Introduction to Python

Words of the Author

> “Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another’s code; too little and expressiveness is endangered.”
> — Guido van Rossum

Concept Overview

Python is a high-level, interpreted programming language known for its simplicity and readability. Created by Guido van Rossum in 1991, it has become a popular choice for both beginners and experienced developers.

Key Concepts

- **Open Source**: Python is free to use and distribute, with a large community contributing to its development.
- **Simple and Readable**: Python's syntax is designed to be clear and straightforward, making it easier to learn and write code.
- **Minimalist**: The language emphasizes simplicity and minimalism, allowing developers to focus on problem-solving rather than complex syntax.
- **Cross-Platform Compatible**: Python can run on various operating systems, including Windows, macOS, and Linux.
- **Dynamically Typed Variables**: Variables in Python do not require an explicit declaration, and their types can change at runtime.
- **Object-Oriented**: Python supports object-oriented programming, allowing for the creation of classes and objects.
- **High-Level**: Python abstracts complex details of the computer, enabling developers to focus on programming logic rather than hardware specifics.

Important Definitions

- **Interpreted Language**: Python code is executed line by line, which makes debugging easier but may impact performance compared to compiled languages.

Execution of Programming Languages

Key Concepts

- **Translation to Machine Code**: The process of converting human-readable source code into machine code that the CPU can understand and execute.

Types of Execution

1. Compiler

- **Definition**: A compiler translates the entire source code into machine code before the program is run.
- **Examples**:
    - C
    - C++
    - C#

2. Interpreter

- **Definition**: An interpreter translates source code line-by-line and executes each line as it goes, which allows for immediate feedback but can be slower than compiled languages.
- **Examples**:
    - Python
    - JavaScript
    - Ruby
    - PHP

3. Hybrid / JIT (Just-In-Time Compilation)

- **Definition**: Combines aspects of both compilation and interpretation. The source code is translated to bytecode, which is then interpreted at runtime, often using a virtual machine (e.g., JVM for Java).
- **Examples**:
    - Java

Additional Notes

- Compiled languages typically have faster execution times due to the pre-compiled machine code, while interpreted languages offer greater flexibility and ease of debugging.
- Hybrid approaches aim to combine the benefits of both methods, optimizing performance and flexibility.

Compiler vs. Interpreter

Concept Overview

Compilers and interpreters are tools used to translate high-level programming languages into machine code, but they do so in different ways and with different implications for program execution.

Key Differences

Feature	Compiler	Interpreter
Translation	Translates the entire program at once before execution.	Translates and executes code line-by-line.
Execution Speed	Generally faster execution since the program is pre-compiled into machine code.	Slower execution due to real-time translation.
Error Detection	Errors are detected after the entire code is compiled, making it easier to spot certain issues.	Errors are detected line-by-line, allowing immediate feedback but potentially making debugging harder.
Output	Produces a standalone executable file.	Does not produce an intermediate output; executes directly.
Memory Usage	Typically uses more memory during compilation but less during execution.	May use more memory during execution due to ongoing translation.
Examples	C, C++, C#	Python, JavaScript, Ruby, PHP

Use Cases

- **Compilers**: Suitable for applications requiring high performance, such as system software and game development.
- **Interpreters**: Useful for scripting, rapid prototyping, and situations where flexibility and ease of debugging are prioritized.

Additional Notes

- Some languages (like Java) use both compilation and interpretation, compiling to bytecode which is then interpreted at runtime by a virtual machine (JVM).

Python Data Types

Numeric Types

1. Integer (`int`)

- **Description**: Whole numbers without a fractional part.
- **Example**: `x = 42`, `y = -7`

2. Floating-Point (`float`)

- **Description**: Numbers that contain a decimal point, representing real numbers.
- **Example**: `a = 3.14`, `b = -0.001`

3. Complex (`complex`)

- **Description**: Numbers that have a real and an imaginary part, represented as `a + bj` where `a` is the real part and `b` is the imaginary part.
- **Example**: `c = 2 + 3j`

Sequence Types

1. String (`str`)

- **Description**: A sequence of characters, used to represent text.
- **Example**: `name = "Alice"`

2. List (`list`)

- **Description**: An ordered, mutable collection of items that can be of different data types.
- **Example**: `fruits = ['apple', 'banana', 'cherry']`

3. Tuple (`tuple`)

- **Description**: An ordered, immutable collection of items that can be of different data types.
- **Example**: `coordinates = (10.0, 20.0)`

4. Set (`set`)

- **Description**: An unordered collection of unique items. Sets are mutable and do not allow duplicate values.
- **Example**: `unique_numbers = {1, 2, 3, 4, 5}`

5. Range (`range`)

- **Description**: Represents a sequence of numbers, commonly used for iteration. It generates numbers on demand and is often used in loops.
- **Example**: `numbers = range(1, 10)`

Mapping Type

1. Dictionary (`dict`)

- **Description**: A collection of key-value pairs, where each key is unique and maps to a value. Dictionaries are mutable.
- **Example**: `student = {'name': 'John', 'age': 21}`

None Type

1. None (`NoneType`)

- **Description**: Represents the absence of a value or a null value.
- **Example**: `result = None`

Boolean Type

1. Boolean (`bool`)

- **Description**: Represents one of two values: `True` or `False`. Used for conditional logic.
- **Example**: `is_active = True`

Here's a concise Markdown snippet covering variables in Python:

Variables in Python

Overview

Variables are placeholders that allow you to store, modify, and retrieve data during program execution.

Key Concepts

- **Variable Assignment**: Assigning a value to a variable using the `=` operator.

```
x = 10
```

- **Dynamic Typing**: Variables can change type at runtime.

```
x = 5        # integer
x = "Hello"  # now a string
```

- **Reassigning Variables**: Variables can be reassigned any time.

```
count = 1
count = count + 1  # count is now 2
```

- **Multiple Assignments**: Assign values to multiple variables in one line.

```
a, b, c = 1, 2, 3
```

- **Constants**: Use uppercase names to indicate that a variable should not change.

```
PI = 3.14159
```

- **Dunder Variables**: Special variables with predefined meanings.

```
__init__  # Constructor method
__str__   # String representation method
```

Here's a concise Markdown snippet covering naming conventions for variables in Python:

Naming Conventions in Python

Key Rules

- **Start with a Letter or Underscore**: Variable names must begin with a letter (a-z, A-Z) or an underscore (_), but not a number.

```
valid_name = 10
_valid_name = 20
```

- **Allowed Characters**: Variable names can include letters, numbers, and underscores.

```
variable_name_1 = "Hello"
```

- **Case Sensitivity**: Variable names are case-sensitive; `myVar` and `myvar` are considered different variables.

```
myVar = 1
myvar = 2  # Different variable
```

- **Avoid Keywords**: Do not use Python keywords (like `if`, `for`, `while`) as variable names to prevent conflicts.

```
# Invalid
if = 5  # This will cause an error
```

Operators in Python

Types of Operators

1. Arithmetic Operators

- **Description**: Used to perform mathematical operations.
- **Operators**: `+`, `-`, `*`, `/`, `%`, `//`, `**`
- **Example**:

```
result = (5 + 3) * 2  # Follows BODMAS
```

2. Assignment Operators

- **Description**: Used to assign values to variables.
- **Operators**: `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`
- **Example**:

```
x = 5
x += 3  # Equivalent to x = x + 3
```

4. Logical Operators

- **Description**: Used to combine Boolean values.
- **Operators**: `and`, `or`, `not`
- **Example**:

```
if (True and False):  # Evaluates to False
    print("True")
```

5. Comparison (Relational) Operators

- **Description**: Used to compare values.
- **Operators**: `==`, `!=`, `>`, `<`, `>=`, `<=`
- **Example**:

```
is_equal = (5 == 5)  # True
```

6. Membership Operators

- **Description**: Used to test if a value is in a sequence (like a list or string).
- **Operators**: `in`, `not in`
- **Example**:

```
fruits = ['apple', 'banana']
is_in = 'apple' in fruits  # True
```

7. Identity Operators

- **Description**: Used to compare the memory locations of two objects.
- **Operators**: `is`, `is not`
- **Example**:

```
a = [1, 2, 3]
b = a
are_same = (a is b)  # True
```

8. Ternary Operator

- **Description**: A shorthand for an if-else statement.
- **Syntax**: `value_if_true if condition else value_if_false`
- **Example**:

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

Here's a concise Markdown snippet summarizing the bitwise operators in Python:

Bitwise Operators in Python

Operators

- **Bitwise AND (`&`)**: Returns 1 if both bits are 1.

```
result = a & b  # e.g., 5 & 3 = 1
```

- **Bitwise OR (`|`)**: Returns 1 if at least one bit is 1.

```
result = a | b  # e.g., 5 | 3 = 7
```

- **Bitwise XOR (`^`)**: Returns 1 if bits are different.

```
result = a ^ b  # e.g., 5 ^ 3 = 6
```

- **Bitwise NOT (`~`)**: Inverts all bits.

```
result = ~a  # e.g., ~5 = -6
```

- **Left Shift (`<<`)**: Shifts bits to the left (multiplies by 2).

```
result = a << 1  # e.g., 5 << 1 = 10
```

- **Right Shift (`>>`)**: Shifts bits to the right (divides by 2).

```
result = a >> 1  # e.g., 5 >> 1 = 2
```

Here's a concise Markdown snippet summarizing type casting in Python:

Type Casting in Python

Type Casting Functions

- `int()`: Converts a value to an integer.

```
num = int("10")  # Converts string to integer
```

- `float()`: Converts a value to a floating-point number.

```
num = float("10.5")  # Converts string to float
```

- `str()`: Converts a value to a string.

```
text = str(100)  # Converts integer to string
```

- `bool()`: Converts a value to a boolean (True or False).

```
flag = bool(0)  # Converts 0 to False
```

- `list()`: Converts an iterable (like a string or tuple) to a list.

```
my_list = list("abc")  # Converts string to list of characters
```

- `tuple()`: Converts an iterable to a tuple.

```
my_tuple = tuple([1, 2, 3])  # Converts list to tuple
```

- `set()`: Converts an iterable to a set (removes duplicates).

```
my_set = set([1, 2, 2, 3])  # Converts list to set
```

- `dict()`: Converts a sequence of key-value pairs into a dictionary.

```
my_dict = dict([(1, 'a'), (2, 'b')])  # Converts list of tuples to dictionary
```

Input/Output in Python

Overview

Input/output serves as the interface between end users and programs, allowing for interaction through the console.

Input

- **Description**: Reads a line of text from the user and returns it as a string.
- **Syntax**:

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

Output

- **Basic Output**: Use `print()` to display text.

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

- **Multiple Arguments**: Print multiple items in one statement.

```
print("Value 1:", 10, "Value 2:", 20)
```

- **Custom Separators**: Use `sep` to change the separator between arguments.

```
print("Hello", "World", sep=", ")
```

- **Custom Endings**: Use `end` to change what is printed at the end.

```
print("Hello", end="!")  # Output: Hello!
```

- **Formatted Output**: Use `format()` or f-strings for formatting.

```
name = "Alice"
age = 30
print("My name is {} and I am {} years old.".format(name, age))
# or using f-string
print(f"My name is {name} and I am {age} years old.")
```

- **Truncate Float Numbers**: Format float numbers to a specific number of decimal places.

```
var_name = 3.14159
print(f"{var_name:.2f}")  # Output: 3.14
```

Conditional Statements in Python

Key Concepts

1. `if` Statement

- **Description**: Executes a block of code if the condition is true.
- **Syntax**:

```
    if condition:
        # Code to execute if condition is true
```

2. `else` Statement

- **Description**: Executes a block of code if the preceding `if` condition is false.
- **Syntax**:

```
    if condition:
        # Code if condition is true
    else:
        # Code if condition is false
```

3. `elif` Statement

- **Description**: Allows checking multiple conditions. It stands for "else if."
- **Syntax**:

```
    if condition1:
        # Code if condition1 is true
    elif condition2:
        # Code if condition2 is true
    else:
        # Code if both conditions are false
```

4. Nested Conditionals

- **Description**: You can place an `if` statement inside another `if` statement.
- **Example**:

```
    if condition1:
        if condition2:
            # Code if both conditions are true
```

Example

```
age = 20
if age >= 18:
    print("You are an adult.")
elif age >= 13:
    print("You are a teenager.")
else:
    print("You are a child.")
```

Here's a concise Markdown snippet summarizing loops in Python:

Loops in Python

Types of Loops

1. For Loop

- **Description**: Executes a block of code a specific number of times, iterating over a sequence (like a list, tuple, string, or range).

- **Syntax**:

```
for item in sequence:
    # Code to execute for each item
```

- **Example**:

```
for i in range(5):  # Iterates from 0 to 4
    print(i)
```

2. While Loop

- **Description**: Repeatedly executes a block of code as long as a specified condition is true.
- **Syntax**:

```
while condition:
    # Code to execute while condition is true
```

- **Example**:

```
count = 0
while count < 5:
    print(count)
    count += 1
```

Loop Control Statements in Python

Types of Control Statements

1. `break`

- **Description**: Exits the loop completely, terminating further iterations.
- **Usage**: Typically used when a certain condition is met.
- **Example**:

```
for i in range(10):
    if i == 5:
        break  # Exits the loop when i equals 5
    print(i)
```

2. `continue`

- **Description**: Skips the current iteration of the loop and proceeds to the next iteration.
- **Usage**: Used when you want to bypass certain conditions within the loop.
- **Example**:

```
for i in range(5):
    if i == 2:
        continue  # Skips the iteration when i equals 2
    print(i)
```

Here's a concise Markdown snippet summarizing nested loops and the use of `else` clauses in Python:

Nested Loops in Python

Types of Nested Loops

1. Nested For Loop

- **Description**: A for loop inside another for loop, typically used for iterating over multi-dimensional data structures.
- **Example**:

```
for i in range(3):  # Outer loop
    for j in range(2):  # Inner loop
        print(f"i: {i}, j: {j}")
```

2. Nested While Loop

- **Description**: A while loop inside another while loop, useful for repetitive conditions within a larger repetitive structure.
- **Example**:

```
i = 0
while i < 3:  # Outer loop
    j = 0
    while j < 2:  # Inner loop
        print(f"i: {i}, j: {j}")
        j += 1
    i += 1
```

Else Clause in Loops

- **Description**: An `else` clause can be added to loops, which executes when the loop terminates normally (not via `break`).
- **Usage**: Useful for actions that should occur after the loop completes.
- **Example**:

```
for i in range(3):
    print(i)
else:
    print("Loop finished without interruption.")
```

Here's a concise Markdown snippet summarizing loops with an `else` clause in Python:

Loops with Else

1. For Loop with Else

- **Description**: The `else` block runs after the loop completes all iterations without encountering a `break`.
- **Example**:

```
for i in range(5):
    print(i)
else:
    print("For loop finished without interruption.")
```

2. While Loop with Else

- **Description**: Similar to the `for` loop, the `else` block runs after the `while` loop completes normally.
- **Example**:

```
count = 0
while count < 5:
    print(count)
    count += 1
else:
    print("While loop finished without interruption.")
```

3. Using Break with Else

- **Description**: If a `break` statement is encountered, the `else` block will not execute.
- **Example**:

```
for i in range(5):
    if i == 3:
        break
    print(i)
else:
    print("This will not print if the loop is broken.")
```

Strings in Python

Overview

Strings are sequences of characters used to represent text. They can be created using single, double, or triple quotes.

Properties

- **Immutable**: Strings cannot be changed after creation. Operations return new strings.

String Creation

- **Single Quotes**: `'quote'`
- **Double Quotes**: `"double quote"`
- **Triple Quotes**: `"""triple quotes"""` (useful for multi-line strings)

String Indexing

- **Access Characters**: Use indexing to access individual characters.

```
str = "Hello"
char = str[1]  # 'e'
```

String Slicing

- **Extract Substrings**: Use slicing to get a substring from the string.

```
substring = str[1:4]  # 'ell'
```

String Concatenation

- **Combine Strings**: Use the `+` operator to concatenate strings.

```
combined = "Hello" + " " + "World"  # 'Hello World'
```

String Repetition

- **Repeat Strings**: Use the `*` operator to repeat a string.

```
repeated = "Hi! " * 3  # 'Hi! Hi! Hi! '
```

String Length

- **Get Length**: Use `len()` to find the length of a string.

```
length = len(str)  # 5
```

String Membership

- **Check Presence**: Use `in` or `not in` to check for substring presence.

```
exists = "H" in str  # True
```

String Methods

- **Common Methods**:
  - `upper()`: Converts to uppercase.

```
str.upper()  # 'HELLO'
```

  - `lower()`: Converts to lowercase.

```
str.lower()  # 'hello'
```

  - `capitalize()`: Capitalizes the first letter.

```
str.capitalize()  # 'Hello'
```

  - `title()`: Capitalizes the first letter of each word.

```
str.title()  # 'Hello'
```

String Operations in Python

1. Stripping Whitespace

- **`strip()`**: Removes leading and trailing whitespace.

```
text = "  Hello  "
stripped = text.strip()  # 'Hello'
```

- **`lstrip()`**: Removes leading whitespace.

```
left_stripped = text.lstrip()  # 'Hello  '
```

- **`rstrip()`**: Removes trailing whitespace.

```
right_stripped = text.rstrip()  # '  Hello'
```

2. Finding and Replacing

- **`find(sub)`**: Returns the index of the first occurrence of `sub`, or -1 if not found.

```
index = "Hello".find("l")  # 2
```

- **`replace(old, new)`**: Replaces all occurrences of `old` with `new`.

```
new_text = "Hello".replace("l", "p")  # 'Heppo'
```

3. Splitting and Joining

- **`split(delimiter)`**: Splits the string into a list based on the delimiter.

```
words = "Hello World".split()  # ['Hello', 'World']
```

- **`join(iterable)`**: Joins elements of an iterable into a single string, using the string as a separator.

```
sentence = " ".join(['Hello', 'World'])  # 'Hello World'
```

4. Checking String Properties

- `startswith(prefix)`: Checks if the string starts with the specified prefix.

```
is_start = "Hello".startswith("He")  # True
```

- `endswith(suffix)`: Checks if the string ends with the specified suffix.

```
is_end = "Hello".endswith("lo")  # True
```

- `isalpha()`: Returns True if all characters are alphabetic.
- `isdigit()`: Returns True if all characters are digits.
- `isalnum()`: Returns True if all characters are alphanumeric.

5. String Formatting

- **f-Strings**: Format strings using `{}` for variable interpolation.

```
name = "Alice"
greeting = f"Hello, {name}!"  # 'Hello, Alice!'
```

- `format()`: Another method for formatting strings.

```
formatted = "Hello, {}".format(name)  # 'Hello, Alice!'
```

- `%` **operator**: Old-style formatting.

```
formatted = "Hello, %s" % name  # 'Hello, Alice!'
```

6. Escaping Characters

- **Backslash (`\`)**: Used to escape special characters within strings.

```
escaped = "He said, "Hello!""  # 'He said, "Hello!"'
```

Lists in Python

Overview

Lists are versatile data structures that store an ordered collection of items.

Key Characteristics

- **Mutable**: Lists can be modified after creation (e.g., items can be added, removed, or changed).
- **Heterogeneous**: Can hold elements of different data types (e.g., integers, strings, objects).
- **Duplicates**: Allows duplicate items, meaning the same value can appear multiple times.
- **Order**: Maintains the order of insertion, so the order of elements is preserved.

Example

```
my_list = [1, "apple", 3.14, "banana", 1]  # A list with mixed data types and duplicates
```

List Operations in Python

Basic List Creation

- **Creation**:

```
empty_list = []
```

Accessing List Elements

- **Indexing**: Access elements using an index.

```
element = my_list[index]
```

- **Slicing**: Extract a sublist using slicing.

```
sublist = my_list[startInd:endInd]
```

Modifying Lists

- **Modifying an Element**: Change the value at a specific index.

```
my_list[index] = new_value
```

Adding Elements

- `append(value)`: Adds an element to the end of the list.

```
my_list.append(4)
```

- `insert(index, value)`: Inserts an element at a specific index.

```
my_list.insert(1, "banana")
```

- `extend(iterable)`: Adds multiple elements from an iterable (e.g., another list).

```
my_list.extend([5, 6])
```

Removing Elements

- `remove(value)`: Removes the first occurrence of a value.

```
my_list.remove("banana")
```

- `pop(index)`: Removes and returns the element at the specified index (or the last element if no index is specified).

```
last_element = my_list.pop()
```

- `clear()`: Removes all elements from the list.

```
my_list.clear()
```

Copying and Concatenation

- `copy()`: Creates a shallow copy of the list.

```
new_list = my_list.copy()
```

- **Concatenation**: Use the `+` operator to combine lists.

```
combined = my_list + [7, 8]
```

- **Repetition**: Use the `*` operator to repeat the list.

```
repeated = my_list * 2
```

Membership

- **Check Membership**: Use `in` and `not in` to check for the presence of an element.

```
exists = 3 in my_list  # True or False
```

Nested Lists

- **Create Nested Lists**: Lists can contain other lists.

```
nested_list = [1, [2, 3], [4, 5, 6]]
```

List Methods

- `sort()`: Sorts the list in ascending order.

```
my_list.sort()
```

- `reverse()`: Reverses the order of elements in the list.

```
my_list.reverse()
```

- `index(value)`: Returns the index of the first occurrence of a value.

```
index_of_value = my_list.index(2)
```

- `count(value)`: Returns the number of occurrences of a value.

```
count_of_value = my_list.count(1)
```

Working with Lists in Python

Create a list

```
my_list = [1, "apple", 3.14, "banana", 1]
```

Accessing elements

```
print("First element:", my_list[0])          # Output: 1
print("Sliced list:", my_list[1:4])          # Output: ['apple', 3.14, 'banana']
```

Modifying an element

```
my_list[2] = "orange"
print("Modified list:", my_list)              # Output: [1, 'apple', 'orange',
'banana', 1]
```

Adding elements

```
my_list.append(4)
print("After appending 4:", my_list)          # Output: [1, 'apple', 'orange',
'banana', 1, 4]
my_list.insert(1, "grape")
print("After inserting 'grape':", my_list)    # Output: [1, 'grape', 'apple',
'orange', 'banana', 1, 4]
my_list.extend([5, 6])
print("After extending with [5, 6]:", my_list) # Output: [1, 'grape', 'apple',
'orange', 'banana', 1, 4, 5, 6]
```

Removing elements

```
my_list.remove("banana")
print("After removing 'banana':", my_list)    # Output: [1, 'grape', 'apple',
'orange', 1, 4, 5, 6]
last_element = my_list.pop()
print("After popping last element:", my_list, "| Popped:", last_element)  # Output:
[1, 'grape', 'apple', 'orange', 1, 4, 5] | Popped: 6
```

Copying the list

```
new_list = my_list.copy()
print("Copied list:", new_list)               # Output: [1, 'grape', 'apple',
'orange', 1, 4, 5]
```

Concatenation and repetition

```
combined = my_list + [7, 8]
print("Concatenated list:", combined)         # Output: [1, 'grape', 'apple',
'orange', 1, 4, 5, 7, 8]
repeated = my_list * 2
print("Repeated list:", repeated)             # Output: [1, 'grape', 'apple',
'orange', 1, 4, 5, 1, 'grape', 'apple', 'orange', 1, 4, 5]
```

Checking membership

```
exists = 3 in my_list
print("Is 3 in my_list?", exists)              # Output: False
```

Nested list

```
nested_list = [1, [2, 3], [4, 5, 6]]
print("Nested list:", nested_list)              # Output: [1, [2, 3], [4, 5, 6]]
```

List methods

```
my_list.sort()
print("Sorted list:", my_list)                  # Output: [1, 1, 4, 'apple', 'grape',
'orange']
my_list.reverse()
print("Reversed list:", my_list)                # Output: ['orange', 'grape',
'apple', 4, 1, 1]
index_of_value = my_list.index('apple')
print("Index of 'apple':", index_of_value)      # Output: 2
count_of_value = my_list.count(1)
print("Count of 1:", count_of_value)            # Output: 2
```

Tuples in Python

Overview

Tuples are a data structure in Python that stores an ordered collection of elements.

Key Characteristics

- **Immutable**: Once created, the elements in a tuple cannot be modified, added, or removed.
- **Ordered**: The order of elements is preserved, just like lists.
- **Heterogeneous**: Tuples can hold elements of different data types (e.g., integers, strings, objects).
- **Memory Efficient**: Tuples require less memory than lists, making them more memory-efficient for storing data.
- **Faster than Lists**: Tuples can be faster than lists for iteration due to their immutable nature.

Example

```
my_tuple = (1, "apple", 3.14, "banana")  # A tuple with mixed data types
```

Tuple Operations in Python

Creating a Tuple

- **Creation**:

```
empty_tuple = ()
```

Accessing Tuple Elements

- **Indexing**: Access elements using an index.

```
element = my_tuple[ind]
```

- **Slicing**: Extract a subtuple using slicing.

```
subtuple = my_tuple[startInd:endInd]
```

Operations

- **Concatenation**: Use the `+` operator to combine tuples.

```
combined = my_tuple + (7, 8)
```

- **Repetition**: Use the `*` operator to repeat the tuple.

```
repeated = my_tuple * 2
```

Membership

- **Check Membership**: Use `in` and `not in` to check for the presence of an element.

```
exists = 3.14 in my_tuple  # True or False
```

Built-in Methods

- `count(item)`: Returns the number of occurrences of an item.

```
count_of_item = my_tuple.count("apple")
```

- `index(item)`: Returns the index of the first occurrence of an item.

```
index_of_item = my_tuple.index("banana")
```

Tuple Unpacking

- **Unpacking**: Assigns elements of a tuple to individual variables.

```
item1, item2, item3 = my_tuple
```

Nested Tuples

- **Create Nested Tuples**: Tuples can contain other tuples.

```
nested_tuple = (1, (2, 3), (4, 5, 6))
```

Converting Tuples to Lists

- **Conversion**: Use `list()` to convert a tuple to a list.

```
my_list = list(my_tuple)
```

Example: Working with Tuples in Python

Create a tuple

```
my_tuple = (1, "apple", 3.14, "banana")
```

Accessing elements

```
first_element = my_tuple[0]
print("First element:", first_element)          # Output: 1
print("Sliced tuple:", my_tuple[1:3])          # Output: ('apple', 3.14)
```

Concatenation

```
combined_tuple = my_tuple + (7, 8)
print("Combined tuple:", combined_tuple)        # Output: (1, 'apple', 3.14,
'banana', 7, 8)
```

Repetition

```
repeated_tuple = my_tuple * 2
print("Repeated tuple:", repeated_tuple)        # Output: (1, 'apple', 3.14,
'banana', 1, 'apple', 3.14, 'banana')
```

Membership check

```
exists = "apple" in my_tuple
print("Is 'apple' in my_tuple?", exists)       # Output: True
```

Built-in methods

```
count_of_apple = my_tuple.count("apple")
print("Count of 'apple':", count_of_apple)     # Output: 1
index_of_banana = my_tuple.index("banana")
print("Index of 'banana':", index_of_banana)    # Output: 3
```

Tuple unpacking

```
item1, item2, item3, item4 = my_tuple
print("Unpacked items:", item1, item2, item3, item4)  # Output: 1 apple 3.14 banana
```

Nested tuples

```
nested_tuple = (1, (2, 3), (4, 5, 6))
print("Nested tuple:", nested_tuple)              # Output: (1, (2, 3), (4, 5, 6))
```

Converting tuple to list

```
my_list = list(my_tuple)
print("Converted to list:", my_list)            # Output: [1, 'apple', 3.14,
'banana']
```

Tuple vs. List in Python

Feature	Tuple	List
Mutability	Immutable (cannot be changed)	Mutable (can be modified)
Syntax	Defined with parentheses `()`	Defined with square brackets `[]`
Performance	Faster due to immutability	Slower for certain operations
Memory Usage	More memory efficient	Generally uses more memory
Heterogeneous	Can store different data types	Can also store different data types
Duplicates	Allows duplicate values	Allows duplicate values
Use Case	Used for fixed collections or data that shouldn't change	Used for collections that may change
Methods	Fewer built-in methods	More built-in methods available (e.g., `append()`, `remove()`)
Indexing & Slicing	Supports indexing and slicing	Supports indexing and slicing

- **Tuples** are best suited for data that should remain constant and are faster and more memory-efficient.
- **Lists** are ideal for data that may need to be changed, allowing for more flexible operations and methods.

Dictionaries in Python

Overview

Dictionaries are data structures that store data in key-value pairs, allowing for efficient data retrieval and management.

Key Characteristics

- **Mutable**: Dictionaries can be modified after creation, allowing for dynamic updates.

```
my_dict = {'name': 'Alice', 'age': 30}
```

- **Powerful and Flexible**: They can store a wide variety of data types, including lists, tuples, and even other dictionaries.
- **Complex Data Management**: Ideal for managing complex data relationships, such as representing objects or records.

```
user_info = {
    'name': 'Alice',
    'age': 30,
    'hobbies': ['reading', 'traveling']
}
```

- **Indexed by Unique Keys**: Each value is accessed via a unique key, which can be of various immutable types (e.g., strings, numbers, tuples).

```
print(user_info['name'])  # Output: Alice
```

Example

Creating and accessing a dictionary:

```
person = {'name': 'Bob', 'age': 25, 'city': 'New York'}
print(person['age'])  # Output: 25
```

Summary

Dictionaries provide a powerful way to store and manage data in key-value pairs, making them essential for a wide range of programming tasks in Python.

Here's a concise Markdown snippet summarizing dictionary operations in Python:

Dictionary Operations in Python

Overview

Dictionaries are versatile data structures that store data in key-value pairs and support various operations for managing that data.

Creating a Dictionary

- **Creation**:

```
empty_dict = {}
```

Accessing Dictionary Elements

- **Using Keys**: Access elements directly using keys.

```
value = my_dict[key]
```

- **Using `get()`**: Safely retrieve values without raising an error if the key doesn't exist.

```
age = my_dict.get("age")  # Returns None if "age" doesn't exist
```

Removing Key-Value Pairs

- `pop(key)`: Removes the specified key and returns its value.

```
removed_value = my_dict.pop("key")
```

- `popitem()`: Removes and returns the last inserted key-value pair as a tuple.

```
last_item = my_dict.popitem()
```

- `del`: Deletes a specified key-value pair.

```
del my_dict["key"]
```

- `clear()`: Removes all items from the dictionary.

```
my_dict.clear()
```

Modifying a Dictionary

- **Adding Key-Value Pairs**: Add new key-value pairs or update existing ones.

```
my_dict["new_key"] = new_value
```

Checking for Key Existence

- **Using `in` and `not in`**: Check if a key exists in the dictionary.

```
exists = "key" in my_dict  # Returns True or False
```

Dictionary Methods

- `keys()`: Returns a view of all keys in the dictionary.
- `values()`: Returns a view of all values in the dictionary.
- `items()`: Returns a view of all key-value pairs as tuples.
- `copy()`: Creates a shallow copy of the dictionary.

```
new_dict = my_dict.copy()
```

Summary

Dictionaries in Python offer powerful operations for accessing, modifying, and managing data stored in key-value pairs, making them essential for various programming tasks.

Dictionary Keys in Python

Valid Keys

Dictionaries can have various types of valid keys, which must be immutable:

1. **Strings**:

```
my_dict = {"key": "value"}
```

2. **Numbers**:

```
my_dict = {1: "one", 2: "two"}
```

3. **Tuples**:

```
my_dict = {(1, 2): "pair"}
```

Invalid Keys

Certain types cannot be used as keys because they are mutable:

4. **Lists**:

```
my_dict = {[1, 2]: "list"}  # Raises TypeError
```

5. **Dictionaries**:

```
my_dict = {{1: 2}: "dict"}  # Raises TypeError
```

Summary

When creating dictionaries in Python, ensure that keys are immutable types such as strings, numbers, or tuples to avoid errors.

```

```

Functions in Python

Overview

Functions are blocks of reusable code that perform a specific task. They help in organizing code, making it more readable, and reducing redundancy.

Key Characteristics

- **First-Class Objects**: Functions can be passed as arguments, returned from other functions, and assigned to variables.

Building Blocks of Functions

1. Defining a Function

- Functions are defined using the `def` keyword.

```
def my_function():
    print("Hello, World!")
```

2. Calling a Function

- Functions are called by their name followed by parentheses.

```
my_function()  # Output: Hello, World!
```

3. Parameters

- Parameters are variables defined in the function declaration that accept values.

```
def greet(name):
    print(f"Hello, {name}!")
```

4. Arguments

- Arguments are the actual values passed to the function when calling it.

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

5. Return Values

- Functions can return values using the `return` statement.

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

result = add(2, 3)
print(result)  # Output: 5
```

Summary

Functions are essential for writing clean, organized, and efficient code. They allow for better code reuse and modular design.

Return Values in Python Functions

Overview

Return values allow functions to send data back to the caller. They can be single or multiple values.

Single Return Value

- A function can return a single value of any type, and the type is determined at runtime.

```
def square(x):
    return x ** 2

result = square(4)  # result is 16 (int)
```

Multiple Return Values

- Functions can return multiple values, which are bundled into a tuple.

How It Works

6. **Tuple Creation**: When multiple values are returned, Python automatically creates a tuple.

```
def min_max(numbers):
    return min(numbers), max(numbers)

result = min_max([1, 2, 3, 4, 5])  # result is (1, 5)
```

7. **Return Value**: The created tuple becomes the return value of the function.

8. **Unpacking (Optional)**: The returned tuple can be unpacked into individual variables.

```
minimum, maximum = min_max([1, 2, 3, 4, 5])
print("Minimum:", minimum)  # Output: Minimum: 1
print("Maximum:", maximum)  # Output: Maximum: 5
```

Summary

Return values enhance the functionality of functions by allowing them to produce outputs, which can be single values or multiple values packed into tuples for easy handling.

Default Arguments in Python Functions

Overview

Default arguments allow function parameters to have default values. These values are used if no argument is provided during the function call.

Key Points

- **Specification**: Default arguments are defined in the function declaration.

```
def greet(name="Guest"):
    print(f"Hello, {name}!")
```

- **Usage**: If a value is not provided for the parameter during the function call, the default value is used.

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

- **Order of Arguments**: Default arguments must be placed after non-default arguments in the function definition.

```
def calculate(area, height=10):
    return area * height

result = calculate(5)         # height defaults to 10
print(result)                 # Output: 50
```

Summary

Default arguments provide flexibility in function calls, allowing parameters to have default values while maintaining the option for explicit input.

Keyword Arguments in Python Functions

Overview

Keyword arguments, also known as named arguments, allow you to pass arguments to a function by explicitly specifying the parameter names. This enhances code clarity and flexibility.

Key Points

- **Flexible Order**: Arguments can be passed in any order as long as the parameter names are specified.

```
def describe_pet(animal_type, pet_name):
    print(f"I have a {animal_type} named {pet_name}.")

describe_pet(pet_name="Buddy", animal_type="dog")  # Output: I have a dog
named Buddy.
```

- **Improved Clarity**: Using keyword arguments makes the function calls more understandable, as itâ€™s clear what each argument represents.

```
describe_pet(animal_type="cat", pet_name="Whiskers")  # Output: I have a cat
named Whiskers.
```

- **Combination with Default Arguments**: Keyword arguments can be used alongside default arguments for additional flexibility.

```
def introduce(name, age=18):
    print(f"My name is {name} and I am {age} years old.")

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

Summary

Keyword arguments enhance the readability of function calls and provide the flexibility to pass arguments in any order, making functions easier to use and understand.

Global vs. Local Variables in Python

Global Variables

- **Definition**: Global variables are defined outside of any function or block.

```
global_var = "I am global!"

def display_global():
    print(global_var)  # Accessing global variable

display_global()  # Output: I am global!
```

- **Accessibility**: They are accessible throughout the entire script or module, including within functions.

Local Variables

- **Definition**: Local variables are defined within a specific function or block.

```
def display_local():
    local_var = "I am local!"
    print(local_var)  # Accessing local variable

display_local()  # Output: I am local!
```

- **Accessibility**: They are accessible only within that specific function or block and cannot be accessed from outside.

```
print(local_var)  # Raises NameError: name 'local_var' is not defined
```

Summary

- **Global variables** provide a way to share data across functions, while
- **local variables** are used to maintain state within a function, ensuring encapsulation and avoiding unintended interactions.

Lambda Functions in Python

Overview

A **lambda function** is a small, anonymous function defined using the `lambda` keyword. It is a single expression that returns a value.

Key Characteristics

- **Anonymous**: Lambda functions are often referred to as anonymous functions because they do not require a name.
- **Single Expression**: They consist of a single expression, which is evaluated and returned.

```
square = lambda x: x ** 2
print(square(5))  # Output: 25
```

Use Cases

1. **Creating Small, One-Time Functions**: Ideal for short functions that are used temporarily.

```
add = lambda a, b: a + b
print(add(3, 4))  # Output: 7
```

2. **Higher-Order Functions**: Used as arguments for functions that expect other functions as input, such as `map()`, `filter()`, and `reduce()`.

    - **Example with `map()`**:

    ```
    numbers = [1, 2, 3, 4]
    squared = list(map(lambda x: x ** 2, numbers))
    print(squared)  # Output: [1, 4, 9, 16]
    ```

    - **Example with `filter()`**:

    ```
    even_numbers = list(filter(lambda x: x % 2 == 0, numbers))
    print(even_numbers)  # Output: [2, 4]
    ```

Syntax

The syntax for defining a lambda function is:

```
lambda arguments: expression
```

Summary

Lambda functions are a powerful feature for creating quick, inline functions, particularly useful in functional programming contexts where functions are passed as arguments.

Recursion in Python

Overview

Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem.

Key Components

1. Base Case

- The base case is a condition that stops the recursion.
- It prevents the function from calling itself indefinitely, ensuring that the recursive calls eventually terminate.

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:
        return n * factorial(n - 1)  # Recursive case
```

2. Recursive Case

- The recursive case is the part of the function where the function calls itself.
- It reduces the problem into smaller instances, gradually approaching the base case.

```
def factorial(n):
    if n == 0:
        return 1  # Base case
    else:
        return n * factorial(n - 1)  # Recursive case
```

Example

A classic example of recursion is calculating the factorial of a number:

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

Summary

Recursion is a powerful technique for solving problems by breaking them down into smaller, more manageable subproblems. The key to effective recursion is defining clear base and recursive cases to ensure termination and correct results.

When to Use and Avoid Recursion in Python

Use Recursion When:

1. **Divisible into Sub-Problems**: The problem can be divided into similar sub-problems that can be solved independently.
    - Example: Factorial calculation, Fibonacci sequence.
2. **Naturally Recursive Problems**: The problem structure is inherently recursive, such as tree and graph traversal.
    - Example: Depth-first search (DFS) in trees or graphs.
3. **Simplicity and Clarity**: The recursive solution is simpler and clearer than its iterative counterpart, enhancing code readability.
    - Example: Recursive implementations often have less boilerplate code.

Avoid Recursion When:

4. **Performance is Critical**: When the performance is crucial, and the depth of recursion can lead to stack overflow or excessive function calls.
    - Example: Very deep recursion, like in large Fibonacci calculations.
5. **Iterative Solutions are Simpler**: If an iterative approach is more straightforward and easier to understand, it may be preferable.
    - Example: Looping through lists or collections without needing the overhead of recursive calls.

Summary

Choosing between recursion and iteration depends on the problem's nature, the importance of performance, and the need for clarity in the solution. Use recursion for naturally recursive problems and when code simplicity is a priority, but opt for iteration when performance or straightforwardness is essential.

File Handling in Python

Overview

File handling allows you to read from and write to files, enabling data storage and retrieval.

Opening a File

Use the `open()` function with the desired mode:

- **Modes**:
  - `'r'`: Read mode.
  - `'w'`: Write mode (creates or truncates a file).
  - `'a'`: Append mode.
  - `'b'`: Binary mode.

Example

```
file = open("example.txt", "w")  # Open file for writing
```

Closing a File

Always close a file to free resources.

```
file.close()  # Close the file
```

Checking if a File Exists

Use `os.path.exists()` to check file existence.

```
import os
exists = os.path.exists("example.txt")  # Returns True or False
```

Writing to a File

- **Write a String**:

```
file = open("example.txt", "w")
file.write("Hello, World!")  # Write a string
```

- **Write Multiple Strings**:

```
lines = ["Line 1\n", "Line 2\n"]
file.writelines(lines)  # Write a list of strings
```

Reading a File

- **Read the Entire File**:

```
file = open("example.txt", "r")
content = file.read()  # Read the whole file
```

- **Read One Line**:

```
line = file.readline()  # Read the next line
```

- **Read All Lines**:

```
lines = file.readlines()  # Read all lines into a list
```

Summary

File handling in Python involves opening, reading, writing, and closing files using various modes. Proper file management ensures efficient data handling and resource utilization.

Working with CSV Files in Python

Overview

The `csv` module in Python provides functionality to read from and write to CSV files, making it easy to handle tabular data.

Importing the CSV Module

```
import csv
```

Methods in the `csv` Module

1. `csv.reader()`

Reads a CSV file and returns a reader object, which can be iterated over to access rows.

Example

```
with open("example.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)  # Prints each row as a list
```

2. `csv.DictReader()`

Reads a CSV file and maps the data into a dictionary where the keys are taken from the first row of the file.

Example

```
with open("example.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(row)  # Prints each row as a dictionary
```

3. `csv.writer()`

Creates a writer object to write data to a CSV file.

Example

```
data = [
    ["Name", "Age", "City"],
    ["Alice", 30, "New York"],
    ["Bob", 25, "Los Angeles"]
]
```

```
with open("example.csv", "w", newline='') as file:
    writer = csv.writer(file)
    writer.writerows(data)  # Writes multiple rows
```

4. `csv.DictWriter()`

Creates a writer object that maps dictionaries to CSV rows.

Example

```
data = [
    {"Name": "Alice", "Age": 30, "City": "New York"},
    {"Name": "Bob", "Age": 25, "City": "Los Angeles"}
]

with open("example_dict.csv", "w", newline='') as file:
    fieldnames = ["Name", "Age", "City"]
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()  # Write header row
    writer.writerows(data)  # Writes multiple rows
```

Summary

The `csv` module simplifies working with CSV files in Python through various methods for reading and writing data, enabling easy manipulation of tabular information.

Exception Handling in Python

Overview

An **exception** is an event that disrupts the normal flow of instructions in a program. Exception handling is the process of managing errors gracefully.

Why Use Exception Handling?

- **Prevent Crashes**: Avoid abrupt termination of the program due to errors.
- **Meaningful Error Messages**: Provide clear feedback to users regarding issues.
- **Graceful Handling**: Manage unexpected situations without compromising program flow.

Basic Syntax

Use `try`, `except`, `finally`, and `else` blocks for exception handling.

Example

```
try:
    # Code that may raise an exception
    result = 10 / 0  # Division by zero
except ZeroDivisionError as e:
    print(f"Error: {e}")  # Handle the specific exception
except Exception as e:
    print(f"An unexpected error occurred: {e}")  # Handle all other exceptions
else:
    print("Operation successful:", result)  # Executes if no exceptions occur
finally:
    print("This block always executes.")  # Executes regardless of exceptions
```

Exception Handling Examples in Python

1. `try` and `except`

Handles a specific exception.

Example

```
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Error: Division by zero!")
```

2. `try`, `except`, and `else`

The `else` block runs if no exceptions are raised.

Example

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("That's not a valid number!")
except ZeroDivisionError:
    print("Error: Division by zero!")
else:
    print("Result is:", result)  # Runs if no exceptions occur
```

3. `try` and Multiple `except`

Handles multiple exceptions separately.

Example

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ValueError:
    print("Please enter valid integers.")
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

4. `try`, `except`, and `finally`

The `finally` block runs regardless of whether an exception occurred.

Example

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()  # Ensures the file is closed
    print("File closed.")
```

Summary

These examples illustrate different ways to handle exceptions in Python, allowing you to manage errors effectively while ensuring your program runs smoothly.

Best Practices for Exception Handling in Python

1. Avoid Using a Bare `except` Clause

Using a bare `except` clause can catch all exceptions, including system exit and keyboard interrupts, making it hard to identify the root cause of errors. Always specify the exception type you want to handle.

Example

```
try:
    # Some code
except ValueError:
    print("Caught a ValueError.")
```

2. Provide Meaningful Error Messages

When handling exceptions, ensure that the error messages are clear and informative. This helps users understand what went wrong and how to fix it.

Example

```
try:
    num = int(input("Enter a number: "))
except ValueError:
    print("Error: Please enter a valid integer.")
```

3. Always Clean Up Resources with `finally`

Use the `finally` block to clean up resources, such as closing files or database connections, regardless of whether an exception occurred.

Example

```
try:
    file = open("example.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found.")
finally:
    file.close()  # Ensures the file is closed
    print("File closed.")
```

Summary

Following these best practices helps ensure that your code is robust, maintainable, and user-friendly when handling exceptions.

Object-Oriented Programming (OOP) Concepts in Python

Overview

Object-Oriented Programming (OOP) is a programming paradigm that models real-world concepts as objects. It focuses on code organization, data modeling, reusability, and maintainability.

Key Concepts

1. Classes and Objects

- **Class**: A blueprint for creating objects. Defines properties (attributes) and behaviors (methods).
- **Object**: An instance of a class.

Example

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return f"{self.name} says Woof!"

# Creating an object
my_dog = Dog("Buddy", "Golden Retriever")
print(my_dog.bark())  # Output: Buddy says Woof!
```

2. Encapsulation

Encapsulation restricts access to certain components of an object and allows for the bundling of data and methods that operate on that data.

Example

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(100)
account.deposit(50)
print(account.get_balance())  # Output: 150
```

3. Inheritance

Inheritance allows a class to inherit properties and methods from another class, promoting code reusability.

Example

```
class Animal:
    def speak(self):
        return "Animal speaks"

class Cat(Animal):  # Cat inherits from Animal
    def speak(self):
        return "Cat meows"

my_cat = Cat()
print(my_cat.speak())  # Output: Cat meows
```

4. Polymorphism

Polymorphism allows methods to do different things based on the object that it is acting upon, often implemented via method overriding or interfaces.

Example

```
def animal_sound(animal):
    print(animal.speak())

animal_sound(Cat())  # Output: Cat meows
animal_sound(Dog("Buddy", "Beagle"))  # Output: Buddy says Woof!
```

Benefits of OOP

- **Code Organization**: Helps in structuring code logically.
- **Data Modeling**: Models real-world entities accurately.
- **Reusability**: Promotes reusing code through inheritance and composition.
- **Maintainability**: Simplifies code maintenance and updates.

Summary

OOP is a powerful paradigm that enhances code quality and facilitates a structured approach to programming.

Here's a concise Markdown snippet summarizing classes and objects in Python:

Classes and Objects in Python

Class

- **Definition**: A class is a custom datatype and a blueprint for creating objects.
- **Purpose**: Defines a set of attributes (data) and methods (functions) that describe the behavior of the objects.
- **Encapsulation**: Encapsulates data for reusability and modularity.

Example

```
class Car:
    def __init__(self, make, model):
        self.make = make  # Attribute
        self.model = model  # Attribute

    def start_engine(self):  # Method
        return f"{self.make} {self.model}'s engine started!"

# Creating a class instance (object)
my_car = Car("Toyota", "Corolla")
print(my_car.start_engine())  # Output: Toyota Corolla's engine started!
```

Object

- **Definition**: An object is an instance of a class created from the class blueprint.
- **Characteristics**:
    - **State**: Represented by attributes (e.g., make, model).
    - **Behavior**: Represented by methods (e.g., start_engine).
- **Flexibility**: Objects can have different attribute values, allowing for diverse instances.

Example

```
my_car_1 = Car("Honda", "Civic")  # Different object with different values
my_car_2 = Car("Ford", "Focus")  # Another object
print(my_car_1.start_engine())  # Output: Honda Civic's engine started!
print(my_car_2.start_engine())  # Output: Ford Focus's engine started!
```

Summary

Classes and objects are fundamental concepts in OOP, enabling the creation of modular and reusable code that models real-world entities.

Constructor in Python

Definition

A constructor is a special method that is automatically called when an object is created. It is used to initialize the object's attributes.

Purpose

- **Initialize Attributes**: Set initial values for the object's attributes.
- **Simplify Object Creation**: Makes it easier to create objects with predefined states.

Syntax

The constructor is defined using the `__init__()` method.

Components

- `self`: Refers to the current instance of the class.
- **Parameters**: Allows passing initial values to set the attributes.

Example

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

# Creating an object
person1 = Person("Alice", 30)
print(f"Name: {person1.name}, Age: {person1.age}")  # Output: Name: Alice, Age: 30

person2 = Person("Bob", 25)
print(f"Name: {person2.name}, Age: {person2.age}")  # Output: Name: Bob, Age: 25
```

Summary

Constructors play a crucial role in OOP by allowing objects to be created with specific initial states, enhancing code readability and maintainability.

Encapsulation

Encapsulation in Python

Definition

Encapsulation is the bundling of data (attributes) and methods (functions) that operate on that data within a single class. This concept restricts direct access to some of the object's components, promoting data hiding.

Benefits

- **Improved Security**: Protects the internal state of an object from unintended interference or modification.
- **Better Collaboration**: Simplifies interactions between different parts of a program, making it easier for teams to work together.
- **Reduced Complexity**: Hides complex implementation details, exposing only necessary interfaces.
- **Improved Code Maintainability**: Changes to internal implementation can be made without affecting external code that uses the class.

Key Components

1. Constructors

Constructors (`__init__` method) initialize an object's attributes when it is created, ensuring that the object starts with a valid state.

Example

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name  # Private attribute
        self.__salary = salary  # Private attribute
```

2. Getter and Setter Methods

These methods provide controlled access to private attributes, allowing you to get (retrieve) or set (modify) values while enforcing rules or validation.

Example

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    def get_salary(self):  # Getter method
        return self.__salary

    def set_salary(self, salary):  # Setter method
        if salary >= 0:  # Validation
            self.__salary = salary
        else:
            raise ValueError("Salary must be non-negative.")
```

3. Access Modifiers

Access modifiers control the visibility of class members:

- **Private (`__`)**: Members are not accessible from outside the class.
- **Protected (`_`)**: Members are intended for internal use but can be accessed in subclasses.
- **Public**: Members are accessible from outside the class.

Summary

Encapsulation is a fundamental OOP concept that enhances data security, simplifies complexity, and improves code maintainability by bundling data and behavior within a class.

Class Variables in Python

Definition

Class variables are used to store data that is shared among all instances of a class. They are defined within the class but outside of any instance methods.

Characteristics

- **Shared**: Class variables are shared among all instances of the class. Any modification to a class variable affects all instances.
- **Access**: Class variables can be accessed using both the class name and instances of the class.

Syntax

```
class ClassName:
    class_variable = value  # Class variable defined here
```

Example

```
class Dog:
    species = "Canis familiaris"  # Class variable

    def __init__(self, name):
        self.name = name  # Instance variable

# Creating instances
dog1 = Dog("Buddy")
dog2 = Dog("Max")

# Accessing class variable
print(Dog.species)  # Output: Canis familiaris
print(dog1.species)  # Output: Canis familiaris
print(dog2.species)  # Output: Canis familiaris

# Modifying class variable
Dog.species = "Canis lupus familiaris"

print(dog1.species)  # Output: Canis lupus familiaris
print(dog2.species)  # Output: Canis lupus familiaris
```

Summary

Class variables provide a way to define attributes that should be shared across all instances of a class, promoting memory efficiency and consistency within the class.

Abstract Classes in Python

Definition

An abstract class is a class that contains one or more abstract methods. It cannot be instantiated directly and serves as a blueprint for its subclasses, enforcing a contract that ensures certain methods are implemented.

Key Features

- **Abstract Methods**: Methods that are declared but contain no implementation. They must be overridden in subclasses.
- **Decorator**: Abstract methods are marked with the `@abstractmethod` decorator.
- **Importing**: To use abstract classes, you need to import the `abc` module.

Usage

Example of an Abstract Class

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Inherit from ABC
    @abstractmethod
    def sound(self):  # Abstract method
        pass

class Dog(Animal):
    def sound(self):  # Implementing the abstract method
        return "Bark"

class Cat(Animal):
    def sound(self):  # Implementing the abstract method
        return "Meow"

# Cannot instantiate an abstract class
# animal = Animal()  # Raises TypeError

# Instantiating subclasses
dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

Summary

Abstract classes in Python provide a way to define common interfaces for a group of related classes while ensuring that specific methods are implemented in those subclasses. This promotes code consistency and enforces a design contract.

Polymorphism in Python

Definition

Polymorphism is the ability of a method to take on many forms. It allows methods to be defined in different ways depending on the object invoking them.

Types of Polymorphism

1. Overriding

- **Definition**: Subclasses can modify or extend the behavior of inherited methods from parent classes.
- **Benefit**: Helps avoid redundant code by leveraging inherited methods, allowing for customized behavior in subclasses.

Example of Overriding

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):  # Overriding the inherited method
        return "Bark"

class Cat(Animal):
    def sound(self):  # Overriding the inherited method
        return "Meow"

# Using polymorphism
animals = [Dog(), Cat()]
for animal in animals:
    print(animal.sound())  # Output: Bark, Meow
```

2. Overloading

- **Note**: Method overloading (same method name with different parameters) is not supported directly in Python. Instead, default arguments or variable-length arguments can be used to achieve similar behavior.

Example of Default Arguments

```
class MathOperations:
    def add(self, a, b, c=0):  # Default argument for overloading
        return a + b + c

math = MathOperations()
print(math.add(2, 3))       # Output: 5
print(math.add(2, 3, 4))    # Output: 9
```

Summary

Polymorphism is a powerful feature in Python that enhances flexibility and maintainability in code by allowing methods to be redefined in subclasses and enabling more dynamic method calls.

`super()` in Python

Definition

The `super()` function is used to call a method from a parent class within a subclass. It provides a way to access inherited methods that have been overridden in the subclass.

Key Features

- **Method Resolution Order (MRO)**: `super()` follows the method resolution order, allowing it to find the next method in line to be executed from the parent class hierarchy.

Usage

Example of Using `super()`

```
class Parent:
    def greet(self):
        return "Hello from Parent!"

class Child(Parent):
    def greet(self):
        parent_greeting = super().greet()  # Call the parent class method
        return f"{parent_greeting} And hello from Child!"

# Creating an instance of Child
child_instance = Child()
print(child_instance.greet())
# Output: Hello from Parent! And hello from Child!
```

MRO Example

To demonstrate the method resolution order:

```
class A:
    def method(self):
        return "Method from A"

class B(A):
    def method(self):
        return super().method() + " and Method from B"

class C(A):
    def method(self):
        return super().method() + " and Method from C"

class D(B, C):
    def method(self):
        return super().method() + " and Method from D"
```

```
d_instance = D()
print(d_instance.method())
# Output: Method from A and Method from C and Method from B and Method from D
```

Summary

The `super()` function is essential for leveraging inheritance in Python, allowing subclasses to access and extend the functionality of parent class methods while maintaining the integrity of the method resolution order.

Python Inheritance

Definition

Inheritance is a fundamental concept in object-oriented programming that allows one class (child or subclass) to inherit attributes and methods from another class (parent or superclass). This promotes code reusability and establishes a hierarchical relationship between classes.

Key Features

- **Code Reusability**: Inherited attributes and methods can be reused in subclasses, reducing code duplication.
- **Method Overriding**: Subclasses can modify or extend the behavior of methods inherited from the parent class.
- **Polymorphism**: Inheritance allows for polymorphic behavior, where a method can be used in different contexts depending on the object.

Types of Inheritance

1. Single Inheritance

A subclass inherits from a single superclass.

```
class Parent:
    def greet(self):
        return "Hello!"

class Child(Parent):
    def farewell(self):
        return "Goodbye!"

child = Child()
print(child.greet())  # Output: Hello!
```

2. Multiple Inheritance

A subclass inherits from multiple superclasses.

```
class Father:
    def skills(self):
        return "Gardening"

class Mother:
    def skills(self):
        return "Cooking"

class Child(Father, Mother):
    pass

child = Child()
print(child.skills())  # Output: Gardening (MRO determines which method is called)
```

3. Multilevel Inheritance

A subclass inherits from a superclass, which in turn inherits from another superclass.

```
class Grandparent:
    def heritage(self):
        return "Family Heritage"

class Parent(Grandparent):
    def values(self):
        return "Family Values"

class Child(Parent):
    def beliefs(self):
        return "Personal Beliefs"

child = Child()
print(child.heritage())  # Output: Family Heritage
```

4. Hierarchical Inheritance

Multiple subclasses inherit from a single superclass.

```
class Animal:
    def sound(self):
        return "Some sound"

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

5. Hybrid Inheritance

A combination of two or more types of inheritance.

```
class A:
    def method_A(self):
        return "Method A"

class B(A):
    def method_B(self):
        return "Method B"

class C(A):
    def method_C(self):
        return "Method C"

class D(B, C):
```

```
    def method_D(self):
        return "Method D"

d = D()
print(d.method_A())  # Output: Method A
print(d.method_B())  # Output: Method B
print(d.method_C())  # Output: Method C
```

Method Resolution Order (MRO)

MRO determines the order in which base classes are searched when executing a method. Python uses the C3 linearization algorithm to create a consistent order.

Example of MRO

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.__mro__)
# Output: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>)
```

Summary

Inheritance in Python enhances code organization, reusability, and maintainability. Understanding the different types of inheritance and how to leverage them effectively is essential for designing robust object-oriented systems.

Regular Expressions in Python

Overview

Regular expressions (regex) are sequences of characters that define a search pattern, primarily used for matching, searching, and manipulating strings.

Library

- `re`: Python's built-in library for working with regular expressions.

Key Functions

1. `re.search(pattern, string)`
    - Searches for the first occurrence of the specified pattern in the string.
    - Returns a match object if found; otherwise, returns `None`.
    - **Example**:

```
import re
result = re.search(r'\d+', 'The year is 2024')
```

2. `re.match(pattern, string)`
    - Checks for a match only at the beginning of the string.
    - Returns a match object if found; otherwise, returns `None`.
    - **Example**:

```
result = re.match(r'The', 'The year is 2024')
```

3. `re.findall(pattern, string)`
    - Finds all non-overlapping matches of the pattern in the string.
    - Returns a list of all matches.
    - **Example**:

```
matches = re.findall(r'\d+', 'There are 2 apples and 3 oranges')
```

4. `re.sub(pattern, repl, string)`
    - Substitutes matches of the pattern with a specified string (`repl`).
    - Returns the modified string.
    - **Example**:

```
modified_string = re.sub(r'apples', 'bananas', 'I like apples')
```

5. `re.split(pattern, string)`
    - Splits the string by the occurrences of the pattern.
    - Returns a list of substrings.

- **Example**:

```
parts = re.split(r'\s+', 'Split this string by spaces')
```

Conclusion

Regular expressions are a powerful tool for string manipulation and pattern matching in Python. The `re` library provides various functions to search, match, and modify strings using regex patterns.

metacharacters used in regular expressions:

Metacharacter	Description	Example
`.`	Matches any single character except newline	`a.c` matches "abc", "a1c", etc.
`^`	Matches the start of the string	`^Hello` matches "Hello, world"
`$`	Matches the end of the string	`world$` matches "Hello, world"
`*`	Matches 0 or more repetitions of the preceding element	`a*` matches "", "a", "aa", etc.
`+`	Matches 1 or more repetitions of the preceding element	`a+` matches "a", "aa", etc.
`?`	Matches 0 or 1 repetition of the preceding element	`a?` matches "", "a"
`{n}`	Matches exactly n repetitions of the preceding element	`a{2}` matches "aa"
`{n,}`	Matches n or more repetitions of the preceding element	`a{2,}` matches "aa", "aaa", etc.
`{n,m}`	Matches between n and m repetitions	`a{1,3}` matches "a", "aa", "aaa"
`[...]`	Matches any single character within the brackets	`[abc]` matches "a", "b", or "c"
`[^...]`	Matches any single character not in the brackets	`[^abc]` matches any character except "a", "b", or "c"
`	`	Acts as a logical OR between expressions
`(...)`	Groups expressions and captures matches	`(abc)` captures "abc"
`\`	Escapes a metacharacter to treat it as a literal	`.` matches the dot character "."

Basic Syntax â€“ Anchors and boundaries

Anchor/Boundary	Description	Example
`^`	Matches the start of a string	`^Hello` matches "Hello world"

`$`	Matches the end of a string	`world$` matches "Hello world"
`\b`	Matches a word boundary	`\bword\b` matches "word" in "word play" but not in "sword"
`\B`	Matches a non-word boundary	`\Bword\B` matches "word" in "swordplay" but not in "word"
`(?=...)`	Positive lookahead: asserts that what follows matches the pattern	`\d(?= dollars)` matches "5" in "5 dollars"
`(?!...)`	Negative lookahead: asserts that what follows does not match the pattern	`\d(?! dollars)` matches "5" in "5 apples" but not in "5 dollars"
`(?<=...)`	Positive lookbehind: asserts that what precedes matches the pattern	`(?<=$)\d+` matches "100" in "$100"
`(?<!...)`	Negative lookbehind: asserts that what precedes does not match the pattern	`(?<!$)\d+` matches "100" in "100 apples" but not in "$100"

Basic Syntax - Character Classes

Character Class	Description	Example
`[abc]`	Matches any single character within the brackets	`[abc]` matches "a", "b", or "c"
`[^abc]`	Matches any single character not in the brackets	`[^abc]` matches "d", "e", etc.
`[a-z]`	Matches any single character in the range a to z	`[a-z]` matches "c", "m", etc.
`[A-Z]`	Matches any single character in the range A to Z	`[A-Z]` matches "C", "M", etc.
`[0-9]`	Matches any single digit	`[0-9]` matches "1", "5", etc.
`[a-zA-Z]`	Matches any letter (lowercase or uppercase)	`[a-zA-Z]` matches "A", "b", etc.
`[a-zA-Z0-9]`	Matches any alphanumeric character	`[a-zA-Z0-9]` matches "1", "a", etc.
`.`	Matches any character except newline	`a.c` matches "abc", "a1c", etc.
`\d`	Matches any digit (equivalent to `[0-9]`)	`\d` matches "5", "0", etc.
`\D`	Matches any non-digit character	`\D` matches "a", " ", etc.
`\w`	Matches any word character (alphanumeric + underscore)	`\w` matches "a", "1", "_", etc.
`\W`	Matches any non-word character	`\W` matches " ", "!", etc.
`\s`	Matches any whitespace character	`\s` matches " ", "\t", etc.
`\S`	Matches any non-whitespace character	`\S` matches "a", "1", etc.

Basic Syntax - Quantifiers

Quantifier	Description	Example
`*`	Matches 0 or more occurrences of the preceding element	`a*` matches "", "a", "aa", etc.
`+`	Matches 1 or more occurrences of the preceding element	`a+` matches "a", "aa", "aaa", etc.
`?`	Matches 0 or 1 occurrence of the preceding element	`a?` matches "", "a"
`{n}`	Matches exactly n occurrences of the preceding element	`a{3}` matches "aaa"
`{n,}`	Matches n or more occurrences of the preceding element	`a{2,}` matches "aa", "aaa", "aaaa", etc.
`{n,m}`	Matches between n and m occurrences of the preceding element	`a{1,3}` matches "a", "aa", or "aaa"
`?`	Matches 0 or 1 occurrence (used after a quantifier for lazy matching)	`a??` matches "" or "a" (lazy match)