Functions:

```
# A function to calculate the total cost of items
def calculate_total(items, tax_rate):
    total = sum(items)
    tax = total * tax_rate
    return total + tax

items = [100, 200, 300]
print(calculate_total(items, 0.05))  # Output: 630.0
```

Arguments and Parameters

- **Positional Arguments**: Arguments are matched by their position.
- **Keyword Arguments**: Arguments are matched by name.
- **Default Arguments**: Arguments that take a default value if not provided.
- **Variable-Length Arguments**: Allow arbitrary numbers of arguments (`*args` and `**kwargs`

```
# Variable-Length Arguments
def print_all(*args, **kwargs):
    print("Positional args:", args)
    print("Keyword args:", kwargs)

print_all(1, 2, 3, key1="value1", key2="value2")
# Output:
# Positional args: (1, 2, 3)
# Keyword args: {'key1': 'value1', 'key2': 'value2'}

**# A function to dynamically create a shopping list
def create_shopping_list(*items):
    return list(items)

shopping_list = create_shopping_list("Milk", "Eggs", "Bread")
print(shopping_list)  # Output: ['Milk', 'Eggs', 'Bread']
```

Lambda Functions

- Lambda functions are **anonymous functions** created using the `lambda` keyword.
- They are typically used for short, single-expression functions.

```
lambda arguments: expression

# Lambda function to calculate the square of a number
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Lambda with multiple arguments
add = lambda x, y: x + y
print(add(10, 20))  # Output: 30
```

```
# Sorting with a lambda function
data = [("Alice", 25), ("Bob", 20), ("Charlie", 30)]
sorted_data = sorted(data, key=lambda x: x[1])  # Sort by age
print(sorted_data)  # Output: [('Bob', 20), ('Alice', 25), ('Charlie', 30)]
```

----

1. Lists

- Ordered, mutable, and allows duplicates.
- Great for maintaining sequences of items.

2. Tuples

- Ordered, immutable, and allows duplicates.
- Useful for data that should not change.

```
# Storing GPS coordinates
locations = [("Home", (12.34, 56.78)), ("Office", (98.76, 54.32))]
for name, (lat, lon) in locations:
    print(f"{name}: Latitude {lat}, Longitude {lon}")
```

3. Sets

- Unordered, mutable, and no duplicates.
- Ideal for operations like intersection, union, and difference.

```
# Basic set operations
even = {2, 4, 6, 8}
odd = {1, 3, 5, 7}
all_numbers = even.union(odd)         # Combines sets
common = even.intersection({4, 5, 6})  # Finds common elements
print(all_numbers, common)  # Output: {1, 2, 3, 4, 5, 6, 7, 8}, {4, 6}


# Identify unique items purchased
purchases = ["apple", "banana", "apple", "cherry"]
unique_items = set(purchases)
print(unique_items)  # Output: {'apple', 'banana', 'cherry'}
```

Dict:

```
# Store product details
products = {
    101: {"name": "Laptop", "price": 75000},
    102: {"name": "Smartphone", "price": 20000},
}
for product_id, details in products.items():
    print(f"{product_id}: {details['name']} costs {details['price']}.")
```

----

5. List Comprehensions

A concise way to create lists based on existing iterables.

```
[expression for item in iterable if condition]

# Squares of even numbers
numbers = [1, 2, 3, 4, 5, 6]
squares = [x ** 2 for x in numbers if x % 2 == 0]
print(squares)  # Output: [4, 16, 36]

# Convert temperatures from Celsius to Fahrenheit
celsius = [0, 20, 30]
fahrenheit = [(c * 9/5) + 32 for c in celsius]
print(fahrenheit)  # Output: [32.0, 68.0, 86.0]
```

6. Dictionary Comprehensions

A compact way to create dictionaries.

```
{key_expression: value_expression for item in iterable if condition}

# Create a dictionary with numbers and their squares
squares_dict = {x: x ** 2 for x in range(1, 6)}
print(squares_dict)  # Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Map employee IDs to their salaries
employee_ids = [101, 102, 103]
salaries = [50000, 60000, 70000]
salary_dict = {emp_id: salary for emp_id, salary in zip(employee_ids, salaries)}
print(salary_dict)  # Output: {101: 50000, 102: 60000, 103: 70000}
```

Mutable Objects

- **Definition**: Mutable objects can be modified after their creation.
- **Examples**: `list`, `dict`, `set`, `bytearray`, user-defined objects.
- **Implications**: Changes made to mutable objects are reflected wherever the object is referenced.

Immutable Objects

- **Definition**: Immutable objects cannot be changed after they are created.
- **Examples**: `int`, `float`, `tuple`, `str`, `frozenset`, `bytes`.
- **Implications**: Any operation that appears to modify an immutable object creates a new object instead.

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

```
b = a  # b refers to the same object as a
b.append(4)
print(a)  # Output: [1, 2, 3, 4] (a is modified because b refers to it)

name = "Alice"
new_name = name.replace("A", "E")  # Creates a new string
print(name)       # Output: Alice (original string is unchanged)
print(new_name)   # Output: Ellice
```

Immutable objects are often faster because they don’t need mechanisms to track changes.

Strings:

String Methods

Python provides numerous built-in methods for string manipulation.

Method	Description	Example
`lower()` / `upper()`	Convert to lowercase/uppercase	`"Python".upper()` → `"PYTHON"`
`strip()`	Remove leading and trailing spaces	`" hello ".strip()` → `"hello"`
`replace()`	Replace substrings	`"Hello".replace("H", "J")` → `"Jello"`
`split()`	Split string into a list	`"a,b,c".split(",")` → `['a', 'b', 'c']`
`join()`	Join elements of a list into a string	`" ".join(['a', 'b', 'c'])` → `"a b c"`
`find()` / `index()`	Find position of a substring	`"Python".find("th")` → `2`
`startswith()` / `endswith()`	Check if string starts/ends with a substring	`"Python".startswith("Py")` → `True`
`isalpha()`, `isdigit()`	Check if all characters are alphabetic/numeric	`"123".isdigit()` → `True`

Python Regex:

2. Common Functions

Function	Description
`re.match()`	Matches a pattern at the start of the string.
`re.search()`	Searches the entire string for a match.
`re.findall()`	Returns all occurrences of a pattern as a list.
`re.sub()`	Replaces occurrences of a pattern with a new string.

Common Regex Patterns

Pattern	Matches	Example
¥d	Any digit	¥d+ → "123" matches 123
¥w	Any alphanumeric character	¥w+ → "Hello123" matches Hello123
¥s	Any whitespace	"Hello¥sWorld" matches "Hello World"
.	Any character except newline	a.c → Matches abc, axc
^	Start of string	^Hello → Matches Hello World
$	End of string	World$ → Matches Hello World
[]	Any character in brackets	[aeiou] → Matches any vowel
`	`	Logical OR
*	0 or more repetitions	a* → Matches aaaa, a, or empty

```
import re

# Match and Search
text = "The price of the item is $100."
match = re.match(r"The", text)  # Matches 'The' at the start
search = re.search(r"\$\d+", text)  # Matches '$100' anywhere
print(search.group())  # Output: $100

# Find All
email_text = "Contact us at support@example.com or sales@example.com."
emails = re.findall(r"\b[\w.-]+@[\w.-]+\.\w+\b", email_text)
print(emails)  # Output: ['support@example.com', 'sales@example.com']

# Substitution
censored = re.sub(r"\$\d+", "[REDACTED]", text)
print(censored)  # Output: The price of the item is [REDACTED].
```

Creating Templates:

```
template = "Dear {name}, your order {order_id} is confirmed."
message = template.format(name="John", order_id=12345)
print(message)  # Output: Dear John, your order 12345 is confirmed.
```

File Handling in Python

File handling allows Python programs to read, write, and manipulate files on the system.

1. Reading and Writing Files

Reading a File

The `open()` function is used to open files for reading. Common modes:

- `'r'`: Read (default mode).

```
# Sample file content: "Hello, World!\nWelcome to Python."
file_path = "sample.txt"

# Reading the entire file
with open(file_path, 'r') as file:
    content = file.read()
    print(content)
# Output:
# Hello, World!
# Welcome to Python.

# Reading line by line
with open(file_path, 'r') as file:
    for line in file:
        print(line.strip())  # Removes the newline character
```

Writing to a File

The `w` mode overwrites the file, while the `a` mode appends to it.

```
file_path = "output.txt"

# Writing new content (overwrites if the file exists)
with open(file_path, 'w') as file:
    file.write("This is a new file.\n")
    file.write("File handling in Python is easy!")

# Appending to the file
with open(file_path, 'a') as file:
    file.write("\nAppending this line to the file.")
```

Reading and Writing

Using the `'r+'` mode allows simultaneous reading and writing.

```
with open(file_path, 'r+') as file:
    content = file.read()
    file.write("\nAdding more content after reading.")
```

Context Managers (The `with` Statement)

The `with` statement ensures proper resource management:

- Automatically closes the file after execution.
- Reduces the risk of leaving files open accidentally.

JSON

Python `json` Module

Loading JSON Data (Parsing)

- Use `json.loads()` to parse JSON strings into Python objects.
- Use `json.load()` to parse JSON from a file.

```
import json


json_string = '{"name": "John", "age": 30, "skills": ["Python", "JavaScript"]}'
data = json.loads(json_string)


print(data['name'])  # Output: John
print(data['skills'])  # Output: ['Python', 'JavaScript']
----


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


print(data)  # Outputs the JSON content as a Python dictionary
```

Dumping JSON Data (Writing)

- Use `json.dumps()` to convert Python objects into JSON strings.
- Use `json.dump()` to write Python objects into JSON files.

```
with open("output.json", 'w') as file:
    json.dump(data, file, indent=4)
```

Handling Errors

- **JSONDecodeError**: Occurs when JSON is malformed.
- Always handle exceptions to ensure robustness.

```
import json

try:
    json_string = '{"name": "John", "age": 30,}'
    data = json.loads(json_string)
except json.JSONDecodeError as e:
    print(f"Invalid JSON: {e}")
```

Raising Exceptions

You can manually raise exceptions using the `raise` keyword.

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

try:
    print(check_age(16))
except ValueError as e:
    print(e)
```

Custom Exceptions

You can create user-defined exceptions by subclassing `Exception`.

```
class InvalidAgeError(Exception):
    def __init__(self, age, message="Age must be 18 or above"):
        self.age = age
        self.message = message
        super().__init__(self.message)

def check_age(age):
    if age < 18:
        raise InvalidAgeError(age)
    return "Access granted"
```

```
try:
    print(check_age(16))
except InvalidAgeError as e:
    print(f"InvalidAgeError: {e.message}. You entered: {e.age}")
```

Packages

A **package** is a collection of related Python modules grouped together in a directory with an `__init__.py` file.

```
my_package/
    __init__.py
    math_operations.py
    string_operations.py
```

```
# __init__.py
from .math_operations import add, subtract
from .string_operations import to_upper, to_lower
```

using the package functions (inside math and string) in another file

----

```
from my_package import add, to_upper

print(add(10, 5))           # Output: 15
print(to_upper("hello"))     # Output: HELLO
```

Dynamically importing moduels

```
module_name = "math"
math_module = __import__(module_name)

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

`random` Module

The `random` module in Python is used to generate pseudo-random numbers and sequences.

`random.random()`: Generates a random float between 0 and 1.
`random.randint(a, b)`: Generates a random integer between `a` and `b` (inclusive).
`random.choice(sequence)`: Picks a random element from a non-empty sequence.
`random.shuffle(sequence)`: Shuffles a sequence in place.

random.choices(population, k): Returns a list of k elements with replacement.
print(random.choices(['a', 'b', 'c'], k=3)) # Example: ['c', 'a', 'b']

datetime.date: Represents only the date
datetime.datetime: Represents date and time.
datetime.time: Represents only the time.

```
now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))  # Example: 2024-12-25 14:30:45


from datetime import datetime

date_str = "2024-12-25"
parsed_date = datetime.strptime(date_str, "%Y-%m-%d")
print(parsed_date)  # Output: 2024-12-25 00:00:00

from datetime import datetime, timedelta

today = datetime.now()
next_week = today + timedelta(days=7)
print(next_week)  # Example: 2025-01-01
```

-0----------------------------

```
import random

def random_number(length):
    if length <= 0:
        return ""
    return ''.join(random.choices('0123456789', k=length))

print(random_number(5))  # Example: '48392'
```

```
import random
import string

def random_alphabets(length):
    if length <= 0:
        return ""
    return ''.join(random.choices(string.ascii_letters, k=length))

print(random_alphabets(5))  # Example: 'aBcDe'
```

```
import random
import string

def random_combination(length):
    if length <= 0:
        return ""
    characters = string.ascii_letters + string.digits
    return ''.join(random.choices(characters, k=length))

print(random_combination(8))  # Example: 'A9b3Cd4E'
```

Date to string and vice versa

```
from datetime import datetime, timedelta

def add_days_to_date(date_str, days_to_add):
    # Parse the input string into a datetime object
    date_obj = datetime.strptime(date_str, "%Y-%m-%d")

    # Add the specified number of days
    new_date = date_obj + timedelta(days=days_to_add)

    # Convert back to a string
    return new_date.strftime("%Y-%m-%d")

# Example usage
input_date = "2024-12-25"
result = add_days_to_date(input_date, 5)
print(result)  # Output: 2024-12-30
```

Input Parsing: Use `datetime.strptime` to convert the input string into a `datetime` object.

- **Adding Days**: Use `timedelta(days=5)` to represent 5 days and add it to the `datetime` object.
- **Formatting the Result**: Use `strftime` to convert the resulting `datetime` object back into a string.

-0--

Concurrency vs Parallelism

---

1. What is Concurrency?

- Concurrency means dealing with multiple tasks **at the same time**.
- Tasks can start, run, and complete in overlapping time periods, but they may not necessarily run simultaneously.
- It's about **task switching**, often used when tasks involve waiting (like I/O operations).

2. What is Parallelism?

- Parallelism means running multiple tasks **exactly at the same time** on different processors or cores.
- It's about **task execution**, used to speed up compute-heavy operations by distributing work.

Concurrency vs Parallelism: Key Differences

Feature	Concurrency	Parallelism
Focus	Task switching and coordination	Simultaneous execution of tasks
Resource Use	Can run on a single CPU	Requires multiple CPUs/cores
Example	Handling multiple I/O requests	Matrix multiplication with multiple threads

---

Multithreading and Multiprocessing

---

1. Multithreading

- **Threads** share the same memory space.
- Suitable for I/O-bound tasks (e.g., reading files, making API calls).
- Limited by Python's **Global Interpreter Lock (GIL)**, which prevents multiple threads from running Python code simultaneously in a single process.

More on thread in the end of this module with examples:

2. Multiprocessing

- **Processes** have separate memory spaces.
- Suitable for CPU-bound tasks (e.g., image processing, data analysis).
- Not affected by the GIL since each process has its own Python interpreter.

```
from multiprocessing import Process

import time


def print_numbers():
    for i in range(5):
        print(i)
        time.sleep(1)

# Creating processes
process1 = Process(target=print_numbers)
process2 = Process(target=print_numbers)

# Starting processes
process1.start()
process2.start()

# Wait for processes to complete
process1.join()
process2.join()
```

Key Differences Between Multithreading and Multiprocessing

Feature	Multithreading	Multiprocessing
Memory	Threads share memory space	Processes have separate memory
GIL Impact	Affected by the GIL	Not affected by the GIL
Best Use Case	I/O-bound tasks	CPU-bound tasks
Complexity	Easier to implement	Requires more memory and setup

What is Async/Await?

- **Async/await** is a way to write **asynchronous code** in Python, primarily for I/O-bound tasks.
- Tasks run **concurrently**, using an event loop to handle waiting tasks without blocking the program.

How It Works

1. `async def`: Defines an asynchronous function.
2. `await`: Pauses the function until the awaited task completes.

```
import asyncio


async def task1():
    print("Task 1: Start")
    await asyncio.sleep(2)  # Simulate I/O operation
    print("Task 1: End")


async def task2():
    print("Task 2: Start")
    await asyncio.sleep(1)  # Simulate I/O operation
    print("Task 2: End")


async def main():
    await asyncio.gather(task1(), task2())  # Run tasks concurrently


# Run the event loop
asyncio.run(main())
```

```
Task 1: Start
Task 2: Start
Task 2: End
Task 1: End
```

What is Multithreading?

- **Thread**: A thread is like a mini-program running inside a program.
- **Multithreading**: Running multiple threads in the same program to do tasks concurrently (not necessarily in parallel).

When to Use Multithreading?

Multithreading is useful for tasks that involve waiting (e.g., downloading files, reading/writing from/to a database, or making API calls). It won't speed up CPU-heavy tasks due to Python's **Global Interpreter Lock (GIL)**.

- **Function Definition**:
- `print_numbers` is the task that threads will execute.

- It prints numbers with a delay (`time.sleep(delay)`) to simulate some work.
- **Thread Creation**:
- We create two threads (`thread1`, `thread2`) using the `Thread` class.
- `args=("Thread 1", 1)` passes arguments to the `print_numbers` function.
- **Thread Start**:
- `thread1.start()` and `thread2.start()` begin execution of the threads.
- **Thread Join**:
- `thread1.join()` and `thread2.join()` ensure the main program waits for the threads to finish.

Key Points

1. Threads **don't necessarily execute in order**; they share CPU time.
2. The `time.sleep()` simulates work and makes it easier to see interleaving.
3. Without `join()`, the main program could exit before threads finish.

```
import threading

import time


# Define a function to run in a thread

def print_numbers(name, delay):

    for i in range(5):

        time.sleep(delay)  # Simulate a task that takes time

        print(f"{name} prints: {i}")


# Create threads

thread1 = threading.Thread(target=print_numbers, args=("Thread 1", 1))

thread2 = threading.Thread(target=print_numbers, args=("Thread 2", 0.5))


# Start threads

thread1.start()

thread2.start()


# Wait for threads to finish

thread1.join()
```

```
thread2.join()

print("Both threads have finished!")

##########Output

Thread 1 prints: 0
Thread 2 prints: 0
Thread 2 prints: 1
Thread 1 prints: 1
Thread 2 prints: 2
Thread 1 prints: 2
Thread 2 prints: 3
Thread 1 prints: 3
Thread 2 prints: 4
Thread 1 prints: 4
Both threads have finished!
-------------------
```

Detailed Explanation of `target`

1. `target` **Parameter**:
    - The `target` parameter takes a reference to the function you want to run in a separate thread.
    - The function itself is not executed when passed. It is only **linked** to the thread and will be executed when the thread's `start()` method is called.
2. `args` **Parameter**:
    - If your function takes arguments, you pass them as a tuple to the `args` parameter.

In Python, due to the **Global Interpreter Lock (GIL)**, threads do not run in true parallel on multiple CPU cores. Instead, they take turns running on a single core. The operating system rapidly switches between threads (called **context switching**), making it appear as if they are running simultaneously.

Threads share the same memory space, which makes communication between them easy, but also introduces risks like **race conditions** if not handled properly.

Key Observations

1. **Context Switching**:

    - Threads "yield" execution when they hit `time.sleep()` or when the GIL is released. This allows another thread to execute.
2. **Interleaving**:
    - The exact interleaving depends on the operating system's thread scheduler. You may see different interleaving on different runs.
3. **Shared Memory**:
    - Both threads share the same memory space, so changes made to global variables in one thread are visible to the oth

What Happens Internally

1. **Thread Scheduling**:
    - The Python interpreter asks the OS to schedule threads. The OS gives each thread a small time slice.
2. **GIL Management**:
    - Python enforces that only one thread runs Python bytecode at a time. Threads frequently release and reacquire the GIL (e.g., during `time.sleep()` or I/O operations).
3. **Execution Stops and Resumes**:
    - When a thread is paused (e.g., during `time.sleep()`), its state (local variables, current instruction) is saved. When it resumes, execution continues from where it stopped.

Difference Between Iterable and Iterator

Feature	Iterable	Iterator
Definition	Can be iterated over.	Generates items one by one.
Methods Implemented	`__iter__()`	`__iter__()` and `__next__()`
Examples	Lists, tuples, strings, etc.	Result of calling `iter()`
When Exhausted	Can be reused repeatedly.	Can't be reused; create a new iterator.

Generators and `yield`

1. What Are Generators?

- Generators are a simpler way to create iterators.
- Instead of using `__iter__()` and `__next__()`, you define a generator function using `def` and `yield`.
- A generator is an iterator but is defined with less boilerplate code.

2. `yield` Statement

- The `yield` statement pauses the function and saves its state, returning a value to the caller.
- When the generator is resumed, it picks up where it left off

ow Generators Work

- A **generator function** contains one or more `yield` statements.
- When called, it doesn’t execute the function. Instead, it returns a generator object.
- Use `next()` to retrieve values from the generator.

```
---
def count_up_to(n):
    count = 1
    while count <= n:
        yield count  # Pause and return the current value
        count += 1

# Create generator
counter = count_up_to(3)

# Use the generator
print(next(counter))  # Output: 1
print(next(counter))  # Output: 2
print(next(counter))  # Output: 3

# Exhausted generator
try:
    print(next(counter))  # Raises StopIteration
except StopIteration:
    print("No more values!")
----
```

Practical example Generator:

```
def fibonacci(limit):
    a, b = 0, 1
    while a < limit:
        yield a
        a, b = b, a + b

# Create generator
fib_gen = fibonacci(10)

# Use generator in a loop
for value in fib_gen:
    print(value, end=" ")  # Output: 0 1 1 2 3 5 8
```

----------Implement class vs generator example:

```
class Counter:
    def __init__(self, max_count):
        self.current = 1
        self.max_count = max_count

    def __iter__(self):
        return self

    def __next__(self):
        if self.current <= self.max_count:
            value = self.current
            self.current += 1
            return value
        else:
            raise StopIteration

# Usage
counter = Counter(5)
for num in counter:
    print(num)
```

```
def counter(max_count):
    current = 1
    while current <= max_count:
        yield current
        current += 1

# Usage
for num in counter(5):
    print(num)
```

--Its much easier to write using generators --

------------===-----------

Object-Oriented Programming (OOP): Classes and Objects

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which are instances of classes. OOP allows you to organize your code into reusable and modular structures.

**Instance Attributes vs Class Attributes

- **Class Attributes**:
  - Shared across all instances of the class.
  - Defined directly within the class.
- **Instance Attributes**:
  - Specific to each object (instance).
  - Usually defined within a special method like `__init__`.

The `__init__` Method

The `__init__` method is a special method called when an object is created. It is used to initialize instance attributes.

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand  # Instance attribute
        self.model = model  # Instance attribute

    def drive(self):
        return f"The {self.brand} {self.model} is driving."


# Creating objects with different attributes
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")


print(car1.brand)  # Output: Toyota
print(car2.model)  # Output: Civic
print(car1.drive())  # Output: The Toyota Camry is driving.
```

Understanding `self` in Python Classes

The `self` keyword is a **reference to the current instance of the class**. It is used to access the attributes and methods of the object (instance) within the class.

Think of `self` as the way an object can refer to itself while the class is being executed. Every method in a class has to take `self` as its first parameter (except static methods).

Why Do We Need `self`?

1. **To Access Instance Attributes**:
    - Attributes belong to a specific object. Using `self`, we can differentiate between the attributes of one object and another.
    - Without `self`, Python wouldn't know which object's attributes to access.
2. **To Call Other Methods**:
    - When a method calls another method of the same object, `self` is used to refer to it.

Key Points About `self`

1. `self` **is Not a Keyword**:
    - It’s just a naming convention in Python. You can name it something else (like `this`), but it’s strongly recommended to use `self` for clarity.
2. `self` **is Passed Automatically**:
    - When you call an instance method, Python automatically passes the instance (`self`) as the first argument.

```
car1.display_info()
Car.display_info(car1)
```

** both are same

Different Objects Have Different `self`:

- Each object’s `self` refers only to itself, ensuring independence between objects.

When Do You Not Use `self`?

- **In Static Methods**:
    - Static methods don’t operate on instance data, so they don’t require `self`.
    - Use the `@staticmethod` decorator for such method

*** All three methods class vs instance vs static

```python
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Instance Methods

The first method on MyClass, called method, is a regular instance method. That’s the basic, no-frills method type you’ll use most of the time. You can see the method takes one parameter, self, which points to an instance of MyClass when the method is called (but of course instance methods can accept more than just one parameter).

Through the self parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object’s state.

Not only can they modify object state, instance methods can also access the class itself through the self.__class__ attribute. This means instance methods can also modify class state

--

Class Methods

Let’s compare that to the second method, MyClass.classmethod. I marked this method with a `@classmethod` decorator to flag it as a class method.

Instead of accepting a self parameter, class methods take a cls parameter that points to the class—and not the object instance—when the method is called.

Because the class method only has access to this cls argument, it can’t modify object instance state. That would require access to self. However, class methods can still modify class state that applies across all instances of the class.

--

Static Methods

The third method, MyClass.staticmethod was marked with a `@staticmethod` decorator to flag it as a static method.

This type of method takes neither a self nor a cls parameter (but of course it’s free to accept an arbitrary number of other parameters).

Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access - and they’re primarily a way to namespace your methods.

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x101a2f4c8>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: unbound method method() must
    be called with MyClass instance as first
    argument (got nothing instead)
```

Simple example to learn all 3

```
import string

class Car:
    myvlsvar = 10
    def __init__(self,year:int,name:str):
        self.name = name
        self.year = year

    def carDetails(self):
        Car.myvlsvar = 90
        print(f"This car is from {self.year} and its {self.name}")

    def classMethod(cls):
        print(f"This is a class method and class var is {cls.myvlsvar}")

    def mystaticMethod():
        Car.myvlsvar = 10
        print("This is my static method")

car = Car(2000,"BMW")
car.carDetails()
car.classMethod()

newcar = Car(300,"NJ")
newcar.classMethod()
```

—

static cant change anything , instance everything, class only class variable.

INHERITANCE:

```
class Employee:
    def __init__(self, name, id):
        self.name = name
        self.id = id


    def display_info(self):
        print(f"Name: {self.name}, ID: {self.id}")


class Manager(Employee):
    def __init__(self, name, id, department):
        super().__init__(name, id)  # Call the constructor of the parent class
        self.department = department


    def manage_team(self):
        print(f"Manager {self.name} is managing the {self.department} team.")
```

Create instances

employee1 = Employee("John Doe", 123)

manager1 = Manager("Jane Smith", 456, "Marketing")

employee1.display_info() # Output: Name: John Doe, ID: 123

manager1.display_info() # Output: Name: Jane Smith, ID: 456

manager1.manage_team() # Output: Manager Jane Smith is managing the Marketing team.

----

python supports multiple inheritance

Method Overloading in Python (Using Default Arguments)

While Python doesn't directly support method overloading like some other languages, you can achieve similar behavior using default arguments.

What is `super()`?

- `super()` is a special function in Python that refers to the parent class (superclass) of a class.
- It's used to:
    - **Call the constructor of the parent class:** This is crucial when you need to initialize attributes that are defined in the parent class.
    - **Access and call methods of the parent class:** This is useful when you want to extend or modify existing behavior from the parent class while still using the original implementation.

--- we can use super to also access super class variables and also change them (change without need of super just Class.var = newval)

-- To access

```
class Parent:
    class_var = 10


class Child(Parent):
    def access_parent_var(self):
        return super().class_var
```

```
child_obj = Child()
print(child_obj.access_parent_var())  # Output: 10
```

-------MULTIPLE INHERITANCE WITH SAME METHOD INSIDE TWO CLASS IT MOVES LINERALY------------

```
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")
        super().method()  # Calls A.method()

class C(A):
    def method(self):
        print("C")
        super().method()  # Calls A.method()

class D(B, C):
    def method(self):
        print("D")
        super().method()  # Calls B.method() which then calls A.method()

d = D()
d.method()
# Output:
# D
# B
# A
```

Encapsulation and Abstraction in Python

Encapsulation

- **Concept:** Encapsulation is the bundling of data (attributes) and methods (functions) that operate on that data within a single unit, typically a class. It's about restricting direct access to some components of an object.
- **Implementation in Python:**
  - **Private Attributes:** While Python doesn't have strict private access modifiers like some other languages, you can indicate a variable as "private" by prefixing it with a double underscore (e.g., `__name`). This makes it harder to access directly from outside the class.
  - **Getters and Setters:** You can create public methods (getters and setters) to control access to the attributes. This allows you to validate data before setting it or perform other actions when the attribute is accessed or modified.

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

    def get_name(self):
        return self.__name

    def set_salary(self, new_salary):
        if new_salary < 0:
            raise ValueError("Salary cannot be negative.")
        self.__salary = new_salary

# Create an instance
employee = Employee("Alice", 50000)

# Access name using getter
print(employee.get_name())  # Output: Alice

# Set salary using setter
employee.set_salary(55000)
```

Encapsulation and Abstraction in Python

Encapsulation

- **Concept:** Encapsulation is the bundling of data (attributes) and methods (functions) that operate on that data within a single unit, typically a class. It's about restricting direct access to some components of an object.
- **Implementation in Python:**
    - **Private Attributes:** While Python doesn't have strict private access modifiers like some other languages, you can indicate a variable as "private" by prefixing it with a double underscore (e.g., `__name`). This makes it harder to access directly from outside the class.
    - **Getters and Setters:** You can create public methods (getters and setters) to control access to the attributes. This allows you to validate data before setting it or perform other actions when the attribute is accessed or modified.

Example:

Python

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

    def get_name(self):
        return self.__name

    def set_salary(self, new_salary):
        if new_salary < 0:
            raise ValueError("Salary cannot be negative.")
        self.__salary = new_salary

# Create an instance
employee = Employee("Alice", 50000)

# Access name using getter
print(employee.get_name())  # Output: Alice

# Set salary using setter
employee.set_salary(55000)
```

Abstraction

- **Concept:** Abstraction is the process of hiding the implementation details of a class and only exposing the essential features. It's about simplifying complex systems by providing a high-level view.
- **Implementation in Python:**

    - **Abstract Base Classes (ABCs):** Python provides the `abc` module for creating abstract base classes. An abstract class cannot be instantiated directly and serves as a blueprint for other classes. It often contains abstract methods (methods declared but not implemented).

```
from abc import ABC, abstractmethod


class Shape(ABC):
    @abstractmethod
    def area(self):
        pass


class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height


    def area(self):
        return self.width * self.height


# Cannot create an instance of the abstract class
# shape = Shape()  # Would raise an error


rectangle = Rectangle(5, 3)
print(rectangle.area())  # Output: 15
```

`__str__` (String Representation)

- **Purpose:**
    - The `__str__` method is used to define how an object of the class should be represented as a string.
    - When you use the `print()` function on an object, the `__str__` method is called to determine the string representation.
-

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

    def __str__(self):
        return f"Person: {self.name}, Age: {self.age}"

person1 = Person("Alice", 30)
print(person1)  # Output: Person: Alice, Age: 30
```

`__main__`

- **Purpose:**
    - `__main__` is a special variable that holds the name of the current module.
    - If a Python script is executed directly (e.g., `python my_script.py`), the value of `__main__` becomes `″__main__″`.
    - This is commonly used to include code that should only be executed when the script is run directly, not when it's imported as a module.