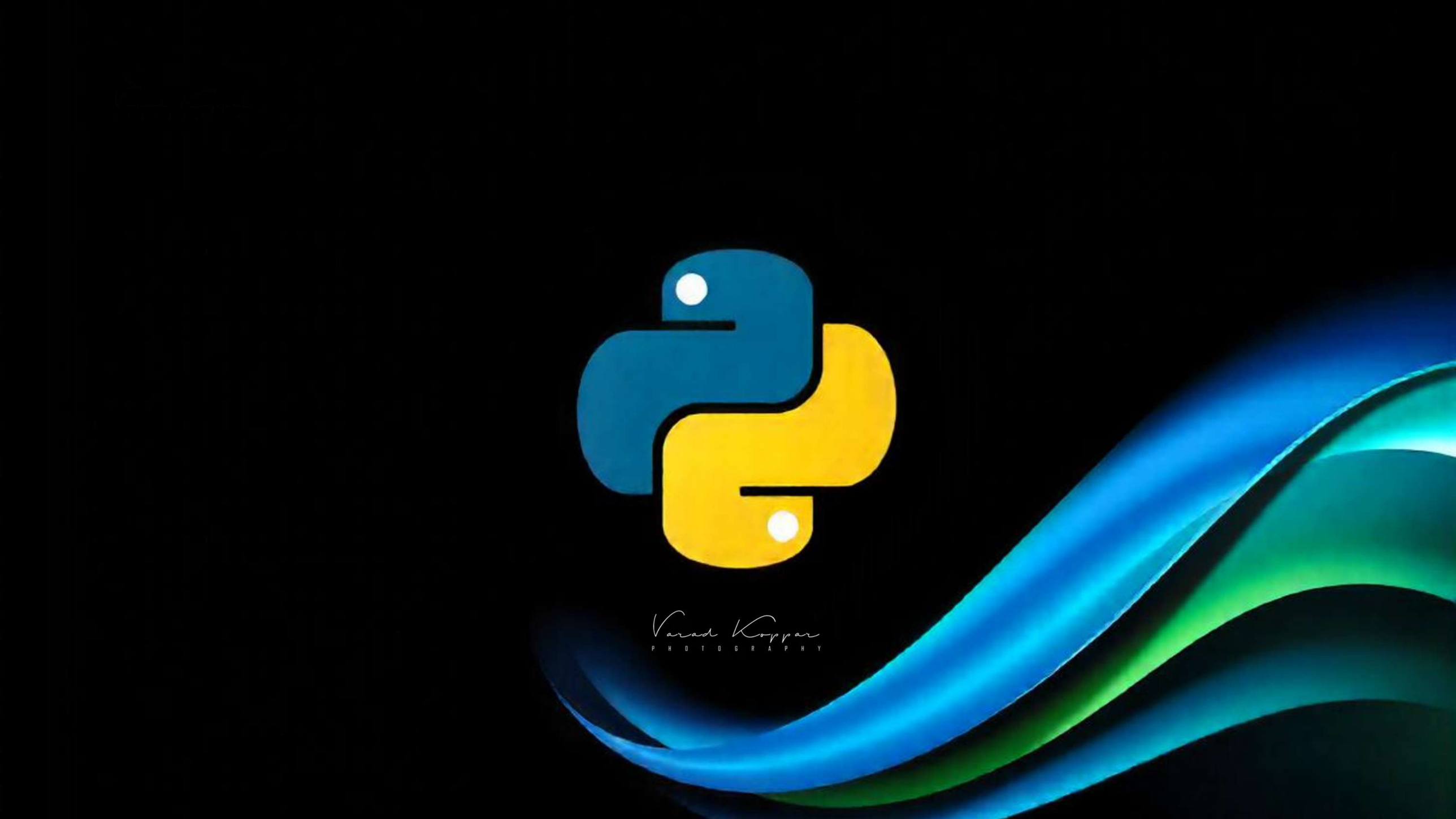

Varad Kappar
PHOTOGRAPHY

Content Overview with Page References

Quick access to topics in python

Topic
Introduction to Python – Features, uses, setup
Operations in Python – Arithmetic, comparison, logical operators
User Input & Type Conversion – input(), int(), float(), str()
Control Flow – if, elif, else
Loops – for, while, basic patterns
Strings – Basics, slicing, common string methods
Lists – Creating, indexing, aliasing & cloning, basic operations
Tuples – Creating, Indexing, Immutability, Basic Operations
Dictionaries – Creating, accessing, updating
Functions in Python – Defining functions, parameters, return values
Scope of Variables – Local vs Global
File Handling (Basic) – Reading/writing text files
Exception Handling (Basic) – try/except

Guido van Rossum

father of python

Introduction to Python

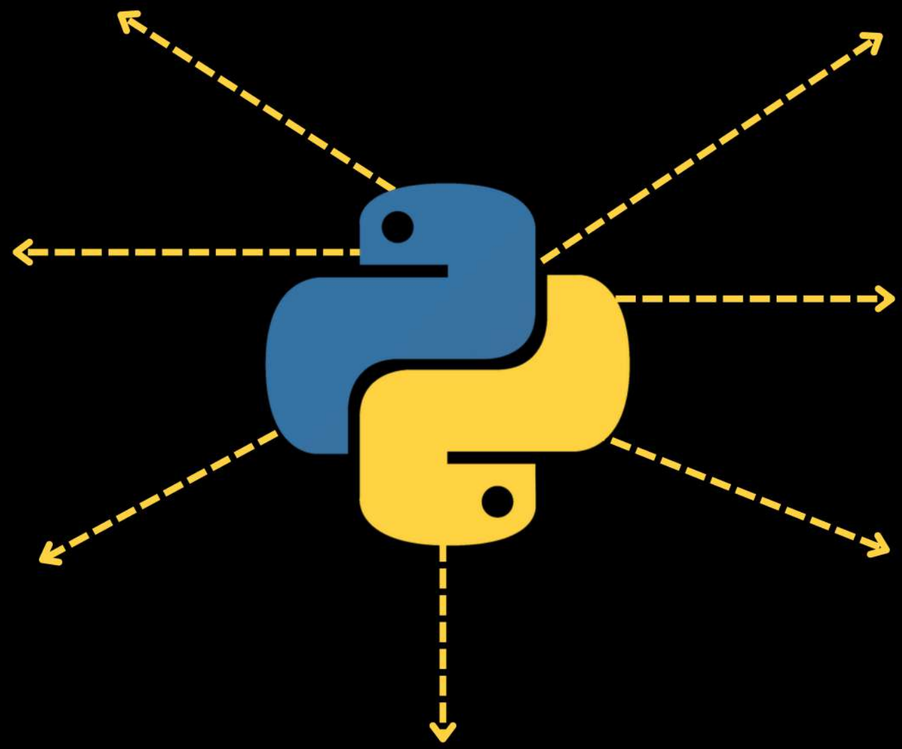
Python is a high-level, general-purpose programming language known for its simplicity and readability. Created by Guido van Rossum in 1989 and released in 1991, Python has evolved through versions 1.0, 2.0, and 3.0, each adding important features like functions, exception handling, list comprehensions, and improved language consistency. Today, Python is widely used in areas such as web development, data science, and machine learning.

Its English-like syntax makes it easy to learn for beginners, while features like dynamic typing, high-level data structures, and support for modules and packages enable rapid application development and promote code modularity and reuse. Python's readability and simplicity also help reduce maintenance effort, making it a popular choice for developers of all levels.

General-purpose

High-level

Interpreted



Scripting

Extensive libraries

Dynamically typed

Object-Oriented Programming (OOP)

Key Features of Python

```python
# Arithmetic operators: +, -, /, *, //, %, **
a = 20
b = 40
print(a + b)

x=60
y=4
print(x-y)

p=15
c=8
print (p*c)

o=766
p=12

print(o/p)

a=10
d=20
print(a/d)

x=10
d=20
print(x//d)


w=10
d=20
print(w**d)
```

Arithmetic Operators: Used to perform mathematical operations.

Addition (+) x = 5; y = 3; print(x + y) # Output: 8

Subtraction (-) a = 10; b = 4; print(a - b) # Output: 6

Multiplication (*) m = 7; n = 6; print(m * n) # Output: 42

Division (/) p = 15; q = 3; print(p / q) # Output: 5.0

Floor Division (//) r = 17; s = 4; print(r // s) # Output: 4

Modulus (%) t = 10; u = 3; print(t % u) # Output: 1

Exponentiation (**) v = 2; w = 3; print(v ** w) # Output: 8

```
a=20
b=50
print(a>b)

a=20
b=50
print(a<b)


a=20
b=50
print(a>=b)

a=20
b=10
print(a<=b)


a=20
b=5
print(a==b)


a=20
b=50
print(a!=b)
```

Comparison Operators: Used to compare two values.

Equal to (==) x = 5; y = 3; print(x == y) # Output: False

Not equal to (!=) a = 10; b = 4; print(a != b) # Output: True

Greater than (>) m = 7; n = 6; print(m > n) # Output: True

Less than (<) p = 15; q = 20; print(p < q) # Output: True

Greater than or equal to (>=) r = 17; s = 17; print(r >= s) # Output: True

Less than or equal to (<=) t = 10; u = 12; print(t <= u) # Output: True

```python
#Logical Operators in Python

#Logical AND (and)

x = 7
print(x < 5 and x < 10)#Both conditions are not true
# Output: False

#Logical OR (or)

y = 3
print(y < 5 or y > 10)#At least one condition is true
# Output: True

# Logical NOT (not)

z = 5
print(not (z < 10))  # Reverses the condition's truth value
# Output: False
```

Logical Operators: Used to combine conditional statements, allowing more complex conditions. Python has three main logical operators:

Logical AND (and): Returns True if both conditions are True; otherwise, False.

Logical OR (or): Returns True if at least one condition is True; only False if both are False.

Logical NOT (not): Reverses the truth value of a condition; True becomes False and vice versa.

```python
#assignment Operator

a=10
print(a)#10
a+=1 #a=a+1
print (a) #11

b=40
b-=13
print(b)

c=15
c*=4
print(c)


d=150
d/=4
print(d)

e=150
e%=4
print(e)


f=1150
f//=14
print(f)


p=1120
p**=14
print(p)

g=120
g**=3
print(g)
```

Assignment Operators: Used to assign values to variables.

Assignment (=) x = 10; print(x) # Output: 10

Add and Assign (+=) y = 5; y += 3; print(y) # Output: 8

Subtract and Assign (-=) a = 10; a -= 4; print(a) # Output: 6

Multiply and Assign (*=) m = 6; m *= 2; print(m) # Output: 12

Divide and Assign (/=) p = 20; p /= 4; print(p) # Output: 5.0

Modulus and Assign (%=) t = 10; t %= 3; print(t) # Output: 1

Exponentiation and Assign (**=) v = 2; v **= 3; print(v) # Output: 8

Floor Division and Assign (//=) r = 17; r //= 4; print(r) # Output: 4

```
#membership operators in , not in

s="aditya"
print("a" in s)

s="aditya"
print("A" in s)

a="prathamesh"
print("o" in a)

a="prathamesh"
print("p" in a)


a="varad"
print("o" not in a)

a="varad"
print("d" not in a)
```

Membership Operators: Used to test if a value exists in a sequence (like lists, strings, or tuples).

in Operator: Returns True if the value exists in the sequence, otherwise False.

numbers = [1, 2, 3, 4, 5]
print(3 in numbers) # Output: True
print(6 in numbers) # Output: False

name = "aditya"
print("a" in name) # Output: True
print("b" in name) # Output: False

not in Operator: Returns True if the value does NOT exist in the sequence, otherwise False

letters = "Hello"
print("z" not in letters) # Output: True
print("H" not in letters) # Output: False

fruits = ["banana", "cherry"]
print("orange" not in fruits) # Output: True
print("banana" not in fruits) # Output: False

```
#identity operator is, is not

a=10
b=10
print (a is b ) #true

a=20
b=10
print(id(a))
print(id(b))

print (a is b )#false


a=20
b=20
print(id(a))
print(id(b))

print (a is not b )#false



a = 20
b = 30
print(a is not b)  # True, because a and b have different values


#id()= return address #id() Function

a=40
print (id(a))

b=90
print (id(b))

''' id(b) will output a unique identifier (an integer)
for the memory location where 90 is stored.'''
```

Identity Operators: Used to check whether two variables point to the same object in memory.

is Operator: Returns True if two variables refer to the same object

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

c = [1, 2, 3]
print(a is c) # Output: False

is not Operator: Returns True if two variables refer to different objects

x = "hello"
y = x
print(x is not y) # Output: False

m = "world"
print(x is not m) # Output: True

id() Function: Returns the unique identity (memory address) of an object. Useful to verify object identity.

a = 70
print(id(a)) # Output: (some unique memory address, varies by system)
# Example: 140706552699200

```
#input() = used to take input from user

#a= input("Enter the value if a")
#print("The entered value is:", a)

x = 122092839
print(x)
print(type(str(x)))

a = 12.2092839
print(a)
print(type(float(a)))


d = 1220
print(d)
print(type(int(d)))

a = 12.2092839
print(a)
print(type(float(a)))

a= int ( input("Enter the value if a"))
b= int (input ("Enter the value if b"))
print("addtion of entered value",a+b)


x=122092839
print(x)


a= input("Enter the value if a")
b= input ("Enter the value if b")
print("addtion of entered value",a+b)

'''type casting the process of converting one data type into
another datatype called as conversion method or type casting.'''
```

User Input and Type Conversion

User Input:
Input() is used to get input from the user.
The value is always returned as a string, regardless of what is entered.

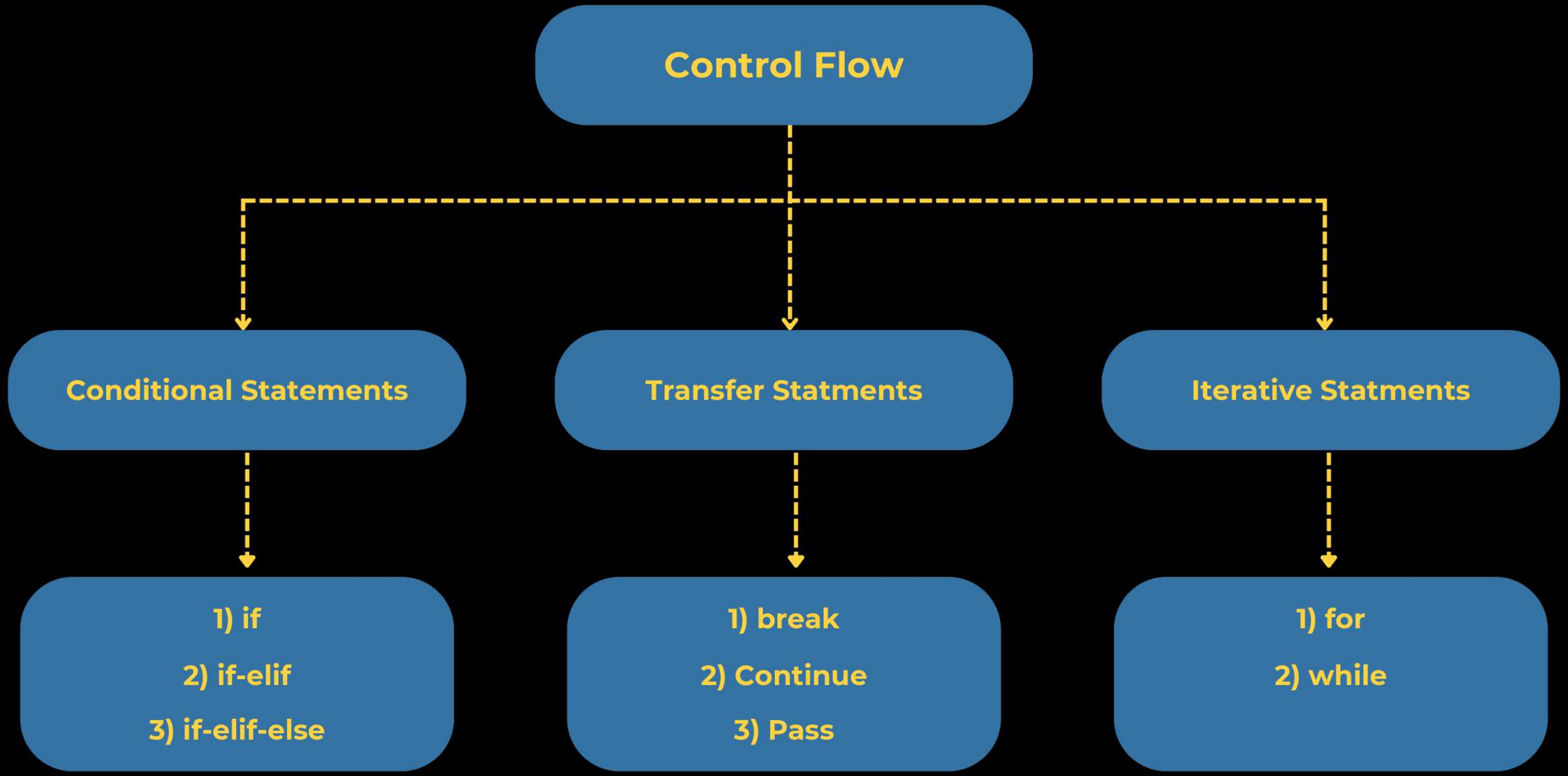
```
a = input("Enter the value of a: ")
print("The entered value is:", a)
```

Type Conversion:
Changing one data type to another enables operations between different types.
Common functions: int(), float(), str().

```
x = 122092839
print(x, type(str(x)))  # Convert integer to string

a = 12.2092839
print(a, type(float(a)))  # Ensure value is float

d = 1220
print(d, type(int(d)))  # Ensure value is integer
```


Control Flow
Conditional Statements
Transfer Statments
Iterative Statments
1) if
2) if-elif
3) if-elif-else
1) break
2) Continue
3) Pass
1) for
2) while

```
#if = if perticular condtional. is true then return if block

'''
if condition:
#code to execute / if block condition'''

if a>10:
    print ("a is greater than 10")


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

Understanding if Statements

Syntax: Executes a block of code if the condition is True.

Check numeric condition

```
a = 15
if a > 10:
   print("a is greater than 10")
```

Check string condition

```
word = "Python"
if word == "Python":
   print("This is Python programming.")
```

```
'''if condition is true then if block will be executed
    otherwise else block will be executed'''

''' if condition:
if block code

else condition:
else block code '''



a = 10
b = 13

if a > b:
    print('a is greater than b')
else:
    print('b is greater than a')
```

Introduction to if-else Statement

Executes one block of code if a condition is True, and another block if the condition is False.

 # Example: Loan approval based on credit score

```
a = 720  # Customer's credit score
b = 550  # Minimum required credit score

if a > b:
    print("Loan approved!")
else:
    print("Loan denied. Your credit score is too low.")
```

Key Points:

- if block runs when the condition is True.
- else block runs when the condition is False.
- Proper indentation is mandatory to define blocks in Python.

```python
x = 10

if x > 15:
    print("x is greater than 15")
elif x > 5:
    print("x is greater than 5 but less than or equal to 15")
else:
    print("x is less than or equal to 5")

# WAP to compare three variables

x = int(input("Enter value of X: "))
y = int(input("Enter value of Y: "))
z = int(input("Enter value of Z: "))

if x > y and x > z:
    print("X is the greatest.")
elif y > z:
    print("Y is the greatest.")
else:
    print("Z is the greatest.")
```

Understanding if-elif-else Statements

Used to check multiple conditions one by one.

Executes the block for the first condition that is True, then skips the rest.

If no condition is True, the else block (if provided) runs.

 # Example: Categorize a student based on marks

```
marks = 75

if marks >= 90:
    print("Grade: A+")
elif marks >= 60:
    print("Grade: B")
else:
    print("Grade: C")
```

Output:
Grade: B

Key Points :

- Only the first True condition executes its block.
- elif is optional; else is executed when all conditions are False.
- Indentation is mandatory for all blocks.

```
# Print numbers from 1 to 5 using a for loop
for i in range(1, 6):
    print("Number:", i)
'''
output
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5 '''
```

Introduction – Why Loops Are Used

Loops let you repeat tasks without rewriting the same code multiple times.
They make programs shorter, easier to read, and easier to maintain, especially when you need to run the same set of statements many times or process each item in a collection.

Syntax:

```
for i in range(start, end, step):
    # code to execute
```

- start → starting number (default is 0)
- end → loop runs until end-1
- step → how much to increment each time (default is 1)

```python
for i in range (10):
    print (i,"Good Morning")


for i in range (1,11):
    print (i)


for a in range(2, 101, 2):
    print(a)


for w in range (5):
    for j in range(5):
        print(w,j)
```

Introduction – Why Loops Are Used

Loops let you repeat tasks without rewriting the same code multiple times.
They make programs shorter, easier to read, and easier to maintain, especially when you need to run the same set of statements many times or process each item in a collection.

Syntax:

for i in range(start, end, step):
    # code to execute

- start → starting number (default is 0)
- end → loop runs until end-1
- step → how much to increment each time (default is 1)

```python
i = 1
while i <=10:
  print(i)
  i += 1


i = 0
while i <50:
  print(i)
  i += 2
print("while loop completed")

i = 0
while i <=50:
  print(i)
  i += 5
print("while loop completed")


b = 150
while b >= 3:
    print(b)
    b -= 3
print("while loop completed")
```

Exploring Python Iteration with while Loops

If you want to execute code repeatedly until a specific condition is met, use a while loop.
It is useful when you don't know in advance how many times the code needs to run or when the condition can change during execution.
The loop keeps running while the condition is True and stops once it becomes False.

Syntax:

```
while condition:
   # code to execute
   # increment/decrement to avoid infinite loop
```

Python Iteration with for Loops

If you want to run a block of code for each item in a sequence or for a fixed number of times, use a for loop.
It is ideal when the number of iterations is known or when you need to iterate over elements of a collection (like a list, string, or range).
The loop automatically takes the next item in the sequence on each iteration until all items are processed.

```python
# break

for i in range(10):
    if i == 7:
        print("Reached 7, exiting the loop.")
        break
    print(i)


# continue
#Skip even numbers in the loop

for i in range(10):
    if i % 2 == 0:  # Check if the number is even
        continue    # Skip the rest of the loop for this iteration
    print(i)        # Print the number if it is odd

#continue
# Print only even numbers

for i in range(10):
    if i % 2 != 0:  # If the number is odd
        continue     # Skip this iteration (odd numbers)
    print(i)  # Print only even numbers

#pass
# Check for numbers in a range

for i in range(6):
    if i == 3:
        pass  # Placeholder, no action taken for 3
    else:
        print(i)
```

Transfer Statements in Python

Transfer statements are used to alter the normal flow of a program inside loops or conditional blocks.
They give you control over when to stop, skip, or temporarily do nothing in a block of code.

Types of Transfer Statements:

break – Stops the loop immediately and exits.

continue – Skips the current iteration and goes to the next.

pass – Does nothing; acts as a placeholder when a statement is required.

Centered Pyramid Pattern of Stars

A pyramid of stars, starting with one star at the top and increasing by one star per row, all centrally aligned.

```
row = 5
for i in range(1, row + 1):
    print(" " * (row - i) + "*" * (2 * i - 1))
```

```python
#Creating Strings in Python

# Single quotes
s1 = 'Hello'

# Double quotes
s2 = "Hello"

# Triple quotes
s3 = '''Hello'''
print(s1, s2, s3)
```

What is a String?

A string is a sequence of characters used to store text in Python. Strings are immutable, meaning their content cannot be changed after creation.

Creating Strings in Python

trings can be created using single quotes ('...'), double quotes ("..."), or triple quotes ('''...''' or """...""") for multi-line strings. Examples:

```python
#Positive Indexing: Counts from the start, beginning with 0.

word = "Python"
print(word[0])  # Output: P
print(word[3])  # Output: h

#Negative Indexing: Counts from the end, starting with -1.

print(word[-1]) # Output: n
print(word[-3]) # Output: h


#Slicing allows you to get a part of the string by specifying a start and end index:

print(word[0:4])    # Output: Pyth
print(word[-4:-1]) # Output: tho
```

Accessing Characters in a String

In Python, a string is a sequence of characters, and you can access each character individually using indexing or a portion of the string using slicing.

Indexing lets you pick characters based on their position:
Positive Indexing: Counts from the start, beginning with 0.

Negative Indexing: Counts from the end, starting with -1.

Slicing allows you to get a part of the string by specifying a start and end index:

```python
#Concatenation (+): Combines two or more strings.

greeting = "Hello " + "World"
print(greeting)  # Output: Hello World

#Repetition (*): Repeats a string multiple times.

echo = "Hi" * 3
print(echo)  # Output: HiHiHi


#Slicing: Extract a substring using [start:end].

text = "Python"
print(text[1:4])  # Output: yth
```

String Operations

Strings in Python support several basic operations to manipulate, combine, and extract information. These operations allow you to work efficiently with text, making it easy to create, modify, or analyze string data. Python provides simple syntax for concatenation, repetition, indexing, slicing, and other manipulations.

Concatenation (+): Combines two or more strings.

Repetition (*): Repeats a string multiple times.

Indexing: Access individual characters using positions.

Slicing: Extract a substring using [start:end].

```python
#.upper() → Changes all characters in the string to uppercase.

text = "hello"
print(text.upper())  # Output: HELLO

#.lower() → Converts all characters to lowercase.

text = "HELLO"
print(text.lower())  # Output: hello

#.strip() → Removes any spaces at the beginning or end of a string.

text = "  Python  "
print(text.strip())  # Output: Python

#.split() → Divides a string into a list of words or substrings.

text = "Python is fun"
print(text.split())  # Output: ['Python', 'is', 'fun']

#.replace() → Replaces a specific substring with another.

text = "I like Java"
print(text.replace("Java", "Python"))  # Output: I like Python
```

Useful String Methods

Python has useful string methods to make text handling easier. .upper() converts text to uppercase, .lower() to lowercase, .strip() removes extra spaces, .split() breaks a string into a list, and .replace() substitutes parts of a string. These methods help clean, format, and process text efficiently.

.upper() → Changes all characters in the string to uppercase.

.lower() → Converts all characters to lowercase.

.strip() → Removes any spaces at the beginning or end of a string.

.split() → Divides a string into a list of words or substrings.

.replace() → Replaces a specific substring with another.

```python
name = "VK"
age = 28
print(f"My name is {name} and I am {age} years old.")
# Output: My name is Alice and I am 25 years old.

a = 5
b = 3
print(f"The sum of {a} and {b} is {a + b}.")
# Output: The sum of 5 and 3 is 8.

pi = 3.14159265
print(f"Value of pi up to 2 decimal places: {pi:.2f}")
# Output: Value of pi up to 2 decimal places: 3.14
```

f-strings / Formatting

f-strings, introduced in Python 3.6, allow you to embed variables and expressions directly inside strings. This makes it easier to create readable and dynamic text without using complicated concatenation or formatting methods.

```python
# Empty list

my_list = []
print(my_list)  # Output: []

# List of integers

numbers = [1, 2, 3, 4, 5]
print(numbers)  # Output: [1, 2, 3, 4, 5]

# List with mixed types

mixed = [1, "Python", 3.14, True]
print(mixed)  # Output: [1, 'Python', 3.14, True]

# Modify an element in a list - lists are mutable, so we can change items by index

fruits = ["apple", "banana", "cherry"]
fruits[1] = "orange"  # Modify second item
print(fruits)  # Output: ['apple', 'orange', 'cherry']

# Mixed list example

my_list = [10, "Python", 3.14, False]
print(my_list)  # Output: [10, 'Python', 3.14, False]

# Shows ordered, allows mixed types, mutable, allows duplicates

my_list[0] = 20
print(my_list)  # Output: [20, 'Python', 3.14, False]
```

Introduction to Python Lists

A list in Python is an ordered collection of items, which can store elements of different data types, such as numbers, strings, or even other lists. Lists are mutable, meaning you can modify, add, or remove elements after creation.

Key Features of Python Lists

Ordered: Elements maintain the order in which they are added.

Mutable: Items can be modified using indexing.

Heterogeneous: Can contain elements of different types.

Dynamic: Size can increase or decrease at runtime.

Supports Nesting: A list can contain another list as an element.

```python
# Create a list
numbers = [1, 2, 3]

# append() - Adds an element to the end of the list
numbers.append(4)
print(numbers)  # Output: [1, 2, 3, 4]

# insert() - Adds an element at a specific index
numbers.insert(1, 10)
print(numbers)  # Output: [1, 10, 2, 3, 4]

# remove() - Removes the first occurrence of a value
numbers.remove(10)
print(numbers)  # Output: [1, 2, 3, 4]

# pop() - Removes element at a given index (default is last)
last_item = numbers.pop()
print(last_item)  # Output: 4
print(numbers)    # Output: [1, 2, 3]

# index() - Returns the index of the first occurrence of a value
print(numbers.index(2))  # Output: 1

# count() - Returns the number of times a value occurs
numbers.append(2)
print(numbers.count(2))  # Output: 2

# sort() - Sorts the list in ascending order
numbers.sort()
print(numbers)  # Output: [1, 2, 2, 3]

# reverse() - Reverses the elements of the list
numbers.reverse()
print(numbers)  # Output: [3, 2, 2, 1]

# len() - Returns the number of items in the list
print(len(numbers))  # Output: 4
```

Essential List Functions in Python

Python provides several built-in functions and methods to make working with lists easier. These functions help you add, remove, find, and count elements, as well as manipulate the list efficiently.

Common List Functions:

append() – Adds an element at the end of the list.

insert() – Adds an element at a specific index.

remove() – Removes the first occurrence of a value.

pop() – Removes and returns an element at a given index (default is the last).

index() – Returns the index of the first occurrence of a value.

count() – Returns how many times a specific value appears in the list.

sort() – Sorts the list in ascending order.Returns how many times a specific value appears in the list.

reverse() – Reverses the order of elements.

len() – Returns the total number of elements in a list.

```python
#Positive Indexing

fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry

#Negative Indexing

print(fruits[-1])  # Output: cherry
print(fruits[-2])  # Output: banana


#Aliasing and Cloning Lists

original = [1, 2, 3]
alias = original
alias[0] = 100
print(original)  # Output: [100, 2, 3]


#To create a separate copy that doesn't affect the original list

original = [1, 2, 3]
clone = original.copy()
clone[0] = 100
print(original)  # Output: [1, 2, 3]
print(clone)     # Output: [100, 2, 3]
```

Indexing and Accessing Elements

Lists in Python are ordered sequences, which means each element has a unique position called an index. You can access elements using positive or negative indexin

Positive Indexing starts at 0 for the first element:

Negative Indexing starts from the end, with -1 representing the last element:

Aliasing and Cloning Lists

Python lists are mutable, which means their contents can be changed. When you assign a list to another variable using =, both variables point to the same list. This is called aliasing.

```python
#Concatenation

list1 = [1, 2]
list2 = [3, 4]
combined = list1 + list2
print(combined)
# Output: [1, 2, 3, 4]

#Repetition (*)

letters = ["a", "b"]
print(letters * 3)
# Output: ['a', 'b', 'a', 'b', 'a', 'b']

#Membership Testing

fruits = ["apple", "banana", "cherry"]
print("apple" in fruits)      # Output: True
print("orange" not in fruits) # Output: Tru

#Iterating Through Lists
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num * 2)
# Output:
# 2
# 4
# 6
# 8
# 10
#Nested Lists
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(matrix[0][2])  # Output: 3
print(matrix[2][1])  # Output: 8
```

Basic List Operations

Python supports several basic operations on lists:

Concatenation (+) – Combine two lists:

Repetition (*) – Repeat list elements multiple times:

Membership Testing (in, not in) – Check if an element exists:

Iterating Through Lists

You can use for loops to go through each element in a list, performing operations on them one by one without manually accessing each index.

Nested Lists

Python allows lists to contain other lists, called nested lists. You can access elements inside these inner lists using multiple indices, which is useful for handling multi-dimensional data.

```python
#Creating Tuples

# Single element tuple
single = (5,)
print(single)  # Output: (5,)

# Multiple elements
numbers = (1, 2, 3, 4)
print(numbers)  # Output: (1, 2, 3, 4)

# Nested tuple
nested = (1, 2, (3, 4))
print(nested)  # Output: (1, 2, (3, 4))


#Accessing Tuple Elements

numbers = (10, 20, 30, 40)
print(numbers[1])    # Output: 20
print(numbers[-1])   # Output: 40
print(numbers[1:3])  # Output: (20, 30)


# Concatenation
t1 = (1, 2); t2 = (3, 4)
print(t1 + t2)  # Output: (1, 2, 3, 4)

# Repetition
print(t1 * 2)   # Output: (1, 2, 1, 2)

# Membership testing
print(3 in t2)  # Output: True
```

Introduction to Tuples

Tuples are ordered collections of elements, similar to lists, but immutable, meaning their contents cannot be changed once created. They are useful when you want a sequence of items that should remain constant throughout a program.

- Tuples can be created in different ways:

Accessing Tuple Elements

Indexing: Use positive or negative indices to get a single element.

Slicing: Extract a subset of the tuple using [start:end].

Tuple Operations

Concatenation (+): Combine two tuples.

Repetition (*): Repeat a tuple multiple times.

Membership testing (in, not in): Check if an element exists in a tuple.

```python
# Tuple Functions / Methods

numbers = (10, 20, 30, 20, 40)

# len() - returns number of elements
print(len(numbers))    # Output: 5

# count() - counts occurrences of a value
print(numbers.count(20))  # Output: 2

# index() - returns first index of a value
print(numbers.index(30))  # Output: 2


# Immutability of Tuples

my_tuple = (1, 2, 3)

# Tuples are immutable - trying to change an element will raise an error
# my_tuple[1] = 5  # Uncommenting this line will cause TypeError

# We can access and use elements, but cannot modify them
print(my_tuple[0])    # Output: 1

# We can create a new tuple by concatenation instead
new_tuple = my_tuple + (4, 5)
print(new_tuple)      # Output: (1, 2, 3, 4, 5)
```

Tuple Functions / Methods

Tuples have a few built-in functions that help you work with their data:

len() → Returns the total number of elements in the tuple.

count(value) → Returns the number of times a specific value appears.

index(value) → Returns the index of the first occurrence of a value.

Immutability of Tuples

Tuples are immutable, meaning once created, their elements cannot be changed, added, or removed. This is different from lists and ensures that the data remains constant. Immutability makes tuples safer and faster for storing fixed data, such as coordinates or configuration settings.

```python
# Nested Tuple
nested = (1, 2, (3, 4, 5), 6)

# Access the inner tuple
print(nested[2])        # Output: (3, 4, 5)

# Access an element inside the inner tuple
print(nested[2][1])     # Output: 4
```

Nested tuples

Nested tuples are tuples that contain other tuples as elements. You can access elements in a nested tuple by using multiple indices: the first index selects the inner tuple, and the second index selects the element within it. This allows structured data to be stored and accessed efficiently.

```python
#Dictionaries
student = {"name": "Shaktiman", "age": 20, "course": "Python"}
print(student)
#Output: {'name': 'Shaktiman', 'age': 20, 'course': 'Python'}


# Empty dictionary
my_dict = {}
print(my_dict)  # Output: {}

# Dictionary with elements
person = {"name": "Bob", "age": 25, "city": "Delhi"}
print(person)  # Output: {'name': 'Bob', 'age': 25, 'city': 'Delhi'}

# Nested dictionary
students = {
    "Alice": {"age": 20, "course": "Python"},
    "Bob": {"age": 25, "course": "Java"}
}
print(students)

#{}
#{'name': 'Bob', 'age': 25, 'city': 'Delhi'}
#{'Alice': {'age': 20, 'course': 'Python'}, 'Bob': {'age': 25, 'course': 'Java'}}
```

Introduction to Dictionaries

A dictionary in Python is a collection of key-value pairs, where each key is unique. Dictionaries are unordered, meaning the items are not stored in any particular sequence. They are mutable, allowing you to add, modify, or remove elements. Dictionaries are ideal for storing structured data, such as student records, product details, or configuration settings.

Creating Dictionaries

You can create dictionaries using curly braces { } or the dict() function. They can store empty dictionaries, single/multiple elements, or even nested dictionaries.

```python
#Accessing and Updating Elements
person = {"name": "Bob", "age": 25, "city": "Delhi"}

# Access value
print(person["name"])  # Output: Bob

# Add new element
person["country"] = "India"
print(person)  # Output: {'name': 'Bob', 'age': 25, 'city': 'Delhi', 'country': 'India'}

# Update value
person["age"] = 26
print(person)  # Output: {'name': 'Bob', 'age': 26, 'city': 'Delhi', 'country': 'India'}


#Dictionary Methods

person = {"name": "Bob", "age": 26, "city": "Delhi"}

# keys(), values(), items()
print(person.keys())    # Output: dict_keys(['name', 'age', 'city'])
print(person.values())  # Output: dict_values(['Bob', 26, 'Delhi'])
print(person.items())   # Output: dict_items([('name', 'Bob'), ('age', 26), ('city', 'Delhi')])

# get() method
print(person.get("name"))       # Output: Bob
print(person.get("country", "Not Found"))  # Output: Not Found

# pop() and update()
person.pop("city")
print(person)  # Output: {'name': 'Bob', 'age': 26}
person.update({"city": "Mumbai", "country": "India"})
print(person)  # Output: {'name': 'Bob', 'age': 26, 'city': 'Mumbai', 'country': 'India'}
```

Accessing and Updating Elements

Values in a dictionary can be accessed using their keys. You can also add new key-value pairs or update existing values.

Dictionary Methods

Python dictionaries provide built-in functions to simplify tasks such as retrieving keys, values, or items, and adding/removing elements.

```python
#Iterating Through Dictionaries

person = {"name": "Bob", "age": 26, "city": "Mumbai"}

# Iterate keys
for key in person:
    print(key, person[key])

# Iterate values
for value in person.values():
    print(value)

# Iterate key-value pairs
for key, value in person.items():
    print(key, ":", value)

'''Output
name Bob
age 26
city Mumbai
Bob
26
Mumbai
name : Bob
age : 26
city : Mumbai'''

students = {
    "Alice": {"age": 20, "course": "Python"},
    "Bob": {"age": 25, "course": "Java"}
}

# Access nested value
print(students["Alice"]["course"])  # Output: Python

# Add a nested value
students["Bob"]["grade"] = "A"
print(students)
#Output Python
{'Alice': {'age': 20, 'course': 'Python'}, 'Bob': {'age': 25, 'course': 'Java', 'grade': 'A'}}
```

Iterating Through Dictionaries

You can loop through keys, values, or key-value pairs using a for loop. This is useful when you want to process or display dictionary data.

Nested Dictionaries

Dictionaries can contain other dictionaries, allowing you to model hierarchical data. Access nested values using multiple keys.

```python
# Defining a simple function
def greet():
    print("Hello, welcome to Python!")

# Calling the function
greet()
#Output: Hello, welcome to Python!

#Positional Parameters

def add(a, b):
    print("Sum:", a + b)

add(5, 3)
#output: Sum: 8


#Default Parameters
def greet(name="Guest"):
    print("Hello,", name)

greet("Varad")
greet()
#output: Hello, Varad , #Hello, Guest


#Keyword Parameters

def multiply(a, b):
    print("Product:", a * b)

multiply(b=4, a=3)
#Output:Product: 12

#Return Values

def square(num):
    return num ** 2

result = square(5)
print("Square:", result) #output:Square: 25
```

Functions in Python

Functions are reusable blocks of code that perform a specific task. They help make programs modular, easier to read, and reduce repetition.

Defining Functions
You can define a function using the def keyword followed by the function name and parentheses. The code inside the function is indented.

Function Parameters

Functions can accept parameters to pass information. There are different types:

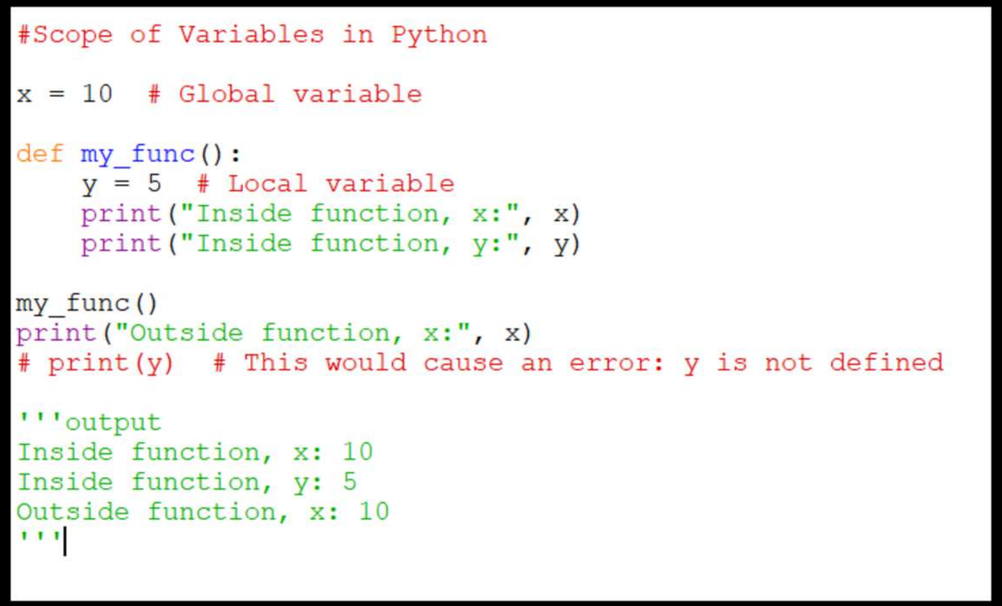
Positional Parameters – Values are passed in order.

Default Parameters – Provide a default value if none is passed.

Keyword Parameters – Pass values using parameter names.

Return Values

Functions can return a value using the return keyword. This allows you to use the result elsewhere in your program.



```python
#Scope of Variables in Python

x = 10  # Global variable

def my_func():
    y = 5  # Local variable
    print("Inside function, x:", x)
    print("Inside function, y:", y)

my_func()
print("Outside function, x:", x)
# print(y)  # This would cause an error: y is not defined

'''output
Inside function, x: 10
Inside function, y: 5
Outside function, x: 10
'''
```

Scope of Variables in Python

In Python, a variable's scope determines where it can be accessed in a program. Local variables are defined inside a function and are only accessible within that function, ensuring they don't interfere with other parts of the program. Global variables, on the other hand, are defined outside functions and can be accessed anywhere in the program, allowing shared data across multiple functions. Understanding variable scope helps prevent unexpected errors and keeps code organized.

Local Variables: Defined inside a function, accessible only within it.

Global Variables: Defined outside all functions, accessible anywhere in the program.

```python
# Writing to a file
file = open("example.txt", "w")
file.write("Hello Python!\n")
file.write("File handling is easy.")
file.close()

# Reading from a file
file = open("example.txt", "r")
print(file.read())
file.close()

# Appending to a file
file = open("example.txt", "a")
file.write("\nAdding more content.")
file.close()

# Safe file handling with 'with'
with open("example.txt", "r") as file:
    print(file.read())
```

File Handling in Python

Python allows you to read from and write to files using built-in functions. You can open a file in different

- 'r' → Read (default)
- 'w' → Write (creates or overwrites a file)
- 'a' → Append (add content to the end)
- 'r+' → Read and write

Writing to a file
with open("example.txt", "w") as f:
  f.write("Hello, Python!")

Note: You can use a text editor or Python's IDLE to create your own .txt files and practice reading/writing using these modes.

```python
# Try to divide 10 by user input; handle invalid or zero input gracefully

try:
    num = int(input("Enter a number: "))
    print("10 divided by your number is:", 10 / num)
except:
    print("Oops! Something went wrong. Make sure you enter a number other than 0.")


# Example: Simulated Payment Checkout with Exception Handlin
print("🛒 Welcome to the Checkout Page")

try:
    amount = float(input("Enter the amount to pay: ₹"))
    card_number = input("Enter your 4-digit card number: ")

    if len(card_number) != 4 or not card_number.isdigit():
        raise ValueError("Invalid card number format.")

    if amount <= 0:
        raise ValueError("Amount must be greater than zero.")

    print(f"☑ Payment of ₹{amount} processed successfully using card ending in {card_number}.")

except ValueError as ve:
    print("❌ Error:", ve)

except Exception as e:
    print("❌ Something went wrong. Please try again.")
```

Exception Handling in Python

Python allows you to handle runtime errors using try and except blocks. This prevents the program from crashing and lets you provide a response or fallback action when an error occurs.

Thank You!

We hope you found these notes helpful and easy to follow for building your Python foundations and learning effectively.