Python Fundamentals

Variables and Data Types in Python

What are Variables?

- Variables are used to store data that can be used and manipulated in a program.
- A variable is created when you assign a value to it using the `=` operator.
- Example:

```
name = "Alice"
age = 25
height = 5.6
```

Variable Naming Rules

- Variable names can contain letters, numbers, and underscores.
- Variable names must start with a letter or underscore.
- Variable names are case-sensitive.
- Avoid using Python keywords as variable names (e.g., `print`, `if`, `else`).

Best Practices

- Use descriptive names that reflect the purpose of the variable.
- Use lowercase letters for variable names.
- Separate words using underscores for readability (e.g., `first_name`, `total_amount`).

Data Types in Python

Python supports several built-in data types:

- **Integers (`int`)**: Whole numbers (e.g., `10` , `-5`).
- **Floats (`float`)**: Decimal numbers (e.g., `3.14` , `-0.001`).
- **Strings (`str`)**: Text data enclosed in quotes (e.g., `"Hello"` , `'Python'`).
- **Booleans (`bool`)**: Represents `True` or `False` .
- **Lists**: Ordered, mutable collections (e.g., `[1, 2, 3]`).
- **Tuples**: Ordered, immutable collections (e.g., `(1, 2, 3)`).
- **Sets**: Unordered collections of unique elements (e.g., `{1, 2, 3}`).
- **Dictionaries**: Key-value pairs (e.g., `{"name": "Alice", "age": 25}`).

Checking Data Types

- Use the `type()` function to check the data type of a variable.

```python
print(type(10))       # Output: <class 'int'>
print(type("Hello"))  # Output: <class 'str'>
```

Typecasting in Python

What is Typecasting?

- Typecasting is the process of converting one data type to another.
- Python provides built-in functions for typecasting:
  - `int()` : Converts to integer.
  - `float()` : Converts to float.
  - `str()` : Converts to string.
  - `bool()` : Converts to boolean.

Examples:

```
# Convert string to integer
num_str = "10"
num_int = int(num_str)
print(num_int)  # Output: 10

# Convert integer to string
num = 25
num_str = str(num)
print(num_str)  # Output: "25"

# Convert float to integer
pi = 3.14
pi_int = int(pi)
print(pi_int)   # Output: 3
```

Taking User Input in Python

Using the `input()` Function

- The `input()` function allows you to take user input from the keyboard.
- By default, `input()` returns a string. You can convert it to other data types as needed.
- Example:

```
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f"Hello {name}, you are {age} years old.")
```

Comments, Escape Sequences & Print Statement

Comments

- Comments are used to explain code and are ignored by the Python interpreter.
- Single-line comments start with `#`.
- Multi-line comments are enclosed in `'''` or `"""`.

```
# This is a single-line comment
'''
This is a
multi-line comment
'''
```

Escape Sequences

- Escape sequences are used to include special characters in strings.
- Common escape sequences:
  - `\n` : Newline
  - `\t` : Tab
  - `\\` : Backslash
  - `\"` : Double quote
  - `\'` : Single quote
- Example:

```
print("Hello\nWorld!")
print("This is a tab\tcharacter.")
```

Print Statement

- The `print()` function is used to display output.

- You can use `sep` and `end` parameters to customize the output.

```
print("Hello", "World", sep=", ", end="!\n")
```

Operators in Python

Types of Operators

1. **Arithmetic Operators**:
    1. `+` (Addition), `-` (Subtraction), `*` (Multiplication), `/` (Division), `%` (Modulus), `**` (Exponentiation), `//` (Floor Division).
    2. Example:

    ```
    print(10 + 5)   # Output: 15
    print(10 ** 2)  # Output: 100
    ```

2. **Comparison Operators**:
    1. `==` (Equal), `!=` (Not Equal), `>` (Greater Than), `<` (Less Than), `>=` (Greater Than or Equal), `<=` (Less Than or Equal).
    2. Example:

    ```
    print(10 > 5)   # Output: True
    print(10 == 5)  # Output: False
    ```

3. **Logical Operators**:
    1. `and`, `or`, `not`.
    2. Example:

    ```
    print(True and False)  # Output: False
    print(True or False)   # Output: True
    print(not True)        # Output: False
    ```

4. **Assignment Operators**:

    1. `=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `//=`.

    2. Example:

```python
x = 10
x += 5  # Equivalent to x = x + 5
print(x)  # Output: 15
```

5. **Membership Operators**:

    1. `in`, `not in`.

    2. Example:

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

6. **Identity Operators**:

    1. `is`, `is not`.

    2. Example:

```python
x = 10
y = 10
print(x is y)  # Output: True
```

Summary

- Variables store data, and Python supports multiple data types.
- Typecasting allows you to convert between data types.
- Use `input()` to take user input and `print()` to display output.
- Comments and escape sequences help make your code more readable.
- Python provides a variety of operators for performing operations on data.

Control Flow and Loops

If-Else Conditional Statements

What are Conditional Statements?

- Conditional statements allow you to execute code based on certain conditions.
- Python uses `if`, `elif`, and `else` for decision-making.

Syntax:

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if all conditions are False
```

Example:

```
age = 18

if age < 18:
    print("You are a minor.")
elif age == 18:
    print("You just became an adult!")
else:
    print("You are an adult.")
```

Match Case Statements in Python (Python 3.10+)

What is Match-Case?

- Match-case is a new feature introduced in Python 3.10 for pattern matching.
- It simplifies complex conditional logic.

Syntax:

```
match value:
    case pattern1:
        # Code to execute if value matches pattern1
    case pattern2:
        # Code to execute if value matches pattern2
    case _:
        # Default case (if no patterns match)
```

Example:

```
status = 404

match status:
    case 200:
        print("Success!")
    case 404:
        print("Not Found")
    case _:
        print("Unknown Status")
```

For Loops in Python

What are For Loops?

- For loops are used to iterate over a sequence (e.g., list, string, range).
- They execute a block of code repeatedly for each item in the sequence.

Syntax:

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

Example:

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

for fruit in fruits:
    print(fruit)
```

Using `range()`:

- The `range()` function generates a sequence of numbers.
- Example:

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

While Loops in Python

What are While Loops?

- While loops execute a block of code as long as a condition is `True`.
- They are useful when the number of iterations is not known in advance.

Syntax:

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

Example:

```
count = 0

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

Infinite Loops:

- Be careful to avoid infinite loops by ensuring the condition eventually becomes `False`.
- Example of an infinite loop:

  ```
  while True:
      print("This will run forever!")
  ```

Break, Continue, and Pass Statements

Break

- The `break` statement is used to exit a loop prematurely.
- Example:

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

Continue

- The `continue` statement skips the rest of the code in the current iteration and moves to the next iteration.

- Example:

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

Pass

- The `pass` statement is a placeholder that does nothing. It is used when syntax requires a statement but no action is needed.
- Example:

```python
for i in range(5):
    if i == 3:
        pass  # Do nothing
    print(i)  # Output: 0, 1, 2, 3, 4
```

Summary

- Use `if`, `elif`, and `else` for decision-making.
- Use `match-case` for pattern matching (Python 3.10+).
- Use `for` loops to iterate over sequences and `while` loops for repeated execution based on a condition.
- Control loop execution with `break`, `continue`, and `pass`.

Strings in Python

Introduction

Strings are one of the most fundamental data types in Python. A string is a sequence of characters enclosed within either single quotes (`'`), double quotes (`"`), or triple quotes (`'''` or “””).

Creating Strings

You can create strings in Python using different types of quotes:

```
# Single-quoted string
a = 'Hello, Python!'

# Double-quoted string
b = "Hello, World!"

# Triple-quoted string (useful for multi-line strings)
c = '''This is
a multi-line
string.'''
```

String Indexing

Each character in a string has an index:

```
text = "Python"
print(text[0])  # Output: P
print(text[1])  # Output: y
print(text[-1]) # Output: n (last character)
```

String Slicing

You can extract parts of a string using slicing:

```
text = "Hello, Python!"
print(text[0:5])   # Output: Hello
print(text[:5])    # Output: Hello
print(text[7:])    # Output: Python!
print(text[::2])   # Output: Hlo Pto!
```

String Methods

Python provides several built-in methods to manipulate strings:

```
text = " hello world "
print(text.upper())     # Output: " HELLO WORLD "
print(text.lower())     # Output: " hello world "
print(text.strip())     # Output: "hello world"
print(text.replace("world", "Python")) # Output: " hello Python "
print(text.split())     # Output: ['hello', 'world']
```

String Formatting

Python offers multiple ways to format strings:

```
name = "John"
age = 25

# Using format()
print("My name is {} and I am {} years old.".format(name, age))

# Using f-strings (Python 3.6+)
print(f"My name is {name} and I am {age} years old.")
```

Multiline Strings

Triple quotes allow you to create multi-line strings:

```python
message = '''
Hello,
This is a multi-line string example.
Goodbye!
'''
print(message)
```

Summary

- Strings are sequences of characters.
- Use single, double, or triple quotes to define strings.
- Indexing and slicing allow accessing parts of a string.
- String methods help modify and manipulate strings.
- f-strings provide an efficient way to format strings.

String Slicing and Indexing

Introduction

In Python, strings are sequences of characters, and each character has an index. You can access individual characters using indexing and extract substrings using slicing.

String Indexing

Each character in a string has a unique index, starting from 0 for the first character and -1 for the last character.

```python
text = "Python"
print(text[0])  # Output: P
print(text[1])  # Output: y
print(text[-1]) # Output: n (last character)
print(text[-2]) # Output: o
```

String Slicing

Slicing allows you to extract a portion of a string using the syntax `string[start:stop:step]`.

```
text = "Hello, Python!"
print(text[0:5])   # Output: Hello
print(text[:5])    # Output: Hello (same as text[0:5])
print(text[7:])    # Output: Python! (from index 7 to end)
print(text[::2])   # Output: Hlo Pto!
print(text[-6:-1]) # Output: ython (negative indexing)
```

Step Parameter

The `step` parameter defines the interval of slicing.

```
text = "Python Programming"
print(text[::2])   # Output: Pto rgamn
print(text[::-1])  # Output: gnimmargorP nohtyP (reverses string)
```

Practical Uses of Slicing

String slicing is useful in many scenarios: - Extracting substrings - Reversing strings - Removing characters - Manipulating text efficiently

```
text = "Welcome to Python!"
print(text[:7])   # Output: Welcome
print(text[-7:])  # Output: Python!
print(text[3:-3]) # Output: come to Pyt
```

Summary

- Indexing allows accessing individual characters.
- Positive indexing starts from 0, negative indexing starts from -1.
- Slicing helps extract portions of a string.
- The step parameter defines the interval for selection.

- Using `[::-1]` reverses a string.

String Methods and Functions

Introduction

Python provides a variety of built-in string methods and functions to manipulate and process strings efficiently.

Common String Methods

Changing Case

```
text = "hello world"
print(text.upper())  # Output: "HELLO WORLD"
print(text.lower())  # Output: "hello world"
print(text.title())  # Output: "Hello World"
print(text.capitalize())  # Output: "Hello world"
```

Removing Whitespace

```
text = "  hello world  "
print(text.strip())  # Output: "hello world"
print(text.lstrip()) # Output: "hello world  "
print(text.rstrip()) # Output: "  hello world"
```

Finding and Replacing

```
text = "Python is fun"
print(text.find("is"))    # Output: 7
print(text.replace("fun", "awesome"))  # Output: "Python is aweso
```

Splitting and Joining

```
text = "apple,banana,orange"
fruits = text.split(",")
print(fruits)  # Output: ['apple', 'banana', 'orange']
```

```python
new_text = " - ".join(fruits)
print(new_text)  # Output: "apple - banana - orange"
```

Checking String Properties

```python
text = "Python123"
print(text.isalpha())  # Output: False
print(text.isdigit())  # Output: False
print(text.isalnum())  # Output: True
print(text.isspace())  # Output: False
```

Useful Built-in String Functions

`len()` - Get Length of a String

```python
text = "Hello, Python!"
print(len(text))  # Output: 14
```

`ord()` and `chr()` - Character Encoding

```python
print(ord('A'))  # Output: 65
print(chr(65))   # Output: 'A'
```

`format()` and f-strings

```python
name = "Alice"
age = 30
print("My name is {} and I am {} years old.".format(name, age))
print(f"My name is {name} and I am {age} years old.")
```

Summary

- Python provides various string methods for modification and analysis.

- Case conversion, trimming, finding, replacing, splitting, and joining are commonly used.
- Functions like `len()`, `ord()`, and `chr()` are useful for working with string properties.

String Formatting and f-Strings

Introduction

String formatting is a powerful feature in Python that allows you to insert variables and expressions into strings in a structured way. Python provides multiple ways to format strings, including the older `.format()` method and the modern `f-strings`.

Using `.format()` Method

The `.format()` method allows inserting values into placeholders `{}`:

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

You can also specify positional and keyword arguments:

```
print("{1} is learning {0}".format("Python", "Alice"))  # Output:
print("{name} is {age} years old".format(name="Bob", age=25))
```

f-Strings (Formatted String Literals)

Introduced in Python 3.6, f-strings are the most concise and readable way to format strings:

```
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

Using Expressions in f-Strings

You can perform calculations directly inside f-strings:

```
x = 10
y = 5
print(f"The sum of {x} and {y} is {x + y}")
```

Formatting Numbers

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

Padding and Alignment

```
text = "Python"
print(f"{text:>10}")  # Right align
print(f"{text:<10}")  # Left align
print(f"{text:^10}")  # Center align
```

Important Notes

> - **Escape Sequences**: Use `\n`, `\t`, `\'`, `\"`, and `\\` to handle special characters in strings.
> - **Raw Strings**: Use `r"string"` to prevent escape sequence interpretation.
> - **String Encoding & Decoding**: Use `.encode()` and `.decode()` to work with different text encodings.
> - **String Immutability**: Strings in Python are immutable, meaning they cannot be changed after creation.
> - **Performance Considerations**: Using `''.join(list_of_strings)` is more efficient than concatenation in loops.

Summary

- `.format()` allows inserting values into placeholders.
- f-strings provide an intuitive and readable way to format strings.
- f-strings support expressions, calculations, and formatting options.

Functions and Modules

1. Defining Functions in Python

Functions help in reusability and modularity in Python.

Syntax:

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

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

Key Points:

- Defined using `def` keyword.
- Function name should be meaningful.
- Use `return` to send a value back.

---

2. Function Arguments & Return Values

Functions can take parameters and return values.

Types of Arguments:

1. **Positional Arguments**

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

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

2. **Default Arguments**

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

print(greet())  # Output: Hello, Guest!
```

3. **Keyword Arguments**

```python
def student(name, age):
    print(f"Name: {name}, Age: {age}")

student(age=20, name="Bob")
```

---

3. Lambda Functions in Python

Lambda functions are anonymous, inline functions.

Syntax:

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

Example:

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

---

4. Recursion in Python

A function calling itself to solve a problem.

Example: Factorial using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

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

Important Notes:

- Must have a base case to avoid infinite recursion.
- Used in algorithms like Fibonacci, Tree Traversals.

---

5. Modules and Pip - Using External Libraries

Importing Modules

Python provides built-in and third-party modules.

Example: Using the `math` module

```
import math

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

Creating Your Own Module

Save this as `mymodule.py` :

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

Import in another file:

```python
import mymodule
print(mymodule.greet("Alice"))  # Output: Hello, Alice!
```

Installing External Libraries with `pip`

```
pip install requests
```

Example usage:

```python
import requests

response = requests.get("https://api.github.com")
print(response.status_code)
```

6. Function Scope and Lifetime

In Python, variables have **scope** (where they can be accessed) and **lifetime** (how long they exist). Variables are created when a function is called and destroyed when it returns. Understanding scope helps avoid unintended errors and improves code organization.

Types of Scope in Python

1. **Local Scope** (inside a function) – Variables declared inside a function are accessible only within that function.
2. **Global Scope** (accessible everywhere) – Variables declared outside any function can be used throughout the program.

Example:

```python
x = 10  # Global variable

def my_func():
    x = 5  # Local variable
    print(x)  # Output: 5

my_func()
print(x)  # Output: 10 (global x remains unchanged)
```

Using the `global` Keyword

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

```python
x = 10  # Global variable

def modify_global():
    global x
    x = 5  # Modifies the global x

modify_global()
print(x)  # Output: 5
```

This allows functions to change global variables, but excessive use of `global` is discouraged as it can make debugging harder.

7. Docstrings - Writing Function Documentation

Docstrings are used to document functions, classes, and modules. In Python, they are written in triple quotes. They are accessible using the `__doc__` attribute. Here's an example:

```python
def add(a, b):
    """Returns the sum of two numbers."""
    return a + b
```

```python
print(add.__doc__)  # Output: Returns the sum of two numbers.
```

Here is even proper way to write docstrings:

```python
def add(a, b):
    """
    Returns the sum of two numbers.

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of the two numbers.
    """
    return a + b
```

Summary

- Functions help in reusability and modularity.
- Functions can take arguments and return values.
- Lambda functions are short, inline functions.
- Recursion is a technique where a function calls itself.
- Modules help in organizing code and using external libraries.
- Scope and lifetime of variables decide their accessibility.
- Docstrings are used to document functions, classes, and modules.

Data Structures in Python

Python provides powerful built-in data structures to store and manipulate collections of data efficiently.

---

1. Lists and List Methods

Lists are ordered, mutable (changeable) collections of items.

Creating a List:

```
numbers = [1, 2, 3, 4, 5]
mixed = [10, "hello", 3.14]
```

Common List Methods:

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

my_list.append(4)    # [1, 2, 3, 4]
my_list.insert(1, 99)  # [1, 99, 2, 3, 4]
my_list.remove(2)    # [1, 99, 3, 4]
my_list.pop()        # Removes last element -> [1, 99, 3]
my_list.reverse()    # [3, 99, 1]
my_list.sort()       # [1, 3, 99]
```

List Comprehensions (Efficient List Creation)

```
squared = [x**2 for x in range(5)]
print(squared)  # Output: [0, 1, 4, 9, 16]
```

---

2. Tuples and Operations on Tuples

Tuples are ordered but **immutable** collections (cannot be changed after creation).

Creating a Tuple:

```
my_tuple = (10, 20, 30)
single_element = (5,)  # Tuple with one element (comma required)
```

Accessing Tuple Elements:

```
print(my_tuple[1])  # Output: 20
```

Tuple Unpacking:

```
a, b, c = my_tuple
print(a, b, c)  # Output: 10 20 30
```

Common Tuple Methods:

Method	Description	Example	Output
`count(x)`	Returns the number of times `x` appears in the tuple	`(1, 2, 2, 3).count(2)`	`2`
`index(x)`	Returns the index of the first occurrence of `x`	`(10, 20, 30).index(20)`	`1`

```
my_tuple = (1, 2, 2, 3, 4)
print(my_tuple.count(2))  # Output: 2

print(my_tuple.index(3))   # Output: 3
```

Why Use Tuples?

- **Faster than lists** (since they are immutable)
- **Used as dictionary keys** (since they are hashable)
- **Safe from unintended modifications**

3. Sets and Set Methods

Sets are **unordered, unique collections** (no duplicates).

Creating a Set:

```
fruits = {"apple", "banana", "cherry"}
```

Key Set Methods:

```
my_set = {1, 2, 3, 4}

my_set.add(5)          # {1, 2, 3, 4, 5}
my_set.remove(2)       # {1, 3, 4, 5}
my_set.discard(10)     # No error if element not found
my_set.pop()           # Removes random element
```

Set Operations:

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a.union(b))         # {1, 2, 3, 4, 5}
print(a.intersection(b))  # {3}
print(a.difference(b))    # {1, 2}
```

Use Case: Sets are great for eliminating duplicate values.

---

4. Dictionaries and Dictionary Methods

Dictionaries store key-value pairs and allow fast lookups.

Creating a Dictionary:

```
student = {"name": "Alice", "age": 21, "grade": "A"}
```

Accessing & Modifying Values:

```
print(student["name"])  # Output: Alice
student["age"] = 22      # Updating value
student["city"] = "New York"  # Adding new key-value pair
```

Common Dictionary Methods:

```
print(student.keys())    # dict_keys(['name', 'age', 'grade', 'ci
print(student.values())  # dict_values(['Alice', 22, 'A', 'New Yo
print(student.items())   # dict_items([('name', 'Alice'), ('age',

student.pop("age")  # Removes "age" key
student.clear()  # Empties dictionary
```

Dictionary Comprehensions:

```
squares = {x: x**2 for x in range(5)}
print(squares)  # {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

5. When to Use Each Data Structure?

Data Structure	Features	Best For
List	Ordered, Mutable	Storing sequences, dynamic data
Tuple	Ordered, Immutable	Fixed collections, dictionary keys

Data Structure	Features	Best For
Set	Unordered, Unique	Removing duplicates, set operations
Dictionary	Key-Value Pairs	Fast lookups, structured data

Object-Oriented Programming (OOP) in Python

We'll now explore how to organize and structure your Python code using objects, making it more manageable, reusable, and easier to understand.

1. What is OOP Anyway?

Imagine you're building with LEGOs. Instead of just having a pile of individual bricks (like in *procedural programming*), OOP lets you create pre-assembled units – like a car, a house, or a robot. These units have specific parts (data) and things they can do (actions).

That's what OOP is all about. It's a way of programming that focuses on creating "objects." An object is like a self-contained unit that bundles together:

- **Data (Attributes):** Information about the object. For a car, this might be its color, model, and speed.
- **Actions (Methods):** Things the object can do. A car can accelerate, brake, and turn.

Why Bother with OOP?

OOP offers several advantages:

- **Organization:** Your code becomes more structured and easier to navigate. Large projects become much more manageable.
- **Reusability:** You can use the same object "blueprints" (classes) multiple times, saving you from writing the same code over and over.
- **Easier Debugging:** When something goes wrong, it's often easier to pinpoint the problem within a specific, self-contained object.
- **Real-World Modeling:** OOP allows you to represent real-world things and their relationships in a natural way.

The Four Pillars of OOP

OOP is built on four fundamental principles:

1. **Abstraction:** Think of driving a car. You use the steering wheel, pedals, and gearshift, but you don't need to know the complex engineering under the hood. Abstraction means hiding complex details and showing only the essential information to the user.
2. **Encapsulation:** This is like putting all the car's engine parts inside a protective casing. Encapsulation bundles data (attributes) and the methods that operate on that data *within* a class. This protects the data from being accidentally changed or misused from outside the object. It controls access.
3. **Inheritance:** Imagine creating a "SportsCar" class. Instead of starting from scratch, you can build it upon an existing "Car" class. The "SportsCar" *inherits* all the features of a "Car" (like wheels and an engine) and adds its own special features (like a spoiler). This promotes code reuse and reduces redundancy.
4. **Polymorphism:** "Poly" means many, and "morph" means forms. This means objects of different classes can respond to the same "message" (method call) in their own specific way. For example, both a "Dog" and a "Cat" might have a `make_sound()` method. The dog will bark, and the cat will meow – same method name, different behavior.

2. Classes and Objects: The Blueprint and the Building

- **Class:** Think of a class as a blueprint or a template. It defines what an object *will be like* – what data it will hold and what actions it can perform. It doesn't create the object itself, just the instructions for creating it. It's like an architectural plan for a house.

- **Object (Instance):** An object is a *specific instance* created from the class blueprint. If "Car" is the class, then *your* red Honda Civic is an object (an instance) of the "Car" class. Each object has its own unique set of data. It's like the actual house built from the architectural plan.

Let's see this in Python:

```
class Dog:  # We define a class called "Dog"
    species = "Canis familiaris" # A class attribute (shared by a
```

```
    def __init__(self, name, breed):  # The constructor (explaine
        self.name = name      # An instance attribute to store the
        self.breed = breed    # An instance attribute to store the

    def bark(self):           # A method (an action the dog can do
        print(f"{self.name} says Woof!")

# Now, let's create some Dog objects:
my_dog = Dog("Buddy", "Golden Retriever")  # Creating an object c
another_dog = Dog("Lucy", "Labrador")     # Creating another obje

# We can access their attributes:
print(my_dog.name)        # Output: Buddy
print(another_dog.breed)  # Output: Labrador

# And make them perform actions:
my_dog.bark()             # Output: Buddy says Woof!
print(Dog.species)        # Output: Canis familiaris
```

- `self` **Explained:** Inside a class, `self` is like saying "this particular object." It's a way for the object to refer to *itself*. It's *always* the first parameter in a method definition, but Python handles it automatically when you call the method. You don't type `self` when *calling* the method; Python inserts it for you.
- **Class vs. Instance Attributes:**
    - **Class Attributes:** These are shared by *all* objects of the class. Like `species` in our `Dog` class. All dogs belong to the same species. They are defined outside of any method, directly within the class.
    - **Instance Attributes:** These are specific to *each individual object*. `name` and `breed` are instance attributes. Each dog has its own name and breed. They are usually defined within the `__init__` method.

3. The Constructor: Setting Things Up (`__init__`)

The `__init__` method is special. It's called the **constructor**. It's automatically run whenever you create a *new* object from a class.

What's it for? The constructor's job is to *initialize* the object's attributes – to give them their starting values. It sets up the initial state of the object.

```python
class Dog:
    def __init__(self, name, breed):  # The constructor
        self.name = name      # Setting the name attribute
        self.breed = breed    # Setting the breed attribute

# When we do this:
my_dog = Dog("Fido", "Poodle")  # The __init__ method is automati

# It's like we're saying:
# 1. Create a new Dog object.
# 2. Run the __init__ method on this new object:
#    - Set my_dog.name to "Fido"
#    - Set my_dog.breed to "Poodle"
```

You can also set default values for parameters in the constructor, making them optional when creating an object:

```python
class Dog:
    def __init__(self, name="Unknown", breed="Mixed"):
        self.name = name
        self.breed = breed

dog1 = Dog()          # name will be "Unknown", breed will be "Mi
dog2 = Dog("Rex")     # name will be "Rex", breed will be "Mixed"
dog3 = Dog("Bella", "Labrador") # name will be "Bella", breed wil
```

4. Inheritance: Building Upon Existing Classes

Inheritance is like a family tree. A child class (or *subclass*) inherits traits (attributes and methods) from its parent class (or *superclass*). This allows you to create new classes that are specialized versions of existing classes, without rewriting all the code.

```
class Animal:  # Parent class (superclass)
    def __init__(self, name):
        self.name = name

    def speak(self):
        print("Generic animal sound")

class Dog(Animal):  # Dog inherits from Animal (Dog is a subclass
    def speak(self):  # We *override* the speak method (more on t
        print("Woof!")

class Cat(Animal):  # Cat also inherits from Animal
    def speak(self):
        print("Meow!")

# Create objects:
my_dog = Dog("Rover")
my_cat = Cat("Fluffy")

# They both have a 'name' attribute (inherited from Animal):
print(my_dog.name)  # Output: Rover
print(my_cat.name)  # Output: Fluffy

# They both have a 'speak' method, but it behaves differently:
my_dog.speak()  # Output: Woof!
my_cat.speak()  # Output: Meow!
```

- **`super()`**: Inside a child class, `super()` lets you call methods from the parent class. This is useful when you want to *extend* the parent's behavior instead of completely replacing it. It's especially important when initializing the parent class's part of a child object.

```python
# Calling Parent Constructor with super()
class Bird(Animal):
    def __init__(self, name, wingspan):
        super().__init__(name)  # Call Animal's __init__ to set t
        self.wingspan = wingspan # Add a Bird-specific attribute

my_bird = Bird("Tweety", 10)
print(my_bird.name)      # Output: Tweety (set by Animal's constr
print(my_bird.wingspan)  # Output: 10   (set by Bird's constructo
```

5. Polymorphism: One Name, Many Forms

Polymorphism, as we saw with the `speak()` method in the inheritance example, means that objects of different classes can respond to the same method call in their own specific way. This allows you to write code that can work with objects of different types without needing to know their exact class.

6. Method Overriding: Customizing Inherited Behavior

Method overriding is *how* polymorphism is achieved in inheritance. When a child class defines a method with the *same name* as a method in its parent class, the child's version *overrides* the parent's version *for objects of the child class*. This allows specialized behavior in subclasses. The parent class's method is still available (using `super()`), but when you call the method on a child class object, the child's version is executed.

7. Operator Overloading: Making Operators Work with Your Objects

Python lets you define how standard operators (like `+`, `-`, `==`) behave when used with objects of your own classes. This is done using special methods called "magic methods" (or "dunder methods" because they have double underscores before and after the name).

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # Overloading the + operator
        #  'other' refers to the object on the *right* side of th
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self): # String representation (for print() and s
        return f"({self.x}, {self.y})"

    def __eq__(self, other): # Overloading == operator
        return self.x == other.x and self.y == other.y

p1 = Point(1, 2)
p2 = Point(3, 4)

p3 = p1 + p2  # This now works!  It calls p1.__add__(p2)
print(p3)      # Output: (4, 6)  (This uses the __str__ method)

print(p1 == p2) # Output: False (This uses the __eq__ method)
```

Other useful magic methods: (You don't need to memorize them all, but be aware they exist!)

- `__sub__` (`-`), `__mul__` (`*`), `__truediv__` (`/`), `__eq__` (`==`), `__ne__` (`!=`), `__lt__` (`<`), `__gt__` (`>`), `__len__` (`len()`), `__getitem__` , `__setitem__` , `__delitem__` (for list/dictionary-like behavior – allowing you to use `[]` with your objects).

8. Getters and Setters: Controlling Access to Attributes

Getters and setters are methods that you create to control how attributes of your class are accessed and modified. They are a key part of the principle of *encapsulation*. Instead of directly accessing an attribute (like `my_object.attribute`), you use methods to get and set its value. This might seem like extra work, but it provides significant advantages.

Why use them?

- **Validation:** You can add checks within the setter to make sure the attribute is set to a *valid* value. For example, you could prevent an age from being negative.
- **Read-Only Attributes:** You can create a getter *without* a setter, making the attribute effectively read-only from outside the class. This protects the attribute from being changed accidentally.
- **Side Effects:** You can perform other actions when an attribute is accessed or modified. For instance, you could update a display or log a change whenever a value is set.
- **Maintainability and Flexibility:** If you decide to change *how* an attribute is stored internally (maybe you switch from storing degrees Celsius to Fahrenheit), you only need to update the getter and setter methods. You don't need to change every other part of your code that uses the attribute. This makes your code much easier to maintain and modify in the future.

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age  # Convention: _age indicates it's intend

    def get_age(self):  # Getter for age
        return self._age

    def set_age(self, new_age):  # Setter for age
        if new_age >= 0 and new_age <= 150:  # Validation
            self._age = new_age
        else:
            print("Invalid age!")

person = Person("Alice", 30)
print(person.get_age())  # Output: 30

person.set_age(35)
print(person.get_age())  # Output: 35
```

```python
person.set_age(-5)   # Output: Invalid age!
print(person.get_age())   # Output: 35 (age wasn't changed)
```

The Pythonic Way: `@property` Decorator

Python offers a more elegant and concise way to define getters and setters using the `@property` decorator. This is the preferred way to implement them in modern Python.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age  # Convention: _age for "private" attribu

    @property  # This makes 'age' a property (the getter)
    def age(self):
        return self._age

    @age.setter # This defines the setter for the 'age' property
    def age(self, new_age):
        if new_age >= 0 and new_age <= 150:
            self._age = new_age
        else:
            print("Invalid age!")

person = Person("Bob", 40)
print(person.age)    # Output: 40  (Looks like direct attribute a
person.age = 45      # (Calls the setter - looks like attribute a
print(person.age)
person.age = -22 #Output: Invalid age!
```

With `@property`, accessing and setting the `age` attribute *looks* like you're working directly with a regular attribute, but you're actually using the getter and setter methods behind the scenes. This combines the convenience of direct access with the control and protection of encapsulation.

Private Variables (and the `_` convention):

It’s important to understand that Python does *not* have truly private attributes in the same way that languages like Java or C++ do. There’s no keyword that completely prevents access to an attribute from outside the class.

Instead, Python uses a *convention*: An attribute name starting with a single underscore (`_`) signals to other programmers that this attribute is intended for *internal use within the class*. It’s a strong suggestion: “Don’t access this directly from outside the class; use the provided getters and setters instead.” It’s like a “Please Do Not Touch” sign.

```
class MyClass:
    def __init__(self):
        self._internal_value = 0  #  Convention: _ means "private

    def get_value(self):
        return self._internal_value

obj = MyClass()
# print(obj._internal_value)  # This *works*, but it's against co
print(obj.get_value())       # This is the preferred way
```

While you *can* still access `obj._internal_value` directly, doing so is considered bad practice and can lead to problems if the internal implementation of the class changes. Always respect the underscore convention! It’s about good coding style and collaboration.

Python: Advanced Concepts

This section covers several advanced concepts in Python, including decorators, getters and setters, static and class methods, magic methods, exception handling, map/filter/reduce, the walrus operator, and *args/**kwargs.

Decorators in Python

Introduction

Decorators in Python are a powerful and expressive feature that allows you to modify or enhance functions and methods in a clean and readable way. They provide a way to wrap additional functionality around an existing function without permanently modifying it. This is often referred to as *metaprogramming*, where one part of the program tries to modify another part of the program at compile time.

Decorators use Python’s higher-order function capability, meaning functions can accept other functions as arguments and return new functions.

Understanding Decorators

A decorator is simply a callable (usually a function) that takes another function as an argument and returns a replacement function. The replacement function typically *extends* or *alters* the behavior of the original function.

Basic Example of a Decorator

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is call
        func()
        print("Something is happening after the function is calle
    return wrapper
```

```python
@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Output:

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

Here, `@my_decorator` is syntactic sugar for `say_hello = my_decorator(say_hello)`. It modifies the behavior of `say_hello()` by wrapping it inside `wrapper()`. The `wrapper` function adds behavior before and after the original function call.

Using Decorators with Arguments

Decorators themselves can also accept arguments. This requires another level of nesting: an outer function that takes the decorator's arguments and returns the actual decorator function.

```python
def repeat(n):
    def decorator(func):
        def wrapper(a):
            for _ in range(n):
                func(a)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")
```

```python
greet("world")
```

Output:

```
Hello, world!
Hello, world!
Hello, world!
```

In this example, `repeat(3)` *returns* the decorator function. The `@` syntax then applies that returned decorator to `greet`. The argument in the `wrapper` function ensures that the decorator can be used with functions that take any number of positional and keyword arguments.

Chaining Multiple Decorators

You can apply multiple decorators to a single function. Decorators are applied from bottom to top (or, equivalently, from the innermost to the outermost).

```python
def uppercase(func):
    def wrapper():
        return func().upper()
    return wrapper

def exclaim(func):
    def wrapper():
        return func() + "!!!"
    return wrapper

@uppercase
@exclaim
def greet():
    return "hello"

print(greet())
```

Output:

```
HELLO!!!
```

Here, `greet` is first decorated by `exclaim`, and then the result of *that* is decorated by `uppercase`. It's equivalent to `greet = uppercase(exclaim(greet))`.

Recap

Decorators are a key feature in Python that enable code reusability and cleaner function modifications. They are commonly used for:

- **Logging:** Recording when a function is called and its arguments.
- **Timing:** Measuring how long a function takes to execute.
- **Authentication and Authorization:** Checking if a user has permission to access a function.
- **Caching:** Storing the results of a function call so that subsequent calls with the same arguments can be returned quickly.
- **Rate Limiting:** Controlling how often a function can be called.
- **Input Validation:** Checking if the arguments to a function meet certain criteria.
- **Instrumentation:** Adding monitoring and profiling to functions.

Frameworks like Flask and Django use decorators extensively for routing, authentication, and defining middleware.

Getters and Setters in Python

Introduction

In object-oriented programming, **getters** and **setters** are methods used to control access to an object's attributes (also known as properties or instance variables). They provide a way to *encapsulate* the internal representation of an object, allowing you to validate data, enforce constraints, and perform other operations when an attribute is accessed or modified. While Python doesn't have private variables in the same way as languages like Java, the convention is to use a leading underscore (`_`) to indicate that an attribute is intended for internal use.

Using getters and setters helps:

- **Encapsulate data and enforce validation:** You can check if the new value meets certain criteria before assigning it.
- **Control access to "private" attributes:** By convention, attributes starting with an underscore are considered private, and external code should use getters/setters instead of direct access.
- **Make the code more maintainable:** Changes to the internal representation of an object don't necessarily require changes to code that uses the object.
- **Add additional logic**: Logic can be added when getting or setting attributes.

Using Getters and Setters

Traditional Approach (Using Methods)

A basic approach is to use explicit getter and setter methods:

```
class Person:
    def __init__(self, name):
        self._name = name  # Convention: underscore (_) denotes a

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        self._name = new_name

p = Person("Alice")
print(p.get_name())  # Alice
p.set_name("Bob")
print(p.get_name())  # Bob
```

Using `@property` (Pythonic Approach)

Python provides a more elegant and concise way to implement getters and setters using the `@property` decorator. This allows you to access and modify attributes using the usual dot notation (e.g., `p.name`) while still having the benefits of getter and setter methods.

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

    @property
    def name(self):  # Getter
        return self._name

    @name.setter
    def name(self, new_name):  # Setter
        self._name = new_name

p = Person("Alice")
print(p.name)  # Alice (calls the getter)

p.name = "Bob"  # Calls the setter
print(p.name)  # Bob
```

Benefits of `@property`:

- **Attribute-like access:** You can use `obj.name` instead of `obj.get_name()` and `obj.set_name()`, making the code cleaner and more readable.
- **Consistent interface:** The external interface of your class remains consistent even if you later decide to add validation or other logic to the getter or setter.
- **Read-only properties:** You can create read-only properties by simply omitting the `@property.setter` method (see the next section).
- `@property.deleter`: deletes a property. Here is an example:

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

    @property
    def name(self):  # Getter
        return self._name

    @name.setter
    def name(self, new_name):  # Setter
        self._name = new_name

    @name.deleter
    def name(self):
        del self._name

p = Person("Alice")
print(p.name)  # Alice
del p.name
print(p.name)  # AttributeError: 'Person' object has no attrib
```

Read-Only Properties

If you want an attribute to be **read-only**, define only the `@property` decorator (the getter) and omit the `@name.setter` method. Attempting to set the attribute will then raise an `AttributeError`.

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @property
```

```python
    def area(self):  # Read-only computed property
        return 3.1416 * self._radius * self._radius

c = Circle(5)
print(c.radius)  # 5
print(c.area)  # 78.54

# c.radius = 10  # Raises AttributeError: can't set attribute
# c.area = 20 # Raises AttributeError: can't set attribute
```

Recap

- **Getters and Setters** provide controlled access to an object's attributes, promoting encapsulation and data validation.
- The `@property` decorator offers a cleaner and more Pythonic way to implement getters and setters, allowing attribute-like access.
- You can create **read-only properties** by defining only a getter (using `@property` without a corresponding `@<attribute>.setter`).
- Using `@property` , you can dynamically compute values (like the `area` in the `Circle` example) while maintaining an attribute-like syntax.

Static and Class Methods in Python

Introduction

In Python, methods within a class can be of three main types:

- **Instance Methods:** These are the most common type of method. They operate on *instances* of the class (objects) and have access to the instance's data through the `self` parameter.
- **Class Methods:** These methods are bound to the *class* itself, not to any particular instance. They have access to class-level attributes and can be used to modify the class state. They receive the class itself (conventionally named `cls`) as the first argument.

- **Static Methods:** These methods are associated with the class, but they don't have access to either the instance (`self`) or the class (`cls`). They are essentially regular functions that are logically grouped within a class for organizational purposes.

Instance Methods (Default Behavior)

Instance methods are the default type of method in Python classes. They require an instance of the class to be called, and they automatically receive the instance as the first argument (`self`).

```
class Dog:
    def __init__(self, name):
        self.name = name  # Instance attribute

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

dog = Dog("Buddy")
print(dog.speak())  # Buddy says Woof!
```

Class Methods (`@classmethod`)

A class method is marked with the `@classmethod` decorator. It takes the class itself (`cls`) as its first parameter, rather than the instance (`self`). Class methods are often used for:

- **Modifying class attributes:** They can change the state of the class, which affects all instances of the class.
- **Factory methods:** They can be used as alternative constructors to create instances of the class in different ways.

```
class Animal:
    species = "Mammal"  # Class attribute
```

```
    @classmethod
    def set_species(cls, new_species):
        cls.species = new_species  # Modifies class attribute

    @classmethod
    def get_species(cls):
        return cls.species

print(Animal.get_species())  # Mammal
Animal.set_species("Reptile")
print(Animal.get_species())  # Reptile

# You can also call class methods on instances, but it's less com
a = Animal()
print(a.get_species()) # Reptile
```

Example: Alternative Constructor

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

    @classmethod
    def from_string(cls, data):
        name, age = data.split("-")
        return cls(name, int(age))  # Creates a new Person instan

p = Person.from_string("Alice-30")
print(p.name, p.age)  # Alice 30
```

In this example, `from_string` acts as a factory method, providing an alternative way to create `Person` objects from a string.

Static Methods (`@staticmethod`)

Static methods are marked with the `@staticmethod` decorator. They are similar to regular functions, except they are defined within the scope of a class.

- They **don't** take `self` or `cls` as parameters.
- They are useful when a method is logically related to a class but doesn't need to access or modify the instance or class state.
- Often used for utility functions that are related to the class

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b

print(MathUtils.add(3, 5))  # 8

#Can also be called on an instance
m = MathUtils()
print(m.add(4,5)) # 9
```

When to Use Static Methods?

- When a method is logically related to a class but doesn't require access to instance-specific or class-specific data.
- For utility functions that perform operations related to the class's purpose (e.g., mathematical calculations, string formatting, validation checks).

Key Differences Between Method Types

Method Type	Requires `self` ?	Requires `cls` ?	Can Access Instance Attributes?	Can Modify Class Attributes?
Instance Method	☑ Yes	✖ No	☑ Yes	☑ Yes (indirectly)

Method Type	Requires `self` ?	Requires `cls` ?	Can Access Instance Attributes?	Can Modify Class Attributes?
Class Method	❌ No	✅ Yes	❌ No (directly)	✅ Yes
Static Method	❌ No	❌ No	❌ No	❌ No

Recap

- **Instance methods** are the most common type and operate on individual objects (`self`).
- **Class methods** operate on the class itself (`cls`) and are often used for factory methods or modifying class-level attributes.
- **Static methods** are utility functions within a class that don't depend on the instance or class state. They're like regular functions that are logically grouped with a class.

Magic (Dunder) Methods in Python

Introduction

Magic methods, also called **dunder (double underscore) methods**, are special methods in Python that have double underscores at the beginning and end of their names (e.g., `__init__` , `__str__` , `__add__`). These methods allow you to define how your objects interact with built-in Python operators, functions, and language constructs. They provide a way to implement *operator overloading* and customize the behavior of your classes in a Pythonic way.

They are used to:

- Customize object creation and initialization (`__init__` , `__new__`).
- Enable operator overloading (e.g., `+` , `-` , `*` , `==` , `<` , `>`).

- Provide string representations of objects (`__str__` , `__repr__`).
- Control attribute access (`__getattr__` , `__setattr__` , `__delattr__`).
- Make objects callable (`__call__`).
- Implement container-like behavior (`__len__` , `__getitem__` , `__setitem__` , `__delitem__` , `__contains__`).
- Support with context managers (`__enter__` , `__exit__`)

Common Magic Methods

1. `__init__` – Object Initialization

The `__init__` method is the constructor. It's called automatically when a new instance of a class is created. It's used to initialize the object's attributes.

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

p = Person("Alice", 30)
print(p.name, p.age)  # Alice 30
```

---

2. `__str__` and `__repr__` – String Representation

- `__str__` : This method should return a human-readable, informal string representation of the object. It's used by the `str()` function and by `print()` .
- `__repr__` : This method should return an unambiguous, official string representation of the object. Ideally, this string should be a valid Python expression that could be used to recreate the object. It's used by the `repr()` function and in the interactive interpreter when you just type the object's name and press Enter.

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

    def __str__(self):
        return f"Person({self.name}, {self.age})"  # User-friendl

    def __repr__(self):
        return f"Person(name='{self.name}', age={self.age})"  # U

p = Person("Alice", 30)
print(str(p))    # Person(Alice, 30)
print(repr(p))   # Person(name='Alice', age=30)
print(p)          # Person(Alice, 30)  # print() uses __str__ if
```

If `__str__` is not defined, Python will use `__repr__` as a fallback for `str()` and `print()`. It's good practice to define *at least* `__repr__` for every class you create.

---

3. `__len__` – Define Behavior for `len()`

This method allows objects of your class to work with the built-in `len()` function. It should return the "length" of the object (however you define that).

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __len__(self):
        return self.pages

b = Book("Python 101", 250)
print(len(b))  # 250
```

---

4. `__add__`, `__sub__`, `__mul__`, etc. – Operator Overloading

These methods allow you to define how your objects behave with standard arithmetic and comparison operators.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
      return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2  # Calls __add__
print(v3)     # Vector(6, 8)
v4 = v3 - v1
print(v4)       # Vector(4, 5)
v5 = v1 * 5
print(v5) # Vector(10, 15)
```

Other common operator overloading methods include:

- `__eq__` (==)
- `__ne__` (!=)
- `__lt__` (<)
- `__gt__` (>)
- `__le__` (<=)

- `__ge__` (>=)
- `__truediv__` (/)
- `__floordiv__` (//)
- `__mod__` (%)
- `__pow__` (**)

Recap

Magic (dunder) methods are a powerful feature of Python that allows you to:

- Customize how your objects interact with built-in operators and functions.
- Make your code more intuitive and readable by using familiar Python syntax.
- Implement operator overloading, container-like behavior, and other advanced features.
- Define string representation.

Exception Handling and Custom Errors in Python

Introduction

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. Python provides a robust mechanism for handling exceptions using `try-except` blocks. This allows your program to gracefully recover from errors or unexpected situations, preventing crashes and providing informative error messages. You can also define your own custom exceptions to represent specific error conditions in your application.

Basic Exception Handling

The `try-except` block is the fundamental construct for handling exceptions:

- The `try` block contains the code that might raise an exception.

- The `except` block contains the code that will be executed if a specific exception occurs within the `try` block.

```python
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

Output:

```
Cannot divide by zero!
```

Handling Multiple Exceptions

You can handle multiple types of exceptions using multiple `except` blocks or by specifying a tuple of exception types in a single `except` block.

```python
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Invalid input! Please enter a number.")

# Alternative using a tuple:
try:
  num = int(input("Enter a number: "))
  result = 10 / num
except (ZeroDivisionError, ValueError) as e:
  print(f"An error occurred: {e}")
```

Using `else` and `finally`

- **`else`**: The `else` block is optional and is executed only if *no* exception occurs within the `try` block. It's useful for code that should run only when the `try` block succeeds.
- **`finally`**: The `finally` block is also optional and is *always* executed, regardless of whether an exception occurred or not. It's typically used for cleanup operations, such as closing files or releasing resources.

```
try:
    file = open("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("File not found!")
else:
    print("File read successfully.")
    print(f"File contents:\n{content}")
finally:
    file.close()  # Ensures the file is closed no matter what
```

Raising Exceptions (`raise`)

You can manually raise exceptions using the `raise` keyword. This is useful for signaling error conditions in your own code.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older!")
    return "Access granted."

try:
  print(check_age(20))  # Access granted.
  print(check_age(16))  # Raises ValueError
except ValueError as e:
  print(f"Error: {e}")
```

Custom Exceptions

Python allows you to define your own custom exception classes by creating a new class that inherits (directly or indirectly) from the built-in `Exception` class (or one of its subclasses). This makes your error handling more specific and informative.

```python
class InvalidAgeError(Exception):
    """Custom exception for invalid age."""
    def __init__(self, message="Age must be 18 or older!"):
        self.message = message
        super().__init__(self.message)

def verify_age(age):
    if age < 18:
        raise InvalidAgeError()  # Raise your custom exception
    return "Welcome!"

try:
    print(verify_age(16))
except InvalidAgeError as e:
    print(f"Error: {e}")
```

By defining custom exceptions, you can:

- Create a hierarchy of exceptions that reflect the specific error conditions in your application.
- Provide more informative error messages tailored to your application's needs.
- Make it easier for other parts of your code (or other developers) to handle specific errors appropriately.

Conclusion

- `try-except` blocks are essential for handling errors and preventing program crashes.
- Multiple `except` blocks or a tuple of exception types can be used to handle different kinds of errors.

- The `else` block executes only if no exception occurs in the `try` block.
- The `finally` block *always* executes, making it suitable for cleanup tasks.
- The `raise` keyword allows you to manually trigger exceptions.
- Custom exceptions (subclasses of `Exception`) provide a way to represent application-specific errors and improve error handling clarity.

Map, Filter, and Reduce

Introduction

`map`, `filter`, and `reduce` are higher-order functions in Python (and many other programming languages) that operate on iterables (lists, tuples, etc.). They provide a concise and functional way to perform common operations on sequences of data without using explicit loops. While they were more central to Python's functional programming style in earlier versions, list comprehensions and generator expressions often provide a more readable alternative in modern Python.

Map

The `map()` function applies a given function to each item of an iterable and returns an iterator that yields the results.

Syntax: `map(function, iterable, ...)`

- `function`: The function to apply to each item.
- `iterable`: The iterable (e.g., list, tuple) whose items will be processed.
- `...`: map can take multiple iterables. The function must take the same number of arguments

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

# Square each number using map
squared_numbers = map(lambda x: x**2, numbers)
print(list(squared_numbers))  # Output: [1, 4, 9, 16, 25]
```

```python
#Example with multiple iterables
numbers1 = [1, 2, 3]
numbers2 = [4, 5, 6]
summed = map(lambda x, y: x + y, numbers1, numbers2)
print(list(summed)) # Output: [5, 7, 9]


# Equivalent list comprehension:
squared_numbers_lc = [x**2 for x in numbers]
print(squared_numbers_lc)  # Output: [1, 4, 9, 16, 25]
```

Filter

The `filter()` function constructs an iterator from elements of an iterable for which a function returns `True`. In other words, it filters the iterable based on a condition.

Syntax: `filter(function, iterable)`

- `function`: A function that returns `True` or `False` for each item. If `None` is passed, it defaults to checking if the element is True (truthy value).
- `iterable`: The iterable to be filtered.

```python
numbers = [1, 2, 3, 4, 5, 6]

# Get even numbers using filter
even_numbers = filter(lambda x: x % 2 == 0, numbers)
print(list(even_numbers))  # Output: [2, 4, 6]

# Equivalent list comprehension:
even_numbers_lc = [x for x in numbers if x % 2 == 0]
print(even_numbers_lc) # Output: [2, 4, 6]

# Example with None as function
values = [0, 1, [], "hello", "", None, True, False]
truthy_values = filter(None, values)
print(list(truthy_values)) # Output: [1, 'hello', True]
```

Reduce

The `reduce()` function applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value. `reduce` is not a built-in function; it must be imported from the `functools` module.

Syntax: `reduce(function, iterable[, initializer])`

- `function` : A function that takes two arguments.
- `iterable` : The iterable to be reduced.
- `initializer` (optional): If provided, it's placed before the items of the iterable in the calculation and serves as a default when the iterable is empty.

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

# Calculate the sum of all numbers using reduce
sum_of_numbers = reduce(lambda x, y: x + y, numbers)
print(sum_of_numbers)  # Output: 15

# Calculate the product of all numbers using reduce
product_of_numbers = reduce(lambda x, y: x * y, numbers)
print(product_of_numbers)  # Output: 120

#reduce with initializer
empty_list_sum = reduce(lambda x,y: x+y, [], 0)
print(empty_list_sum) # 0

# Without the initializer:
# empty_list_sum = reduce(lambda x,y: x+y, []) # raises TypeError

# Equivalent using a loop (for sum):
total = 0
for x in numbers:
    total += x
print(total)  # 15
```

When to use map, filter, reduce vs. list comprehensions/generator expressions:

- **Readability:** List comprehensions and generator expressions are often more readable and easier to understand, especially for simple operations.
- **Performance:** In many cases, list comprehensions/generator expressions can be slightly faster than `map` and `filter`.
- **Complex Operations:** `reduce` can be useful for more complex aggregations where
- **Complex Operations:** `reduce` can be useful for more complex aggregations where the logic is not easily expressed in a list comprehension. `map` and `filter` may also be preferable when you already have a named function that you want to apply.
- **Functional Programming Style:** If you're working in a more functional programming style, `map`, `filter`, and `reduce` can fit naturally into your code.

Walrus Operator (:=)

Introduction

The walrus operator (`:=`), introduced in Python 3.8, is an *assignment expression* operator. It allows you to assign a value to a variable *within an expression*. This can make your code more concise and, in some cases, more efficient by avoiding repeated calculations or function calls. The name "walrus operator" comes from the operator's resemblance to the eyes and tusks of a walrus.

Use Cases

1. **Conditional Expressions:** The most common use case is within `if` statements, `while` loops, and list comprehensions, where you need to both test a condition *and* use the value that was tested.

   ```
   # Without walrus operator
   data = input("Enter a value (or 'quit' to exit): ")
   ```

```
while data != "quit":
    print(f"You entered: {data}")
    data = input("Enter a value (or 'quit' to exit): ")

# With walrus operator
while (data := input("Enter a value (or 'quit' to exit): "))
    print(f"You entered: {data}")
```

In the "with walrus" example, the input is assigned to `data` *and* compared to "quit" in a single expression.

2. **List Comprehensions:** You can avoid repeated calculations or function calls within a list comprehension.

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

# Without walrus operator:  calculate x * 2 twice
results = [x * 2 for x in numbers if x * 2 > 5]

# With walrus operator: calculate x * 2 only once
results = [y for x in numbers if (y := x * 2) > 5]
```

3. **Reading Files:** You can read lines from a file and process them within a loop.

```
# Without Walrus
with open("my_file.txt", "r") as f:
    line = f.readline()
    while line:
        print(line.strip())
        line = f.readline()

# With Walrus
with open("my_file.txt", "r") as f:
    while (line := f.readline()):
        print(line.strip())
```

Considerations

- **Readability:** While the walrus operator can make code more concise, it can also make it harder to read if overused. Use it judiciously where it improves clarity.
- **Scope:** The variable assigned using `:=` is scoped to the surrounding block (e.g., the `if` statement, `while` loop, or list comprehension).
- **Precedence:** The walrus operator has lower precedence than most other operators. Parentheses are often needed to ensure the expression is evaluated as intended.

Args and Kwargs

Introduction

`*args` and `**kwargs` are special syntaxes in Python function definitions that allow you to pass a variable number of arguments to a function. They are used when you don't know in advance how many arguments a function might need to accept.

- `*args` : Allows you to pass a variable number of *positional* arguments.
- `**kwargs` : Allows you to pass a variable number of *keyword* arguments.

`*args` (Positional Arguments)

`*args` collects any extra positional arguments passed to a function into a *tuple*. The name `args` is just a convention; you could use any valid variable name preceded by a single asterisk (e.g., `*values` , `*numbers`).

```
def my_function(*args):
    print(type(args))  # <class 'tuple'>
    for arg in args:
        print(arg)

my_function(1, 2, 3, "hello")  # Output: 1 2 3 hello
```

```python
my_function()  # No output (empty tuple)
my_function("a", "b")  # Output: a b
```

In this example, `*args` collects all positional arguments passed to `my_function` into the `args` tuple.

`**kwargs` (Keyword Arguments)

`**kwargs` collects any extra *keyword* arguments passed to a function into a *dictionary*. Again, `kwargs` is the conventional name, but you could use any valid variable name preceded by two asterisks (e.g., `**data`, `**options`).

```python
def my_function(**kwargs):
    print(type(kwargs))  # <class 'dict'>
    for key, value in kwargs.items():
        print(f"{key}: {value}")

my_function(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York

my_function()  # No output (empty dictionary)
my_function(a=1, b=2)
# Output:
# a: 1
# b: 2
```

In this example, `**kwargs` collects all keyword arguments into the `kwargs` dictionary.

Combining `*args` and `**kwargs`

You can use both `*args` and `**kwargs` in the same function definition. The order is important: `*args` must come *before* `**kwargs`. You can also include regular positional and keyword parameters.

```python
def my_function(a, b, *args, c=10, **kwargs):
    print(f"a: {a}")
    print(f"b: {b}")
    print(f"args: {args}")
    print(f"c: {c}")
    print(f"kwargs: {kwargs}")

my_function(1, 2, 3, 4, 5, c=20, name="Bob", country="USA")
# Output:
# a: 1
# b: 2
# args: (3, 4, 5)
# c: 20
# kwargs: {'name': 'Bob', 'country': 'USA'}

my_function(1,2)
# Output:
# a: 1
# b: 2
# args: ()
# c: 10
# kwargs: {}
```

Use Cases

- **Flexible Function Design:** `*args` and `**kwargs` make your functions more flexible, allowing them to handle a varying number of inputs without needing to define a specific number of parameters.
- **Decorator Implementation:** Decorators often use `*args` and `**kwargs` to wrap functions that might have different signatures.
- **Function Composition:** You can use `*args` and `**kwargs` to pass arguments through multiple layers of function calls.
- **Inheritance:** Subclasses can accept extra parameters to those defined by parent classes.

```python
# Example showing use in inheritance
class Animal:
```

```
  def __init__(self, name):
    self.name = name

class Dog(Animal):
  def __init__(self, name, breed, *args, **kwargs):
    super().__init__(name)
    self.breed = breed
    # Process any additional arguments or keyword arguments here
    print(f"args: {args}")
    print(f"kwargs: {kwargs}")

dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Lucy", "Labrador", 1,2,3, color="Black", age = 5)
```

Section 9: File Handling and OS Operations

This section introduces you to file handling in Python, which allows your programs to interact with files on your computer. We'll also explore basic operating system (OS) interactions using Python's built-in modules.

File I/O in Python

File Input/Output (I/O) refers to reading data from and writing data to files. Python provides built-in functions to make this process straightforward. Working with files generally involves these steps:

1. **Opening a file:** You need to open a file before you can read from it or write to it. This creates a connection between your program and the file.
2. **Performing operations:** You can then read data from the file or write data to it.
3. **Closing the file:** It's crucial to close the file when you're finished with it. This releases the connection and ensures that any changes you've made are saved.

Read, Write, and Append Files

Python provides several modes for opening files:

- **'r' (Read mode):** Opens the file for reading. This is the default mode. If the file doesn't exist, you'll get an error.
- **'w' (Write mode):** Opens the file for writing. If the file exists, its contents will be overwritten. If the file doesn't exist, a new file will be created.
- **'a' (Append mode):** Opens the file for appending. Data will be added to the end of the file. If the file doesn't exist, a new file will be created.

Here are some examples:

Reading from a file:

```
try:
    file = open("my_file.txt", "r")  # Open in read mode
```

```
    content = file.read()  # Read the entire file content
    print(content)
    file.close()  # Close the file
except FileNotFoundError:
    print("File not found.")


# Reading line by line
try:
    file = open("my_file.txt", "r")
    for line in file: # Efficient for large files
        print(line.strip()) # Remove newline characters
    file.close()
except FileNotFoundError:
    print("File not found.")
```

Writing to a file:

```
file = open("new_file.txt", "w")  # Open in write mode (creates o
file.write("Hello, world!\n")  # Write some text
file.write("This is a new line.\n")
file.close()
```

Appending to a file:

```
file = open("my_file.txt", "a")  # Open in append mode
file.write("This is appended text.\n")
file.close()
```

Using `with` statement (recommended):

The `with` statement provides a cleaner way to work with files. It automatically closes the file, even if errors occur.

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

```python
except FileNotFoundError:
    print("File not found.")

with open("output.txt", "w") as file:
    file.write("Data written using 'with'.\n")
```

OS and Shutil Modules in Python

Python's `os` module provides functions for interacting with the operating system, such as working with directories and files. The `shutil` module offers higher-level file operations.

`os` module examples:

```python
import os

# Get the current working directory
current_dir = os.getcwd()
print("Current directory:", current_dir)

# Create a new directory
# os.mkdir("new_directory")  # creates only one level of director
# os.makedirs("path/to/new_directory") # creates nested directori

# Change the current directory
# os.chdir("new_directory")

# List files and directories in a directory
files = os.listdir(".") # "." represents current directory
print("Files in current directory:", files)

# Remove a file or directory
# os.remove("my_file.txt")
# os.rmdir("new_directory") # removes empty directory
# shutil.rmtree("path/to/new_directory") # removes non-empty dire

# Rename a file or directory
# os.rename("old_name.txt", "new_name.txt")
```

```python
# Check if a file or directory exists
if os.path.exists("my_file.txt"):
    print("File exists")

# Join path components in a platform-independent way
path = os.path.join("folder", "subfolder", "file.txt")
print("Joined path:", path)
```

`shutil` module examples:

```python
import shutil

# Copy a file
# shutil.copy("my_file.txt", "my_file_copy.txt")

# Move a file or directory
# shutil.move("my_file.txt", "new_directory/")
```

Creating Command Line Utilities

You can use Python to create simple command-line utilities. The `argparse` module makes it easier to handle command-line arguments.

```python
import argparse

parser = argparse.ArgumentParser(description="A simple command-li
parser.add_argument("filename", help="The file to process.")
parser.add_argument("-n", "--number", type=int, default=1, help="

args = parser.parse_args()

try:
    with open(args.filename, "r") as file:
        content = file.read()
        for _ in range(args.number):
            print(content)
```

```
except FileNotFoundError:
    print("File not found.")
```

To run this script from the command line:

```
python my_script.py my_file.txt -n 3
```

This will print the contents of `my_file.txt` three times. You can learn more about `argparse` in the Python documentation.