Python Documentation

Master's Python With Elight Python Series Documentation

Table of Contents

Python — Introduction

Python is one of the most popular programming languages in the world today. It was created by **Guido van Rossum** and first released in **1991**. Python is known for its **simple and easy-to-read syntax**, which makes it a great choice for both beginners and professionals.

Why Learn Python?

Python is widely used because it is:

- **Beginner-Friendly:** Its syntax is close to English, making it easier to learn.
- **Interpreted:** You don't need to compile code — it runs line by line, which is faster for testing and debugging.
- **Cross-Platform:** Python works on Windows, macOS, Linux, and even mobile devices.

- **Versatile:** Used in web development, automation, data science, artificial intelligence, game development, and more.
- **Community Support:** Millions of developers and thousands of free tutorials (like W3Schools) are available to help you learn.

Key Features of Python

- **High-level language:** Python handles complex details like memory management for you.
- **Dynamically typed:** You don't need to declare variable types — Python figures it out automatically.
- **Readable syntax:** Uses indentation instead of braces, making the code cleaner.
- **Large standard library:** Comes with built-in modules for handling files, math, dates, databases, and more.

Where is Python Used?

Python is extremely flexible. Some common areas include:

- **Web Development:** Frameworks like Django and Flask.
- **Data Science & Machine Learning:** Libraries like Pandas, NumPy, TensorFlow, and PyTorch.
- **Automation (Scripting):** Automating repetitive tasks like file handling or sending emails.
- **Game Development:** Libraries like Pygame.
- **Education:** Because it's simple, Python is often the first programming language taught in schools.

First Python Example

Here's the famous "Hello, World!" program in Python:

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

When you run this code, Python will display:

```
Hello, World!
```

How to Start with Python

1. Download and install Python 3 from python.org.
2. Open your terminal or command prompt.
3. Type `python --version` or `python3 --version` to confirm installation.
4. Use an editor like VS Code, PyCharm, or even W3Schools' online editor to write and run code.

Quick Recap

Python is a high-level, beginner-friendly, interpreted language created in 1991. It is easy to read, supports many applications (from web apps to AI), and has a huge global community. If you are new to coding, Python is one of the best languages to start with.

Practice Questions (Python Intro)

1. Who created Python and in which year was it first released?
2. Why is Python considered beginner-friendly?
3. List three real-world applications of Python.
4. Write a Python program to print: `Welcome to Python`.
5. What is the difference between "compiled" and "interpreted" languages, and where does Python fit?

Python — Getting Started

Before you can start writing and running Python programs, you need to set up Python on your computer. This topic explains how to install Python, check if it's working, and run your first program.

1. Install Python

- **Download:** Visit python.org and download the latest version of **Python 3**.
- **Windows:** Run the installer and make sure you tick the option *"Add Python to PATH"* during installation.
- **macOS:** Python is pre-installed, but often it is an older version. Install Python 3 from python.org for the latest updates.
- **Linux:** Most Linux systems already have Python installed. To install or update, use a package manager like:

  ```
  sudo apt-get install python3
  ```

2. Verify Installation

After installation, check if Python is working properly:

```
python --version
```

or

```
python3 --version
```

If Python is installed, you'll see something like:

```
Python 3.12.6
```

3. Write Your First Program

Open your terminal (Command Prompt, PowerShell, or shell) and type:

```
python
```

This opens the **Python Interactive Shell**, where you can run Python commands directly.

Type this command:

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

You should see:

```
Hello, World!
```

4. Running Python from a File

Instead of typing code in the interactive shell, you can write code in a file with the extension `.py`.

1. Open a text editor like Notepad, VS Code, or PyCharm.
2. Write the following code:

   ```
   print("Welcome to Python programming!")
   ```

3. Save the file as `first_program.py`.
4. Run it using:

   ```
   python first_program.py
   ```

5. Online Python Editors

If you don't want to install Python immediately, you can use online interpreters such as:

- W3Schools Tryit Editor
- Replit (replit.com)
- Google Colab (colab.research.google.com)

These let you write and run Python code directly in your browser.

Quick Recap

- Download and install Python 3 from python.org.
- Check installation using `python --version`.
- Run code in the Python interactive shell or save it as a `.py` file.
- You can also practice Python online without installation.

Practice Questions (Python Getting Started)

1. Where can you download the latest version of Python?
2. Which command is used to check the installed version of Python?
3. Write and run a program that prints: `Python Setup Successful!`
4. What is the file extension used for Python scripts?
5. Name two online platforms where you can run Python without installation.

Python Syntax

The **syntax of Python** is what makes it one of the easiest programming languages to learn. Unlike other languages that use symbols and complex rules, Python uses simple and clean code that almost looks like normal English sentences.

Key Points About Python Syntax

- **Indentation Instead of Braces:** Python uses `indentation (spaces or tabs)` to define code blocks. Other languages like C, C++ or Java use curly braces `{ }`, but in Python indentation is mandatory.
- **Case Sensitive:** Variables and keywords are case-sensitive in Python. For example, `name` and `Name` are different.
- **Statements End Automatically:** In Python, you do not need a semicolon (`;`) at the end of each line. Each line is treated as a complete statement.
- **Comments:** You can add comments using the `#` symbol. Comments are ignored by the Python interpreter.

Example of Python Syntax

```
 # This is a comment
print("Hello, World!")  # This line prints text
```

In this example:

- The first line is a **comment**.
- The second line uses the **print()** function to display text on the screen.

Indentation in Python

Indentation is one of the most important parts of Python syntax. It tells the interpreter which block of code belongs together.

```
 if 5 > 2:
    print("Five is greater than two!")
```

If you forget to indent or use the wrong indentation, Python will throw an error. For example:

```
 if 5 > 2:
print("Five is greater than two!")  # ❌ This will cause an error
```

Multi-Line Statements

You can write a statement across multiple lines using a backslash (`\`).

```
 x = 10 + 20 + 30 + \
    40 + 50
print(x)
```

Python Identifiers

Identifiers are the names you give to variables, functions, classes, etc. They must start with a letter or underscore and cannot contain special characters or spaces.

Conclusion

Python's syntax is designed to be clean, simple, and easy to read. It helps beginners focus on problem-solving instead of worrying about complex grammar rules. Understanding indentation, case sensitivity, and basic rules of Python syntax is the first step toward writing error-free code.

Python Comments

Comments in Python are lines of text that are ignored by the Python interpreter. They are used to explain the code, make it more readable, and leave helpful notes for yourself or other developers. Good commenting practices make your code easier to understand and maintain.

Why Use Comments?

- **Explain Code:** Helps beginners and teams understand the logic behind the code.
- **Debugging Aid:** You can temporarily "disable" code by turning it into a comment.
- **Improves Readability:** Well-commented code is easier to maintain in the long run.

Types of Comments in Python

1. Single-Line Comments
In Python, single-line comments start with the `#` symbol. Everything written after `#` on that line is treated as a comment.

```
 # This is a single-line comment
print("Hello, World!")  # This prints a message
```

2. Multi-Line Comments
Python does not have a special syntax for multi-line comments like some other languages. Instead, you can use multiple `#` symbols or triple quotes (`''' ... '''` or `""" ... """`) to write multi-line comments.

```
 # This is line one of a comment
# This is line two of the comment
# This is line three of the comment
```

```
 """
This is a multi-line comment.
It can span across multiple lines.
Useful for long explanations or documentation.
"""
print("Python comments example")
```

Best Practices for Writing Comments

- Keep comments short and meaningful.

- Use comments to explain *why* you are doing something, not just *what* the code does.
- Avoid obvious comments like `# increase x by 1` if the code already shows `x = x + 1`.
- Update comments when you update the code.

Conclusion

Comments are not executed by Python, but they play a huge role in making your code professional and easy to understand. As a beginner, try to develop the habit of writing clear and meaningful comments in your programs.

Python Variables

In Python, **variables** are used to store data that can be used and modified later in your program. Think of a variable as a *container* that holds information like numbers, text, or more complex data types.

Creating a Variable

To create a variable in Python, you simply assign a value using the `=` operator:

```
 name = "Ashish"
age = 20
height = 5.9
```

Here:

- `name` stores a string.
- `age` stores an integer.
- `height` stores a float (decimal number).

Rules for Naming Variables

- Variable names must start with a letter or an underscore (`_`).
- Variable names can only contain letters, numbers, and underscores.
- Variable names are case-sensitive. For example, `age` and `Age` are different.
- Do not use Python keywords (like `if`, `for`, `while`) as variable names.

Assigning Values to Multiple Variables

You can assign values to multiple variables in a single line:

```
 x, y, z = 5, 10, 15
print(x, y, z)  # Output: 5 10 15
```

Or assign the same value to multiple variables:

```
 a = b = c = 100
print(a, b, c)  # Output: 100 100 100
```

Changing the Value of a Variable

Variables in Python are **dynamic**, meaning their value can change at any time:

```
 score = 50
print(score)  # Output: 50

score = 75
print(score)  # Output: 75
```

Global vs Local Variables

- **Global Variables:** Declared outside a function and can be used anywhere in the program.
- **Local Variables:** Declared inside a function and can only be used within that function.

```
 global_var = 100  # Global variable

def my_function():
    local_var = 50  # Local variable
    print(local_var)

my_function()  # Output: 50
print(global_var)  # Output: 100
```

Conclusion

Variables are the foundation of any Python program. They allow you to store and manipulate data easily. Understanding how to create, name, and use variables correctly is essential for writing efficient and readable Python code.

Python Data Types

In Python, **data types** define the kind of value a variable holds. Each variable in Python has a data type that determines what operations can be performed on it. Understanding data types is essential for writing correct and efficient code.

Common Python Data Types

- **Numbers:** Used to store numeric values. Python has three types of numbers:
  - **int:** Integer values, e.g., 5, -10, 1000
  - **float:** Decimal numbers, e.g., 3.14, -0.5
  - **complex:** Complex numbers, e.g., 2 + 3j
- **String (str):** Text values enclosed in single or double quotes, e.g., `"Hello"` or `'Python'`
- **Boolean (bool):** Represents True or False values
- **List:** A collection of multiple items in a specific order, e.g., `[1, 2, 3, "Python"]`
- **Tuple:** Similar to lists, but **immutable** (cannot be changed), e.g., `(1, 2, 3)`
- **Set:** An unordered collection of unique items, e.g., `{1, 2, 3}`
- **Dictionary (dict):** Stores data in **key-value pairs**, e.g., `{"name": "Ashish", "age": 20}`
- **NoneType:** Represents a **null value**, e.g., `None`

Checking Data Types

You can check the data type of a variable using the `type()` function:

```
 x = 10
y = "Python"
z = 3.14

print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'str'>
print(type(z))  # Output: <class 'float'>
```

Type Conversion (Casting)

Python allows you to convert one data type to another using type conversion functions:

```
 x = 10      # int
y = float(x)   # convert to float
z = str(x)     # convert to string

print(y)  # Output: 10.0
print(z)  # Output: '10'
```

Conclusion

Understanding Python data types is crucial for programming because it helps you choose the right type of variable for the data you want to store and manipulate. Using data types correctly improves your code's reliability, performance, and readability.

Python Numbers

In Python, **numbers** are used to store numeric values. Python supports several types of numbers, and understanding them is essential for performing calculations and mathematical operations in your programs.

Types of Numbers in Python

- **Integer (int):** Whole numbers without decimals. Example: `10, -5, 1000`
- **Float (float):** Numbers with a decimal point. Example: `3.14, -0.5, 2.0`
- **Complex (complex):** Numbers with a real and imaginary part. Example: `2 + 3j`

Creating Numbers

Assign numbers to variables like this:

```
 int_num = 10
float_num = 3.14
complex_num = 2 + 3j

print(int_num)
print(float_num)
print(complex_num)
```

Basic Number Operations

Python allows you to perform arithmetic operations easily:

- **Addition (+):** `5 + 3 = 8`
- **Subtraction (-):** `10 - 4 = 6`
- **Multiplication (*):** `2 * 5 = 10`

- **Division (/):** `10 / 2 = 5.0`
- **Floor Division (//):** Divides and returns the integer part. `7 // 2 = 3`
- **Modulus (%):** Returns the remainder. `7 % 2 = 1`
- **Exponent (**):** Power operation. `2 ** 3 = 8`

Type Conversion

Convert numbers between types using built-in functions:

```
 x = 10      # int
y = float(x)   # convert to float
z = complex(x) # convert to complex

print(type(y))  # <class 'float'>
print(type(z))  # <class 'complex'>
```

Python Number Functions

Python has useful built-in functions for numbers:

- `abs(x)` - Returns absolute value of x
- `round(x, n)` - Rounds x to n decimal places
- `pow(x, y)` - Returns x raised to the power y
- `min(x, y, ...)` - Returns the smallest number
- `max(x, y, ...)` - Returns the largest number

Conclusion

Numbers are fundamental in Python programming. By understanding the different types of numbers and how to perform operations on them, you can easily handle calculations, mathematical problems, and even complex scientific tasks in your Python programs.

Python Casting

In Python, **casting** means converting a variable from one data type to another. Python is dynamically typed, which means the type is decided automatically when you assign a value. However, you may want to convert data into a specific type for calculations or formatting.

Why Use Casting?

- To ensure values are in the right format for calculations.
- To avoid errors when working with different data types.
- To convert user input (which is always a string) into numbers.

Types of Casting in Python

`int()` **— Convert to Integer**
Converts a number or a numeric string into an integer. Decimals are truncated, not rounded.

```
 x = int(1)       # 1
y = int(2.9)     # 2
z = int("10")    # 10
print(x, y, z)   # Output: 1 2 10
```

`float()` **— Convert to Floating Point**
Converts a number or numeric string into a floating-point (decimal) number.

```
 a = float(1)       # 1.0
b = float(2.5)     # 2.5
c = float("3.14")  # 3.14
print(a, b, c)     # Output: 1.0 2.5 3.14
```

`str()` **— Convert to String**
Converts numbers and other values into text.

```
 p = str(10)      # "10"
q = str(3.14)    # "3.14"
r = str(True)    # "True"
print(p, q, r)   # Output: 10 3.14 True
```

Practical Examples

```
 # Example 1: Math with input
age = input("Enter your age: ")  # input is string
age_num = int(age)
print(age_num + 5)

# Example 2: String concatenation
price = 99
print("The price is " + str(price))
```

Key Points

- `int()` removes the decimal part (does not round).
- `float()` can handle decimal values and numeric strings.
- `str()` is used when combining numbers with text.
- Invalid conversions (e.g. `int("abc")`) cause `ValueError`.

Practice Questions

1. What is the difference between `int(2.9)` and `round(2.9)`?
2. Write a program that asks for age and prints: *In 5 years, you will be X years old.*
3. Convert the string `"15.75"` into both float and int. What happens?
4. Why does `int("3.5")` raise an error while `float("3.5")` works?
5. Write a program that takes a float price, casts it to int, and prints both values.

Python Strings

In Python, a **string** is a sequence of characters enclosed in either single quotes (`' '`) or double quotes (`" "`). Strings are one of the most commonly used data types in Python.

Creating Strings

```
 # Single or double quotes
a = "Hello"
b = 'World'
print(a, b)   # Output: Hello World
```

Multiline Strings

Use triple quotes (`'''` or `"""`) for strings that span multiple lines.

```
 text = """This is
a multiline
string."""
print(text)
```

Accessing Characters

Strings are arrays of characters, and indexing starts at 0.

```
 word = "Python"
print(word[0])   # P
print(word[5])   # n
```

Slicing Strings

You can extract a portion of a string using slicing.

```
 word = "Python"
print(word[0:4])   # Pyth
print(word[2:])    # thon
print(word[:3])    # Pyt
```

String Length

```
 txt = "Hello World"
print(len(txt))   # 11
```

String Methods

Python has many built-in string methods:

- `upper()` → Converts to uppercase
- `lower()` → Converts to lowercase
- `strip()` → Removes whitespace
- `replace()` → Replaces a substring
- `split()` → Splits string into a list

```
 txt = " Hello World "
print(txt.upper())      # HELLO WORLD
print(txt.lower())      # hello world
print(txt.strip())      # "Hello World"
print(txt.replace("World", "Python"))  # Hello Python
print(txt.split())      # ['Hello', 'World']
```

String Concatenation

```
 a = "Hello"
b = "World"
print(a + " " + b)   # Hello World
```

String Formatting

You can insert variables into strings using `f-strings` or `format()`.

```
 name = "Alice"
age = 25
print(f"My name is {name}, and I am {age} years old.")
print("My name is {}, and I am {} years old.".format(name, age))
```

Escape Characters

Use the backslash (`\`) to escape special characters.

```
 txt = "We are \"Python\" developers"
print(txt)   # We are "Python" developers
```

Practice Questions

1. Create a multiline string about your favorite hobby and print it.
2. Write a program to display the first and last character of the string `"Programming"`.
3. Take a user's full name as input and print it in uppercase.
4. Replace the word `"World"` with `"Python"` in `"Hello World"`.
5. Format a string that says: "*My favorite number is X*", where *X* is an integer variable.

Python Booleans

In Python, a **boolean** represents one of two values: `True` or `False`. Booleans are often used in conditional statements, loops, and logical operations.

Boolean Values

```
 print(True)   # True
print(False)  # False
print(type(True))   # <class 'bool'>
```

Boolean Expressions

A boolean expression evaluates to either `True` or `False`.

```
 x = 10
y = 5

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

Using Booleans in If Statements

```
 is_raining = True

if is_raining:
    print("Take an umbrella")
else:
    print("Enjoy the sunshine")
```

bool() Function

The `bool()` function can convert values to a boolean:

```
 print(bool(1))       # True
print(bool(0))       # False
print(bool("Hello")) # True
print(bool(""))      # False
```

Falsy Values in Python

The following are considered **False** in Python:

- `0`

- `""` (empty string)
- `[]` (empty list)
- `()` (empty tuple)
- `{}` (empty dictionary)
- `None`
- `False`

Booleans in Logical Operations

```
 a = True
b = False

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

Practice Questions

1. Write a boolean expression to check if a number is greater than 100.
2. Use an `if` statement to check if a string is empty or not.
3. Convert the values `0`, `"Python"`, and `[]` into booleans using `bool()`.
4. What will be the result of `5 > 2 and 3 < 1`?
5. Create a program that checks if a person's age is 18 or older and prints "*Adult*" or "*Minor*".

Python Operators

In Python, **operators** are special symbols that perform operations on variables and values. Operators are the building blocks of programming because they help us perform calculations, compare values, and make decisions.

Types of Python Operators

- **Arithmetic Operators**
- **Comparison (Relational) Operators**
- **Assignment Operators**
- **Logical Operators**
- **Identity Operators**
- **Membership Operators**
- **Bitwise Operators**

1. Arithmetic Operators

Used for mathematical calculations:

```
 x = 10
y = 3

print(x + y)  # Addition → 13
print(x - y)  # Subtraction → 7
print(x * y)  # Multiplication → 30
print(x / y)  # Division → 3.33
print(x % y)  # Modulus → 1 (remainder)
print(x ** y) # Exponentiation → 1000
print(x // y) # Floor division → 3
```

2. Comparison Operators

Used to compare two values. Returns `True` or `False`.

```
 a = 5
b = 10

print(a == b)  # False
print(a != b)  # True
print(a > b)   # False
print(a < b)   # True
print(a >= b)  # False
print(a <= b)  # True
```

3. Assignment Operators

Used to assign values to variables.

```
 x = 5
x += 3   # x = x + 3 → 8
x -= 2   # x = x - 2 → 6
x *= 4   # x = x * 4 → 24
x /= 6   # x = x / 6 → 4.0
```

```
x %= 3   # x = x % 3 → 1
```

4. Logical Operators

Used to combine conditional statements.

```
 a = True
b = False

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

5. Identity Operators

Check if two objects are the same in memory.

```
 x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)     # True (same object)
print(x is y)     # False (different objects with same values)
print(x == y)     # True (values are equal)
```

6. Membership Operators

Check if a value is present in a sequence.

```
 fruits = ["apple", "banana", "cherry"]

print("apple" in fruits)   # True
print("mango" not in fruits) # True
```

7. Bitwise Operators

Used to compare binary numbers.

```
 a = 6   # 110 in binary
b = 3   # 011 in binary

print(a & b)  # AND → 2 (010)
print(a | b)  # OR  → 7 (111)
print(a ^ b)  # XOR → 5 (101)
print(~a)     # NOT → -7
print(a << 1) # Left shift → 12
print(a >> 1) # Right shift → 3
```

Practice Questions

1. Write a program to calculate the square of a number using the exponent operator (`**`).
2. Check if two numbers are equal or not using comparison operators.
3. Create a program that uses logical operators to check if a number is between 10 and 50.
4. Use membership operators to check if `"python"` exists in the list `["java", "python", "c++"]` .
5. Write a small code that uses bitwise operators to perform AND, OR, and XOR on `5` and `7` .

Python Lists

In Python, a **list** is an ordered collection of items. Lists are one of the most versatile data types and allow you to store multiple values in a single variable. You can store numbers, strings, or even other lists inside a list.

Creating a List

You can create lists using square brackets `[]` :

```
 fruits = ["apple", "banana", "cherry"]
numbers = [1, 2, 3, 4, 5]
mixed = [1, "apple", 3.14, True]
```

Accessing List Items

Items in a list can be accessed using their **index**. Python indexes start at 0.

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

List Slicing

Extract a portion of a list using slicing:

```
 fruits = ["apple", "banana", "cherry", "date"]
print(fruits[1:3])  # Output: ['banana', 'cherry']
print(fruits[:2])   # Output: ['apple', 'banana']
print(fruits[2:])   # Output: ['cherry', 'date']
```

Modifying Lists

- **Change an Item:** `fruits[1] = "blueberry"`
- **Add Items:** `append()` or `insert()` methods
- **Remove Items:** `remove()` , `pop()` , or `del`

```
 fruits = ["apple", "banana", "cherry"]

fruits.append("date")       # Add at the end
fruits.insert(1, "orange")  # Add at index 1
fruits.remove("banana")     # Remove specific item
fruits.pop()                # Remove last item

print(fruits)  # Output: ['apple', 'orange', 'cherry']
```

List Operations

- **Concatenation (+):** Join two lists → `[1,2] + [3,4] = [1,2,3,4]`
- **Repetition (*):** Repeat a list → `[1,2] * 2 = [1,2,1,2]`
- **Check Membership:** `3 in [1,2,3] → True`

Useful List Methods

- `len(list)` - Returns the number of items
- `list.sort()` - Sorts the list
- `list.reverse()` - Reverses the list
- `list.clear()` - Removes all items
- `list.copy()` - Returns a copy of the list

Conclusion

Python lists are flexible and powerful tools for storing and manipulating data. By mastering list operations and methods, you can efficiently handle collections of items in your programs.

Python Tuples

In Python, a **tuple** is an ordered collection of items, similar to a list. However, the key difference is that tuples are **immutable**, which means their items cannot be changed, added, or removed after creation.

Creating a Tuple

You can create tuples using parentheses `()` or the `tuple()` function:

```
 # Using parentheses
my_tuple = (1, 2, 3, "Python")

# Using tuple() function
another_tuple = tuple([4, 5, 6])

print(my_tuple)
print(another_tuple)
```

Accessing Tuple Items

Items in a tuple can be accessed using **indexing**:

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

Tuple Slicing

You can extract parts of a tuple using slicing:

```
 fruits = ("apple", "banana", "cherry", "date")
print(fruits[1:3])  # Output: ('banana', 'cherry')
print(fruits[:2])   # Output: ('apple', 'banana')
print(fruits[2:])   # Output: ('cherry', 'date')
```

Why Use Tuples?

- **Immutability:** Tuples cannot be changed, which makes them safer for storing data that should not be modified.
- **Performance:** Tuples are faster than lists because of their immutability.
- **Can be used as keys:** Tuples can be used as keys in dictionaries, unlike lists.

Tuple Operations

- **Concatenation (+):** Join two tuples → `(1,2) + (3,4) = (1,2,3,4)`
- **Repetition (*):** Repeat a tuple → `(1,2) * 2 = (1,2,1,2)`
- **Check Membership:** `3 in (1,2,3) → True`

Tuple Methods

Tuples have only two built-in methods:

- `count(x)` - Returns the number of times x appears in the tuple
- `index(x)` - Returns the first index of x in the tuple

Conclusion

Tuples are simple, immutable sequences that are ideal for storing fixed collections of items. They are faster than lists and can be safely used in programs where data should not change.

Python Sets

In Python, a **set** is an unordered collection of unique items. Sets are useful when you want to store multiple items without duplicates and perform operations like union, intersection, and difference.

Creating a Set

You can create a set using curly braces `{}` or the `set()` function:

```
 # Using curly braces
my_set = {1, 2, 3, 4}

# Using set() function
another_set = set([3, 4, 5, 6])

print(my_set)
print(another_set)
```

Key Features of Sets

- **No duplicates:** Sets automatically remove duplicate values.
- **Unordered:** Items in a set do not have a specific order.
- **Mutable:** You can add or remove items, but the items themselves must be immutable.

Accessing Set Items

Since sets are unordered, you cannot access items using an index. You can, however, loop through the set:

```
 fruits = {"apple", "banana", "cherry"}
for fruit in fruits:
    print(fruit)
```

Modifying Sets

- **Add Items:** `set.add(item)`
- **Add Multiple Items:** `set.update([item1, item2])`
- **Remove Items:** `set.remove(item)` or `set.discard(item)`
- **Clear Set:** `set.clear()` removes all items

```
 fruits = {"apple", "banana"}
fruits.add("cherry")
fruits.update(["orange", "mango"])
fruits.remove("banana")

print(fruits)  # Output: {'apple', 'cherry', 'orange', 'mango'}
```

Set Operations

- **Union (|):** Combine sets → `{1,2} | {2,3} = {1,2,3}`
- **Intersection (&):** Common items → `{1,2} & {2,3} = {2}`
- **Difference (-):** Items in first but not in second → `{1,2,3} - {2,3} = {1}`
- **Symmetric Difference (^):** Items in either set but not both → `{1,2} ^ {2,3} = {1,3}`

Conclusion

Python sets are powerful for storing unique items and performing mathematical set operations. They are ideal for removing duplicates, checking membership, and handling collections without caring about order.

Python Dictionaries

In Python, a **dictionary** is an unordered collection of **key-value pairs**. Dictionaries are used to store data where each item has a unique key associated with a value, making it easy to retrieve, update, and manage information.

Creating a Dictionary

You can create a dictionary using curly braces `{}` or the `dict()` function:

```
 # Using curly braces
my_dict = {"name": "Ashish", "age": 20, "city": "Delhi"}

# Using dict() function
another_dict = dict(name="Python", version=3.11)

print(my_dict)
print(another_dict)
```

Accessing Dictionary Items

You can access dictionary values using their keys:

```
 print(my_dict["name"])  # Output: Ashish
print(my_dict.get("age"))  # Output: 20
```

Modifying Dictionaries

- **Update a Value:** `my_dict["age"] = 21`
- **Add a New Key-Value Pair:** `my_dict["email"] = "ashish@example.com"`
- **Remove Items:** `del my_dict["city"]` or `my_dict.pop("age")`

Dictionary Keys and Values

You can get all the keys or values in a dictionary:

```
 print(my_dict.keys())    # Output: dict_keys(['name', 'age', 'city'])
print(my_dict.values())  # Output: dict_values(['Ashish', 20, 'Delhi'])
```

Dictionary Iteration

You can loop through dictionaries using `for` loops:

```
 for key in my_dict:
    print(key, ":", my_dict[key])

# Output:
# name : Ashish
# age : 20
# city : Delhi
```

Nested Dictionaries

Dictionaries can contain other dictionaries as values, allowing complex data storage:

```
 student = {
    "name": "Ashish",
    "marks": {"math": 95, "science": 90}
}

print(student["marks"]["math"])  # Output: 95
```

Conclusion

Python dictionaries are extremely useful for storing and managing structured data. With unique keys and flexible values, dictionaries provide fast access and efficient data manipulation, making them one of the most important data types in Python programming.

Python If.....Else

In Python, **if-else** statements are used to make decisions in your programs. They allow the program to execute certain code only if a specific condition is `True`, and optionally execute another block of code if the condition is `False`.

Basic If Statement

The `if` statement runs a block of code only if a condition is True:

```
 age = 18
if age >= 18:
    print("You are an adult.")
```

If-Else Statement

The `else` block runs if the `if` condition is False:

```
 age = 16
if age >= 18:
    print("You are an adult.")
else:
    print("You are a minor.")
```

If-Elif-Else Statement

Use `elif` (else if) to check multiple conditions:

```
 marks = 75

if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 50:
    print("Grade: C")
else:
    print("Grade: F")
```

Nested If Statements

You can place an `if` statement inside another `if` statement:

```
 age = 20
citizen = True

if age >= 18:
    if citizen:
        print("You are eligible to vote.")
    else:
        print("You must be a citizen to vote.")
else:
    print("You are not eligible to vote.")
```

Python If Else Best Practices

- Keep conditions simple and readable.
- Use meaningful variable names for clarity.
- Use `elif` instead of multiple `if` statements for better performance.

Conclusion

Python **if-else** statements are essential for controlling the flow of your program. They allow you to make decisions based on conditions and handle multiple scenarios effectively. Mastering if-else statements is crucial for writing intelligent and responsive Python programs.

Python Loops

In Python, **loops** are used to execute a block of code repeatedly until a condition is met. Loops help automate repetitive tasks, making your programs more efficient and concise.

1. While Loop

The `while` loop repeats a block of code as long as a condition is `True`:

```
 count = 1
while count <= 5:
    print("Count:", count)
    count += 1  # increment count
```

2. For Loop

The `for` loop is used to iterate over a sequence like a list, tuple, string, or range:

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

# Using range()
for i in range(5):
    print(i)  # Output: 0 1 2 3 4
```

Loop Control Statements

- **break:** Exits the loop immediately.
- **continue:** Skips the current iteration and continues with the next one.
- **pass:** Does nothing; acts as a placeholder.

```
 # Break example
for i in range(10):
    if i == 5:
        break
    print(i)  # Output: 0 1 2 3 4

# Continue example
for i in range(5):
    if i == 2:
        continue
    print(i)  # Output: 0 1 3 4
```

Nested Loops

You can use loops inside another loop:

```
 for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

Conclusion

Python loops are powerful tools for repeating tasks efficiently. By mastering `for` and `while` loops along with control statements like `break` and `continue`, you can write dynamic and efficient programs.

Python Functions

In Python, a **function** is a block of reusable code that performs a specific task. Functions help organize your code, avoid repetition, and make programs easier to read and maintain.

Defining a Function

You can define a function using the `def` keyword:

```
 def greet():
    print("Hello, welcome to Python!")

greet()  # Calling the function
```

Functions with Parameters

Functions can accept input values, called **parameters**:

```
 def greet(name):
    print("Hello, " + name + "!")

greet("Ashish")  # Output: Hello, Ashish!
```

Functions with Return Values

Functions can return a value using the `return` keyword:

```
 def add(a, b):
    return a + b

result = add(5, 3)
print(result)  # Output: 8
```

Default Parameters

You can assign default values to parameters if no value is provided:

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

greet()       # Output: Hello, Guest!
greet("Ashish")  # Output: Hello, Ashish!
```

Keyword Arguments

You can call functions using parameter names:

```
 def greet(name, age):
    print(name, "is", age, "years old.")

greet(age=20, name="Ashish")  # Output: Ashish is 20 years old.
```

Advantages of Using Functions

- Code Reusability – Write once, use multiple times.
- Modular Programming – Break complex problems into smaller pieces.
- Improved Readability – Clear and organized code.

Conclusion

Python functions are essential for creating structured, efficient, and reusable code. By learning to define functions with parameters and return values, you can build scalable and maintainable programs.

Python Lambda

In Python, a **lambda function** is a small, anonymous function. It can take any number of arguments but can only have one expression. Lambda functions are often used when you need a function for a short period of time.

Syntax

```
lambda arguments : expression
```

- `lambda` keyword is used to define the function.
- `arguments` are the inputs (like normal function parameters).
- `expression` is evaluated and returned. (It is always a single line.)

Example: Simple Lambda

```
add = lambda a, b: a + b
print(add(5, 3))  # Output: 8
```

Why Use Lambda Functions?

- When you need a simple function for a short time.
- To make code shorter and cleaner.
- Useful with functions like `map()`, `filter()`, and `sorted()`.

Lambda with Single Argument

```
square = lambda x: x * x
print(square(4))  # Output: 16
```

Lambda with Multiple Arguments

```
multiply = lambda a, b, c: a * b * c
print(multiply(2, 3, 4))  # Output: 24
```

Using Lambda with `map()`

`map()` applies a function to each element in a list.

```
nums = [1, 2, 3, 4, 5]
squares = list(map(lambda x: x**2, nums))
print(squares)  # [1, 4, 9, 16, 25]
```

Using Lambda with `filter()`

`filter()` selects items based on a condition.

```
nums = [10, 15, 20, 25, 30]
even = list(filter(lambda x: x % 2 == 0, nums))
print(even)  # [10, 20, 30]
```

Using Lambda with `sorted()`

`sorted()` can use a lambda to customize sorting.

```
students = [("Alice", 25), ("Bob", 20), ("Charlie", 23)]
# Sort by age
sorted_students = sorted(students, key=lambda x: x[1])
print(sorted_students)
# Output: [('Bob', 20), ('Charlie', 23), ('Alice', 25)]
```

Practice Questions

1. Write a lambda function to add 15 to a number.
2. Use a lambda function to check if a number is even.
3. With `map()` and lambda, convert a list of temperatures from Celsius to Fahrenheit.
4. Use `filter()` and lambda to find numbers divisible by 5 in a list.
5. Sort a list of tuples `[("apple", 50), ("banana", 20), ("cherry", 30)]` by price using a lambda.

Python Arrays

In Python, an **array** is a collection of elements of the same type. Arrays are useful when you need to store multiple values in a single variable and perform operations on them. Unlike lists, arrays are more memory-efficient and require importing the `array` module.

Creating an Array

To use arrays, you need to import the `array` module:

```
 import array as arr

numbers = arr.array('i', [1, 2, 3, 4, 5])
print(numbers)  # array('i', [1, 2, 3, 4, 5])
```

- The first argument `'i'` specifies the **type code** (here, integer). - Common type codes: `'i'` → int, `'f'` → float, `'u'` → Unicode character.

Accessing Array Elements

```
 print(numbers[0])   # 1
print(numbers[3])   # 4
```

Modifying Array Elements

```
 numbers[1] = 10
print(numbers)   # array('i', [1, 10, 3, 4, 5])
```

Array Methods

- `append(x)` → Add an element at the end
- `insert(i, x)` → Insert element at position i
- `remove(x)` → Remove first occurrence of x
- `pop()` → Remove last element
- `index(x)` → Returns first index of x
- `reverse()` → Reverse the array

```
 numbers.append(6)
numbers.insert(2, 7)
numbers.remove(3)
numbers.pop()
print(numbers)   # array('i', [1, 10, 7, 4])
```

Iterating Through an Array

```
 for num in numbers:
    print(num)
# Output:
# 1
# 10
# 7
# 4
```

Practice Questions

1. Create an array of integers from 1 to 10 and print it.
2. Add the number 15 at the end of the array.
3. Insert the number 20 at the 3rd position in the array.
4. Remove the first occurrence of 7 from the array.
5. Reverse the array and print all elements using a loop.

Python Classes / Objects

Python is an **object-oriented programming (OOP)** language. In Python, a **class** is a blueprint for creating objects, and an **object** is an instance of a class. Classes allow you to group data (attributes) and functions (methods) together.

Creating a Class

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

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")
```

Creating Objects

```
 p1 = Person("Alice", 25)
p2 = Person("Bob", 30)

p1.greet()  # Hello, my name is Alice and I am 25 years old.
p2.greet()  # Hello, my name is Bob and I am 30 years old.
```

Attributes and Methods

- **Attributes** are variables that belong to an object.
- **Methods** are functions that belong to an object.

```
 print(p1.name)  # Alice
print(p2.age)   # 30
```

The `__init__` Method

The `__init__` method is called automatically when an object is created. It is used to initialize the attributes of the object.

Practice Questions

1. Create a class `Car` with attributes `brand` and `year`, and a method that prints these details.
2. Create two objects of the `Car` class and display their details.
3. Add a method to the `Person` class that calculates the year of birth using the age attribute.
4. Modify an attribute of an object after it is created and display the updated value.
5. Create a class `Rectangle` with width and height, and a method to calculate the area.

Python Inheritance

In Python, **inheritance** is a feature of object-oriented programming that allows a class (child class) to inherit attributes and methods from another class (parent class). Inheritance helps in code reusability and makes programs easier to maintain.

Syntax

```
 class Parent:
    # parent class code

class Child(Parent):
    # child class code
```

Example: Simple Inheritance

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

    def greet(self):
        print(f"Hello, my name is {self.name}")

# Child class inherits from Person
class Student(Person):
    def __init__(self, name, age, student_id):
        super().__init__(name, age)  # call parent constructor
        self.student_id = student_id

    def display_id(self):
        print(f"My student ID is {self.student_id}")

# Creating objects
s1 = Student("Alice", 20, "S101")
s1.greet()         # Inherited method → Hello, my name is Alice
s1.display_id()    # Child class method → My student ID is S101
```

Types of Inheritance

- **Single Inheritance:** Child inherits from one parent.
- **Multiple Inheritance:** Child inherits from multiple parents.
- **Multilevel Inheritance:** Chain of inheritance (grandparent → parent → child).
- **Hierarchical Inheritance:** Multiple children inherit from the same parent.

- **Hybrid Inheritance:** Combination of two or more types of inheritance.

Practice Questions

1. Create a parent class `Vehicle` with a method `vehicle_type()`. Create a child class `Car` that inherits it and adds a method `car_brand()`.
2. Implement multilevel inheritance: `Grandparent → Parent → Child` with a method in each class and call them from the child object.
3. Create two parent classes and a child class using multiple inheritance, then call methods from both parents.
4. Modify the child class to override a method from the parent class and test the output.
5. Design a `Teacher` class that inherits from a `Person` class and adds a subject attribute. Create an object and display all details.

Python Iterators

In Python, an **iterator** is an object that allows you to traverse through all the elements of a collection (like lists, tuples, or dictionaries) one by one. Iterators are used with loops and are memory-efficient because they do not store all elements at once.

Iterator vs Iterable

- **Iterable:** Any Python object capable of returning its elements one by one (e.g., list, tuple, string).
- **Iterator:** The object returned by the `iter()` function, which is used to traverse an iterable.

Creating an Iterator

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

# Get an iterator using iter()
my_iter = iter(my_list)

# Access elements using next()
print(next(my_iter))  # 1
print(next(my_iter))  # 2
print(next(my_iter))  # 3
print(next(my_iter))  # 4
# next(my_iter)  # Raises StopIteration
```

Using Iterators in Loops

Iterators are commonly used with `for` loops, which handle the StopIteration automatically.

```
 fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Output:
# apple
# banana
# cherry
```

Creating a Custom Iterator

```
 class Counter:
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        return self

    def __next__(self):
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1

# Using the custom iterator
for num in Counter(1, 5):
    print(num)
# Output: 1 2 3 4 5
```

Practice Questions

1. Convert a list of numbers into an iterator and print all elements using `next()`.
2. Create a string iterator and print each character one by one.
3. Write a custom iterator that generates the first N even numbers.
4. Explain the difference between `iter()` and `next()` with examples.
5. Create a loop that prints elements of a tuple using an iterator instead of a `for` loop.

Python Polymorphism

In Python, **polymorphism** means the ability of a function, method, or object to take multiple forms. It allows the same operation to behave differently on different data types or classes.

Polymorphism with Functions

The same function name can work with different data types.

```
 def add(a, b):
    return a + b

print(add(5, 10))        # Output: 15 (integers)
print(add("Hello ", "World"))  # Output: Hello World (strings)
print(add([1, 2], [3, 4]))     # Output: [1, 2, 3, 4] (lists)
```

Polymorphism with Methods

Different classes can have methods with the same name, performing different tasks.

```
 class Dog:
    def speak(self):
        print("Woof!")

class Cat:
    def speak(self):
        print("Meow!")

animals = [Dog(), Cat()]

for animal in animals:
    animal.speak()
# Output:
# Woof!
# Meow!
```

Polymorphism with Inheritance

Child classes can override methods of the parent class while maintaining the same method name.

```
 class Vehicle:
    def start(self):
        print("Vehicle starts")

class Car(Vehicle):
    def start(self):
        print("Car starts")

v = Vehicle()
c = Car()
v.start()  # Vehicle starts
c.start()  # Car starts
```

Key Points

- Polymorphism increases flexibility and reusability of code.
- It allows the same interface to work for different types of objects.
- Commonly used in object-oriented programming with method overriding.

Practice Questions

1. Write a function that adds two numbers or concatenates two strings using the same function name.
2. Create two classes `Bird` and `Fish` with a method `move()`, each printing different outputs. Call `move()` for both objects.
3. Demonstrate method overriding with a parent class `Shape` and child classes `Circle` and `Square`.
4. Explain how polymorphism is useful when working with a list of different objects.
5. Create a list of objects from different classes that share a method name and call that method in a loop.

Python Scope

In Python, **scope** refers to the region of the program where a variable is accessible. Understanding scope is important to avoid naming conflicts and to know where a variable can be used.

Types of Scope in Python

- **Local Scope:** Variables defined inside a function. Accessible only within that function.
- **Global Scope:** Variables defined outside any function. Accessible anywhere in the file.
- **Enclosing Scope:** Variables in the local scope of enclosing functions (used in nested functions).
- **Built-in Scope:** Names preassigned in Python, like `print()`, `len()`, etc.

Local Scope Example

```
 def greet():
    name = "Alice"  # Local variable
    print(name)

greet()           # Output: Alice
# print(name)     # Error: name is not defined outside the function
```

Global Scope Example

```
name = "Bob"  # Global variable

def greet():
    print(name)   # Can access global variable

greet()         # Output: Bob
print(name)     # Output: Bob
```

Using `global` Keyword

To modify a global variable inside a function, use the `global` keyword.

```
x = 10

def modify():
    global x
    x = 20

modify()
print(x)  # Output: 20
```

Enclosing (Nested) Scope Example

```
def outer():
    x = "Outer variable"
    def inner():
        print(x)  # Accesses variable from enclosing function
    inner()

outer()  # Output: Outer variable
```

Practice Questions

1. Define a local variable inside a function and try to print it outside the function. What happens?
2. Create a global variable and modify it inside a function using the `global` keyword.
3. Write a nested function and access the variable from the outer function.
4. Explain the difference between local and global scope with an example.
5. List some Python built-in functions and explain their scope.

Python Modules

In Python, a **module** is a file containing Python code, such as functions, classes, or variables, that can be imported and used in other Python programs. Modules help organize code, make it reusable, and improve readability.

Creating a Module

Create a Python file (e.g., `mymodule.py`) with functions or variables:

```
# mymodule.py
def greet(name):
    print(f"Hello, {name}!")

pi = 3.14159
```

Using a Module

```
# main.py
import mymodule

mymodule.greet("Alice")  # Output: Hello, Alice!
print(mymodule.pi)       # Output: 3.14159
```

Importing Specific Items

```
from mymodule import greet

greet("Bob")  # Output: Hello, Bob!
```

Renaming Modules

```
import mymodule as mm
```

```
mm.greet("Charlie")  # Output: Hello, Charlie!
```

Built-in Modules

Python provides many built-in modules, such as:

- `math` – mathematical operations
- `random` – generate random numbers
- `datetime` – work with dates and times
- `os` – operating system functionality
- `sys` – system-specific parameters and functions

```
 import math
print(math.sqrt(16))  # Output: 4.0

import random
print(random.randint(1, 10))  # Output: Random number between 1 and 10
```

Practice Questions

1. Create a module `calculator.py` with functions for addition, subtraction, multiplication, and division. Import and use it in another file.
2. Import only the `sqrt()` function from the `math` module and calculate the square root of 49.
3. Use the `random` module to generate a random number between 50 and 100.
4. Rename a module during import and use its function.
5. List 5 other built-in modules in Python and explain their purpose.

Python Dates

In Python, the **datetime** module is used to work with dates and times. It allows you to create, manipulate, and format date and time objects easily.

Importing the datetime Module

```
 import datetime
```

Getting the Current Date and Time

```
 now = datetime.datetime.now()
print(now)            # Output: 2025-09-29 13:45:12.345678
print(now.year)       # Current year
print(now.month)      # Current month
print(now.day)        # Current day
```

Creating a Specific Date

```
 my_date = datetime.datetime(2025, 12, 25)
print(my_date)        # Output: 2025-12-25 00:00:00
```

Formatting Dates

You can format dates using the `strftime()` method:

```
 today = datetime.datetime.now()
print(today.strftime("%Y-%m-%d"))     # 2025-09-29
print(today.strftime("%d/%m/%Y"))     # 29/09/2025
print(today.strftime("%A, %B %d"))    # Monday, September 29
```

Timedelta: Date Arithmetic

The `timedelta` class allows you to add or subtract days, seconds, or other time intervals.

```
 from datetime import datetime, timedelta

today = datetime.now()
tomorrow = today + timedelta(days=1)
yesterday = today - timedelta(days=1)

print("Tomorrow:", tomorrow)
print("Yesterday:", yesterday)
```

Practice Questions

1. Get the current date and time and print only the year, month, and day separately.
2. Create a datetime object for your birthday and print it.
3. Format the current date as `DD-MM-YYYY` and `Month Day, Year`.

4. Calculate the date 10 days from today using `timedelta`.
5. Write a program to calculate the number of days between two dates.

Python Math Module

In Python, the **math** module provides a wide range of mathematical functions and constants. It is commonly used for advanced mathematical operations like square root, trigonometry, factorials, and more.

Importing the math Module

```
import math
```

Common Math Functions

- `math.sqrt(x)` – Returns the square root of `x`.
- `math.pow(x, y)` – Returns `x` raised to the power `y`.
- `math.ceil(x)` – Returns the smallest integer greater than or equal to `x`.
- `math.floor(x)` – Returns the largest integer less than or equal to `x`.
- `math.factorial(x)` – Returns the factorial of `x`.
- `math.gcd(x, y)` – Returns the greatest common divisor of `x` and `y`.
- `math.sin(x), math.cos(x), math.tan(x)` – Trigonometric functions (x in radians).

Examples

```
 import math

print(math.sqrt(16))      # 4.0
print(math.pow(2, 3))     # 8.0
print(math.ceil(4.2))     # 5
print(math.floor(4.8))    # 4
print(math.factorial(5))  # 120
print(math.gcd(12, 18))   # 6
print(math.sin(math.pi/2))# 1.0
```

Constants in Math Module

- `math.pi` – Value of π (3.14159...)
- `math.e` – Value of Euler's number e (2.71828...)

Practice Questions

1. Calculate the square root of 49 using the math module.
2. Find 3 to the power of 4 using `math.pow()`.
3. Use `math.ceil()` and `math.floor()` on 7.3 and 7.7 and print results.
4. Calculate factorial of 6 using `math.factorial()`.
5. Compute the sine of 90 degrees (Hint: convert degrees to radians first).

Python JSON

In Python, **JSON (JavaScript Object Notation)** is a popular data format used for exchanging data between a server and a client. Python provides the `json` module to work with JSON data, allowing you to convert between Python objects and JSON strings.

Importing the json Module

```
import json
```

Converting Python Object to JSON

Use `json.dumps()` to convert Python objects to JSON strings:

```
 import json

data = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
}

json_string = json.dumps(data)
print(json_string)
# Output: {"name": "Alice", "age": 25, "city": "New York"}
```

Converting JSON to Python Object

Use `json.loads()` to parse a JSON string back into a Python object:

```
json_data = '{"name": "Bob", "age": 30, "city": "London"}'
python_obj = json.loads(json_data)
print(python_obj)
# Output: {'name': 'Bob', 'age': 30, 'city': 'London'}
print(python_obj["name"])  # Bob
```

Reading JSON from a File

```
import json

with open("data.json", "r") as file:
    data = json.load(file)
print(data)
```

Writing JSON to a File

```
import json

data = {"name": "Charlie", "age": 28, "city": "Paris"}

with open("data.json", "w") as file:
    json.dump(data, file)
```

Practice Questions

1. Convert a Python dictionary `{"fruit": "apple", "quantity": 10}` to a JSON string.
2. Parse the JSON string `'{"name": "Eve", "age": 22}'` into a Python dictionary and print the age.
3. Write a Python list of dictionaries to a JSON file.
4. Read the JSON file created above and print each item.
5. Explain the difference between `json.load()` and `json.loads()`.

Python RegEx (Regular Expressions)

In Python, **RegEx (Regular Expressions)** is a powerful tool for searching, matching, and manipulating strings. Python provides the `re` module to work with regular expressions.

Importing the re Module

```
import re
```

Basic Functions in re Module

- `re.match(pattern, string)` – Checks if the string starts with the pattern.
- `re.search(pattern, string)` – Searches the entire string for the pattern.
- `re.findall(pattern, string)` – Returns a list of all matches in the string.
- `re.split(pattern, string)` – Splits the string by the pattern.
- `re.sub(pattern, replacement, string)` – Replaces occurrences of the pattern with the replacement.

Example: Searching a Pattern

```
import re

text = "My phone number is 123-456-7890"
pattern = r"\d{3}-\d{3}-\d{4}"

match = re.search(pattern, text)
if match:
    print("Found:", match.group())
# Output: Found: 123-456-7890
```

Example: Finding All Matches

```
text = "Email me at alice@example.com or bob@example.com"
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"
emails = re.findall(pattern, text)
print(emails)
# Output: ['alice@example.com', 'bob@example.com']
```

Example: Replacing Text

```
text = "I love cats"
new_text = re.sub(r"cats", "dogs", text)
print(new_text)
# Output: I love dogs
```

Practice Questions

1. Write a regular expression to match all phone numbers in the format XXX-XXX-XXXX in a string.
2. Find all email addresses in a given text using RegEx.
3. Replace all occurrences of "Python" with "Java" in a string using `re.sub()`.
4. Split a string by one or more spaces using `re.split()`.
5. Explain the difference between `re.match()` and `re.search()` with examples.

Python PIP

PIP stands for **"Python Package Installer"**. It is a tool used to install and manage Python packages from the Python Package Index (PyPI). Using PIP, you can easily add external libraries to your Python projects.

Checking if PIP is Installed

```
 pip --version
# Output example: pip 23.1.2 from /usr/lib/python3/site-packages/pip (python 3.10)
```

Installing a Package

```
 pip install package_name
# Example:
pip install requests
```

Upgrading a Package

```
 pip install --upgrade package_name
# Example:
pip install --upgrade requests
```

Uninstalling a Package

```
 pip uninstall package_name
# Example:
pip uninstall requests
```

Listing Installed Packages

```
 pip list
```

Searching for Packages

```
 pip search keyword
# Example:
pip search flask
```

Practice Questions

1. Check if PIP is installed on your system and note the version.
2. Install the `numpy` package using PIP.
3. Upgrade the `numpy` package to the latest version.
4. Uninstall the `numpy` package using PIP.
5. List all installed packages and find the version of `requests`.

Python Try...Except

In Python, the **try...except** block is used to handle exceptions (errors) gracefully. Instead of stopping the program when an error occurs, you can catch the error and take appropriate action.

Basic Syntax

```
 try:
    # Code that might raise an exception
except ExceptionType:
    # Code to handle the exception
```

Example: Handling Division by Zero

```
 try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
# Output: Cannot divide by zero!
```

Handling Multiple Exceptions

```
try:
    x = int(input("Enter a number: "))
    y = 10 / x
except ValueError:
    print("Invalid input! Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Using `else` and `finally`

- `else` block executes if no exception occurs.
- `finally` block executes regardless of whether an exception occurred or not.

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful!")
finally:
    print("Execution complete.")
# Output:
# Division successful!
# Execution complete.
```

Practice Questions

1. Write a program that catches a `ValueError` when a user enters a non-integer value.
2. Handle `ZeroDivisionError` when dividing two numbers input by the user.
3. Create a try...except block to handle multiple exceptions in one program.
4. Use `finally` to print a message after executing a try...except block, regardless of errors.
5. Write a program that reads a file and handles `FileNotFoundError` gracefully.

Python User Input

In Python, the **input()** function is used to take input from the user. By default, the input is taken as a string, but it can be converted into other data types as needed.

Basic Syntax

```
variable = input("Enter something: ")
```

Example: Taking String Input

```
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

Example: Taking Integer Input

```
age = int(input("Enter your age: "))
print("You are", age, "years old.")
```

Example: Taking Float Input

```
price = float(input("Enter the price: "))
print("The price is", price)
```

Using Input in Calculations

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
sum = a + b
print("Sum:", sum)
```

Practice Questions

1. Take a string input from the user and print it in uppercase.
2. Take two integers as input and print their sum, difference, and product.
3. Take a float input for the radius of a circle and calculate its area.

4. Write a program that asks for the user's name and age and prints a greeting message.
5. Take a number input and check if it is even or odd.

Python String Formatting

In Python, **string formatting** allows you to create strings dynamically by inserting variables or expressions into them. It makes your output readable and customizable.

1. Using the `%` Operator

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

2. Using the `str.format()` Method

```
 name = "Bob"
age = 30
print("My name is {} and I am {} years old.".format(name, age))
# Output: My name is Bob and I am 30 years old.

# Using positional arguments
print("I am {1} years old and my name is {0}.".format(name, age))
```

3. Using f-Strings (Python 3.6+)

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

# Using expressions
print(f"Next year, I will be {age + 1} years old.")
# Output: Next year, I will be 29 years old.
```

4. Formatting Numbers

```
 pi = 3.14159265
print("Pi rounded to 2 decimal places: {:.2f}".format(pi))
# Output: Pi rounded to 2 decimal places: 3.14

print(f"Pi rounded to 3 decimal places: {pi:.3f}")
# Output: Pi rounded to 3 decimal places: 3.142
```

Practice Questions

1. Print your name and age using the `%` operator.
2. Use `str.format()` to print a sentence with your favorite color and hobby.
3. Create an f-string that shows the result of 5 + 10 in a sentence.
4. Format the number 123.456789 to display only 2 decimal places using f-strings.
5. Use string formatting to align text in a table-like output for three items and their prices.

Python File Handling

In Python, **file handling** allows you to create, read, update, and delete files. The built-in `open()` function is used to work with files in different modes.

Opening a File

```
 # Open a file in read mode
file = open("example.txt", "r")  # "r" = read mode
content = file.read()
print(content)
file.close()
```

File Modes

- `"r"` – Read (default), file must exist
- `"w"` – Write, creates a new file or truncates existing file
- `"a"` – Append, adds content to the end of the file
- `"x"` – Create, creates a new file, fails if file exists
- `"b"` – Binary mode (used with other modes, e.g., "rb", "wb")

Writing to a File

```
 file = open("example.txt", "w")
file.write("Hello, World!\n")
file.write("Python File Handling")
file.close()
```

Appending to a File

```
 file = open("example.txt", "a")
file.write("\nThis is appended text.")
file.close()
```

Using `with` Statement

The `with` statement automatically closes the file after its block finishes:

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
```

Reading Line by Line

```
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Remove newline characters
```

Practice Questions

1. Create a file named `data.txt` and write your name and age into it.
2. Read the content of `data.txt` and print it line by line.
3. Append a new line to `data.txt` containing your favorite hobby.
4. Use the `with` statement to read and print the file content.
5. Create a binary file and write some bytes into it, then read the bytes back.

Python Requests

In Python, the **Requests** library is used to send HTTP requests easily. It allows you to interact with web services, APIs, and websites by sending GET, POST, PUT, DELETE, and other HTTP requests.

Installing Requests

```
pip install requests
```

Making a GET Request

```
 import requests

response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.status_code)  # 200
print(response.text)         # JSON data as string
print(response.json())       # JSON data as Python dictionary
```

Making a POST Request

```
 import requests

data = {"title": "foo", "body": "bar", "userId": 1}
response = requests.post("https://jsonplaceholder.typicode.com/posts", json=data)
print(response.status_code)  # 201
print(response.json())       # Response data
```

Adding Headers

```
 headers = {"Authorization": "Bearer YOUR_TOKEN"}
response = requests.get("https://api.example.com/data", headers=headers)
print(response.json())
```

Handling Query Parameters

```
 params = {"userId": 1}
response = requests.get("https://jsonplaceholder.typicode.com/posts", params=params)
print(response.json())
```

Practice Questions

1. Install the Requests library and make a GET request to `https://api.github.com`.
2. Send a POST request to `https://jsonplaceholder.typicode.com/posts` with custom data and print the response.
3. Use query parameters to fetch posts for a specific user from the JSONPlaceholder API.
4. Add custom headers to a GET request and print the JSON response.
5. Explain the difference between `response.text` and `response.json()`.

Python BeautifulSoup

BeautifulSoup is a Python library used for web scraping. It helps you parse HTML and XML documents and extract useful information like headings, paragraphs, links, and more. It works well with the `requests` library to fetch webpage content and then process it.

Finding Elements

```
 # Find first <h1> tag
h1_tag = soup.find("h1")
print(h1_tag.text)

# Find all <p> tags
paragraphs = soup.find_all("p")
for p in paragraphs:
    print(p.text)
```

Accessing Attributes

```
 link = soup.find("a")
print(link["href"])  # Get href attribute of the first <a>
```

Practice Questions

1. Install BeautifulSoup and requests library, and fetch the title of `https://example.com`.
2. Extract and print all paragraph texts from a given webpage.
3. Find all hyperlinks (`<a>` tags) and print their URLs.
4. Parse an HTML string and extract the text from a specific tag.
5. Explain the difference between `find()` and `find_all()` in BeautifulSoup.

Frequently Asked Questions

What is stress testing in networking ?

Stress testing in networking is like pushing your network to its limits to see how much traffic or load it can handle before it starts slowing down or fails. It helps you find weak spots, so you can fix issues before they affect real users. Think of it as a "toughness test" for your network!

What are the 3 types of stress test ?

The three main types of stress tests are:

Application Stress Testing – Checks how well a software app handles heavy loads or extreme conditions.

System Stress Testing – Tests the entire system (hardware + software) under extreme stress to find bottlenecks.

Network Stress Testing – Simulates high traffic to see how the network performs under pressure.

Which tool is best for stress testing ?

The best tool for stress testing depends on your needs, but here are some top options:

NetFloodX – If you're focusing on network-level testing, NetFloodX is a smart, efficient choice.

Apache JMeter – Ideal for web apps and network performance testing.

LoadRunner – Enterprise-grade tool for full system load and stress testing.

Locust – A flexible, Python-based tool for writing custom stress test scenarios.

Wireshark (with traffic generators) – Best for analyzing network behavior under stress.

How can i test my network strength ?

You can easily test your network strength using our tool NetFloodX. Just launch the tool, enter your target IP or domain, set the traffic level, and start the test. NetFloodX floods the network with controlled traffic to check how much load it can handle before slowing down or dropping packets. It’s a quick and powerful way to spot weak points in your network! Want a step-by-step tutorial?.

How can I protect my systems against the types of attacks NetFloodX simulates?

Implement rate limiting, traffic filtering, connection timeout adjustments, and consider DDoS protection services. Regular testing with tools like NetFloodX helps identify and address vulnerabilities.

Does NetFloodX work on mobile devices?

NetFloodX is primarily designed for desktop operating systems. While it might work on rooted/jailbroken mobile devices with the right dependencies, this is not officially supported.

-->