Module 13

Python Fundamentals

**Introduction to Python Theory**

• Introduction to Python and its Features (simple, high-level, interpreted language).

**Python** is a widely used, high-level, interpreted programming language known for its simplicity, readability, and versatility. First released in 1991 by **Guido van Rossum**, Python has since become one of the most popular languages for various applications, from web development to data analysis, machine learning, and more.

• History and evolution of Python. • Advantages of using Python over other programming languages.

Python, created by **Guido van Rossum** and first released in **1991**, has evolved into one of the most widely used programming languages today. Initially developed as a successor to the ABC language, Python was designed to be simple, readable, and easy to learn. Over the years, it has undergone significant improvements, including the release of **Python 2.0** in 2000, which introduced features like garbage collection and list comprehensions. In **2008**, Python 3.0 was released, marking a major shift with improved Unicode support and other changes, although it was not backward compatible with Python 2.x. Python 3 has since become the dominant version, with support for Python 2 ending in 2020.

The advantages of Python over other programming languages are numerous. First, its **simplicity and readability** make it an excellent choice for beginners, allowing developers to focus on solving problems without getting bogged down by complex syntax. Python also offers a **large standard library** and is **cross-platform**, meaning code can run on different operating systems without modification. Its **flexibility** allows it to be used for a wide variety of tasks, from web development and data science to automation, machine learning, and game development.

Additionally, Python's **strong community support** ensures that resources like documentation, libraries, and tutorials are readily available. It supports both **object-oriented** and **functional programming**, giving developers flexibility in their approach. Although Python is an **interpreted** language, meaning it runs line by line, this feature makes it easier to debug and test code. Lastly, Python's integration with other languages and its use in emerging fields like **artificial intelligence** and **data analysis** further solidify its place as one of the most versatile and popular programming languages today.

• Installing Python and setting up the development environment (Anaconda, PyCharm, or VS Code).

To get started with Python, first download and install it from the official [python.org](https://www.python.org/downloads/). You can add Python to your system's PATH during installation to run it from the command line. For a more streamlined experience, you can use **Anaconda**, which simplifies package management and is ideal for data science. Download it from anaconda.com. Alternatively, you can set up an IDE like **PyCharm**, a feature-rich environment for Python development, or **Visual Studio Code (VS Code)**, a lightweight code editor that can be enhanced with extensions for Python. Both PyCharm and VS Code support installing libraries via pip and provide integrated terminals for running Python code.

• Writing and executing your first Python program.

Program:

Print(“Hello World!!”)

Output:

Hello World!!

**2. Programming Style**

• Understanding Python’s PEP 8 guidelines.

**PEP 8** is the official style guide for Python code, which promotes writing code that is easy to read, maintain, and understand. It provides recommendations on how to structure your Python code to make it consistent and readable. Adhering to PEP 8 is considered a best practice and helps improve collaboration in team environments. Below are some of the key aspects of PEP 8.

• Indentation, comments, and naming conventions in Python.

In Python, indentation, comments, and naming conventions are essential for writing clean, readable, and maintainable code.

**Indentation** is used to define the structure of code blocks, particularly in loops, functions, and conditionals. Unlike some other languages that use braces {} for block definition, Python uses indentation (typically four spaces per level) to signify a block of code.

**Comments** are used to explain the code and improve its readability. In Python, comments are written using the # symbol for single-line comments, and docstrings (triple quotes) are used for multi-line comments or to document functions, classes, and modules.

**Naming conventions** in Python follow the PEP 8 style guide. Variable and function names are typically written in lowercase with underscores separating words (e.g., my\_variable), while class names use the CamelCase convention (e.g., MyClass). Constants are usually written in uppercase letters with underscores (e.g., MAX\_SIZE). Adhering to these conventions ensures that code is uniform and easily understandable by other developers, facilitating collaboration and long-term maintenance.

• Writing readable and maintainable code.

Writing readable and maintainable code is crucial for creating software that is not only functional but also easy to understand, extend, and debug. Several key principles guide the process:

1. **Clear Naming Conventions**: Choose descriptive and meaningful names for variables, functions, and classes. This helps developers understand the purpose of the code at a glance. For instance, use get\_user\_info() instead of a vague name like fetch() or do\_something(). Consistently following naming conventions, such as using snake\_case for variables and functions and CamelCase for classes, also promotes consistency across the codebase.
2. **Modular Code**: Break the code into small, focused functions or classes that handle one specific task. This makes the code more manageable and reusable. Functions should ideally not exceed a few lines to stay readable. If a function becomes too complex, it should be refactored into smaller ones.
3. **Consistent Indentation and Formatting**: Proper indentation, usually four spaces per level in Python, visually separates blocks of code, making it easier to follow the structure. Consistent formatting also includes following a style guide (e.g., PEP 8 in Python) for spacing, line length, and comment placement. This consistency reduces cognitive load and helps maintain the codebase over time.
4. **Comments and Documentation**: While code should ideally be self-explanatory, comments are valuable for explaining the intent behind complex logic or decisions. Use comments sparingly to clarify tricky sections of code, but avoid over-commenting obvious code. In addition, docstrings should be used to document the purpose of functions, classes, and modules, as well as their parameters and expected return values.
5. **Error Handling**: Code should be robust, with appropriate error handling and logging to catch and report problems. Use try-except blocks for exceptions in Python and ensure that the program gracefully handles unexpected situations instead of crashing.
6. **Testing**: Writing unit tests and integration tests is an integral part of maintainable code. Tests help ensure that code works as expected and make future changes safer. Automated testing can catch regressions and improve confidence when refactoring or adding new features.
7. **Avoiding Code Duplication**: Repeating the same code in multiple places increases the risk of bugs and makes the code harder to maintain. Instead, reuse code by creating functions or classes that encapsulate common behavior. Following the DRY (Don't Repeat Yourself) principle helps in reducing redundancy.
8. **Refactoring**: Continuously improving the code by refactoring—restructuring code without changing its external behavior—ensures it remains clean, efficient, and easy to understand as requirements evolve.

**3. Core Python Concepts**

• Understanding data types: integers, floats, strings, lists, tuples, dictionaries, sets.

1 **Integers (int)**: Whole numbers, both positive and negative, without decimals.

* Example: 5, -12, 100

2 **Floats (float)**: Numbers that have a decimal point.

* Example: 3.14, -0.99, 10.0

3 **Strings (str)**: A sequence of characters enclosed in single, double, or triple quotes.

* Example: "Hello, world!", 'Python', """This is a string"""

4 **Lists (list)**: Ordered, mutable (changeable) collections of items. Items can be of different types.

* Example: [1, 2, 3, "apple", 4.5]

5 **Tuples (tuple)**: Ordered, immutable (unchangeable) collections of items. Once created, the values cannot be modified.

* Example: (1, 2, 3, "apple")

6 **Dictionaries (dict)**: Unordered, mutable collections of key-value pairs. Keys are unique, and values can be of any data type.

* Example: {"name": "John", "age": 30, "city": "New York"}

7 **Sets (set)**: Unordered collections of unique items. Unlike lists, sets do not allow duplicates.

* Example: {1, 2, 3, 4}

• Python variables and memory allocation.

A variable in Python is a name that refers to a value stored in memory. You can think of a variable as a label or reference pointing to a memory location where the data is kept.

**Example:**

python

Copy code

x = 10

Here, x is the variable name, and it refers to the value 10 stored in memory.

When you create a variable and assign a value to it:

1. **Object Creation**: Python creates an object in memory to store the value.
2. **Variable Reference**: The variable name becomes a reference to that object.
3. **Memory Address**: Each object has a unique memory address, which can be checked using the id() function.

**Example:**

python

Copy code

x = 5

y = 5

print(id(x))

# Output: Memory address of the object 5

print(id(y)) # Output: Same as x, because integers are immutable and shared In this case, both x and y point to the same memory location because Python optimizes memory usage for immutable types like integers by reusing existing objects.

**Types of Memory Allocation**

1. **Static Memory Allocation**:
   * Happens at compile-time.
   * Memory size is fixed, and variables are pre-allocated.
   * **Not typical in Python**, as it is dynamically typed.
2. **Dynamic Memory Allocation**:
   * Happens at runtime.
   * Python dynamically allocates memory when an object is created.
   * Memory for objects is managed by Python’s **garbage collector**.

• Python operators: arithmetic, comparison, logical, bitwise.

**1. Arithmetic Operators**

Arithmetic operators perform basic mathematical operations:

* **Addition (+)**: Adds two numbers. Example: 5 + 3 gives 8.
  + **Subtraction (-)**: Subtracts one number from another. Example: 10 - 4 results in 6.
  + **Multiplication (\*)**: Multiplies two numbers. Example: 7 \* 3 equals 21.
  + **Division (/)**: Divides one number by another. Example: 9 / 2 gives 4.5.
  + **Floor Division (//)**: Performs division but returns only the integer part. Example: 9 // 2 is 4.
  + **Modulus (%)**: Returns the remainder after division. Example: 10 % 3 gives 1.
  + **Exponentiation (\*\*)**: Raises one number to the power of another. Example: 2 \*\* 3 equals 8.

**2. Comparison Operators**

Comparison operators compare two values and return either True or False:

* **Equal to (==)**: Checks if two values are equal. Example: 5 == 5 is True.
* **Not equal to (!=)**: Checks if two values are not equal. Example: 5 != 3 is True.
* **Greater than (>)**: Checks if one value is greater than another. Example: 10 > 5 is True.
* **Less than (<)**: Checks if one value is smaller than another. Example: 2 < 8 is True.
* **Greater than or equal to (>=)**: Checks if one value is greater than or equal to another. Example: 5 >= 5 is True.
* **Less than or equal to (<=)**: Checks if one value is smaller than or equal to another. Example: 3 <= 4 is True.

**3. Logical Operators**

Logical operators are used to combine conditional statements:

* **and**: Returns True if both conditions are true. Example: (5 > 3) and (10 > 5) is True.
* **or**: Returns True if at least one condition is true. Example: (5 > 10) or (3 < 8) is True.
* **not**: Negates the result. Example: not (5 == 5) is False.

**4. Bitwise Operators**

Bitwise operators work on binary numbers:

* **Bitwise AND (&)**: Compares each bit of two numbers; the result has 1 where both bits are 1. Example: 5 & 3 gives 1.
* **Bitwise OR (|)**: Compares each bit of two numbers; the result has 1 where either bit is 1. Example: 5 | 3 gives 7.
* **Bitwise XOR (^)**: Compares each bit of two numbers; the result has 1 where the bits are different. Example: 5 ^ 3 gives 6.
* **Bitwise NOT (~)**: Inverts all bits. Example: ~5 gives -6.
* **Left shift (<<)**: Shifts bits to the left, filling in zeros from the right. Example: 5 << 1 gives 10.
* **Right shift (>>)**: Shifts bits to the right, filling in zeros from the left. Example: 5 >> 1 gives 2.

**4. Conditional Statements**

• Introduction to conditional statements: if, else, elif.

Conditional statements allow you to control the flow of your program based on different conditions. They let the program make decisions and execute specific blocks of code depending on whether a condition is true or false.

**1. if Statement**

The if statement is the most basic conditional. It checks a condition, and if the condition is True, it executes the code block that follows.

**Syntax:**

python

Copy code

if condition:

# Code to execute if the condition is True

**Example:**

python

Copy code

x = 10

if x > 5:

print("x is greater than 5")

*Output: "x is greater than 5"*

**2. else Statement**

The else statement provides an alternative block of code to execute if the if condition is False.

**Syntax:**

python

Copy code

if condition:

# Code to execute if the condition is True

else:

# Code to execute if the condition is False

**Example:**

python

Copy code

x = 3

if x > 5:

print("x is greater than 5")

else:

print("x is not greater than 5")

*Output: "x is not greater than 5"*

**3. elif Statement**

The elif statement stands for "else if" and allows you to check multiple conditions sequentially. It only runs if the previous conditions were False.

**Syntax:**

python

Copy code

if condition1:

# Code to execute if condition1 is True

elif condition2:

# Code to execute if condition2 is True

else:

# Code to execute if none of the conditions are True

**Example:**

python

Copy code

x = 15

if x < 10:

print("x is less than 10")

elif x == 15:

print("x is exactly 15")

else:

print("x is greater than 10 but not 15")

*Output: "x is exactly 15"*

**Nested Conditions**

You can also nest if, else, and elif statements within each other for more complex decision-making.

**Example:**

python

Copy code

x = 20

if x > 10:

if x < 30:

print("x is between 10 and 30")

else:

print("x is greater than or equal to 30")

*Output: "x is between 10 and 30"*

**Using Conditional Expressions (Ternary Operator)**

For simple conditions, Python offers a shorthand way to write if-else statements.

**Syntax:**

python

Copy code

value = condition\_true if condition else condition\_false

**Example:**

python

Copy code

x = 7

result = "Even" if x % 2 == 0 else "Odd"

print(result)

*Output: "Odd"*

• Nested if-else conditions.

**Nested if-else** conditions involve placing one if or else statement inside another. This structure allows you to check multiple conditions sequentially within different levels of logic.

**Syntax:**

python

Copy code

if condition1:

# Outer if block

if condition2:

# Inner if block

# Code to execute if both condition1 and condition2 are True

else:

# Code to execute if condition1 is True but condition2 is False

else:

# Code to execute if condition1 is False

1. **Looping (For, While)**

• Introduction to for and while loops.

**Introduction to for and while Loops in Python**

Loops are used to repeat a block of code multiple times, which makes programming more efficient by reducing redundancy. Python provides two primary types of loops: for loops and while loops.

**1. for Loop**

A for loop is used when you know the number of times you want to iterate or when looping through a collection (like a list, tuple, or string).

**Syntax:**

python

Copy code

for variable in iterable:

# Code block to repeat

**Explanation:**

* **variable**: Takes the value of each element in the iterable (such as a list or range) on each iteration.
* **iterable**: A collection of items or a sequence (like a list or range()).

**2. while Loop**

A while loop is used when you want to repeat a block of code as long as a condition is True. This is useful when the number of iterations isn't known in advance.

**Syntax:**

python

Copy code

while condition:

# Code block to repeat

**Explanation:**

* **condition**: A boolean expression that determines if the loop continues.
* The loop runs until the condition becomes False.

**Key Differences Between for and while Loops:**

* **for loops**: Best for iterating over a known sequence (like lists, strings, or ranges).
* **while loops**: Best for situations where the loop must run until a specific condition changes.

**Break and Continue Statements**

* **break**: Stops the loop immediately.
* **continue**: Skips the current iteration and moves to the next.

• How loops work in Python.

Loops in Python allow you to repeat a block of code multiple times, making it easier to perform repetitive tasks without writing redundant code. Python primarily supports two types of loops: for loops and while loops.

• Using loops with collections (lists, tuples, etc.).

Python collections like lists, tuples, dictionaries, and sets can be easily iterated over using loops. This allows you to perform operations on each element of the collection, such as printing values, modifying elements, or applying calculations.

**1. Iterating Over Lists**

A list is an ordered collection of items. You can loop through each item in the list using a for loop.

**2. Iterating Over Tuples**

A tuple is similar to a list but immutable. You can loop through a tuple the same way as a list.

**3. Iterating Over Strings**

A string is a sequence of characters, so you can loop through each character individually.

**4. Iterating Over Dictionaries**

Dictionaries store key-value pairs. You can iterate through keys, values, or both.

**5. Iterating Over Sets**

A set is an unordered collection of unique elements.

**6. Using enumerate() with Loops**

The enumerate() function provides both the index and the item when looping through a collection.

**7. Nested Loops with Collections**

You can use loops within loops to handle collections of collections (like lists of lists).