane classes also inherit brand, model, and move() from Vehicle, but they both override the move() method.
Because of polymorphism we can execute the same method for all classes.

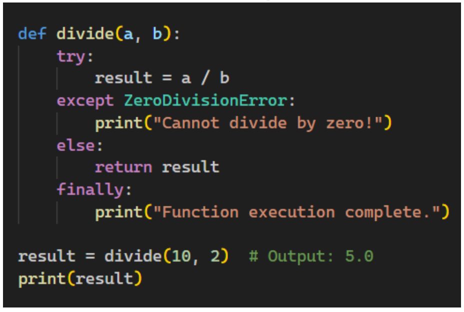
## ➢ EXCEPTION HANDLING:

The **try** block lets you test a block of code for errors.
The **except** block lets you handle the error.
The **else** block lets you execute code when there is no error.
The **finally** block lets you execute code, regardless of the result of the try- and except blocks.



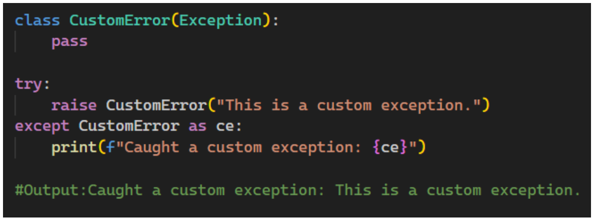
```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code to handle the specific exception
    print("Cannot divide by zero!")
except Exception as e:
    # Code to handle other exceptions
    print(f"An error occurred: {e}")
else:
    # Code that runs if no exception occurred
    print("No exception occurred.")
finally:
    # Code that runs no matter what
    print("This block always runs.")
```

### Exception Handling in Functions:



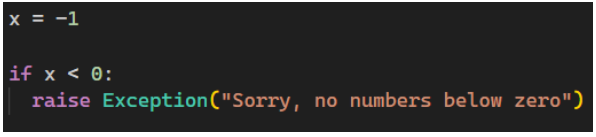
```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Cannot divide by zero!")
    else:
        return result
    finally:
        print("Function execution complete.")

result = divide(10, 2)  # Output: 5.0
print(result)
```

**Custom Exceptions:** You can define your own custom exceptions by creating a new class that inherits from the built-in Exception class.



```
class CustomError(Exception):
    pass

try:
    raise CustomError("This is a custom exception.")
except CustomError as ce:
    print(f"Caught a custom exception: {ce}")

#Output:Caught a custom exception: This is a custom exception.
```

**Raise an Exception:** To throw (or raise) an exception, use the raise keyword.

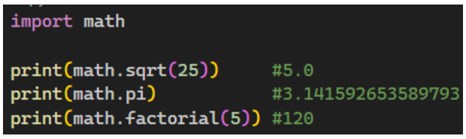


```
x = -1

if x < 0:
  raise Exception("Sorry, no numbers below zero")
```

# MODULES

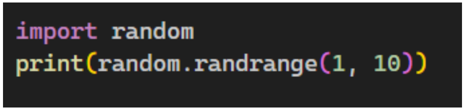
- **Built-in modules:** Python comes with a comprehensive standard library that includes a wide range of built-in modules for various purposes.

1. **math:** Provides mathematical functions- square root, trigonometric functions, etc.



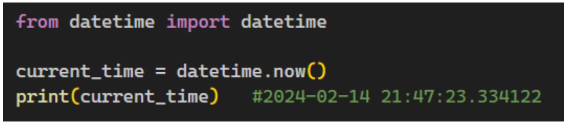
```
import math

print(math.sqrt(25))     #5.0
print(math.pi)           #3.141592653589793
print(math.factorial(5)) #120
```

2. **random:** Offers functions for generating pseudo-random numbers.



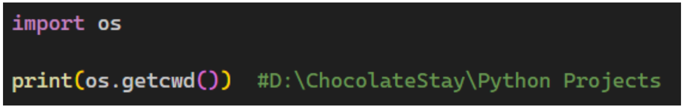
```
import random
print(random.randrange(1, 10))
```

3. **datetime:** Used for working with dates and times.



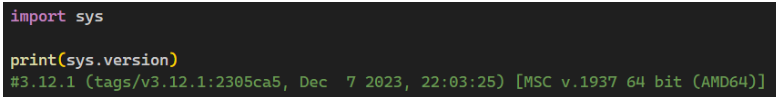
```
from datetime import datetime

current_time = datetime.now()
print(current_time)   #2024-02-14 21:47:23.334122
```

4. **os:** Provides a way to interact with the operating system.



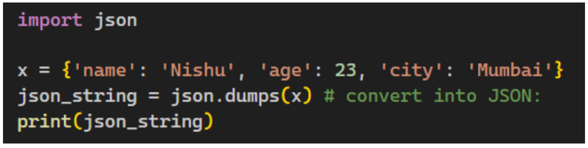
```
import os

print(os.getcwd())  #D:\ChocolateStay\Python Projects
```

5. **sys:** Provides access to some variables used or maintained by the Python interpreter and functions that interact strongly with the interpreter.



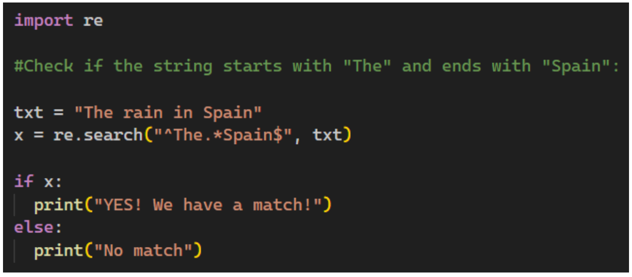
```
import sys

print(sys.version)
#3.12.1 (tags/v3.12.1:2305ca5, Dec  7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)]
```

6. **json:** Enables encoding and decoding of JSON data. You can convert Python objects (dict, list, tuple, string, int, float, True, False, None) into JSON strings:



```
import json

x = {'name': 'Nishu', 'age': 23, 'city': 'Mumbai'}
json_string = json.dumps(x) # convert into JSON:
print(json_string)
```

7. **re:** Allows the use of regular expressions for pattern matching and manipulation.

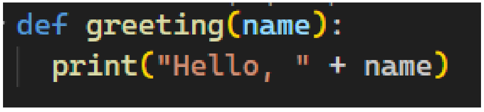


```
import re

#Check if the string starts with "The" and ends with "Spain":

txt = "The rain in Spain"
x = re.search("^The.*Spain$", txt)

if x:
  print("YES! We have a match!")
else:
  print("No match")
```

- **Creating Modules:**

To create a module just save your code with .py extension:

**mymodule.py**

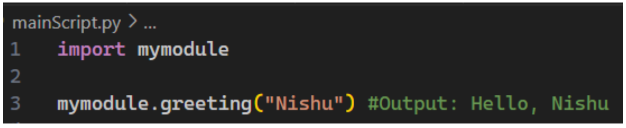


```python
def greeting(name):
    print("Hello, " + name)
```

- **Importing module:**

Now we can use the module we just created, by using the **import** statement.

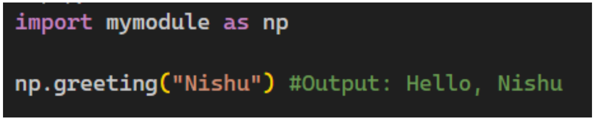
**mainScript.py**



```python
import mymodule

mymodule.greeting("Nishu") #Output: Hello, Nishu
```

- **Renaming the module:**

You can create an alias when you import a module, by using the **as** keyword.



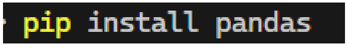
```python
import mymodule as np

np.greeting("Nishu") #Output: Hello, Nishu
```

- **PIP:**

**pip** is the package installer for Python. It is a command-line tool that allows you to install, uninstall, and manage Python packages from the Python Package Index (PyPI). PyPI is a repository of software packages developed and maintained by the Python community.

- **Install Packages:**

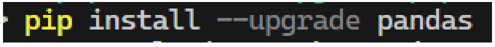
  pip install package_name



```
pip install pandas
```

- **Upgrade packages:**

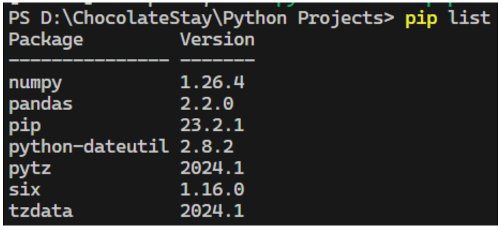
  pip install --upgrade package_name



```
pip install --upgrade pandas
```

- **List installed packages:**

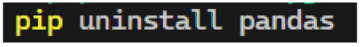
  pip list



```
PS D:\ChocolateStay\Python Projects> pip list
Package         Version
--------------- -------
numpy           1.26.4
pandas          2.2.0
pip             23.2.1
python-dateutil 2.8.2
pytz            2024.1
six             1.16.0
tzdata          2024.1
```

- **Uninstall packages:**

  pip uninstall package_name



```
pip uninstall pandas
```