Python Programming

CT108-3-1-PYP

0003 - Introduction to Programming With Python

Topic Learning Outcomes

At the end of this topic, you should be able to:

- Understand the basic structure and syntax of Python Programming Language
- Write your own simple Python scripts

Contents & Structure

- Introduction to Python
- Python as a Programming Language
- Interactive vs Script Programming

What is a Programming Language?

A **programming language** is a formal set of rules and syntax that allows humans to communicate instructions to a computer. These instructions tell the computer what operations to perform.

In essence, a programming language provides a way to express algorithms (a sequence of steps to solve a problem) in a form that both humans and machines can understand.

1. Characteristics of Programming Languages

1A. Set of Rules for Operations

- Programming languages define a set of rules (syntax and semantics) that programmers must follow to tell a computer what operations to perform. For example, in Python, the rule for defining a function starts with the def keyword.

1B. Communication of Algorithms

- Programming languages allow us to write algorithms in a structured and clear manner. For example, an algorithm to add two numbers can be written in Python as:
    - def add_numbers(a, b):
    - return a + b

1C. Linguistic Framework

Programming languages provide a framework for describing computations. This framework includes data types, control structures, and functions, which allow developers to express complex computational logic.

1. Characteristics of Programming Languages

1D. Notation for Computation

- Programming languages use a specific notation that is machine-readable (so it can be executed by a computer) and human-readable (so programmers can understand and modify it). For instance, the equation x = y + z is both readable by a human and can be executed by a machine when written in a language like Python.

1E. Tool for Developing Executable Models

- Programming languages are tools that developers use to create software that models real-world problems. For example, a program that manages a bank account can be developed using a language like Java.

2. Examples of Influential Programming Languages

2A. FORTRAN (Formula Translation):

- **Domain**: Science and engineering.
- **Example**: Used for numerical and scientific computation, such as simulating physical systems or solving differential equations.
- **Example Code**:

```
PROGRAM ADDITION
   REAL :: A, B, SUM
   A = 3.0
   B = 5.0
   SUM = A + B
   PRINT *, 'Sum = ', SUM
END PROGRAM ADDITION
```

2. Examples of Influential Programming Languages

2B. COBOL (Common Business-Oriented Language):

- **Domain**: Business data processing.
- **Example**: Used for managing large-scale transaction processing systems like banking software.
- **Example Code**:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ADDITION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A PIC 9(3) VALUE 3.
01 B PIC 9(3) VALUE 5.
01 SUM PIC 9(4).
PROCEDURE DIVISION.
ADD A TO B GIVING SUM.
DISPLAY 'Sum = ' SUM.
STOP RUN.
```

2C. LISP (LISt Processing):

- **Domain**: Logic and Artificial Intelligence(AI).
- **Example**: Used in AI research, such as natural language processing or symbolic reasoning.
- **Example Code**:

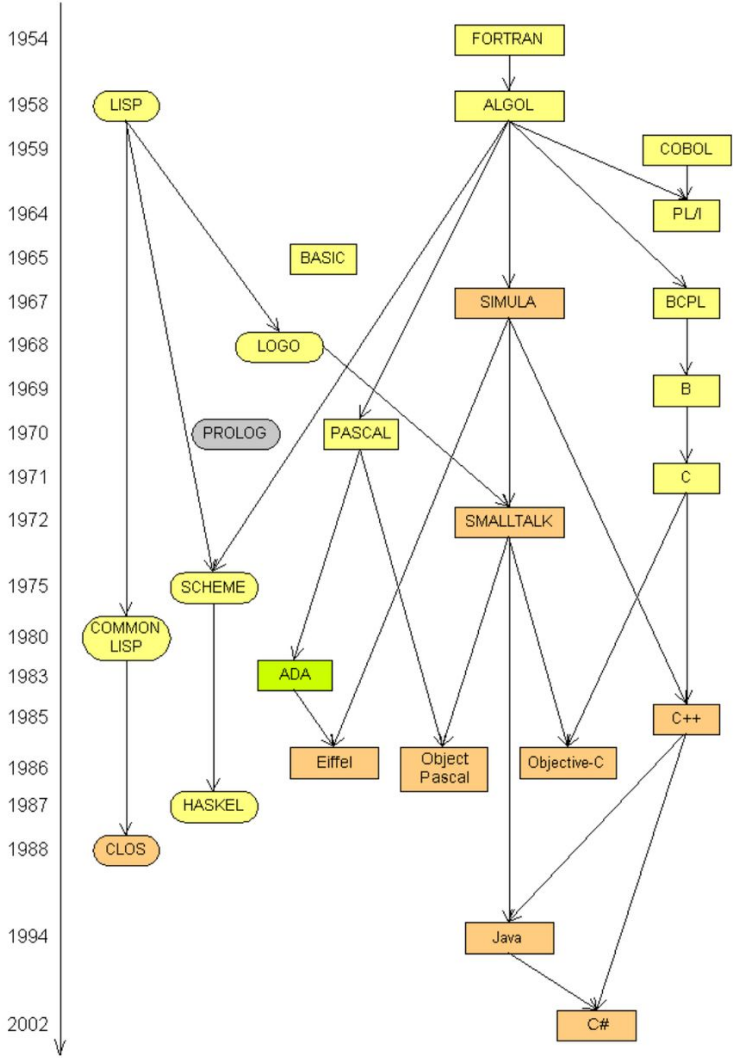
```
(defun add-numbers (a b)
  (+ a b))
(print (add-numbers 3 5))
```

2D. BASIC (Beginner's All-purpose Symbolic Instruction Code):

- **Domain**: Simple, easy-to-learn language for beginners.
- **Example**: Often used for educational purposes or simple automation tasks.
- **Example Code**:

```
10 A = 3
20 B = 5
30 SUM = A + B
40 PRINT "Sum = "; SUM
```

3. History Of High-Level Programming Languages Evolution Form 1954 – 2002



4. Common Instructions in All Programming Languages

4A. Input

- Getting data from the user.
- **Example**: In Python:

```
name = input("Enter your name: ")
```

4B. Output

- Displaying data to the user on a monitor or printout.
- **Example**: In Python

```
print("Hello, " + name)
```

4C. Math

- Performing basic mathematical operations.
- **Example**: In Python

```
result = 3 + 5
```

4D. Conditional Statements

- Checking conditions and executing code based on those conditions.
- **Example**: In Python:

```
if result > 5:
    print("Result is greater than 5")
```

4E. Repetition (Loops)

- Executing a sequence of statements multiple times.
- **Example**: In Python

```
for i in range(5):
    print(i)
```

Summary

Programming languages are powerful tools that allow us to communicate complex instructions to computers. They provide the structure, rules, and syntax needed to express algorithms in a way that can be executed by machines and understood by humans. Each language has its own domain of influence, such as FORTRAN for scientific computation or COBOL for business data processing, and they all share some basic types of instructions like input, output, and loops.

Users vs. Programmers

1. Users

- A **user** is anyone who interacts with a computer, smartphone, or any electronic device to perform tasks. Users typically focus on using applications to accomplish specific goals, such as writing documents, browsing the internet, or managing schedules.
- They don't need to understand the underlying technology that makes the applications work—they just need to know how to operate them effectively.

1A. Example

- Ali is an educated person but is not familiar with the technical aspects of computers. He uses a word processor to write documents, a spreadsheet to manage his finances, and a map application to find directions. Ali views these applications as tools that help him with daily tasks.

2. Programmers

- A **programmer**, on the other hand, is someone who understands the technological inner workings of computers. Programmers write code, which is a series of instructions that tell a computer how to perform specific tasks.
- They have the skills to create new tools (software) by writing programs that can automate tasks, solve problems, or perform complex computations. Programmers understand the "ways" of computers and the languages used to communicate with them.

2B. Example

- Jane is a programmer who writes code in various programming languages. She develops applications, automates repetitive tasks, and creates software tools that others can use. Jane not only uses the tools available on a computer but also has the ability to create new tools that enhance or extend the computer's capabilities.

3. Which Are You?

- This depends on your current skill set and interests. If you primarily use software tools without much knowledge of how they work behind the scenes, you are a user. If you are interested in understanding how these tools work and want to create or modify them, you might be on the path to becoming a programmer.

4. Which Do You Want To Be?

- If you want to gain the skills to create software, automate tasks, and understand the inner workings of computers, you may aspire to be a programmer. If you prefer to focus on using existing tools without delving into their technical aspects, being a user may be more suited to you.

5. Do You Ever Wonder How It Works?

- Curiosity about how software and devices work is often the first step toward becoming a programmer. Understanding how technology operates can lead to deeper exploration and eventually to learning programming skills.

What is a Program/Code/Software?

1. Program

A **program** is a set of instructions that a computer follows to perform a specific task or solve a problem. Computers cannot perform tasks on their own; they rely on these instructions to know what to do. A program is essentially a way to encode human intelligence into a form that a computer can execute.

Example:

Imagine you want your computer to calculate the total cost of items in a shopping cart. You would write a program with instructions like:

- **Input** the price of each item.
- **Sum** all the prices.
- **Output** the total cost.

A simple program in Python might look like this:

```
# Program to calculate total cost
item1 = 10.50  # price of item 1
item2 = 5.25   # price of item 2
item3 = 8.75   # price of item 3

total_cost = item1 + item2 + item3  # summing the prices
print("The total cost is:", total_cost)  # displaying the total cost
```

When this program runs, the computer will follow these instructions step by step and output something like:

```
The total cost is: 24.5
```

2. Code

Code refers to the actual text written by a programmer that makes up a program. Code is written using a specific programming language, such as Python, Java, or C++. Each programming language has its own syntax and rules for writing code. The actual instructions written in a programming language.

Example:

In the Python example above, the lines:

```
item1 = 10.50
item2 = 5.25
item3 = 8.75
total_cost = item1 + item2 + item3
print("The total cost is:", total_cost)
```

are all part of the code that instructs the computer on what to do.

- **Syntax:** The = sign is used to assign values to variables (e.g., item1 = 10.50).
- **Command:** The print() function is a command that tells the computer to display something on the screen.

If you were to write this in a different language, like JavaScript, the code might look a bit different, but the concept would be the same.

Example: JavaScript

```
// Program to calculate total cost in JavaScript
var item1 = 10.50;
var item2 = 5.25;
var item3 = 8.75;

var total_cost = item1 + item2 + item3;
console.log("The total cost is: " + total_cost);
```

3. Software

- **Software** is a broader term that encompasses one or more programs along with the data and documentation needed to use them effectively. Software is what makes the hardware (physical parts of a computer) useful. It can range from small applications to massive systems running complex operations.
- Software is a collection of programs that work together to perform a task or set of tasks. For example, a word processor, a web browser, or an operating system are all types of software.

Example:

Consider Microsoft Word, which is a word processing software. Microsoft Word is made up of multiple programs that work together:

- **Spell Checker Program:** A program that checks the spelling and grammar of the text.
- **Formatting Program:** A program that applies styles, fonts, and layouts to your document.
- **Save/Load Program:** A program that allows you to save your work to disk or load a previously saved document.

These individual programs are part of the larger software package, Microsoft Word, which allows you to create, edit, and manage text documents.

Another Example: Imagine you're using a navigation app on your smartphone, like Google Maps. This software consists of many programs:

- **Map Rendering Program:** This draws the map on your screen.
- **GPS Program:** This communicates with satellites to determine your location.
- **Routing Program:** This calculates the best route to your destination.
- **Traffic Data Program:** This checks real-time traffic information to update your route.

All these programs work together to give you a seamless experience when using the app, and the entire package is what we refer to as **software**.

Conclusion

- **Programs** are the sets of instructions that tell the computer what to do.
- **Code** is the written form of those instructions in a programming language.
- **Software** is the complete package of programs, data, and documentation that allows users to perform tasks on a computer.

Understanding these concepts is crucial for anyone interested in programming or developing software, as they form the foundation of how we interact with and control computers.

Programming Basics

1. Code/Source Code

- This refers to the sequence of instructions that make up a program. For example, if you write a program to calculate the sum of two numbers, the code would include the steps needed to receive the input, perform the calculation, and display the result.

2. Syntax

- Every programming language has a set of rules called syntax that defines how code must be written. Syntax determines how to structure commands, variables, loops, and other elements so that the computer can understand and execute the program. Incorrect syntax leads to errors and prevents the program from running.

2A. Example

- In Python, the syntax for printing text to the console is print("Hello, World!"). If you miss a quotation mark or parentheses, the program will not run and will generate a syntax error.

3. Output

- This is the result produced by a program after it executes the code. The output can be displayed on the screen, saved to a file, or sent to another program.

3A. Example

- If you run a program that adds two numbers and prints the result, the output might be The sum is 10.

4. Console

- The console is a text-based interface that allows you to interact with the computer by typing commands and receiving text output. When you run a program, the console is where you typically see the output. Some development environments have an integrated console, while others open a separate window.

4A. Example

- When you run a Python script in an IDE (Integrated Development Environment) like PyCharm, the output might appear in the console section of the IDE.

These concepts form the foundation of programming, and understanding them is crucial for anyone interested in becoming a programmer.

Learning Python

1. Learning a Programming Language

Learning a programming language is similar to learning a new human language. Just as with English, where you start by learning the alphabet, then progress to words, sentences, and grammar, learning a programming language follows a similar structure:

- **Symbols**: In programming, symbols include the characters and special signs that form the building blocks of code, such as +, -, *, /, = in Python.
- **Variables**: These are named locations in memory used to store data that can be changed during program execution, such as x = 10.
- **Constants**: Constants are similar to variables but their values do not change, such as PI = 3.14.
- **Statements**: Instructions given to the computer to perform specific tasks, such as print("Hello, World!").
- **Control Structures**: These manage the flow of a program by making decisions, repeating actions, or halting a program, such as if, for, while loops.

2. What is Python?

- Python is a widely used programming language that was first implemented in 1989 by Guido van Rossum. It is an open-source(Free To Download) language, meaning its source code is freely available to the public for use and modification. Python's development is community-driven, allowing for continuous improvement and updates.

3. Why Python?

3A. Python is Easy to Learn

- Python's simplicity makes it an excellent choice for beginners. Its syntax is clear and resembles natural language, which helps new programmers focus on learning programming concepts rather than worrying about complex syntax.

3B. Speed and Ease Compared to Other Languages

- **C**: While C is much faster, it is harder to use due to its low-level nature and manual memory management.
- **Java**: Comparable in speed to Python but slightly more complex due to its strict object-oriented structure.
- **Perl**: Slower than Python, but similarly easy to use. However, Perl is not strongly typed, meaning variables can change types, which might lead to errors.

3C. Object-Oriented Programming (OOP)

Python supports OOP concepts like:

- **Polymorphism**: The ability to present the same interface for different underlying data types.
- **Operation Overloading**: Using the same operator for different data types (e.g., + can be used for both addition and string concatenation).
- **Multiple Inheritance**: A class can inherit features from more than one parent class.

3D. Indentation

- Python uses indentation to define blocks of code, which makes the code more readable. This is considered one of Python's greatest features, as it enforces a clean and consistent coding style.

3E. Free and Open Source

- Python is freely available for download and installation. The source code can be accessed, modified, and redistributed, encouraging collaborative development.

3F. Python is Powerful

- **Dynamic Typing**: Variables do not need a declared type; Python infers it at runtime.
- **Built-in Types and Tools**: Python has a rich set of built-in types, like lists, dictionaries, and sets, and tools that make programming more efficient.
- **Library Utilities**: Python has an extensive standard library and third-party utilities like NumPy and SciPy for scientific computing.
- **Automatic Memory Management**: Python manages memory allocation and deallocation automatically, reducing the chance of memory leaks.

3G. Python is Portable

- Python programs run on virtually every major platform, including Windows, macOS, and Linux. As long as the Python interpreter is installed, Python programs can run consistently across different systems.

3H. Python is Mixable

- Python can be easily integrated with components written in other languages, such as C or C++, which is useful for performance-critical applications. This ability to mix languages is particularly advantageous for scientific computing and other computation-heavy tasks.

3I. Python is Easy to Use

- Python does not require an intermediate compilation step as in C or C++. Python programs are automatically compiled to an intermediate form called bytecode, which the interpreter reads. This process combines the speed of an interpreter with the efficiency of compiled languages.

3J. Python is Easy to Learn

- The structure and syntax of Python are designed to be intuitive and straightforward. This makes it an excellent first language for new programmers and a powerful tool for experienced developers.

4. What Can You Do with Python?

Python is versatile and can be used for a wide range of applications, including:

- **System Programming**: Automating and managing system tasks.
- **Graphical User Interface (GUI) Programming**: Creating desktop applications with graphical interfaces using libraries like Tkinter or PyQt.
- **Internet Scripting**: Developing web applications, working with APIs, and automating web-related tasks using frameworks like Django or Flask.
- **Component Integration**: Integrating different software components or systems.
- **Database Programming**: Interacting with databases using libraries like SQLite or SQLAlchemy.
- **Gaming, Images, XML, Robotics**: Developing games, working with images, processing XML data, and programming robots.

4A. Example: A Simple Python Program

```
# Example of a simple Python program to calculate the area of a circle
PI = 3.14  # Constant value of Pi

def calculate_area(radius):
    area = PI * radius ** 2  # Formula for area of a circle
    return area

# Input from user
radius = float(input("Enter the radius of the circle: "))
area = calculate_area(radius)
print(f"The area of the circle with radius {radius} is {area}")
```

This example showcases Python's simplicity and readability, making it easy to understand and use for a wide range of applications.

5. Compiling and Interpreting

A comparison between two methods of executing programs in different programming languages: **compiling** and **interpreting**. Let's break this down and provide examples for each part.

5A. Compiling Process (Java Example)

In the case of languages like **Java**, the process involves compiling the code before it can be executed:

- **Source Code (Hello.java)**: This is the human-readable program written by the programmer in Java. For example:

```
public class Hello {
   public static void main(String[] args) {
      System.out.println("Hello, world!");
   }
}
```

- **Compiling**: Java programs are compiled using a compiler (like javac). The compiler translates the source code (Java code) into **bytecode** that can be executed by the Java Virtual Machine (JVM).
    - Command: javac Hello.java
    - Output: Hello.class (bytecode)

- **Byte Code (Hello.class)**: This is an intermediate code that is platform-independent. It’s not human-readable but is understood by the JVM.

- **Executing**: The JVM interprets the bytecode and runs the program.
    - Command: java Hello
    - Output: The program runs and prints: Hello, world!

5B. Interpreting Process (Python Example)

Languages like **Python** skip the compilation step and directly interpret the code:

- **Source Code (Hello.py)**: This is the Python program written by the programmer. Example:

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

- **Interpreting**: Python does not require a compilation step. The Python interpreter reads the source code and converts it into machine instructions on the fly.
  - Command: python Hello.py
  - Output: The program runs directly and prints: Hello, world!

5C. Key Differences Between Compiling and Interpreting

- **Compiling**:
  - The source code is first transformed into bytecode (in Java) or machine code in other languages like C. This bytecode or machine code is then executed by a virtual machine or processor.
  - Advantages: Once compiled, programs generally run faster since the translation to machine code is already done.
  - Example Languages: C, C++, Java.

- **Interpreting**:
  - The source code is executed line by line by the interpreter without the need for a separate compilation step.
  - Advantages: Interpreted languages like Python are generally more flexible and easier to debug since you can run and test code without compiling.
  - Example Languages: Python, JavaScript, PHP.

Note:

- Many languages require you to compile (translate) your program into a form that the machine understands.
- Python is instead directly interpreted into machine instructions.

5D. Further Example: C Programming (Compiled)

In C, the process is even more direct:

- **Source Code (Hello.c)**:

```c
#include <stdio.h>
int main() {
  printf("Hello, world!\n");
  return 0;
}
```

- **Compiling**: The C compiler (gcc for example) translates the source code directly into machine code.
  - Command: gcc Hello.c -o Hello
  - Output: An executable file (Hello or Hello.exe)

- **Executing**:
  - Command: ./Hello
  - Output: The program runs and prints: Hello, world!

In this case, the machine code produced is platform-specific and cannot be run on other platforms without recompilation.

5E. Further Example: JavaScript (Interpreted)

In JavaScript, the code is usually interpreted by a browser:

- **Source Code (Hello.js)**:

```javascript
console.log("Hello, world!");
```

- **Interpreting**: The browser's JavaScript engine reads the code and runs it immediately, without needing compilation.
  - Output: The message Hello, world! appears in the browser's console.

Summary

- **Compiled languages** like Java and C convert the entire source code to bytecode or machine code before execution, which leads to faster performance at runtime.
- **Interpreted languages** like Python and JavaScript execute the source code line by line without prior compilation, offering more flexibility and ease of debugging.

Each approach has its trade-offs, and the choice between compiled and interpreted languages depends on the specific needs of the program and developer preferences.

Your First Python Program – Python IDLE

In Python, IDLE (Integrated Development and Learning Environment) is an interactive shell where you can type Python commands and get instant feedback. Let's break down the example you provided:

Here, 1 + 2 is a simple mathematical expression. The shell evaluates the expression and returns the result: 3.

```
>>> 1 + 2
3
```

Similarly, 2 * 3 is multiplication. The shell returns 6.

```
>>> 2 * 3
6
```

In this case, you assign the value 1 to the variable x using the = operator. print(x) outputs the value stored in x, which is 1.

```
>>> x = 1
>>> print(x)
1
```

Here, you're updating the value of x by adding 1 to its current value. Now, x is 2, and print(x) outputs 2.

```
>>> x = x + 1
>>> print(x)
2
```

This command ends the interactive session. Alternatively, you can use quit().

The goal of running this in IDLE is to ensure that Python is properly installed and running on your system.

```
>>> exit()
```

1. Understanding the Code

Let’s explore the underlying concepts one by one.

1A. Indentation Matters to Code Meaning

Python uses **indentation** to define code blocks. Unlike other programming languages that use braces ({}) or other symbols to indicate the beginning and end of a block of code (like loops, conditionals, or functions), Python uses indentation (typically 4 spaces or a tab).

For example:

```
if x > 0:
    print("x is positive")
    print("Still inside the block")
```

Here, the lines with print are indented, meaning they belong to the if block. Without correct indentation, Python will raise an error.

Example:

```
if x > 0:
print("x is positive")  # This will result in an IndentationError
```

1B. First Assignment to a Variable Creates It

In Python, variables are created when they are first assigned a value. Unlike languages such as C or Java, you don't need to declare the type of the variable beforehand.

Example:

```
x = 10  # Integer
y = "hello"  # String
z = 3.14  # Float
```

Python determines the variable type based on the assigned value. You can assign different types of values to the same variable later, as Python is dynamically typed.

Example:

```
x = 10  # x is an integer
y = "new value"  # Now x is a string
```

1C. Assignment is = and Comparison is ==

In Python:

- = is used for assignment (storing a value in a variable).
- == is used for comparison (to check if two values are equal).

Example:

```
x = 5  # Assignment: x is now 5
if x == 5:  # Comparison: checks if x equals 5
   print("x equals 5")
```

1D. For Numbers, +, -, *, /, % Are as Expected

Python uses standard arithmetic operators like:

- + for addition
- - for subtraction
- * for multiplication
- / for division (returns a float)
- % for modulus (remainder)

Example:

```
a = 10 + 5  # a = 15
b = 10 - 2  # b = 8
c = 10 * 2  # c = 20
d = 10 / 3  # d = 3.3333...
e = 10 % 3  # e = 1 (remainder of 10 divided by 3)
```

- **Special Use of + for String Concatenation**: In Python, you can use the + operator to concatenate strings.

Example:

```
s1 = "Hello"
s2 = "World"
result = s1 + " " + s2  # result = "Hello World"
```

- **Special Use of % for String Formatting**: The % operator can also be used to format strings, similar to how it's used in C's printf.

Example:

```
name = "John"
age = 25
message = "My name is %s and I am %d years old" % (name, age)
print(message)
# Output: My name is John and I am 25 years old
```

1E. Logical Operators are Words (and, or, not) Not Symbols

In Python, the logical operators are written as words:

- and is used for logical AND.
- or is used for logical OR.
- not is used for logical negation (NOT).

Example:

```
x = True
y = False

if x and not y:  # True AND (NOT False) = True
    print("Condition is True")
```

1F. The Basic Printing Command is print

The print() function is used to output data to the console.

Examples:

```
print("Hello, World!")  # Outputs: Hello, World!
print(42)               # Outputs: 42
```

You can print multiple values by separating them with commas:

```
name = "Alice"
age = 30
print("Name:", name, "Age:", age)  # Outputs: Name: Alice Age: 30
```

2. Further Examples

2A. Simple Calculation in Python

```
>>> a = 3
>>> b = 4
>>> result = a * b
>>> print(result)
12
```

2B. Variable Assignment and Reassignment

```
>>> x = 10
>>> print(x)
10
>>> x = "Hello"
>>> print(x)
Hello
```

2C. Using Logical Operators

```
>>> a = True
>>> b = False
>>> print(a and b)  # False, because both must be True for 'and' to return True
>>> print(a or b)   # True, because only one of them needs to be True for 'or'
>>> print(not b)    # True, because 'not False' is True
```

2D. String Concatenation and Formatting

```
>>> name = "Alice"
>>> greeting = "Hello, " + name + "!"
>>> print(greeting)
Hello, Alice!

# Using formatting with %
>>> age = 25
>>> print("Name: %s, Age: %d" % (name, age))
Name: Alice, Age: 25
```

In summary, Python is designed to be easy to read and write, making use of indentation for code structure, dynamic typing for variables, and straightforward syntax for operators and commands.

Sentences or Lines in Python

1. x = 2 (Assignment Statement)

The assignment statement is used to assign a value to a variable. In this case, the variable x is assigned the value 2.

Example:

```
x = 2
print(x)  # Output: 2
```

2. x + 2 (Assignment with Expression)

This line is an example of an assignment where an expression is used to calculate the new value of the variable. x is currently 2 (from the previous line), so this expression evaluates to 2 + 2, which gives 4. Then, x is updated to hold the new value 4.

Example:

```
x = 2
x = x + 2
print(x)  # Output: 4
```

3. print(x) (Print Statement)

The print() function is used to output data to the console. In this case, it will print the current value of x.

Example:

```
x = 4
print(x)  # Output: 4
```

Basic Data Types in Python

Python has several built-in data types. Let's explore each one you mentioned with further details and examples.

1. Integers (Default For Numbers)

Integers are whole numbers (positive, negative, or zero) with no decimal point.

Example Of Assignment:

```
z = 5 / 2
print(z)  # Output: 2.5
```

However, if you meant z = 5 // 2 (integer division), it results in:

```
z = 5 // 2
print(z)  # Output: 2
```

// is used for integer division in Python and will return the quotient without the remainder.

2. Floats

Floats are numbers that contain a decimal point. They can represent fractions.

Example:

```
z = 3.456
print(z)  # Output: 3.456
```

If you divide two integers, Python will automatically return a float:

```
z = 5 / 2
print(z)  # Output: 2.5
```

3. Strings

Strings in Python are sequences of characters. They can be enclosed in either double quotes (") or single quotes (').

- **Example with Double Quotes and Single Quotes:**

```
s1 = "abc"
s2 = 'abc'
print(s1 == s2)  # Output: True
```

- **Example with a Single Quote inside a String:**

```
s = "Matt's book"
print(s)  # Output: Matt's book
```

- **Triple-quoted strings** (""" """ or ''' ''') allow you to define multi-line strings or strings that contain both single and double quotes without escaping them.

  Example with Triple Double Quotes:

```
s = """This is a 'test' with "quotes"."""
print(s)
# Output:
# This is a 'test' with "quotes".
```

4. Further Explanation with Examples

- **Mathematical Operations:** Python allows simple mathematical operations using these basic data types:

```
x = 10
y = 3

addition = x + y          # 13
subtraction = x - y       # 7
multiplication = x * y    # 30
division = x / y          # 3.3333 (float result)
int_division = x // y     # 3 (integer result)
power = x ** y            # 1000 (10 raised to the power of 3)
```

- **String Operations:** Strings can be concatenated or repeated:

```
s1 = "Hello"
s2 = "World"

# Concatenation
result = s1 + " " + s2
print(result)  # Output: "Hello World"

# Repetition
print(s1 * 3)  # Output: "HelloHelloHello"
```

- **Type Conversion**: You can convert between different data types using type conversion functions:

```
# Convert integer to string
x = 10
s = str(x)
print(s)  # Output: '10'

# Convert string to integer
s = "123"
y = int(s)
print(y)  # Output: 123

# Convert string to float
s = "3.14"
f = float(s)
print(f)  # Output: 3.14
```

- **Working with Mixed Types**: Python allows mathematical operations between floats and integers, and it promotes the result to the most general type (float).

```
x = 3
y = 2.5
result = x + y  # Output: 5.5 (float)
print(result)
```

Summary

- **Assignment Statement (x = 2)**: Stores a value in a variable.
- **Assignment with Expression (x = x + 2)**: Updates a variable using its current value.
- **Print Statement (print(x))**: Outputs the value of a variable.

For **Basic Data Types**:

- **Integers**: Whole numbers (e.g., x = 5).
- **Floats**: Decimal numbers (e.g., x = 3.456).
- **Strings**: Text data, enclosed in single (') or double (") quotes, with triple quotes allowing multiline strings (e.g., "Matt's", '''Hello "World"!''').

Whitespace & Commenting in Python

1. Whitespace in Python

1A. Explanation

Whitespace is crucial in Python as it affects how the code is structured and executed. Python uses indentation (spaces or tabs) to determine the grouping of statements, unlike many other languages which use braces {} to denote code blocks.

- **Newline** (\n) ends a line of code.
- **Indentation** is used to define blocks of code, such as in loops, functions, and classes.
- **Colons (:)** indicate the start of a new block. For example, when defining a function or an if statement.

1B. Key Points

- Each block of code must be consistently indented. Typically, 4 spaces are used, though tabs are also allowed (but mixing tabs and spaces will cause errors).
- The indentation level defines the scope of code, e.g., which statements belong inside a loop, function, or condition.
- A **line of code with less indentation** indicates the end of a block.
- A **line of code with more indentation** starts a new block within the current block.

1C. Example

```
def greet(name):       # The colon (:) indicates a new block starts
    if name:           # Another colon, starting a block for the if statement
        print(f"Hello, {name}")  # Indented block inside the if
    else:              # else starts a new block
        print("Hello, stranger")  # Indented block inside the else

greet("Alice")
greet("")
```

1D. Output

```
Hello, Alice
Hello, stranger
```

In this example:

- The def greet(name): defines a function.
- The indented blocks under if and else define code that runs depending on the condition.

2. Whitespace in Python (Deeper Dive)

2A. Indentation

Python requires proper indentation to understand the flow of the program. If indentation is not consistent or incorrect, Python will throw an IndentationError.

Here are some additional key concepts:

- **Consistent Indentation**: If you choose to use 4 spaces, you must stick with 4 spaces throughout the code block. Mixing tabs and spaces will raise an error.
- **Nested Blocks**: If a block contains another block (like an if statement inside a for loop), each inner block must be indented further.

2B. Example Of Inconsistent Indentation (Will Cause An Error)

```
def calculate_square(num):
 if num > 0:    # First level indentation
       square = num * num  # Inconsistent: this has more spaces than the previous block
   return square  # Less indented than the if block (error)

print(calculate_square(5))
```

Output (Error)

```
IndentationError: unindent does not match any outer indentation level
```

2C. Corrected Example With Consistent Indentation

```
def calculate_square(num):
   if num > 0:  # First level indentation
      square = num * num  # Correct indentation
      return square
   return 0

print(calculate_square(5))
```

Output

```
25
```

Here, the indentation is consistent, making the code structure clear to both the interpreter and the programmer.

2D. Example: Nested Blocks And Control Flow

```
def check_even_odd(numbers):
    for num in numbers:
        if num % 2 == 0:  # First block (inside for)
            print(f"{num} is even")
        else:  # Second block (inside for)
            print(f"{num} is odd")

# Call the function
check_even_odd([1, 2, 3, 4, 5])
```

Output

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
```

Here, the for loop is the outer block, and within it, there's an if-else block. The indentation helps Python understand which statements belong to which block.

2E. Newlines (\n)

In Python, the newline character (\n) is used within strings to create line breaks, or in some cases, we may need to write code on multiple lines (for readability or length). Python generally allows code to span multiple lines using backslashes (\).

Example Of Using Newline In Strings

```
def print_poem():
    print("Roses are red,\nViolets are blue,\nPython is awesome,\nAnd so are you!")

print_poem()
```

Output

```
Roses are red,
Violets are blue,
Python is awesome,
And so are you!
```

Example of Multi-line Statements

```
result = 1 + 2 + 3 + \
         4 + 5 + 6  # Use backslash for multi-line statement

print(result)
```

Output

```
21
```

3. Commenting in Python

3A. Explanation

Comments are used to add explanatory notes to the code. Python ignores everything after the # symbol on that line. Comments can serve various purposes, including:

- **Explaining code logic** for yourself or others.
- **Temporarily disabling** parts of code.
- **Leaving notes** such as who wrote the code or why a certain approach was taken.

3B. Key Points

- **Single-line comments** start with #.
- **Multi-line comments** can be created by placing # at the beginning of each line or using triple quotes (''' or """) for block comments, though the latter is typically reserved for documentation strings (docstrings).

3C. Example of Comments

```
# This function calculates the sum of two numbers
def add_numbers(a, b):  # a and b are input arguments
    return a + b  # return the sum of a and b

# Calling the function with example arguments
result = add_numbers(5, 10)
print(result)  # Output will be 15

# The line below is commented out and won't execute
# print("This line won't run")
```

3D. Output

```
15c
```

In this example:

- The comments describe what each part of the code is doing.
- The last line # print("This line won't run") is disabled by the comment and won't be executed.

3E. Example of Using a Block Comment (Docstring)

```
def subtract_numbers(a, b):
    """
    This function subtracts b from a
    Parameters:
    a (int): The first number
    b (int): The second number

    Returns:
    int: The result of the subtraction
    """
    return a - b

# The result of subtraction
print(subtract_numbers(10, 3))  # Output: 7
```

Summary of Whitespace and Commenting

- **Whitespace** determines code structure through indentation and line breaks, and there are no braces {} in Python to group code.

- **Comments** allow you to annotate your code with explanations or temporarily disable code without deleting it.

4. Commenting in Python (Advanced Concepts)

4A. Types of Comments in Python

- **Inline Comments**: These are comments that occur on the same line as a statement and explain that specific line.
- **Block Comments**: These are longer comments that describe a section of code. You can use multiple # symbols for multi-line comments or docstrings for documentation.

4B. Inline Comments

Useful for short explanations on the same line as the code.

```
x = 10  # Assign 10 to x
y = x + 5  # Add 5 to x and store in y
```

4C. Block Comments

Useful when explaining complex or multiple lines of code.

```
# The following function takes a number as input,
# squares it, and then adds 10 to the result.
def compute(num):
    return num * num + 10
```

4E. Docstrings (Documentation Strings)

While block comments use #, **docstrings** use triple quotes (''' or """). These are used for documentation, especially for functions and classes, and can be accessed with Python's built-in help() function.

Example Of A Docstring

```
def divide_numbers(a, b):
    """
    This function divides two numbers.

    Parameters:
    a (int or float): The numerator
    b (int or float): The denominator

    Returns:
    float: The result of the division, or
    None: If division by zero is attempted.
    """
    if b == 0:
        return None
    return a / b

help(divide_numbers)    # Checking the docstring using help()
```

Output

```
Help on function divide_numbers in module __main__:

divide_numbers(a, b)
    This function divides two numbers.

    Parameters:
    a (int or float): The numerator
    b (int or float): The denominator

    Returns:
    float: The result of the division, or
    None: If division by zero is attempted.
```

4F. Disabling Code Temporarily

Sometimes, you might want to disable a portion of the code without deleting it, for debugging or testing purposes. This is done using the # symbol.

Example of Temporarily Disabling Code

```
# This part of the code is disabled:
# print("This won't run")

# Only this part will execute
print("Hello, World!")
```

Output

```
Hello, World!
```

Best Practices for Commenting

- Keep comments **concise** and **relevant**.
- Avoid obvious comments like # add 2 to x when the code itself is clear: x = x + 2.
- Use comments to explain **why** something is done, not just **what** is done.
- Write docstrings for all functions and classes to improve readability and maintainability.

5. More Advanced Example Combining Both Concepts

5A. Example: Commenting and Using Whitespace Properly

```
def process_numbers(numbers):
    """
    This function processes a list of numbers.
    It filters out negative numbers and calculates
    the sum of positive numbers.

    Parameters:
    numbers (list): A list of integers

    Returns:
    int: The sum of positive numbers
    """
    positive_sum = 0  # Initialize sum

    # Loop through each number in the list
    for num in numbers:
        if num > 0:  # Check if number is positive
            positive_sum += num  # Add to sum if positive
        else:
            # Negative numbers are ignored
            continue
    return positive_sum  # Return the final sum

# Example usage
result = process_numbers([-5, 2, 10, -3, 8])
print(result)  # Should print 20 (2 + 10 + 8)
```

Output

```
20
```

In this example:

- A **docstring** is used to describe the function and its parameters.
- **Inline comments** explain each step of the logic.
- **Whitespace** is used consistently to group the if block inside the for loop, and the function definition is clearly separated from the main code.

Python Scripts

When writing Python programs (scripts), the behavior and execution are controlled by the Python interpreter. Here's a detailed look at how Python scripts work:

- **Execution of a Python Script**: When you run a Python script from the command line (e.g., using python script.py), the Python interpreter processes and evaluates each expression in the file. The interpreter reads the file line by line and executes the code sequentially.

- **Command Line Arguments and I/O Redirection**: Python provides built-in mechanisms to work with command-line arguments and to redirect input/output. For example, you can pass arguments to a script using sys.argv or use tools like argparse to handle more complex argument parsing. Additionally, input and output can be redirected, for instance:

```
python script.py < inputfile > outputfile
```

This means the script will take input from inputfile and direct its output to outputfile.

- **Scripts vs. Modules**: Python scripts can be designed to work both as standalone programs and as modules that can be imported into other scripts. For example, when you write a script with the following at the bottom:

```
if __name__ == "__main__":
    main()
```

- This block ensures that the script can be run directly or imported as a module. When imported, this block won't execute, but when run directly, the main() function will be called.

- **File Suffix (.py)**: By convention, Python scripts are saved with a .py extension, signaling that the file contains Python code. This extension helps the operating system and users identify the file as a Python script.

Example of Python Script

```
# script.py
import sys

def main():
    name = sys.argv[1]
    print(f"Hello, {name}!")

if __name__ == "__main__":
    main()
```

Run the script from the command line:

```
python script.py Alice
```

This will output:

```
Hello, Alice!
```

Interacting with Python

There are two main ways to interact with Python: **Interactive Mode** and **Manual (Script) Mode**.

1. Interactive Mode

This is when you directly interact with Python using the interpreter. It is often used as a **sandbox** for quick experimentation. In this mode, you enter commands one by one, and Python executes them immediately. This is useful for learning, testing small pieces of code, or debugging.

- **REPL (Read-Eval-Print Loop)**: In interactive mode, Python operates in a REPL, meaning it reads the user's input, evaluates it, prints the result, and loops back to await the next input.

- **Example**:

```
$ python
Python 3.10.5 (default, Jun 6 2022) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- Here, you can start typing Python commands and get immediate feedback. For example:

```
>>> 5 + 3
8
>>> print("Hello, World!")
Hello, World!
```

- This mode is great for using Python as a calculator, testing small code snippets, or exploring new features of Python.

1A. Key Characteristics

- You type individual Python commands, one at a time, directly into the interpreter (Python shell).
- Python immediately evaluates and runs the command, returning the result instantly.
- Ideal for quick testing, debugging, and learning new Python features.
- Temporary: once you close the Python shell, any variables or functions you defined are lost unless saved manually.

1B. How to Access Interactive Mode

You can start Python in interactive mode by typing python or python3 in the terminal or command prompt.

```
$ python
Python 3.10.5 (default, Jun 6 2022) on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

1C. Example of Interactive Mode

```
>>> a = 5
>>> b = 10
>>> c = a + b
>>> print(c)
15
```

In this example:

- The user types in a variable assignment (a = 5), and Python executes it immediately.
- After the addition operation (c = a + b), Python instantly computes and stores the result.
- The print() function outputs the result, 15.

Benefits of Interactive Mode

- **Immediate Feedback**: You get instant results after entering a command.
- **Experimentation**: Great for trying small code snippets, exploring functions, or using Python as a calculator.
- **Learning**: Beginners can quickly understand the syntax and behavior of Python by interacting with it.

Limitations of Interactive Mode

- **Temporary Environment**: Once you close the shell, all variables and functions are lost unless saved elsewhere.
- **Not for Long Programs**: Complex, long scripts are difficult to manage in an interactive environment since the code isn't saved unless copied and pasted elsewhere.

2. Manual Mode (Script Execution / Script Mode)

Manual mode refers to executing a Python script from the command line or a terminal. You write all the necessary code in a file and run it in one go, unlike interactive mode, where you enter commands step by step.

- **Execution**: In manual mode, the Python interpreter processes the entire script. This is useful when you have a more complex or long program, and it needs to be executed from start to finish. The script can be executed on any system with Python installed using the command:

```
python script.py
```

In **Script Mode**, you write a series of Python statements in a file (usually with a .py extension) using a text editor, and then execute the entire file as a script.

1A. Key Characteristics

- The entire program is written in a file before execution. This makes it reusable and allows for long, complex programs.
- You run the script file from a terminal or command prompt, and Python executes all the statements in the file sequentially.
- Suitable for developing real-world applications and projects.

1B. How to Run a Python Script

Once you write a Python script (in a file, say script.py), you can execute it by typing:

```
python script.py
```

1C. Example of Script Mode

```
# script.py
a = 5
b = 10
c = a + b
print(c)
```

1D. Save this file as script.py, and then run it:

```
$ python script.py
15
```

In this case:

- The script is stored in a file, so the code can be reused.
- When executed, Python reads the entire file, runs the statements, and prints 15.

Benefits of Script Mode:

- **Reusability**: Once written, the script can be saved, modified, and reused anytime.
- **Complex Programs**: Suitable for writing long, complex programs that need to be executed multiple times.
- **Persistence**: Variables, functions, and logic are stored in a file, so nothing is lost between runs.
- **Version Control**: You can manage versions of your script for different projects or improvements.

Limitations of Script Mode:

- **No Immediate Feedback**: Unlike interactive mode, you must run the entire script to see the output, even if you only want to check a single line.
- **Less Dynamic**: It's not as fast for quick experimentation, as you need to edit the file, save it, and run the script again to see the results.

3. Comparison of Interactive vs. Manual Mode

Feature	Interactive Mode	Manual Mode / Script Mode
Description	REPL: Python takes single inputs, evaluates, and returns a result	Python executes a script all at once via a command-line prompt
Benefits	Useful for debugging, exploring new features, or quick throwaway scripts	Best for long, complex programs or reusable code
Usage	Open a Python prompt with python	Run a Python script file: python script.py
Execution Style	Type one line at a time, and Python responds immediately	Execute an entire file with multiple lines of code
Use Case	Ideal for testing, experimenting, and quick calculations	Suitable for building complex, reusable programs
Persistence	Code is lost when the session ends unless manually saved	Code is saved in a file and can be reused anytime
Feedback	Immediate feedback for each line entered	Feedback after running the whole script
Complexity	Not suitable for long or complex programs	Ideal for long, structured programs
Reusability	No built-in persistence or reusability	Script files can be reused and modified

4. Further Example

4A. Interactive Mode Example (Python as a Calculator)

You want to quickly calculate the square of a number:

```
>>> num = 8
>>> square = num ** 2
>>> print(square)
64
```

This calculation is quick, and you immediately see the result.

4B. Script Mode Example (Simple Program)

Suppose you want to write a program that calculates the area of a rectangle:

```
# rectangle_area.py
def calculate_area(length, width):
    return length * width

length = float(input("Enter the length: "))
width = float(input("Enter the width: "))
area = calculate_area(length, width)
print(f"The area of the rectangle is {area}")
```

You can save this in a file called rectangle_area.py. When you run this script:

```
$ python rectangle_area.py
Enter the length: 5
Enter the width: 10
The area of the rectangle is 50.0
```

This is a more complex, structured program, and script mode allows you to reuse and modify it easily.

Conclusion

- **Interactive Mode** is best for quick experimentation, debugging, and small calculations. It offers an immediate feedback loop but is less suitable for complex programs.
- **Script Mode** is ideal for writing longer, reusable, and structured programs. It's the standard way of developing Python projects but lacks the instant feedback provided by interactive mode.

Both modes have their unique advantages and should be chosen based on the task at hand. If you're experimenting or learning, interactive mode is often the best option, while script mode is more appropriate for real-world applications and projects.

Conclusion

- **Interactive Mode** is best for quick experimentation, debugging, and small calculations. It offers an immediate feedback loop but is less suitable for complex programs.
- **Script Mode** is ideal for writing longer, reusable, and structured programs. It's the standard way of developing Python projects but lacks the instant feedback provided by interactive mode.

Both modes have their unique advantages and should be chosen based on the task at hand. If you're experimenting or learning, interactive mode is often the best option, while script mode is more appropriate for real-world applications and projects.

Print

The print() function in Python is used to output data to the terminal or console. Whenever you execute a Python script, the results of your computations or variable values are not automatically displayed unless you explicitly print them.

1. Explanation

- **Basic Usage of print()**:
  - The print() function displays any type of data (e.g., strings, numbers, lists, etc.) to the console.

- **Syntax**

```
print(value1, value2, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
```

- value1, value2, ...: These are the values you want to print. You can pass multiple values separated by commas.

- sep: Specifies how to separate multiple values. By default, it is a space (' ').

- end: Specifies what to print at the end. By default, it is a newline character ('\n'), so the next output will be printed on a new line.

- file: The file or stream where the output should go (by default, it is sys.stdout, which is the terminal).

- flush: If True, it forces the output to be written immediately to the terminal.

2. Example 1: Printing A Simple Message

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

Output:

```
Hello, World!
```

3. Example 2: Printing Multiple Values

```
name = "Alice"
age = 30
print(name, age)
```

Output:

```
Hello, World!
```

Here, the print() function automatically adds a space between name and age.

4. Example 3: Customizing The Separator

You can customize the separator between values using the sep argument.

```
print("Apple", "Banana", "Cherry", sep=", ")
```

Output:

```
Apple, Banana, Cherry
```

5. Example 4: Customizing The End Character

By default, print() adds a newline at the end, but you can change this with the end argument.

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

Output:

```
Hello World!
```

6. Example 5: Printing Variables Of Different Types

```
age = 25
name = "John"
print("My name is", name, "and I am", age, "years old.")
```

Output:

```
My name is John and I am 25 years old.
```

This works because Python automatically converts variables (e.g., name and age) to a string format when printing. However, in some cases, you might need to convert the variables yourself, which we'll cover next.

7. Example 6: Mixing Types And Type Conversion

When printing, mixing types (such as strings and integers) requires type conversion if you're concatenating them:

```
age = 25
name = "John"
print("My name is " + name + " and I am " + str(age) + " years old.")
```

Output:

```
My name is John and I am 25 years old.
```

Here, str(age) converts the integer age into a string so it can be concatenated with other strings using the + operator. Without conversion, Python will raise an error because you cannot concatenate a string and an integer directly.

8. Example 7: Printing Lists Or Other Data Structures

You can also print complex data types like lists, dictionaries, or tuples.

```
my_list = [1, 2, 3, 4]
print("List contents:", my_list)
```

Output:

```
List contents: [1, 2, 3, 4]
```

9. Example 8: Redirecting Output To A File

By default, print() outputs to the console, but you can redirect the output to a file using the file parameter.

```
with open("output.txt", "w") as f:
    print("Writing to a file!", file=f)
```

This will write "Writing to a file!" into the file output.txt instead of displaying it in the console.

10. Important Note On Mixing Variable Types:

- **When printing multiple values separated by commas**: Python automatically converts them to strings and adds a space between them. No manual type conversion is needed.
- **When concatenating strings using +**: All the values must be strings. If any variable is not a string (e.g., an integer), you must explicitly convert it using str().

10A. Example: Mixing Up Variables

```
age = 30
name = "Alice"
# This will raise an error
print("Name: " + name + ", Age: " + age)
```

This code will raise a TypeError because age is an integer, and you can't concatenate an integer with a string. To fix this:

```
print("Name: " + name + ", Age: " + str(age))
```

Output:

```
Name: Alice, Age: 30
```

Summary

- print() is used to display output in Python.
- You can print multiple variables, strings, numbers, lists, and more using print().
- You can customize how values are separated (using sep) and how the output ends (using end).
- Always be careful when mixing different types (like strings and integers); if necessary, convert them using str().

This allows for much more flexible and informative output, making debugging and displaying results easier in Python programs.

Input

The input() function is used in Python to read a value from the user, typically from the keyboard. Whatever the user types is returned as a **string**, and this value can be stored in a variable for later use. If you need the input as a number (integer or float), you must **explicitly convert** it using functions like int() or float().

1. Key Points

- The result of input() is always a string by default.
- If you expect a number, you need to convert the string to the desired numeric type (e.g., int() for integers or float() for floating-point numbers).

1A. Example

```
age = int(input("How old are you? "))  # The user's input is converted to an integer
print("Your age is", age)  # The user's age is printed

# Calculating retirement time (assuming 65 is the retirement age)
print("You have", 65 - age, "years until retirement")
```

1B. Output

```
How old are you? 53
Your age is 53
You have 12 years until retirement
```

1C. Explanation

- The user is prompted with the message "How old are you?".
- When the user enters a number (e.g., 53), it is converted into an integer using int().
- This number is stored in the variable age, and we print the user's age.
- Finally, we calculate the number of years left until the assumed retirement age of 65 by subtracting the user's age from 65 and displaying that value.

2. input() Statement With Example

In this example, we use the input() function to collect multiple pieces of information from the user, including their name and birth year.

2A. Example

```
print("What's your name?")  # Asking for the user's name
name = input("> ")  # Reading the user's name

print("What year were you born?")  # Asking for the birth year
birthyear = int(input("> "))  # Reading and converting the birth year to an integer

# Calculating the user's age (assuming the current year is 2022)
print("Hi", name + "!", "You are", 2022 - birthyear, "years old.")
```

2B. Output

```
What's your name?
> Alice
What year were you born?
> 1990
Hi Alice! You are 32 years old.
```

2C. Explanation

- The program first asks the user for their name. The input() function is used to collect this data, and it is stored in the variable name.
- Then, the program asks for the user's birth year, which is also taken via input(). Since this is a number, the input is converted to an integer using int().
- The user's current age is calculated by subtracting the birth year from the current year (2022, in this example).
- Finally, the program prints a greeting with the user's name and calculated age.

3. More Examples Of input() Usage

3A. Collecting Multiple Values

```
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
print("Your full name is", first_name, last_name)
```

Output

```
Enter your first name: John
Enter your last name: Doe
Your full name is John Doe
```

3B. Handling Decimal Numbers

```
height = float(input("Enter your height in meters: "))
print("Your height is", height, "meters.")
```

Output

```
Enter your height in meters: 1.75
Your height is 1.75 meters.
```

3C. Basic Calculator

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
print("The sum is", num1 + num2)
```

Output

```
Enter first number: 5
Enter second number: 3.5
The sum is 8.5
```

4. Common Mistakes to Avoid

Forgetting to Convert Input: The input() function always returns a string, even if you are asking for a number. You need to convert it to an integer or float using int() or float() respectively.

- **Example**:

```
age = input("How old are you? ")  # This will store a string, not an integer!
```

- **Output**

```
age = int(input("How old are you? "))  # Properly converts the input to an integer
```

Incorrect Conversion: Trying to convert non-numeric input into an integer or float will cause an error.

- **Example**

```
age = int(input("Enter a number: "))  # If the user enters "abc", it will cause an error
```

Program Steps or Program Flow

1. Definition

This refers to the sequence of actions or operations that a program or recipe follows to achieve a goal. Each step leads to the next, and there may be some conditional or looping steps involved.

2. Example: Pizza Dough Recipe

- **Gather Ingredients**: Collect all the necessary ingredients like sugar, salt, olive oil, and flour.
- **Combine Ingredients**: In a mixing bowl, combine 1 tablespoon of sugar, 1 tablespoon of salt, 1 tablespoon of olive oil, and 1 cup of flour.
- **Turn on Mixer**: Start the mixer to blend the ingredients.
- **Add Flour**: Gradually add 1/4 cup of flour to the mixture.
- **Check Dough Consistency**:
    - If the dough comes off the sides of the bowl, proceed to step 6.
    - Otherwise, go back to step 4 and keep adding flour until the desired consistency is reached.
- **Knead Dough**: Knead the dough for 15 minutes.
- **Let Rest**: Allow the dough to rest for at least 45 minutes in a warm area.

Sequential Steps or Flow

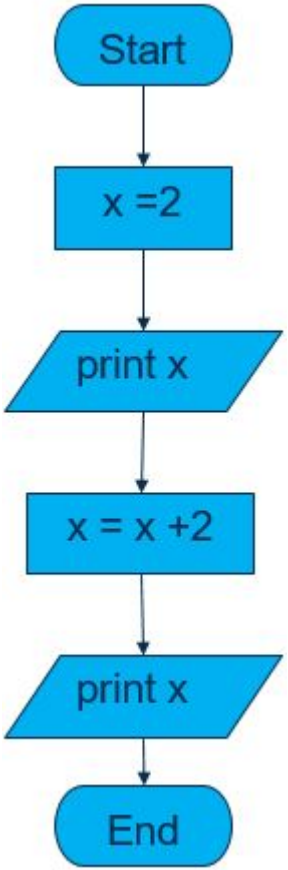
1. Definition

This describes how a program executes instructions in a linear order, one after the other, without skipping any steps unless explicitly directed by conditional statements.

2. Example: Simple Arithmetic Program

- In this example, the program executes each line in order, producing a clear output.

```
x = 2          # Step 1: Assign 2 to variable x
print(x)       # Step 2: Print the value of x (Output: 2)
x = x + 2      # Step 3: Increase x by 2 (x is now 4)
print(x)       # Step 4: Print the new value of x (Output: 4)
```



Conditional Steps or Flow

1. Definition

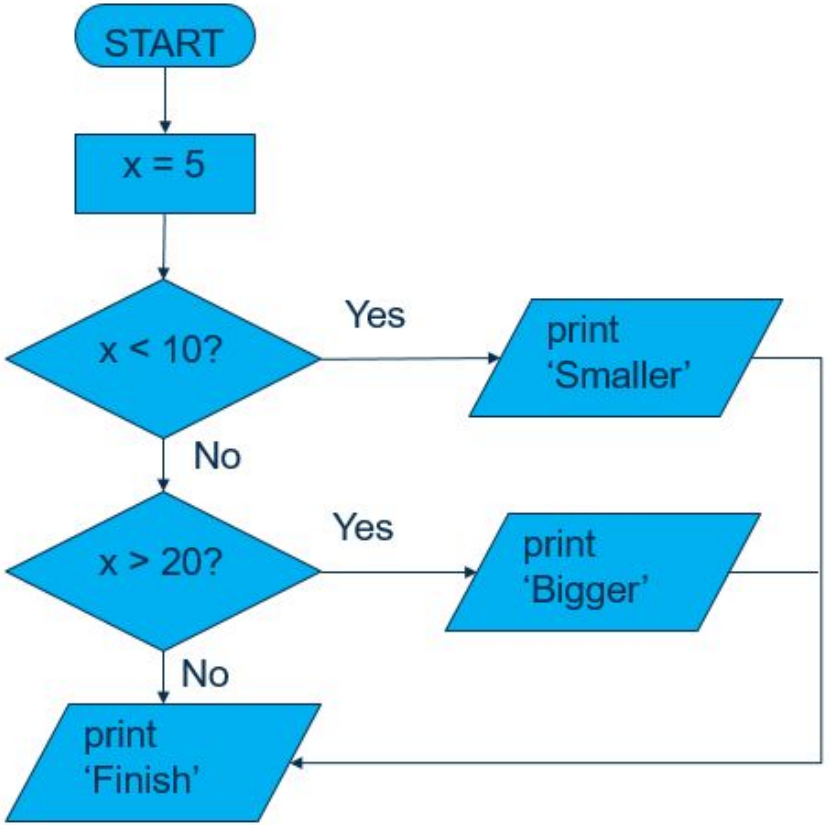
Conditional flow involves making decisions based on certain conditions. The program will execute certain steps only if specific conditions are met.

2. Example: Simple Conditional Statements

```
x = 5
if (x < 10):            # Check if x is less than 10
    print('Smaller')    # This line executes if the condition is true
if (x > 20):            # Check if x is greater than 20
    print('Bigger')     # This line does not execute since the condition is false
print('Finish')         # This line executes regardless of previous conditions
```

In this case, only the first conditional executes, leading to the output:

```
Smaller
Finish
```



Repetition Steps or Flow

1. Definition

Repetition involves executing a block of code multiple times. This is typically done using loops, which can iterate over a sequence or continue until a certain condition is met.

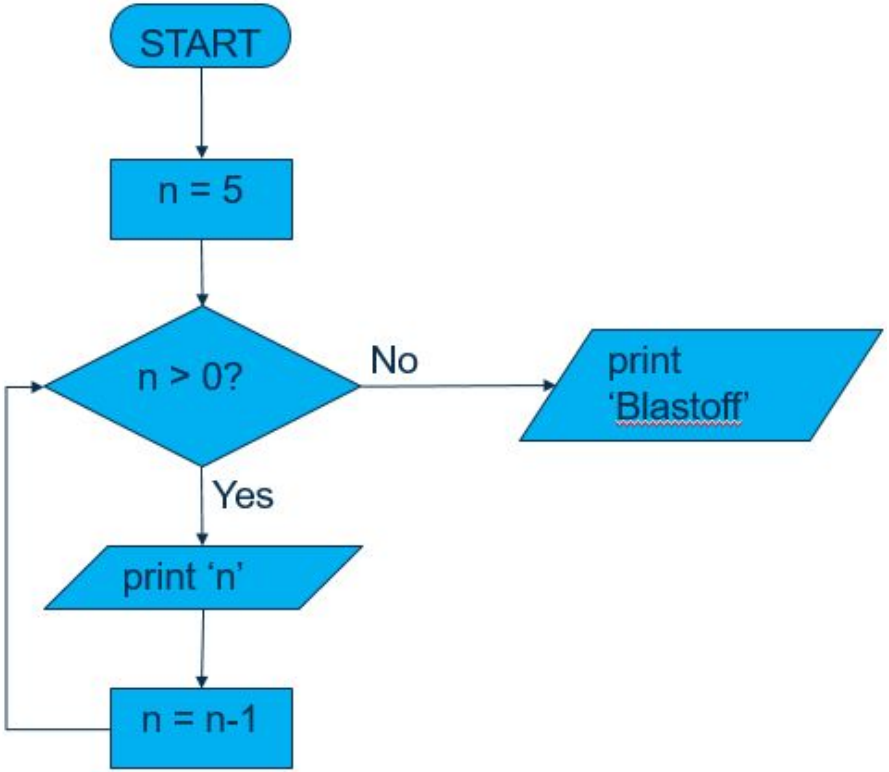
2. Example: Countdown with While Loop

```
n = 5                 # Initialize variable n to 5
while(n > 0):         # Loop while n is greater than 0
   print(n)           # Print the current value of n
   n = n - 1          # Decrement n by 1
print('Blastoff!')    # Print when the loop is finished
```

The output of this program will be:

```
5
4
3
2
1
Blastoff!
```

This demonstrates how the loop repeats the print statement and decrements the value of n until it is no longer greater than 0.



Summary

- **Program Steps**: A structured sequence of actions or operations.
- **Sequential Steps**: Execution of statements in a linear order.
- **Conditional Steps**: Decisions made based on conditions, leading to different execution paths.
- **Repetition Steps**: Repeated execution of a block of code, often using loops.

These concepts are fundamental in programming and help in building logical, structured, and efficient code.