Introduction to Python Programming

Dr. Navaneeth Bhaskar
Dept. of AI&DS
NMAM Institute of Technology, NITTE University

Part -1

Python

- Python is a high-level and general-purpose programming language that is easy to learn and use.
- It was created by Guido van Rossum and first released in 1991.
- Python is widely used for web development, data science, artificial intelligence, machine learning, automation, and game development.

Features of Python Programming

- Easy to learn and use with simple syntax and readability.
- Interpreted language, executing code line by line without compilation.
- Dynamically typed, eliminating the need to specify variable types explicitly.
- Extensive library support for applications like data science, web development, and machine learning.
- Platform independent, running on multiple operating systems without modification.
- Supports multiple programming styles, allowing developers to choose the best approach for their problem.

Building blocks of a Python program

- Variables and Data Types – Store and manipulate data (e.g., int, float, str, list)
- Operators – Perform operations on variables (e.g., +, -, *, /, ==)
- Control Structures – Guide program flow (if-else, loops)
- Functions – Reusable blocks of code (def my_function():)
- Classes and Objects – Support object-oriented programming (class MyClass:)
- Modules and Libraries – Extend functionality (import math)

Variable

- A variable is a label for data in the program. It is a named storage location that holds data, allowing the program to use and manipulate it.
- You assign a value to a variable using the '=' operator.
- Once assigned, you can use the variable to work with that value later.
  Example:

```
# Assigning a value to a variable
age = 25  # 'age' is a variable, and it holds the value 25

# Using the variable
print(age)  # This will print the value stored in the variable 'age' (which is 25)
```

- In this example: age is the variable name and 25 is the value stored in the variable.

Rules for Writing Variable Names

- A variable name must begin with a letter (a-z, A-Z) or an underscore (_)

- After the first character, the name can include letters, digits (0-9), or underscores
- Variable names are case-sensitive
- A variable name cannot be a Python keyword
- Special characters (except underscores) are not allowed
- Variable names cannot contain spaces
- There is no explicit length limit, but names should be meaningful and concise

Data Types in Python

Basic Data Types (Fundamental Data Types)

- int (Integer)
- float (Floating-point)
- str (String)
- bool (Boolean)

Container Data Types (Collection Data Types)

- list (Ordered, mutable collection)
- tuple (Ordered, immutable collection)
- set (Unordered, unique values)
- dict (Key-value pairs)

User-Defined Data Types

- Classes & Objects

Integer (int)

- Used for whole numbers (positive or negative).
- Example: age = 25

Floating Point (float)

- Used for decimal numbers.
- Example: height = 5.9

String (str)

- Used for text data, enclosed in quotes (" " or ' ').
- Example: name = "Edhan"

Boolean (bool)

- Represents True or False values.
- Example: is_student = True

List (list)

- Stores multiple values in one variable.
- Lists are mutable (changeable).
- Example: fruits = ["apple", "banana", "cherry"]

Tuple (tuple)

- Similar to a list but immutable (cannot be changed after creation).

- Example: coordinates = (10, 20)

Set (set)

- Stores multiple unique values in one variable.
- Sets are unordered (the elements do not have a fixed order) and mutable (can be modified).
- Example: fruits = {"apple", "banana", "cherry"}

Dictionary (dict)

- Stores key-value pairs (like a real-world dictionary).
- Example: student = {"name": "Edhan", "age": 25, "course": "Data Science"}

Keywords in Python

- Keywords are reserved words in Python that have special meanings and cannot be used as variable names, function names, or identifiers.
- Keywords are case-sensitive (True is valid, but true is not).
- They must be used only for their intended purpose in Python syntax.
- Cannot be used as variable names.

List of Python Keywords:

```
False      def        if         raise
None       del        import     return
True       elif       in         try
and        else       is         while
as         except     lambda     with
assert     finally    nonlocal   yield
break      for        not
class      from       or
continue   global     pass
```

Identifiers

- An identifier is a name used to identify a variable, function, class, or other objects in Python.
- It is essentially the label you give to a piece of data or a function to reference it in your program.

Rules for Identifiers:

- An identifier must begin with a letter (a-z, A-Z) or an underscore (_).
- After the first character, the identifier can include letters, digits (0-9), and underscores.
- Identifiers are case-sensitive, meaning Variable, variable, and VARIABLE are considered different.
- Identifiers cannot be Python keywords (reserved words like if, else, True, etc.).
- Special characters (except underscores) are not allowed.
- Identifiers cannot contain spaces.

Operators

Operators are symbols that perform operations on variables or values. Python supports several types of operators, each serving a different purpose.

Arithmetic Operators

These operators are used to perform basic mathematical operations.

- + (Addition): Adds two operands.

- - (Subtraction): Subtracts the right operand from the left operand.
- * (Multiplication): Multiplies two operands.
- / (Division): Divides the left operand by the right operand (returns float).
- // (Floor Division): Divides the left operand by the right operand and returns the integer part of the result.
- % (Modulo): Returns the remainder of the division.
- ** (Exponentiation): Raises the left operand to the power of the right operand.

Arithmetic Operations with a = 10 and b = 3

Operator	Expression	Result	Explanation
`+` (Addition)	`a + b`	`10 + 3 = 13`	Adds `a` and `b`.
`-` (Subtraction)	`a - b`	`10 - 3 = 7`	Subtracts `b` from `a`.
`*` (Multiplication)	`a * b`	`10 * 3 = 30`	Multiplies `a` and `b`.
`/` (Division)	`a / b`	`10 / 3 = 3.3333`	Divides `a` by `b`, returns a float.
`//` (Floor Division)	`a // b`	`10 // 3 = 3`	Returns only the integer part of the division.
`%` (Modulo)	`a % b`	`10 % 3 = 1`	Returns the remainder of `10 ÷ 3`.
`**` (Exponentiation)	`a ** b`	`10 ** 3 = 1000`	Raises `a` to the power of `b` (10^3).

Comparison (Relational) Operators

These operators are used to compare two values and return a Boolean result (True or False).

- == (Equal to): Checks if two values are equal.
- != (Not equal to): Checks if two values are not equal.
    - (Greater than): Checks if the left value is greater than the right value.
- < (Less than): Checks if the left value is less than the right value.
- >= (Greater than or equal to): Checks if the left value is greater than or equal to the right value.
- <= (Less than or equal to): Checks if the left value is less than or equal to the right value.

Comparison Operators with a = 10 and b = 3

Operator	Expression	Result	Explanation
`==` (Equal to)	`a == b`	`False`	`10` is **not equal** to `3`.
`!=` (Not equal to)	`a != b`	`True`	`10` is **not equal** to `3`.
`>` (Greater than)	`a > b`	`True`	`10` is **greater than** `3`.
`<` (Less than)	`a < b`	`False`	`10` is **not less than** `3`.
`>=` (Greater than or equal to)	`a >= b`	`True`	`10` is **greater than or equal to** `3`.
`<=` (Less than or equal to)	`a <= b`	`False`	`10` is **not less than or equal to** `3`.

Logical Operators

Logical operators are used to combine conditional statements.

- and: Returns True if both operands are True.
- or: Returns True if at least one operand is True.
- not: Reverses the Boolean value (True becomes False, and False becomes True).

Logical Operators with a = 10 and b = 3

Operator	Expression	Result	Explanation
`and` (Logical AND)	`(a > 5) and (b < 5)`	`True`	Both conditions are **True**, so the result is **True**.
`and`	`(a > 5) and (b > 5)`	`False`	One condition is **False**, so the result is **False**.
`or` (Logical OR)	`(a > 5) or (b > 5)`	`True`	One condition is **True**, so the result is **True**.
`or`	`(a < 5) or (b > 5)`	`False`	Both conditions are **False**, so the result is **False**.
`not` (Logical NOT)	`not (a > 5)`	`False`	`a > 5` is **True**, but `not` reverses it to **False**.
`not`	`not (b > 5)`	`True`	`b > 5` is **False**, but `not` reverses it to **True**.

Assignment Operators

These operators are used to assign values to variables.

- =: Assigns the value of the right operand to the left operand.
- +=: Adds the right operand to the left operand and assigns the result to the left operand.
- -=: Subtracts the right operand from the left operand and assigns the result to the left operand.
- *=: Multiplies the left operand by the right operand and assigns the result to the left operand.
- /=: Divides the left operand by the right operand and assigns the result to the left operand.
- %=: Calculates the modulo and assigns the result to the left operand.
- //=: Performs floor division and assigns the result to the left operand.
- **=: Performs exponentiation and assigns the result to the left operand.

Assignment Operators with a = 10 and b = 3

Operator	Expression	Equivalent To	Result	Explanation
`=` (Assignment)	`a = b`	`a = 3`	`a = 3`	Assigns `b` 's value to `a` .
`+=` (Add & Assign)	`a += b`	`a = a + b`	`13`	`10 + 3 = 13` , then stores in `a` .
`-=` (Subtract & Assign)	`a -= b`	`a = a - b`	`7`	`10 - 3 = 7` , then stores in `a` .
`*=` (Multiply & Assign)	`a *= b`	`a = a * b`	`30`	`10 * 3 = 30` , then stores in `a` .
`/=` (Divide & Assign)	`a /= b`	`a = a / b`	`3.3333`	`10 / 3 = 3.3333` , stores float in `a` .
`%=` (Modulo & Assign)	`a %= b`	`a = a % b`	`1`	`10 % 3 = 1` , stores remainder in `a` .
`//=` (Floor Divide & Assign)	`a //= b`	`a = a // b`	`3`	`10 // 3 = 3` , stores integer part in `a` .
`**=` (Exponentiate & Assign)	`a **= b`	`a = a ** b`	`1000`	`10 ** 3 = 1000` , stores result in `a` .

Bitwise Operators

Bitwise operators are used to perform operations on binary representations of integers.

- & (AND): Performs a bitwise AND operation.
- | (OR): Performs a bitwise OR operation.
- ^ (XOR): Performs a bitwise XOR operation.
- ~ (NOT): Performs a bitwise NOT operation.
- << (Left shift): Shifts the bits to the left by the specified number of positions.
- >> (Right shift): Shifts the bits to the right by the specified number of positions.

Membership Operators

Membership operators in Python are used to check whether a value exists in a sequence (such as a string, list, tuple, or dictionary).

- in: Returns True if the specified value is present in the sequence.
- not in: Returns True if the specified value is not present in the sequence.

Example:

```
my_list = [1, 2, 3, 4, 5]

# Using 'in'
print(3 in my_list)   # Output: True

# Using 'not in'
print(10 not in my_list)   # Output: True
```

Identity Operators

Identity operators are used to compare the memory locations of two objects. They check whether two variables reference the same object in memory.

- is → Returns True if both variables reference the same object.
- is not → Returns True if both variables reference different objects.

Modules in Python

- A module is a file containing Python code (functions, classes, variables) that can be imported and reused in other programs.
- These modules help in mathematical calculations, random number generation, and precise arithmetic operations.
- Every Python file (.py) is considered a module. When we run a Python program, its module name is set to __main__.

Use Multiple Modules

There are two main reasons for breaking a program into multiple modules:

- Managing Large Codebases - If a program is too big, dividing it into multiple .py files makes development and maintenance easier.
- Code Reusability - Instead of copying the same functions into different programs, we can write them once in a module and import them wherever needed.

Importing a Module in Python

- To use a module's functions in another file, we use the import statement.

Example: Creating a Module (functions.py)

```
# functions.py
def display():
    print("Earlier, the rich owned cars while the poor had horses.")

def show():
    print("Now, everyone has cars, and only the rich own horses.")
```

Using the Module:

```
import functions  # Importing the 'functions' module

functions.display()
functions.show()
```

Different Ways to Import Modules:

Importing Specific Functions

- Instead of importing the entire module, we can import only specific functions.

```
from math import sin, cos, tan
from functions import display  # Imports only the 'display' function

display()  # No need to use functions.display()
```

Importing Everything

- We can use * to import all functions from a module.

```
from functions import *  # Imports all functions

display()
show()
```

Renaming a Module

- We can rename a module while importing it.

```
import functions as fun  # Using 'fun' instead of 'functions'

fun.display()
```

Examples for Common Built-in Modules in Python

math Module

- The math module provides mathematical functions like square root, factorial, logarithms, and trigonometric functions.
- Mathematical functions in math module:

Function	Description
`math.pi`	Value of π (3.14159...)
`math.e`	Value of e (2.718...)
`math.sqrt(x)`	Square root of `x`
`math.factorial(x)`	Factorial of `x`
`math.fabs(x)`	Absolute value of `x`
`math.log(x)`	Natural logarithm (base e) of `x`
`math.log10(x)`	Base-10 logarithm of `x`
`math.exp(x)`	e raised to the power of `x`
`math.trunc(x)`	Removes the decimal part of `x`
`math.ceil(x)`	Rounds `x` **up** to the nearest integer
`math.floor(x)`	Rounds `x` **down** to the nearest integer

Trigonometric functions in math module:

Function	Description
`math.sin(x)`	Returns the **sine** of `x` (in radians)
`math.cos(x)`	Returns the **cosine** of `x` (in radians)
`math.tan(x)`	Returns the **tangent** of `x` (in radians)
`math.asin(x)`	Returns the **inverse sine** (arcsin) of `x` (in radians)
`math.acos(x)`	Returns the **inverse cosine** (arccos) of `x` (in radians)
`math.atan(x)`	Returns the **inverse tangent** (arctan) of `x` (in radians)
`math.sinh(x)`	Returns the **hyperbolic sine** of `x`
`math.cosh(x)`	Returns the **hyperbolic cosine** of `x`
`math.tanh(x)`	Returns the **hyperbolic tangent** of `x`
`math.degrees(x)`	Converts **radians to degrees**
`math.radians(x)`	Converts **degrees to radians**
`math.hypot(x, y)`	Returns the **Euclidean distance** `sqrt(x² + y²)`

cmath Module

- The cmath module provides functions for complex numbers.

random Module

- The random module helps generate random numbers.

Function	Description
`random.random()`	Returns a random number **between 0 and 1**
`random.randint(a, b)`	Returns a random integer **between a and b**
`random.seed(x)`	Sets the seed value for random number generation
`random.choice(sequence)`	Returns a **random element** from a list or string
`random.shuffle(sequence)`	Randomly **shuffles elements** in a list

decimal Module

- The decimal module is used for precise arithmetic calculations, especially when floating-point precision is important.

Comments and Indentation in Python

Single-line Comments:

- In Python, comments are written using the # symbol. Any text following the # will be ignored by the Python interpreter.

Multi-line Comments:

- For multi-line comments, you can use triple single quotes (''') or triple double quotes (""").

Indentation:

- Indentation refers to the spaces or tabs used at the beginning of a line to define the structure of the code. Unlike many other languages (like C, C++, or Java) that use braces {} to define code blocks, Python relies on indentation.

Multi-line Statements:

- If an expression or statement is too long, it can be split across multiple lines by using the backslash (\) at the end of each line except the last one. This tells Python to treat the next line as part of the current statement.

Python Programming Modes

- Python can be used in two modes:
  - Interactive Mode- Used for exploring Python syntax, seeking help, and debugging short programs.
  - Script Mode- Used for writing full-fledged Python programs.
- Both modes are supported by IDLE (Integrated Development and Learning Environment).

Using IDLE in Interactive Mode

- Open IDLE
  - In Windows, search for IDLE in the search bar and press Enter.
  - Or, double-click the IDLE icon.
- The Python shell window will open, displaying the >>> prompt.

Using IDLE in Script Mode

- Open IDLE
  - In the IDLE shell window, go to File → New File.
  - A new script editor window will open.
  - Type the script and save.
  - Run the script by selecting Run → Run Module (F5).
- Interactive Mode is for quick testing.
- Script Mode is for writing full programs.
- IDLE supports both modes for efficient coding.

Literals in Python

- Literals are fixed values used directly in Python code, representing different data types like strings, numbers, and booleans.
- String literals are enclosed in single or double quotes, such as "Hello, World!" or 'Python is fun'.
- Numeric literals include integers like 35, floating-point numbers like 3.14159, and complex numbers like 2 + 3j.
- Boolean literals represent truth values and can be either True or False, such as in a condition like is_active = True.
- Collection literals include data structures like lists ["apple", "banana"], tuples (10, 20), sets {1, 2, 3}, and dictionaries {"name": "Bob", "age": 35}.
- Literals are immutable values that Python automatically identifies based on syntax, making them essential for defining constants and structured data.

Precedence and Associativity in Python

- Precedence determines the order in which operators are evaluated in an expression.
- Operators with higher precedence are executed before those with lower precedence.

Operator Precedence (PEMDAS Rule)

Operator	Description	Associativity
`()`	Parentheses	**Highest**
`**`	Exponentiation	Right to Left
`*, /, //, %`	Multiplication, Division, Floor Division, Modulus	Left to Right
`+, -`	Addition, Subtraction	Left to Right

- Associativity determines the order of execution when multiple operators have the same precedence in an expression.
- Most arithmetic operators (+, -, *, /, //, %) follow left-to-right associativity.

Conversions (Mixed-Mode Operations)

- When performing operations between different types of numbers, Python follows these rules:

Operation	Result Type	Example	Output
`int` + `float`	**float**	`5 + 2.5`	`7.5`
`int` + `complex`	**complex**	`4 + (2+3j)`	`(6+3j)`
`float` + `complex`	**complex**	`2.5 + (3+4j)`	`(5.5+4j)`

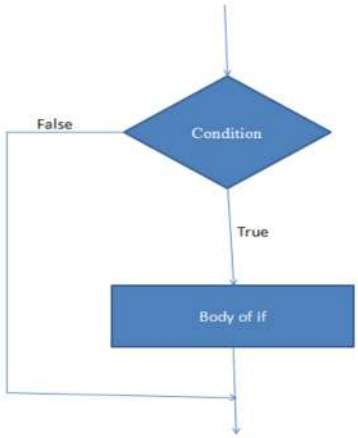
- Python provides built-in functions to convert one data type into another:

Function	What it does?	Example	Output
`int(x)`	Converts `float` or `string` to `int` (removes decimal)	`int(3.99)`	`3`
`float(x)`	Converts `int` or `string` to `float`	`float(7)`	`7.0`
`complex(x)`	Converts `int/float` to `complex` (imaginary part = 0)	`complex(3)`	`(3+0j)`
`complex(x, y)`	Creates a complex number `x + yj`	`complex(4, 5)`	`(4+5j)`
`bool(x)`	Converts `int/float` to `True` (non-zero) or `False` (zero)	`bool(0)`	`False`
`str(x)`	Converts `int/float/bool` to a string	`str(123)`	`'123'`
`chr(x)`	Converts an integer to its corresponding **character** (ASCII)	`chr(65)`	`'A'`

Control Statements

if

- Executes a block of code only if the condition is true.
- Used for simple decision-making.



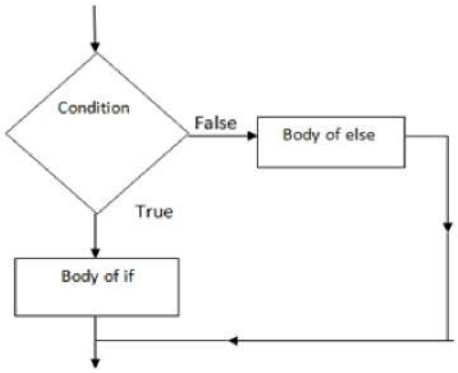

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

Output: You are eligible to vote.

if-else

- Provides an alternative action when the condition is false.
- Executes the if block if the condition is true, otherwise the else block runs.

```
age = 16
if age >= 18:
    print("You are eligible to vote.")
else:
    print("You are not eligible to vote yet.")
```



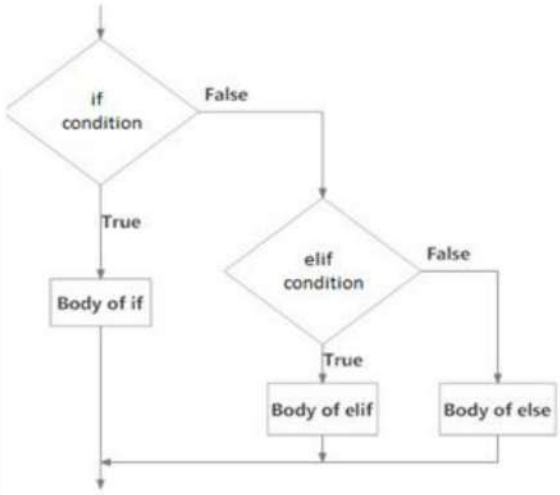
Output: You are not eligible to vote yet.

elif (Else-If)

- Checks multiple conditions one after another.
- Stops checking as soon as a true condition is found.

```
marks = 85

if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
else:
    print("Grade: C")
```



Output: Grade: B

Nested if

- An if statement inside another if statement.
- Useful for making complex decisions with multiple conditions.

```
age = 20
registered = True

if age >= 18:
    if registered:
        print("You can vote.")
    else:
        print("Please register to vote.")
else:
    print("You are not eligible to vote yet.")
```

Output: You can vote.

for Loop

- Repeats a block of code a specific number of times.
- Commonly used to iterate over a range or collection.

```
for i in range(1, 6):
    print("Hello person", i)
```

Output:
Hello person 1
Hello person 2
Hello person 3
Hello person 4
Hello person 5

while Loop

- Repeats a block of code as long as the condition is true.
- Suitable for situations where the number of iterations is not known in advance.

```
count = 1
while count <= 3:
    print("Counting:", count)
    count += 1
```

Output:
Counting: 1
Counting: 2
Counting: 3

do-while Loop (Simulated in Python)

- Python does not have a built-in do-while loop.
- However, you can mimic the behavior of a do-while loop using a while loop by ensuring that the loop body is executed at least once before checking the condition.

```
count = 1  # Initialize count variable with value 1

while True:  # Infinite loop (runs indefinitely)
    print("Count:", count)  # Print the current value of count
    count += 1  # Increment count by 1

    if count > 3:  # Check if count is greater than 3
        break  # Exit the loop when count exceeds 3
```

Output:
Count: 1
Count: 2
Count: 3

Break

- Exits the loop immediately, even if the loop condition is still true.
- Used to stop the loop when a specific condition is met.

```python
for i in range(1, 6):
    if i == 3:
        break
    print(i)
```

Output:

1
2

Continue

- Skips the current iteration and moves to the next one.
- Used to skip specific conditions without stopping the loop.

```python
for i in range(1, 6):
    if i == 3:
        continue
    print(i)
```

Output:

1
2
4
5

Pass

- Does nothing, acts as a placeholder for future code.
- Has no effect on the loop's execution.

```python
for i in range(3):
    if i == 1:
        pass  # Placeholder for future code
    print(i)
```

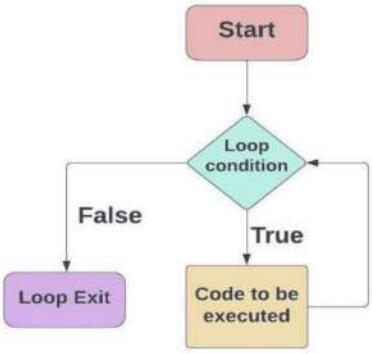
Output:

0
1
2

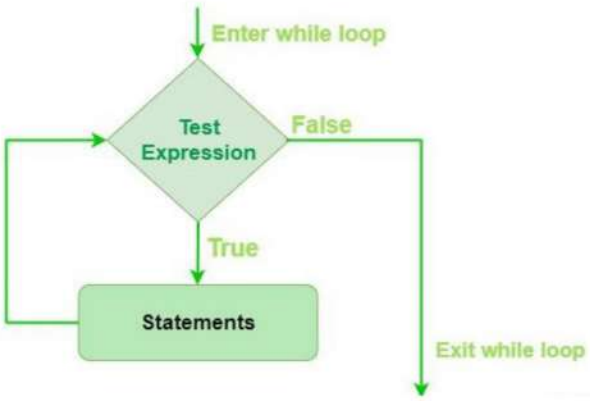
Control Statement	Purpose	Usage
`if`	Executes if condition is true	Simple conditions
`if-else`	Executes alternative code	Decision-making
`elif`	Multiple conditions	Grading, category checks
Nested `if`	Complex conditions	Multiple dependent conditions
`for` loop	Fixed number of iterations	Counting, iterating over lists
`while` loop	Repeats while condition is true	Unknown iteration limits
`do-while` (simulated)	Executes at least once	Menus, login attempts
`break`	Exits loop early	Stop when condition is met
`continue`	Skip current iteration	Skip specific values
`pass`	Placeholder for future code	Code stubs, empty blocks

Looping Statements in Python

- Looping statements are used to execute a block of code multiple times until a condition is met.
- The two types of loops in Python are:
    - **for loop** → Used for iterating over sequences (lists, tuples, strings, ranges).



    - **while loop** → Executes as long as a condition is True.



Parameters /Types of Parameters

Parameters (Formal Parameters)

- These are the parameters defined in the function definition.
- They act as placeholders for values that will be passed during function calls.

Arguments (Actual Parameters)

- Arguments are values passed to a function when calling it.
- They replace the formal parameters during execution.

```
# Function definition with a formal parameter 'x'
def square(x):
    return x * x  # Formal parameter 'x' is used inside the function

# Function call with an actual parameter 5
result = square(5)  # Here, 5 is the actual parameter

print("Square:", result)  # Output: Square: 25
```

Types of Arguments (Actual Parameters)

Positional Arguments

- These are the most basic type of arguments. The values you pass to the function must be in the same order as the parameters in the function definition.

```
def fun(i, j, k):
    print(i + j)
    print(k.upper())

fun(10, 3.14, 'Rigmarole')  # Correct, arguments in the right order
```

Keyword Arguments

- These arguments are passed by specifying the parameter names (keywords) in the function call.
- The order doesn't matter because Python uses the parameter names to match values.

```
def print_it(i, a, str):
    print(i, a, str)

print_it(i=10, a=3.14, str='Sicilian')  # Correct, order doesn't matter
```

Variable-length Positional Arguments (*args)

- These allow you to pass a variable number of positional arguments to a function. Inside the function, args will be treated as a tuple containing all the passed values.

```
def print_it(*args):
    for var in args:
        print(var)

print_it(10, 3.14, 'Sicilian')  # You can pass any number of arguments
```

Variable-length Keyword Arguments (kwargs)**

- These allow you to pass a variable number of keyword arguments to a function.
- Inside the function, kwargs will be treated as a dictionary where the keys are the argument names and the values are the corresponding values.

```
def print_it(**kwargs):
    for name, value in kwargs.items():
        print(name, value)

print_it(a=10, b=3.14, s='Sicilian')  # You can pass any number of keyword arguments
```

Functions in Python

- A function is a block of reusable code that performs a specific task.
- Types of Functions in Python
    - Built-in Functions – Predefined functions in Python (e.g., print(), len(), abs()).
    - User-defined Functions – Functions created by programmers using the def keyword.
- Common Built-in Functions for Numbers

Function	Description
`abs(x)`	Returns the absolute value of `x`
`pow(x, y)`	Returns `x` raised to the power of `y`
`min(x1, x2, ...)`	Returns the smallest value
`max(x1, x2, ...)`	Returns the largest value

`divmod(x, y)`	Returns `(x // y, x % y)` as a tuple
`round(x, n)`	Rounds `x` to `n` decimal places
`bin(x)`	Returns the binary representation of `x`
`oct(x)`	Returns the octal representation of `x`
`hex(x)`	Returns the hexadecimal representation of `x`

Recursive Function

- A recursive function is a function that calls itself inside its own body. This helps in solving problems that can be broken into smaller subproblems of the same type.
- Factorial using Recursion
  Factorial of a number n is: $n!=n \times (n-1) \times (n-2) \times ... \times 1$

```
def factorial(n):
    if n == 1:  # Base case
        return 1
    else:  # Recursive case
        return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

Lambda Functions

- Lambda functions are anonymous functions (functions without a name).
- They are created using the lambda keyword.
- They are typically used for short, simple operations

Syntax:

```
lambda arguments: expression
```

Example:

```
# Lambda function to calculate the average of three numbers
average = lambda x, y, z: (x + y + z) / 3
print(average(10, 20, 30))  # Output: 20.0
```

Program: Write a python program to find the factorial of a number using recursion.

```
def recur_factorial(n):
    if n == 1:  # Base case: Stop when n is 1
        return n
    else:  # Recursive case: Multiply n by factorial of (n-1)
        return n * recur_factorial(n - 1)

num = int(input("Enter the number: "))  # User input

if num < 0:  # Check for negative input
    print("Factorial does not exist for negative numbers")
elif num == 0:  # Special case: Factorial of 0 is 1
    print("The factorial of 0 is 1")
else:  # Compute factorial for positive numbers
    print("The factorial of", num, "is", recur_factorial(num))
```

Program: Write a python program to check whether the number is palindrome or not.

```
n=int(input("Enter number:"))  # User input
temp=n  # Store original number
rev=0  # Variable to store reversed number

while n>0:
    dig=n%10  # Get last digit
    rev=rev*10+dig  # Build reversed number
    n=n//10  # Remove last digit

if temp==rev:
    print("The number is a palindrome!")
else:
    print("The number isn't a palindrome!")
```

Program: Write a Python program to check whether a given number is odd or even.

(An even number is divisible by 2, while an odd number is not)

```
# Taking input from the user
num = int(input("Enter a number: "))

# Checking if the number is even or odd
if num % 2 == 0:
    print(f"{num} is an even number.")
else:
    print(f"{num} is an odd number.")
```

Program: Write a python program to check if a number is prime.

(A prime number is a number greater than 1 that is divisible only by 1 and itself)

```
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n):  # Checking all numbers till n-1
        if n % i == 0:
            return False
    return True

# Taking input from the user
num = int(input("Enter a number: "))

# Checking if the number is prime
if is_prime(num):
    print(f"{num} is a prime number.")
else:
    print(f"{num} is not a prime number.")
```

Program: Write a python program to print all prime numbers in an interval.

```
start=int(input("Enter starting range:"))
end=int(input("Enter ending range:"))
c=0  # Counter for prime numbers

print("Prime numbers between",start,"and",end,"are:")

for num in range(start,end+1):
    if num>1:  # Prime numbers are greater than 1
        for i in range(2,num):
            if num%i==0:
                break
        else:
            c+=1
            print(num)

if c==0:
    print("There are no prime numbers in the range.")
```