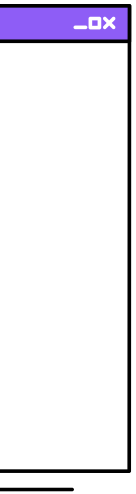
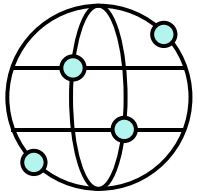
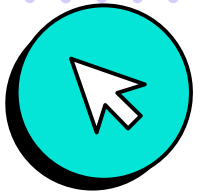
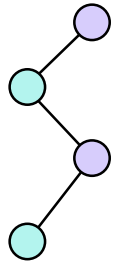




Python 3 Developer's Handbook: Your Quick Reference Guide

Ben Khalfallah Héla - 2025



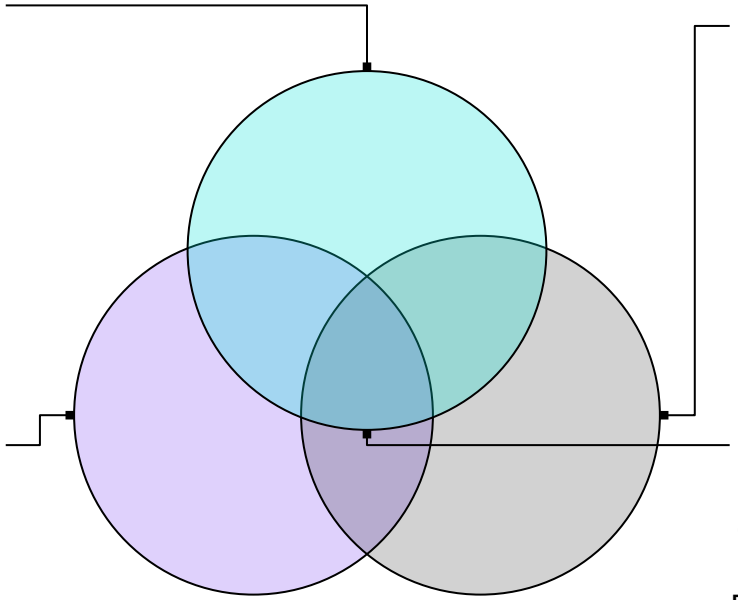

Python Aspects

Dynamically typed

Variables **are not bound to a specific type**, and their **type can change at runtime**.

Interpreted

Python code is executed **line-by-line by the interpreter**.



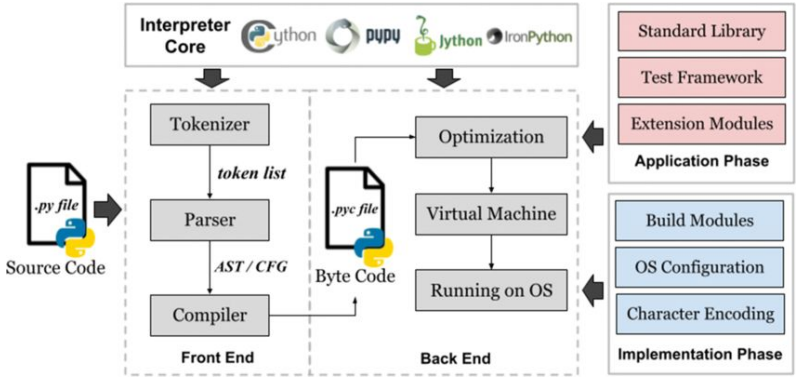
Strongly Typed

Variables **retain their type** and Python **does not perform implicit type conversions** when they don't make sense.

Is not inherently JIT-compiled

Python (specifically **CPython**) **is not inherently JIT-compiled**, but some Python implementations do include JIT compilation (PyPy or Numba).

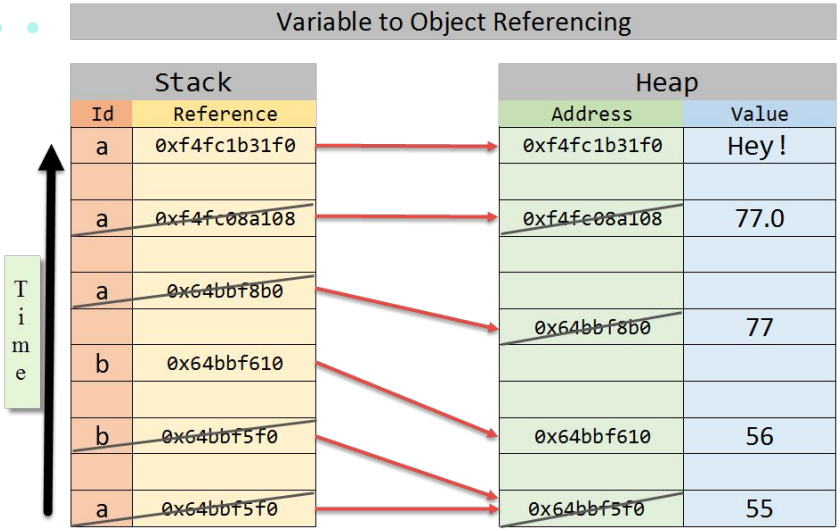
How Python Code is Executed ?



https://www.researchgate.net/figure/The-General-Architecture-of-Python_fig1_366233366

1. **Source Code**:
   a. Human-readable .py files.
   b. Written by developers.
2. **Parsing and Compilation**:
   a. Source code → Abstract Syntax Tree → Bytecode.
   b. Bytecode is stored in .pyc files for reuse.
3. **Execution**:
   a. Bytecode is executed by the Python Virtual Machine (PVM).
   b. The PVM is a **stack-based interpreter.**
4. **Memory Management**:
   a. Objects are stored in the **heap**.
   b. **Reference counting** tracks object usage.
   c. **Garbage collection** handles unused objects and circular references.

Memory Layout



https://austincode.com/cosc1336/variables.php

Variables in Python are **just references (or labels) to values**:

- **Names in the Stack:** Variable names are just keys in the mapping.
- **References in the Stack:** The stack stores the reference (address) pointing to the object in the heap.
- **Objects in the Heap:** The actual objects, including their data and methods, reside in the heap.

Python Types

In Python, all types are objects

Primitive Types Are Objects

- Integers, floats, strings, booleans, etc., are **objects**.
- Even simple values like 10 or "hello" are instances of a class (int and str, respectively).

Functions and Classes Are Objects

- Functions are **objects** and **can be assigned** to variables or **passed** as arguments.
- Classes themselves are objects of the type **metaclass**.

Modules Are Objects

- Modules imported using **import** are also objects.

Everything Has Attributes and Methods

- Because everything is an **object**, **all types have attributes and methods**.
- Since all types are objects, **variables in Python are just references (or labels) to these objects**.

References, Not Values !

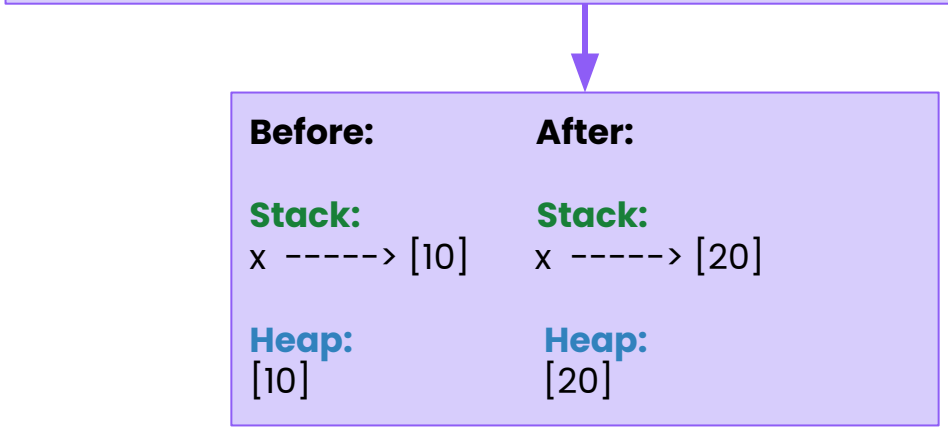
```
a = [1, 2, 3]
b = a  # b now references the same object as a

b.append(4)
print(a)  # [1, 2, 3, 4] (both variables point to the same list)
```

Stack: **Heap:**
a -----> address ----> [[1, 2, 3, 4] (list object)]
b -----> address ----> **(shared reference)**

- Variables do not hold the actual value. They hold a **reference** to where the value is stored.
- When **assigning** or **passing** a variable, **you're copying the reference, not the object itself**.
- **Multiple variables can refer to the same object**. This is why **changes to a mutable object via one variable are visible via another variable**.

Reassignment Creates a New Reference !

```
x = 10
x = 20  # x now references a new int object
```



- **When you reassign a variable, it points to a new object**, leaving the old object **unchanged (if immutable).**
- For **immutable objects** like int, float, str, or tuple, **any operation that "changes" the value actually creates a new object, leaving the old object intact**.
- **Mutable** objects, however, **can be modified in place**.

Main

```
# Example with main() for clarity and modularity
def greet(name):
    print(f"Hello, {name}!")

def main():
    name = input("Enter your name: ")
    greet(name)

if __name__ == "__main__":
    main()
```

- **Main** is considered a **best practice** for larger or more complex applications:
  - **Clarity:** It organizes code and provides a **clear entry point** for larger or complex applications.
  - **Modularity:** Keeps execution logic separate, making the code easier to maintain and reuse.
  - **Collaboration:** Helps collaborators and deployment tools quickly understand **where the program starts**.
- For simple, one-off scripts, you don't need a main() function.

Arithmetic Operators

Operator	Symbol	Description	Example
Addition	+	Adds two numbers.	5 + 3 # 8
Subtraction	-	Subtracts the second from the first.	5 - 3 # 2
Multiplication	*	Multiplies two numbers.	5 * 3 # 15
Division	/	Divides the first by the second **(float division)**.	5 / 2 # 2.5
Floor Division	//	**Divides and floors** the result to an integer.	5 // 2 # 2
Modulus	%	Returns the **remainder of division**.	5 % 2 # 1
Exponentiation	**	**Raises** the first number to the **power** of the second.	2 ** 3 # 8

Comparison Operators

Operator	Symbol	Description	Example
Equal	==	Checks if two values are **equal**.	5 == 5 # True
Not Equal	!=	Checks if two values are **not equal**.	5 != 3 # True
Greater Than	>	Checks if the first is **greater** than the second.	5 > 3 # True
Less Than	<	Checks if the first is **less** than the second.	3 < 5 # True
Greater or Equal	>=	Checks if the first is **greater or equal** to the second.	5 >= 3 # True
Less or Equal	<=	Checks if the first is **less or equal** to the second.	3 <= 5 # True

Assignment Operators

Operator	Symbol	Description	Example
Assign	=	Assigns a value to a variable.	x = 5
Add and Assign	+=	Adds and assigns.	x += 3 # x = x + 3
Subtract and Assign	-=	Subtracts and assigns.	x -= 3 # x = x - 3
Multiply and Assign	*=	Multiplies and assigns.	x *= 3 # x = x * 3
Divide and Assign	/=	Divides and assigns.	x /= 3 # x = x / 3
Floor Divide and Assign	//=	Floor divides and assigns.	x //= 3 # x = x // 3
Modulus and Assign	%=	Modulus and assigns.	x %= 3 # x = x % 3
Exponent and Assign	**=	Exponentiates and assigns.	x **= 3 # x = x ** 3

Logical Operators

Operator	Symbol	Description	Example
Logical AND	and	Returns True if both are true.	True and False # False
Logical OR	or	Returns True if at least one is true.	True or False # True
Logical NOT	not	Negates the value.	not True # False

Identity and Membership Operators

Operator	Symbol	Description	Example
Identity	is	Tests if two objects are the same.	x is y
Not Identity	is not	Tests if two objects are not the same.	x is not y
Membership	in	Tests if a value exists in a sequence.	5 in [1, 2, 5]
Not Membership	not in	Tests if a value does not exist in a sequence.	3 not in [1, 2, 5]

Boolean and NoneType Keywords

Keyword	Purpose
True	Boolean value representing **true**.
False	Boolean value representing **false**.
None	Represents a **null** value or absence.

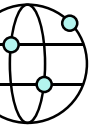
Control Flow Keywords

Keyword	Purpose
if	Starts a **conditional** block.
elif	**"Else if"** block in conditional logic.
else	Defines code to execute if all if and elif conditions fail.
while	Starts a **loop** that executes **while a condition is true**.
for	Starts a **loop** for **iterating over a sequence or range**.
break	**Exits** the current loop immediately.
continue	**Skips** the remaining code in the current loop iteration.
pass	**Placeholder statement**; **does nothing**. Useful for stubs.

Control Flow Keywords

```
age = 20

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

```
You are an adult.
```



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

for num in numbers:
    if num == 3:
        print("Found 3! Breaking the loop.")
        break  # Exit the loop when num is 3
    elif num % 2 == 0:
        continue  # Skip even numbers
    print(f"Processing {num}")
```

```
Processing 1
Processing 3
Found 3! Breaking the loop.
```

```
n = 0

while n < 5:
    if n == 3:
        pass  # Placeholder; does nothing
    else:
        print(f"n = {n}")
    n += 1
else:
    print("Loop ended, n reached 5.")
```

```
n = 0
n = 1
n = 2
n = 4
Loop ended, n reached 5.
```

Function and Class Definition Keywords

Keyword	Purpose
def	Defines a function.
return	Exits a function and returns a value.
class	Defines a class.
lambda	Creates an anonymous function.

Function and Class Definition Keywords

```
def add_numbers(a, b):
    return a + b  # Returns the sum of a and b

result = add_numbers(5, 7)
print(f"The sum is: {result}")

The sum is: 12
```

- **def**: Defines the **add_numbers** function.
- **return**: Sends the result of the addition back to the caller.

```
class Animal:
    def __init__(self, name):
        self.name = name  # Initialize the object with a name

    def speak(self):
        return f"{self.name} makes a sound."

# Create an instance of the class
dog = Animal("Dog")
print(dog.speak())

Dog makes a sound.
```

- **class**: Defines the Animal **class**.
- **def**: Defines methods inside the class, such as **__init__** (constructor) and **speak**.

Exception Handling Keywords

Keyword	Purpose
try	Starts a block for exception handling.
except	Catches and handles exceptions raised in the try block.
finally	Defines code that always executes after the try block.
raise	Raises an exception manually.

Exception Handling Keywords

```
def divide_numbers(a, b):
    try:
        result = a / b  # Attempt to divide
    except ZeroDivisionError:  # Handle division by zero
        print("Error: Division by zero is not allowed.")
        result = None
    finally:
        print("Execution completed.")  # This block always runs
    return result

# Test the function
print(divide_numbers(10, 2))  # Valid division
print(divide_numbers(10, 0))  # Division by zero
```

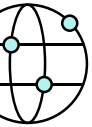
```
Execution completed.
5.0
Error: Division by zero is not allowed.
Execution completed.
None
```

- **try**:
  - Code that **might raise an exception** is placed here.
- **except**:
  - **Catches** and **handles** specific exceptions (like **ZeroDivisionError**).
- **finally**:
  - **Always executes,** whether an exception was raised or not (e.g., for cleanup).

Exception Handling Keywords

```
def withdraw_money(balance, amount):
    try:
        if amount > balance:
            raise ValueError("Insufficient funds!")  # Manually raise an exception
        balance -= amount
        print(f"Withdrawal successful! Remaining balance: ${balance}")
    except ValueError as e:  # Handle the raised exception
        print(f"Error: {e}")
    finally:
        print("Transaction completed.")  # Always executes

# Test the function
withdraw_money(100, 50)  # Successful withdrawal
withdraw_money(100, 150)  # Insufficient funds
```

```
Withdrawal successful! Remaining balance: $50
Transaction completed.
Error: Insufficient funds!
Transaction completed.
```



- **raise**:
  - **Manually raises** a ValueError if the withdrawal amount exceeds the balance.
  - The **message** "Insufficient funds!" **is passed as an argument to the exception**.
- **try**:
  - Contains the logic that **might raise the exception**.
- **except**:
  - **Catches the ValueError** raised by the raise statement and handles it gracefully.
- **finally**:
  - **Executes cleanup logic** (e.g., printing "Transaction completed.") regardless of success or failure.

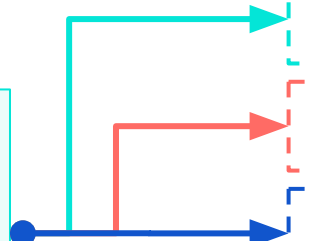
Import and Module Keywords

Keyword	Purpose
import	Imports a module or library.
from	Imports specific items or parts of a module.
as	Creates an alias for imports or context.

Import and Module Keywords

Create a file called math_utils.py

```
# math_utils.py

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

def subtract(a, b):
    return a - b
```



```
# main.py
# Importing the entire module
import math_utils

# Importing specific functions
from math_utils import add

# Importing with an alias
import math_utils as mu

# Using the imported module and functions
result1 = math_utils.add(5, 3)  # Using the full module name
result2 = add(10, 7)            # Using the directly imported function
result3 = mu.subtract(15, 8)    # Using the module alias

print(f"Result 1: {result1}")
print(f"Result 2: {result2}")
print(f"Result 3: {result3}")
```

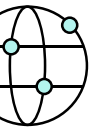
- **import**: Used to **import the entire math_utils** module.
- **from ... import**: Allows **importing specific functions or variables** directly (e.g., add).
- **as**: **Creates an alias** for a module or function for convenience (e.g., import math_utils as mu).

Variable Scope and Namespace Keywords

Keyword	Purpose
global	Declares a variable as global, allowing it to be accessed across the entire program.
nonlocal	Declares a variable as nonlocal, modifying a variable in an outer, non-global scope.

```
# Global variable
counter = 0

def increment_global():
    global counter  # Declare counter as global to modify it
    counter += 1
    print(f"Global counter: {counter}")

def outer_function():
    count = 0  # Local to outer_function

    def inner_function():
        nonlocal count  # Declare count as nonlocal to modify the outer scope's variable
        count += 1
        print(f"Nonlocal count: {count}")

    inner_function()
    inner_function()
    print(f"Final count in outer_function: {count}")

# Test the functions
increment_global()  # Modifies the global counter
increment_global()  # Modifies the global counter again

outer_function()  # Demonstrates nonlocal scope
```

```
Global counter: 1
Global counter: 2
Nonlocal count: 1
Nonlocal count: 2
Final count in outer_function: 2
```



global:

- Declares a variable as **global**, allowing modifications to the variable in the global namespace.
- Without global, attempting to modify a global variable inside a function would raise an UnboundLocalError.

nonlocal:

- Declares a variable as **nonlocal**, allowing modifications to a variable in the nearest enclosing (non-global) scope.
- Useful for nested functions where the variable resides in the enclosing function's scope.

Asynchronous Programming Keywords

Keyword	Purpose
async	Declares an asynchronous function or block.
await	Pauses execution in an asynchronous function until a task completes.

```python
import asyncio

# Define an asynchronous function
async def fetch_data():
    print("Start fetching data...")
    await asyncio.sleep(2)  # Simulate a network delay
    print("Data fetched!")
    return {"data": "Sample data"}

# Another asynchronous function
async def process_data():
    print("Start processing data...")
    await asyncio.sleep(1)  # Simulate processing time
    print("Data processed!")

# Main asynchronous function
async def main():
    # Run the tasks concurrently
    fetch_task = asyncio.create_task(fetch_data())
    process_task = asyncio.create_task(process_data())

    # Wait for both tasks to complete
    await fetch_task
    await process_task

# Run the main function
asyncio.run(main())
```

```
Start fetching data...
Start processing data...
Data processed!
Data fetched!
```

async:

- Used to define an **asynchronous function**.
- Functions marked with **async** can use **await** to pause and resume execution.

await:

- **Used inside an async function to pause execution** until the awaited coroutine completes.
- Allows other tasks to run in the meantime, enabling **concurrency**.

asyncio:

- A **library** for writing asynchronous programs in Python.
- **asyncio.run()** is used to **execute an async function** (like main() in this example).

asynchronous tasks:

- **Improves performance** by allowing tasks to **run concurrently** (e.g., fetching data and processing it simultaneously).
- **Ideal for I/O-bound tasks** like network requests, file reading/writing, or database interactions.

Context Management Keywords

Keyword	Purpose
with	Ensures proper resource management (e.g., file handling).

Context Management Keywords

```
# Using 'with' to manage a file resource
with open("example.txt", "w") as file:
    file.write("Hello, world!")  # Write to the file

# The file is automatically closed when the 'with' block ends
print("File written successfully!")
```

File written successfully!

with:

- Manages a **context** (**e.g., opening a file**) and **ensures that necessary cleanup (like closing the file) is performed automatically**, **even if an exception occurs**.
- Simplifies resource management and reduces the likelihood of resource leaks.
- The with keyword calls the **context manager's __enter__** method **at the start** of the block (e.g., opening the file).
- At **the end** of the block, the **__exit__** method is automatically called (e.g., closing the file).

```
class MyContextManager:
    def __enter__(self):
        print("Entering the context...")
        return self

    def __exit__(self, exc_type, exc_value, traceback):
        print("Exiting the context...")

# Using the custom context manager
with MyContextManager() as manager:
    print("Inside the context!")
```

Entering the context...
Inside the context!
Exiting the context...

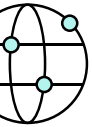
Special Purpose Keywords

Keyword	Purpose
assert	Ensures a condition is True during debugging; raises AssertionError if False.
del	Deletes variables, list items, or dictionary keys.
yield	Creates a generator that lazily produces values one at a time.

Special Purpose Keywords (assert)

The assert keyword is used to test conditions and raise an AssertionError if the condition is False.

```
def divide(a, b):
    assert b != 0, "Divider cannot be zero!"  # Ensures b is not zero
    return a / b

print(divide(10, 2))  # Valid
print(divide(10, 0))  # Triggers AssertionError


5.0
Traceback (most recent call last):
  ...
AssertionError: Divider cannot be zero!
```



Special Purpose Keywords (del)

```
# Deleting a variable
x = 10
print(x)  # 10
del x
# print(x)  # Raises NameError: x is not defined

# Deleting an item from a list
lst = [1, 2, 3, 4]
del lst[2]  # Removes the element at index 2
print(lst)  # [1, 2, 4]

# Deleting a key-value pair from a dictionary
my_dict = {"a": 1, "b": 2}
del my_dict["a"]
print(my_dict)  # {'b': 2}

10
[1, 2, 4]
{'b': 2}
```

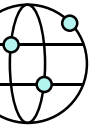
The del keyword deletes objects, variables, or items from a list or dictionary.

Special Purpose Keywords (yield)

- **yield pauses the function's execution** and **saves its state**. When the generator is iterated again, **it resumes where it left off**.
- Unlike return, a function with yield can produce multiple values over time.

```
def generate_numbers():
    for i in range(5):
        yield i  # Pauses and returns the current value

# Using the generator
gen = generate_numbers()

for num in gen:
    print(num)
```

```
0
1
2
3
4
```



Python Types and Mutability

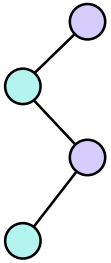
Category	Type	Mutability	Description	Example
Numbers	int	Immutable	Integer numbers.	x = 42
	float	Immutable	Floating-point numbers (real numbers).	x = 3.14
	complex	Immutable	Complex numbers (e.g., a + bi).	x = 1 + 2j
	bool	Immutable	Boolean values (True or False).	x = True
Sequences	str	Immutable	Strings of characters.	x = "hello"
	tuple	Immutable	Ordered collection of elements.	x = (1, 2, 3)
	bytes	Immutable	Immutable sequence of bytes.	x = b"data"
	list	Mutable	Ordered collection of elements.	x = [1, 2, 3]
	bytearray	Mutable	Mutable sequence of bytes.	x = bytearray(b"data")
Mappings	dict	Mutable	Key-value pairs.	x = {"key": "value"}
Sets	set	Mutable	Unordered collection of unique elements.	x = {1, 2, 3}
	frozenset	Immutable	Immutable version of a set.	x = frozenset([1, 2, 3])

Operations in Tuple (Immutable)

Operation	Description	Example
s[i]	Returns the **item at index i**.	t = (1, 2, 3); t[1]
s[i:j]	Returns a **slice** of the tuple from index i to j-1.	t = (1, 2, 3); t[1:3]
s[i:j:k]	Returns a **slice** from i to j-1 with step k.	t = (1, 2, 3, 4); t[::2]
len(s)	Returns the **length** of the tuple.	len((1, 2, 3))
min(s)	Returns the **smallest** item in the tuple.	min((3, 1, 4))
max(s)	Returns the **largest** item in the tuple.	max((3, 1, 4))
s.index(x[, i[, j]])	Returns the **index of the first occurrence of x** (optionally between i and j).	(1, 2, 3, 2).index(2)
s.count(x)	Returns the **number of occurrences of x** in the tuple.	(1, 2, 3, 2).count(2)
x in s	Checks **if x exists** in the tuple.	2 in (1, 2, 3)
x not in s	Checks **if x does not exist** in the tuple.	4 not in (1, 2, 3)
s + t	**Concatenates** tuples s and t.	(1, 2) + (3, 4)
s * n or n * s	**Repeats** the tuple s, n times.	(1, 2) * 3

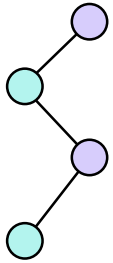
Operations in List (Mutable)

Operation	Description	Example	Result
s[i]	Returns the item at index i.	lst = [1, 2, 3]; lst[1]	2
s[i] = x	**Replaces** the item at index i with x.	lst = [1, 2, 3]; lst[1] = 5	[1, 5, 3]
s[i:j] = t	**Replaces the slice s[i:j]** with the sequence t.	lst = [1, 2, 3]; lst[1:3] = [4]	[1, 4]
del s[i]	**Deletes** the item at index i.	lst = [1, 2, 3]; del lst[1]	[1, 3]
del s[i:j]	**Deletes the slice s[i:j]**.	lst = [1, 2, 3]; del lst[0:2]	[3]
s[i:j:k] = t	Replaces the **slice** from i to j with **step** k by **sequence** t.	lst = [1, 2, 3, 4]; lst[::2] = [5, 6]	[5, 2, 6, 4]
s.append(x)	**Appends x** to the end of the list.	lst = [1, 2]; lst.append(3)	[1, 2, 3]
s.extend(t) or s += t	Extends the list by **appending elements** of t.	lst = [1, 2]; lst.extend([3, 4])	[1, 2, 3, 4]
s.insert(i, x)	**Inserts** x at index i.	lst = [1, 3]; lst.insert(1, 2)	[1, 2, 3]



Operations in List (Mutable)

Operation	Description	Example	Result
s.remove(x)	**Removes the first occurrence** of x.	lst = [1, 2, 3]; lst.remove(2)	[1, 3]
s.pop([i])	**Removes and returns** the item at index i (last by default).	lst = [1, 2, 3]; lst.pop()	3, [1, 2]
s.clear()	**Removes all items** from the list.	lst = [1, 2, 3]; lst.clear()	[]
s.reverse()	**Reverses** the list **in place**.	lst = [1, 2, 3]; lst.reverse()	[3, 2, 1]
s.sort([key][, reverse])	**Sorts** the list in ascending order (or descending with reverse=True).	lst = [3, 1, 2]; lst.sort()	[1, 2, 3]
s.copy()	**Returns a shallow copy** of the list.	lst = [1, 2, 3]; new_lst = lst.copy()	[1, 2, 3]
len(s)	Returns the **length** of the list.	len([1, 2, 3])	3
min(s)	Returns the **smallest** item in the list.	min([3, 1, 4])	1
max(s)	Returns the **largest** item in the list.	max([3, 1, 4])	4
s.index(x[, i[, j]])	Returns the **index of the first occurrence** of x (optionally between i and j).	[1, 2, 3, 2].index(2)	1
s.count(x)	Returns the **number of occurrences** of x in the list.	[1, 2, 3, 2].count(2)	2



Operations in Dictionary (Mutable)

Operation	Description	Example	Result
d[key]	Returns the value associated with key.	d = {"a": 1}; d["a"]	1
d[key] = value	**Adds or updates** the key-value pair.	d = {"a": 1}; d["b"] = 2	{"a": 1, "b": 2}
del d[key]	**Deletes** the key-value pair for key.	d = {"a": 1}; del d["a"]	{}
key in d	**Checks if key exists** in the dictionary.	"a" in {"a": 1}	True
key not in d	**Checks if key does not exist** in the dictionary.	"b" not in {"a": 1}	True
len(d)	Returns the **number of key-value pairs.**	len({"a": 1, "b": 2})	2
d.clear()	**Removes all items** from the dictionary.	d = {"a": 1}; d.clear()	{}
d.copy()	Returns a **shallow copy** of the dictionary.	d = {"a": 1}; copy_d = d.copy()	{"a": 1}
d.get(key[, default])	Returns **the value for key or default** if the key is not found.	d = {"a": 1}; d.get("b", 0)	0
d.pop(key[, default])	**Removes key and returns its value**; returns default if key is not found.	d = {"a": 1}; d.pop("a")	1, {}

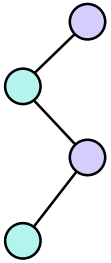
Operations in Dictionary (Mutable)

Operation	Description	Example	Result
d.popitem()	**Removes** and **returns** the **last inserted key-value pair**.	d = {"a": 1, "b": 2}; d.popitem()	("b", 2), {"a": 1}
d.update([other])	**Updates** the dictionary with key-value pairs from other.	d = {"a": 1}; d.update({"b": 2})	{"a": 1, "b": 2}
d.keys()	Returns a view object with **all keys**.	d = {"a": 1}; list(d.keys())	["a"]
d.values()	Returns a view object with **all values**.	d = {"a": 1}; list(d.values())	[1]
d.items()	Returns a view object with **all key-value pairs as tuples**.	d = {"a": 1}; list(d.items())	[("a", 1)]
{d, **other}**	**Merges two dictionaries** into a new one.	d = {"a": 1}; {**d, **{"b": 2}}	{"a": 1, "b": 2}

Operations in Set (Mutable)

Sets are **mutable**, **automatically discard duplicate elements** and are **unordered**.

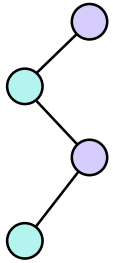
Operation	Description	Example	Result
x in s	Checks if x is in the set.	s = {1, 2, 3}; 2 in s	True
x not in s	Checks if x is not in the set.	s = {1, 2, 3}; 4 not in s	True
s.add(x)	Adds x to the set.	s = {1, 2}; s.add(3)	{1, 2, 3}
s.remove(x)	Removes x from the set. Raises KeyError if x is not found.	s = {1, 2, 3}; s.remove(2)	{1, 3}
s.discard(x)	Removes x from the set if it exists. Does nothing if x is not found.	s = {1, 2, 3}; s.discard(4)	{1, 2, 3}
s.pop()	Removes and returns an arbitrary element from the set.	s = {1, 2, 3}; s.pop()	1 (example), {2, 3}
s.clear()	Removes all elements from the set.	s = {1, 2, 3}; s.clear()	set()
len(s)	Returns the number of elements in the set.	len({1, 2, 3})	3



Operations in Set (Mutable)

Methods like union, intersection, and difference **align with mathematical set theory.**

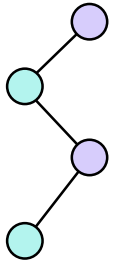
Operation	Description	Example	Result
s.union(t)	Returns a new set with elements from both s and t.	{1, 2}.union({2, 3})	{1, 2, 3}
s.intersection(t)	Returns a new set with elements common to both s and t.	{1, 2}.intersection({2, 3})	{2}
s.difference(t)	Returns a new set with elements in s but not in t.	{1, 2}.difference({2, 3})	{1}
s.symmetric_difference(t)	Returns a new set with elements in either s or t, but not both.	{1, 2}.symmetric_difference({2, 3})	{1, 3}
s.update(t)	Updates s with elements from t.	s = {1}; s.update({2, 3})	{1, 2, 3}
s.intersection_update(t)	Updates s with elements common to both s and t.	s = {1, 2}; s.intersection_update({2, 3})	{2}
s.difference_update(t)	Updates s by removing elements found in t.	s = {1, 2}; s.difference_update({2, 3})	{1}
s.symmetric_difference_update(t)	Updates s with elements in either s or t, but not both.	s = {1, 2}; s.symmetric_difference_update({2, 3})	{1, 3}



Operations in String (Immutable)

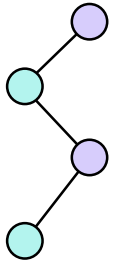
Strings are **immutable**, so methods like replace, upper, or strip **return a new string without modifying the original**.

Operation	Description	Example	Result
s[i]	Returns the character **at index i.**	'hello'[1]	'e'
s[i:j]	Returns a **substring from index i to j-1**.	'hello'[1:4]	'ell'
s[i:j:k]	Returns a **substring from i to j-1 with step k**.	'hello'[::2]	'hlo'
len(s)	Returns the **length** of the string.	len('hello')	5
s + t	**Concatenates** strings s and t.	'hello' + ' world'	'hello world'
s * n or n * s	**Repeats the string s, n times**.	'ha' * 3	'hahaha'
x in s	Checks **if substring x exists** in s.	'ell' in 'hello'	True
x not in s	Checks **if substring x does not exist** in s.	'world' not in 'hello'	True
s.lower()	Converts all characters in the string to **lowercase**.	'Hello'.lower()	'hello'
s.upper()	Converts all characters in the string to **uppercase**.	'Hello'.upper()	'HELLO'
s.title()	Converts the **first character of each word to uppercase**.	'hello world'.title()	'Hello World'



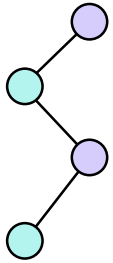
Operations in String (Immutable)

Operation	Description	Example	Result
s.strip([chars])	**Removes leading and trailing** characters (default is whitespace).	' hello '.strip()	'hello'
s.lstrip([chars])	**Removes leading** characters (default is whitespace).	' hello '.lstrip()	'hello '
s.rstrip([chars])	**Removes trailing** characters (default is whitespace).	' hello '.rstrip()	' hello'
s.startswith(x)	Checks if the string **starts with substring** x.	'hello'.startswith('he')	True
s.endswith(x)	Checks if the string **ends with substring** x.	'hello'.endswith('lo')	True
s.find(x[, i[, j]])	Returns the **lowest index of substring** x in s (or -1 if not found).	'hello'.find('l')	2
s.rfind(x[, i[, j]])	Returns the **highest index of substring** x in s (or -1 if not found).	'hello'.rfind('l')	3
s.count(x)	Returns the **number of occurrences of substring** x in s.	'hello'.count('l')	2
s.replace(old, new[, n])	Returns a **copy** of the string with occurrences of old replaced by new.	'hello'.replace('l', 'L', 1)	'heLlo'
s.split([sep[, maxsplit]])	**Splits** the string into a list of substrings using sep as a delimiter.	'a,b,c'.split(',')	['a', 'b', 'c']
s.rsplit([sep[, maxsplit]])	**Splits** the string into a list **from the right** using sep as a delimiter.	'a,b,c'.rsplit(',', 1)	['a,b', 'c']



Operations in String (Immutable)

Operation	Description	Example	Result
s.join(iterable)	**Joins** elements of iterable with string s as a separator.	','.join(['a', 'b', 'c'])	'a,b,c'
s.center(width[, fill])	**Centers the string** in a field of given width with optional fill character.	'hello'.center(10, '-')	'--hello--'
s.ljust(width[, fill])	**Left-justifies** the string in a field of given width with optional fill character.	'hello'.ljust(10, '-')	'hello-----'
s.rjust(width[, fill])	**Right-justifies** the string in a field of given width with optional fill character.	'hello'.rjust(10, '-')	'-----hello'



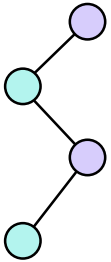
Operations in Numbers (Immutable)

Numbers in Python are **immutable**, meaning **any operation creates a new object**.

Operation	Description	Example	Result
complex(x, y)	Creates a complex number x + yj.	complex(2, 3)	(2+3j)
x.real	Returns the real part of a complex number.	(2+3j).real	2.0
x.imag	Returns the imaginary part of a complex number.	(2+3j).imag	3.0
x.conjugate()	Returns the complex conjugate of x.	(2+3j).conjugate()	(2-3j)
int(x)	Converts x to an integer (truncates the decimal).	int(3.8)	3
float(x)	Converts x to a float.	float(3)	3.0
math.floor(x)	Returns the largest integer less than or equal to x.	math.floor(3.7)	3
math.ceil(x)	Returns the smallest integer greater than or equal to x.	math.ceil(3.2)	4
round(x[, n])	Rounds x to n decimal places (default is 0).	round(3.14159, 2)	3.14
pow(x, y[, z])	Returns x raised to the power of y, optionally modulo z.	pow(2, 3)	8
abs(x)	Returns the absolute value of x.	abs(-5)	5

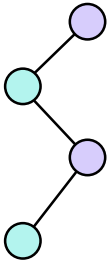
File Operations

Operation	Description	Example	Result
open(file, mode)	**Opens** a file in the **specified mode** ('r', 'w', 'a', etc.).	f = open("example.txt", "r")	Opens example.txt in read mode.
f.read([size])	**Reads the file content** (size bytes if specified, otherwise the entire file).	content = f.read()	Reads and returns the content of the file.
f.readline()	**Reads a single line** from the file.	line = f.readline()	Reads and returns the next line.
f.readlines()	**Reads all lines** and returns them as a list.	lines = f.readlines()	Returns ['line1\n', 'line2\n'].
f.write(data)	**Writes data** to the file.	f.write("Hello, world!")	Writes "Hello, world!" to the file.
f.writelines(lines)	**Writes a list of strings** to the file.	f.writelines(["Line1\n", "Line2\n"])	Writes Line1 and Line2 to the file.
f.close()	**Closes** the file.	f.close()	Ensures that the file is properly closed.
with open(file, mode) as f	**Opens a file and ensures it is closed automatically** after the block ends.	with open("example.txt", "r") as f:	Opens example.txt and automatically closes it after the block.



File Operations

Operation	Description	Example	Result
f.seek(offset, whence)	**Moves the file pointer** to the specified position.	f.seek(0)	Moves the pointer to the beginning of the file.
f.tell()	**Returns the current position** of the file pointer.	position = f.tell()	Returns the current byte position.



Modes for Opening Files

Mode	Description
'r'	Opens the file for reading (default). The file must exist.
'w'	Opens the file for writing. Creates a new file or truncates the existing file.
'a'	Opens the file for appending. Creates the file if it doesn't exist.
'r+'	Opens the file for reading and writing. The file must exist.
'w+'	Opens the file for writing and reading. Creates a new file or truncates the existing file.
'a+'	Opens the file for appending and reading. Creates the file if it doesn't exist.
'b'	Binary mode (e.g., 'rb' for reading binary files).
't'	Text mode (default).

Nested Loop

```
students = ["Alice", "Bob", "Charlie"]
subjects = ["Math", "Science", "History"]

# Outer loop for students
for student in students:
    print(f"Marks for {student}:")

    # Inner loop for subjects
    for subject in subjects:
        mark = input(f"  Enter marks for {subject}: ")
        print(f"  {subject}: {mark}")
    print()  # Newline after each student's marks
```

```
Marks for Alice:
  Enter marks for Math: 85
  Math: 85
  Enter marks for Science: 90
  Science: 90
  Enter marks for History: 78
  History: 78
```

```
students = ["Alice", "Bob", "Charlie", "Daisy", "Eve"]
group = 1

while group <= 3:  # Outer loop for groups
    print(f"Group {group}:")
    for i in range(group - 1, len(students), 3):  # Inner loop assigns students to this group
        print(f"  {students[i]}")
    group += 1
```

```
Group 1:
  Alice
  Daisy
Group 2:
  Bob
  Eve
Group 3:
  Charlie
```

Ternary Operator

value_if_true if condition else value_if_false

```
a = 10
b = 20

# Using ternary operator
larger = a if a > b else b
print(f"The larger number is: {larger}")
```

The larger number is: 20

```
num = 7

# Using ternary operator
result = "Even" if num % 2 == 0 else "Odd"
print(f"The number {num} is {result}.")
```

The number 7 is Odd.

```
age = 16

# Using ternary operator
eligibility = "Eligible" if age >= 18 else "Not Eligible"
print(f"You are {eligibility} to vote.")
```

You are Not Eligible to vote.

Function (Positional Arguments)

- **Positional arguments** are the most basic type of function arguments.
- They must be **passed in the correct order** when calling the function.
- **Every argument corresponds to a parameter** in the function definition, **based on its position**.
- The **order** of arguments passed determines which parameter they map to.

```
def calculate_area(length, width):
    area = length * width
    print(f"The area of the rectangle is: {area}")

# Call with positional arguments
calculate_area(10, 5)
calculate_area(7, 3)
```

```
The area of the rectangle is: 50
The area of the rectangle is: 21
```

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

# Missing argument
# add(10)  # TypeError: add() missing 1 required positional argument: 'b'

# Extra argument
# add(10, 5, 3)  # TypeError: add() takes 2 positional arguments but 3 were given
```

```
def print_order(first, second):
    print(f"First: {first}, Second: {second}")

# Call with arguments in correct order
print_order("Alice", "Bob")

# Call with reversed order
print_order("Bob", "Alice")
```

```
First: Alice, Second: Bob
First: Bob, Second: Alice
```

Function (Optional Arguments)

- Optional arguments, also known as **default arguments**, allow you to define parameters with default values.
- If a **value is not provided** during the function call, the **default value is used**.
- Useful for creating functions with **flexible usage**.

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

# Call without passing the optional argument
greet()

# Call with the optional argument
greet("Alice")
```

Hello, **Guest**!
Hello, **Alice**!

```
def calculate_price(price, tax=0.05, discount=0.1):
    total_price = price + (price * tax) - (price * discount)
    print(f"The total price is: {total_price:.2f}")

# Call with all arguments
calculate_price(100, 0.07, 0.2)

# Call with one optional argument
calculate_price(100, 0.07)

# Call with only the required argument
calculate_price(100)
```

The total price is: **97.00**
The total price is: **102.00**
The total price is: **105.00**

Function (Combining Positional and Optional Arguments)

- **Positional arguments** must come **before** optional arguments in the function definition.
- **Optional arguments can have default values**, and the caller can choose to provide or skip them.

```
def function_name(positional1, positional2, optional1=default_value1, optional2=default_value2):
    # Function body
```

```
def describe_pet(name, species="dog", color="unknown"):
    print(f"{name} is a {color} {species}.")

# Call with all arguments
describe_pet("Buddy", "cat", "black")

# Call with only one optional argument
describe_pet("Charlie", "rabbit")

# Call with only required argument
describe_pet("Daisy")
```

```
Buddy is a black cat.
Charlie is a unknown rabbit.
Daisy is a unknown dog.
```

```
def func(a, b=10): # Valid
    pass

# def func(a=10, b): # Invalid
#     pass
```

```
def calculate(a, b=5): # Valid
    return a + b

# def calculate(a=5, b): # Invalid
#     return a + b
```

Function (Arbitrary Arguments)

- In Python, you can define functions that accept **an arbitrary number of arguments**.
- This is done using the ***args** and ****kwargs** syntax.

```
def function_name(*args):
    # args is a tuple of all positional arguments

def function_name(**kwargs):
    # kwargs is a dictionary of all keyword arguments
```

```
def print_values(*args):
    for index, value in enumerate(args, start=1):
        print(f"Argument {index}: {value}")

print_values("Alice", 25, "Engineer")
```

```
Argument 1: Alice
Argument 2: 25
Argument 3: Engineer
```

```
def display_user_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_user_info(name="Alice", age=25,
profession="Engineer")
```

```
name: Alice
age: 25
profession: Engineer
```

Function (Combining *args and **kwargs)

- You can use **both *args and **kwargs in the same function** to handle both positional and keyword arguments.
- *args lets you **accept multiple positional** arguments.
- **kwargs lets you **handle optional, named arguments**.
- Use *args **when you don't know the number of positional arguments** beforehand.
- Use **kwargs **when you don't know the number of keyword arguments** beforehand.

```
def handle_arguments(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

handle_arguments(1, 2, 3, name="Alice", age=25)
```

Positional arguments: (**1, 2, 3**)
Keyword arguments: **{'name': 'Alice', 'age': 25}**

```
def func(a, b=10, *args, **kwargs):
    pass

# def func(*args, a, **kwargs):  # SyntaxError
#     pass
```

Function (Single Return)

- A function can return a **single value** using the return statement.
- The returned value **can be of any type** (e.g., integer, string, list, etc.).

```
def calculate_circle_area(radius):
    area = 3.14 * radius * radius
    return area  # Single value is returned

# Call the function
result = calculate_circle_area(5)
print(f"The area of the circle with radius 5 is: {result}")
```

The area of the circle with radius 5 is: 78.5

```
def square(number):
    return number * number

result = square(5)
print(f"The square of 5 is {result}")
```

The square of 5 is 25

Function (Multiple Return)

- A function can return **multiple values** by separating them with commas in the return statement.
- The values are returned as a **tuple**.

```
def rectangle_properties(length, width):
    area = length * width
    perimeter = 2 * (length + width)
    return area, perimeter

area, perimeter = rectangle_properties(5, 3)
print(f"Area: {area}, Perimeter: {perimeter}")
```

Area: 15, Perimeter: 16

```
def analyze_marks(marks):
    average = sum(marks) / len(marks)
    highest = max(marks)
    lowest = min(marks)
    return average, highest, lowest

# Call the function
marks = [85, 90, 78, 92, 88]
average, highest, lowest = analyze_marks(marks)

print(f"Average: {average}")
print(f"Highest: {highest}")
print(f"Lowest: {lowest}")
```

Average: 86.6
Highest: 92
Lowest: 78

Function (None Return)

- If a function does **not have a return statement, or the return statement is empty, it returns None**.
- Use this for functions that perform an action but do not produce a value.

```
def log_message(message):
    print(f"[LOG]: {message}")  # Action performed without returning any value

# Call the function
log_message("System initialized successfully.")
result = log_message("User login detected.")

# Check the return value
print(f"Return value of the function: {result}")
```

```
[LOG]: System initialized successfully.
[LOG]: User login detected.
Return value of the function: None
```

```
def print_message(message):
    print(f"Message: {message}")

result = print_message("Hello, World!")
print(f"The function returned: {result}")
```

```
Message: Hello, World!
The function returned: None
```

Function (Return Type)

Type	Description	Use Case
Single Return	Returns a single value using return.	Use for functions that produce **one result**.
Multiple Returns	Returns multiple values as a **tuple**.	Use when **multiple related results** are needed.
No Return (**None**)	Performs an action but **doesn't return a value**.	Use for actions like logging or printing.

Lambda Function

- A **lambda function** is a **small**, **anonymous** function defined with the **lambda** keyword.
- It is often used for **simple, one-time operations** where defining a full function is unnecessary.
- Lambda functions are **limited to a single expression** (no multi-line statements).
- Frequently used with functions like map, filter, and sorted.

lambda arguments: expression

- **arguments**: Input parameters for the lambda function (can be zero or more).
- **expression**: A single expression whose result is returned

```
# Lambda function to add two numbers
add = lambda x, y: x + y

# Call the lambda function
result = add(5, 3)
print(f"The sum is: {result}")
```

The sum is: 8

```
# Lambda function with a default argument
increment = lambda x, inc=1: x + inc

print(increment(5))      # Uses default increment of 1
print(increment(5, 2))   # Overrides default and uses 2 as increment
```

6
7

Lambda Function (sort, map, filter, reduce)

```
# List of tuples
data = [("Alice", 25), ("Bob", 30), ("Charlie", 20)]

# Sort by name
sorted_by_name = sorted(data, key=lambda x: x[0])
print("Sorted by name:", sorted_by_name)

# Sort by age
sorted_by_age = sorted(data, key=lambda x: x[1])
print("Sorted by age:", sorted_by_age)
```

```
Sorted by name: [('Alice', 25), ('Bob', 30), ('Charlie', 20)]
Sorted by age: [('Charlie', 20), ('Alice', 25), ('Bob', 30)]
```

```
# List of numbers
numbers = [10, 15, 20, 25, 30]

# Use lambda to filter numbers divisible by 10
divisible_by_ten = list(filter(lambda x: x % 10 == 0, numbers))
print(divisible_by_ten)
```

```
[10, 20, 30]
```

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use lambda to square each number
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

```
[1, 4, 9, 16, 25]
```

```
from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Use reduce with a lambda to calculate the product
product = reduce(lambda x, y: x * y, numbers)
print(f"The product is: {product}")
```

```
The product is: 120
```

Lambda Function (Inside a Function)

- **create_operation Function**:
  - Accepts a string argument (op_type) that determines the type of operation.
  - **Dynamically returns a lambda function** corresponding to the operation type.
- **Dynamic Behavior**:
  - create_operation("add") returns a lambda for addition.
  - create_operation("subtract") returns a lambda for subtraction.
  - Handles invalid operation types gracefully.

```
def create_operation(op_type):
    # Define a lambda function based on the operation type
    if op_type == "add":
        return lambda x, y: x + y
    elif op_type == "subtract":
        return lambda x, y: x - y
    elif op_type == "multiply":
        return lambda x, y: x * y
    elif op_type == "divide":
        return lambda x, y: x / y if y != 0 else "Cannot divide by zero"
    else:
        return lambda x, y: "Invalid operation"

# Use the function to create specific operations
add_func = create_operation("add")
subtract_func = create_operation("subtract")

# Perform calculations
print(add_func(10, 5))       # Output: 15
print(subtract_func(10, 5))  # Output: 5

# Handle invalid operations
invalid_func = create_operation("mod")
print(invalid_func(10, 5))   # Output: Invalid operation
```

Lambda Function (in Dict and List)

```
# Dictionary of operations
operations = {
    "add": lambda x, y: x + y,
    "subtract": lambda x, y: x - y,
    "multiply": lambda x, y: x * y,
    "divide": lambda x, y: x / y if y != 0 else "Division by zero"
}

# Perform operations
print(operations["add"](10, 5))       # 15
print(operations["subtract"](10, 5))  # 5
print(operations["multiply"](10, 5))  # 50
print(operations["divide"](10, 2))    # 5.0
```

```
15
5
50
5.0
```

Lambda functions **are stored as values in a dictionary**, enabling flexible and dynamic execution of operations.

```
# List of lambda functions
functions = [
    lambda x: x + 2,
    lambda x: x * 3,
    lambda x: x ** 2
]

# Apply each function to a value
for func in functions:
    print(func(4))
```

```
6
12
16
```

A **list of lambda functions** is applied iteratively to the value 4.

Immediately Invoked Lambda Expression (IIFE)

- An **Immediately Invoked Lambda Expression (IIFE)** is a lambda function that is defined and called at the same time.
- It is a quick way to execute small functions without explicitly naming them.
- Ideal for small operations **that don't need to persist beyond a single use**.
- **Reduces** the clutter of **temporary function** definitions.

(lambda arguments: expression)*(arguments_to_pass)*

result = (lambda x, y: x + y)(5, 3)
print(f"The result is: {**result**}")

The result is: 8

result = (lambda x: "Even" if x % 2 == 0 else "Odd")(7)
print(f"The number is: {**result**}")

The number is: Odd

square = (lambda x: x ** 2)(4)
print(f"The square is: {**square**}")

The square is: 16

maximum = (lambda a, b: a if a > b else b)(10, 20)
print(f"The maximum number is: {**maximum**}")

The maximum number is: 20

Closure

- **Nested Function**: A closure is created when a nested function references variables from its enclosing function.
- **Data Encapsulation**: The variables from the enclosing scope are remembered and stored in the closure, even if the enclosing scope has finished execution.
- **Practical Use**: Closures are commonly used in decorators, callback functions, and maintaining state.

```
def outer_function():
    variable = "enclosed"

    def inner_function():
        return variable  # Accesses the variable from the enclosing scope

    return inner_function  # Returns the inner function (closure)
```

```
def outer_function(message):
    def inner_function():
        print(f"Message: {message}")  # 'message' is captured from the enclosing scope
    return inner_function

# Create a closure
closure = outer_function("Hello, World!")

# Call the closure
closure()

Message: Hello, World!
```

Closure

```
def make_counter():
    count = 0  # Variable in the enclosing scope

    def counter():
        nonlocal count  # Allows modification of the enclosing variable
        count += 1
        return count

    return counter

# Create a closure
counter1 = make_counter()
counter2 = make_counter()

# Use the closures
print(counter1())  # 1
print(counter1())  # 2
print(counter2())  # 1
print(counter2())  # 2
```

```
1
2
1
2
```

Each closure (counter1 and counter2) **maintains its own count variable**.

```
def make_filter(threshold):
    def filter_function(value):
        return value > threshold  # Retains access to 'threshold'
    return filter_function

# Create closures with different thresholds
filter_above_10 = make_filter(10)
filter_above_20 = make_filter(20)

# Use the closures
print(filter_above_10(15))  # True
print(filter_above_10(5))   # False
print(filter_above_20(25))  # True
print(filter_above_20(15))  # False
```

```
True
False
True
False
```

The closures **retain** the threshold value set during their creation.

Decorator (Definition)

- A **decorator** in Python **is a function that takes another function as input, adds some functionality to it, and returns it**.
- Decorators are a powerful way to **modify or enhance the behavior of functions or methods without changing their code**.

```
def decorator_function(original_function):
    def wrapper_function(*args, **kwargs):
        # Add functionality here
        print("Wrapper executed before", original_function.__name__)
        result = original_function(*args, **kwargs)
        print("Wrapper executed after", original_function.__name__)
        return result
    return wrapper_function

@decorator_function
def display():
    print("Display function executed")

# Call the decorated function
display()
```

Decorator (with Parameters)

Outer Function (decorator_with_params):

- **Accepts the parameter for the decorator** (**param** in this case).
- Returns the actual decorator (**decorator_function**).

Inner Decorator (decorator_function):

- **Accepts the function to be decorated (original_function)**.
- Returns the **wrapper_function**.

Wrapper Function:

- Adds functionality **before** and **after** calling the original function.
- **Uses the parameter (param) to customize its behavior.**

```
def decorator_with_params(param):
    def decorator_function(original_function):
        def wrapper_function(*args, **kwargs):
            # Add functionality using the decorator parameter
            print(f"Decorator parameter: {param}")
            print("Wrapper executed before", original_function.__name__)
            result = original_function(*args, **kwargs)
            print("Wrapper executed after", original_function.__name__)
        return result
    return wrapper_function
return decorator_function

@decorator_with_params("Custom Parameter")
def display():
    print("Display function executed")

# Call the decorated function
display()
```

Decorator (Examples)

```
def greeting_decorator(func):
  def wrapper():
      print("Hello!")  # Additional functionality
      func()         # Call the original function
  return wrapper

@greeting_decorator
def say_name():
   print("My name is Alice.")

# Call the decorated function
say_name()
```

```
Hello!
My name is Alice.
```

The decorator simple_decorator adds functionality before and after the say_hello function call.

```
def repeat_decorator(times):
  def decorator(func):
    def wrapper(*args, **kwargs):
      for _ in range(times):
        func(*args, **kwargs)
    return wrapper
  return decorator

@repeat_decorator(times=3)
def greet():
   print("Hello!")

greet()
```

```
Hello!
Hello!
Hello!
```

Decorator with Parameters: The repeat_decorator accepts a parameter (times) to control how many times the function is called.

Multiple Decorators

```
def uppercase_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result.upper()
    return wrapper

def exclamation_decorator(func):
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        return result + "!!!"
    return wrapper

@exclamation_decorator
@uppercase_decorator
def say_message():
    return "hello world"

print(say_message())


HELLO WORLD!!!
```

- The **uppercase_decorator** converts the output to **uppercase**.
- The **exclamation_decorator** **adds exclamation marks**.
- The decorators are **applied in the order written (bottom to top execution)**.

Callback Functions

- A **callback function** is a function that is passed as an argument to another function and is executed after the main function completes its operation.
- **Callbacks** are commonly **used** for **event handling, asynchronous programming,** and **modular code**.
- **Modularity**: Decouples the main logic from additional behaviors like notifications or logging.
- **Customizability**: Allows dynamic behavior by passing different callback functions.
- **Asynchronous Programming**: Often used to handle asynchronous events in frameworks like asyncio.

```
def execute_operation(a, b, operation):
    print(f"Executing operation with {a} and {b}")
    result = operation(a, b)  # Call the operation callback
    print(f"Operation result: {result}")

def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

# Pass different callback functions
execute_operation(4, 5, add)
execute_operation(4, 5, multiply)
```

```
Executing operation with 4 and 5
Operation result: 9

Executing operation with 4 and 5
Operation result: 20
```

Higher Order Function (HOF)

```
# Higher-Order Function
def operate_on_numbers(a, b, operation):
    return operation(a, b)

# Define some functions to use as arguments
def add(x, y):
    return x + y

def multiply(x, y):
    return x * y

# Call the Higher-Order Function
result_add = operate_on_numbers(5, 3, add)  # Pass 'add' as the function
result_multiply = operate_on_numbers(5, 3, multiply)  # Pass 'multiply' as the function

print(f"Addition result: {result_add}")
print(f"Multiplication result: {result_multiply}")

Addition result: 8
Multiplication result: 15
```

- The **operate_on_numbers** function takes a, b, and a **function (operation)** as arguments. It applies the passed function (add or multiply) to a and b.
- The function to execute (add or multiply) **is passed as an argument**, enabling **dynamic operation**.

A **Higher-Order Function (HOF)** is a function that either:

1. **Takes another function as an argument**, or
2. **Returns a function as its result.**

Higher-order functions are a key feature of functional programming and are commonly used in Python for clean, modular, and reusable code.

Functions as First-Class Citizens: Python treats functions as **objects**, so they can be passed as arguments, returned, or assigned to variables.

Common Higher-Order Functions: **Built-in** Python functions like **map**, **filter**, **reduce**, and **sorted** are examples of higher-order functions.

HOF vs Callback vs Closure

Concept	Definition	Example
Higher-Order Function (HOF)	Takes a function as input or returns a function. **Performs the main logic dynamically.**	operate_on_numbers(5, 3, add)
Callback Function	Invoked inside another function to **perform additional or follow-up tasks.**	process_data("Sample Data", notify)
Closure	A function that **retains access to variables** in its **enclosing scope** even after the scope ends.	make_counter()

Class

Feature	Description	Syntax
Class	Blueprint for creating objects with attributes and methods.	**class** ClassName:
Single Inheritance	A class inherits from one parent class.	**class** ChildClass**(ParentClass)**:
Multiple Inheritance	A class inherits from multiple parent classes.	**class** ChildClass**(Parent1, Parent2)**:
Method Overriding	Child class redefines a parent class method.	**def** method**(self)**:
Diamond Problem	In **multiple inheritance**, the **diamond problem** arises when a class inherits from two classes that both inherit from a common base class. Python resolves this using the **Method Resolution Order (MRO)**.	**class** ChildClass**(Parent1, Parent2)**:

Iterator and Iterable

Iterable:

- An object capable of **returning its elements one at a time**.
- Any object you can **loop over.**
- Examples: Lists, tuples, dictionaries, sets, strings.
- **It must implement the `__iter__()` method**, which returns an iterator.

Iterator:

- An object that represents a **stream of data**.
- **It must implement both the `__iter__()` and `__next__()` methods**.
- **Created from an iterable using the `iter()`** function.
- Provides **one item at a time using the `next()`** function.
- **StopIteration**: Raised when there are **no more elements to retrieve from an iterator**.

```
# A list is an iterable
numbers = [1, 2, 3, 4]

# Using iter() to get an iterator
iterator = iter(numbers)

print(next(iterator))  # 1
print(next(iterator))  # 2
```

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

# Using a for loop
for fruit in fruits:
    print(fruit)

# Manually using an iterator
iterator = iter(fruits)
print(next(iterator))  # apple
print(next(iterator))  # banana
print(next(iterator))  # cherry
```

Custom Iterator

- The Counter class is a **custom iterator**.
- It **implements both** the **__iter__()** and **__next__()** methods.

```
class Counter:
    def __init__(self, start, end):
        self.current = start
        self.end = end

    def __iter__(self):
        return self

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

# Create an iterator object
counter = Counter(1, 5)

# Iterate using a for loop
for number in counter:
    print(number)
```

Generator

What is a Generator?

- A **generator** is a special type of iterator.
- It uses the yield keyword instead of return to produce values **one at a time** as they are needed.

Key Features:

1. **Memory Efficient**:
   - Generators do not store the entire sequence in memory.
   - Items are produced one at a time, making them ideal for large datasets.
2. **Lazy Evaluation**:
   - Values are computed only when requested.
   - This approach ensures efficient processing of **large** or **infinite sequences**.
3. **Simpler Syntax**:
   - Generators are defined using functions with yield instead of implementing the __iter__() and __next__() methods in a class.

```
def infinite_counter():
    current = 1
    while True:
        yield current
        current += 1

# Create an infinite generator
counter = infinite_counter()

# Use the generator
for _ in range(5):
    print(next(counter))
```

When to Use Generators:

1. **Large Data Handling**: Generators are ideal for **files**, **streams**, or **sequences too large to fit into memory**.
2. **Infinite Data**: Use generators for data that can be generated indefinitely (e.g., **Fibonacci series**, **counters**).
3. **Lazy Access**: When you **don't need all values immediately** but only a subset or need to process items one at a time.

Generator with Send

```
def dynamic_counter():
    count = 0  # Initialize the counter
    while True:
        increment = yield count  # Yield the current count and wait for `send()`
        if increment is None:   # If no value is sent, default to incrementing by 1
            increment = 1
        count += increment      # Update the counter dynamically

# Create the generator
counter = dynamic_counter()

# Start the generator
print(next(counter))  # Output: 0 (initial count)

# Update the counter using send()
print(counter.send(5))  # Output: 5 (incremented by 5)
print(counter.send(2))  # Output: 7 (incremented by 2)

# Increment by the default value (1) by calling next()
print(next(counter))  # Output: 8
print(counter.send(-3))  # Output: 5 (decremented by 3)
```

1. **Generator Setup**:
   a. The **dynamic_counter** generator initializes a **count** variable to 0.
   b. The **yield** keyword pauses execution and outputs the current value of count.
2. **Using next()**: The first call to **next() starts** the generator and **pauses at the yield statement**, returning the initial value (0).
3. **Using send()**:
   a. The **send(value)** method sends a value to the generator, **resuming it from the yield statement**.
   b. The **sent value (increment)** is used to **update the counter dynamically**.
4. **Default Behavior**: If **send(None) or next() is called**, the generator uses a default increment of 1.
5. **Dynamic Updates**: **Subsequent calls to send() or next() continue updating** the counter based on the sent value or default.

Generator to Read Large Files

Generator Function:

- The function **read_large_file()** uses a **for** loop to read the file line by line.
- The **yield** statement **pauses** the function after returning each line.

Memory Efficiency:

- The file is **read one line at a time**, ensuring that **only the current line resides in memory**.
- This is particularly **useful for very large files**.

```
def read_large_file(file_path):
    """
    Generator function to read a large file line by line.
    """
    with open(file_path, 'r') as file:
        for line in file:
            yield line.strip()  # Yield each line after stripping whitespace

# Example usage
file_path = "large_file.txt"  # Replace with the path to your large file

# Process the file line by line
for line in read_large_file(file_path):
    print(line)  # Print or process each line
```

Built-in Exceptions (Base Exceptions)

Exception	Description	Example
BaseException	The base class for all exceptions.	-
Exception	The base class for all built-in, non-system-exiting exceptions.	-

Built-in Exceptions (Arithmetic Errors)

Exception	Description	Example
ArithmeticError	**Base class** for errors in arithmetic operations.	-
ZeroDivisionError	Raised when division or modulo by zero occurs.	1 / 0
OverflowError	Raised when a numerical result exceeds the limit for a data type.	math.exp(1000)
FloatingPointError	Raised when a floating-point operation fails.	Rarely occurs in modern Python.

Built-in Exceptions (Lookups Errors)

Exception	Description	Example
IndexError	Raised when a sequence index is out of range.	lst = [1]; print(lst[5])
KeyError	Raised when a dictionary key is not found.	d = {}; print(d['missing_key'])

Built-in Exceptions (Type and Value Errors)

Exception	Description	Example
TypeError	Raised when an operation is applied to an object of inappropriate type.	1 + 'string'
ValueError	Raised when a function gets an argument of the right type but an inappropriate value.	int('string')

Built-in Exceptions (Input/Output and File Errors)

Exception	Description	Example
EOFError	Raised when the input() function hits end-of-file condition.	input() on empty input stream
OSError	Base class for system-related errors.	open('missing_file.txt')
FileNotFoundError	Raised when a file or directory is requested but cannot be found.	open('non_existent.txt')

Logging

```
import logging

# Configure logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')

# Log messages with different levels
logging.debug("This is a DEBUG message.")
logging.info("This is an INFO message.")
logging.warning("This is a WARNING message.")
logging.error("This is an ERROR message.")
logging.critical("This is a CRITICAL message.")
```

```
2025-01-02 12:00:00,000 - DEBUG - This is a DEBUG message.
2025-01-02 12:00:00,001 - INFO - This is an INFO message.
2025-01-02 12:00:00,002 - WARNING - This is a WARNING message.
2025-01-02 12:00:00,003 - ERROR - This is an ERROR message.
2025-01-02 12:00:00,004 - CRITICAL - This is a CRITICAL message.
```

Python's **built-in logging module** provides a flexible framework for emitting log messages from programs.

Log Formatting Placeholders

Placeholder	Description
%(asctime)s	Timestamp of the log message.
%(levelname)s	Severity level of the log message (e.g., DEBUG, INFO).
%(message)s	The log message itself.
%(name)s	Name of the logger.
%(pathname)s	Full path of the source file that generated the log message.
%(lineno)d	Line number in the source file where the log message was generated.

Logging (Writing to a File)

```
import logging

# Configure logging to write to a file
logging.basicConfig(
   filename='app.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

# Log messages
logging.info("Application started.")
logging.warning("This is a warning message.")
logging.error("An error occurred.")

# File: Logs will be written to app.log.
```

Creating Custom Logger

```
import logging

# Create a custom logger
logger = logging.getLogger('custom_logger')
logger.setLevel(logging.DEBUG)

# Create handlers
console_handler = logging.StreamHandler()
file_handler = logging.FileHandler('custom.log')

# Set levels for handlers
console_handler.setLevel(logging.WARNING)
file_handler.setLevel(logging.DEBUG)

# Create a formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Add formatter to handlers
console_handler.setFormatter(formatter)
file_handler.setFormatter(formatter)

# Add handlers to the logger
logger.addHandler(console_handler)
logger.addHandler(file_handler)

# Log messages
logger.debug("Debug message for the log file only.")
logger.warning("Warning message for both console and log file.")
logger.error("Error message for both console and log file.")
```

Output in Console:

2025-01-02 12:00:00,000 - custom_logger - WARNING - Warning message for **both console and log file.**

2025-01-02 12:00:00,001 - custom_logger - ERROR - Error message for **both console and log file.**

Output in custom.log:

2025-01-02 12:00:00,000 - custom_logger - DEBUG - Debug message for the **log file only.**

2025-01-02 12:00:00,001 - custom_logger - WARNING - Warning message for **both console and log file.**

2025-01-02 12:00:00,002 - custom_logger - ERROR - Error message for **both console and log file.**

Datetime

```
from datetime import datetime

# Get the current date and time
now = datetime.now()
print("Current date and time:", now)
```

```
Current date and time:
2025-01-02 12:00:00.123456
```

```
# Format the current date and time
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print("Formatted datetime:", formatted)

# Parse a string into a datetime object
date_str = "2025-01-02 14:30:45"
parsed_date = datetime.strptime(date_str, "%Y-%m-%d %H:%M:%S")
print("Parsed datetime:", parsed_date)
```

```
Parsed datetime: 2025-01-02 14:30:45
```

```
from datetime import datetime, date, time

# Create a specific date
specific_date = date(2025, 1, 2)
print("Specific date:", specific_date)

# Create a specific time
specific_time = time(14, 30, 45)
print("Specific time:", specific_time)

# Create a datetime object
specific_datetime = datetime(2025, 1, 2, 14, 30, 45)
print("Specific datetime:", specific_datetime)
```

```
Specific date: 2025-01-02
Specific time: 14:30:45
Specific datetime: 2025-01-02 14:30:45
```

Use **strptime()** to **convert strings into datetime objects**.
Use **strftime()** for **formatting** and **strptime()** for **parsing**.

Datetime (Classes)

Class	Description
datetime	Combines both date and time into a single object.
date	Represents a date (year, month, day).
time	Represents time (hours, minutes, seconds, microseconds).
timedelta	Represents a duration or the difference between two dates or times.
tzinfo	Base class for handling time zones.

Formatting Dates and Times

Directive	Description	Example
%Y	Full year	2025
%m	Month as a zero-padded number	01
%d	Day of the month	02
%H	Hour (24-hour clock)	14
%M	Minutes	30
%S	Seconds	45

Date Arithmetic

```
from datetime import timedelta

# Add 5 days to the current date
future_date = now + timedelta(days=5)
print("Future date:", future_date)

# Subtract 3 days from the current date
past_date = now - timedelta(days=3)
print("Past date:", past_date)
```

Future date: 2025-01-07 12:00:00.123456
Past date: 2024-12-30 12:00:00.123456

```
# Difference between two dates
date1 = datetime(2025, 1, 10)
date2 = datetime(2025, 1, 2)

difference = date1 - date2

print("Difference:", difference)
print("Days:", difference.days)
```

Difference: 8 days, 0:00:00
Days: 8

Use timedelta for **adding** or **subtracting** dates and times.

Handling Time Zones

```
from datetime import datetime, timezone, timedelta

# Create a timezone-aware datetime
utc = datetime.now(timezone.utc)
print("UTC time:", utc)

# Convert to a different timezone
offset = timedelta(hours=5, minutes=30)
local_time = utc.astimezone(timezone(offset))
print("Local time:", local_time)

UTC time: 2025-01-02 12:00:00+00:00
Local time: 2025-01-02 17:30:00+05:30
```

Use **timezone** and **astimezone()** for handling time zones effectively.

Getting Day Information

```
# Day details
print("Year:", now.year)
print("Month:", now.month)
print("Day:", now.day)
print("Weekday:", now.weekday())  # Monday=0, Sunday=6

Year: 2025
Month: 1
Day: 2
Weekday: 3
```

Threading vs Coroutines

Feature	Threads	Coroutines
Management	Managed by the **OS**.	Managed by **Python's event loop**.
Concurrency Model	**Preemptive multitasking** (OS-level).	**Cooperative multitasking (software-level).**
Overhead	Higher (OS resources required).	Lower (no OS resources needed).
Switching Control	Controlled by the **OS scheduler**.	Controlled explicitly by **await**.
Suitability	Best for blocking I/O tasks.	Best for non-blocking I/O tasks.
Parallelism	**True parallelism** (except for GIL).	**Single-threaded,** no true parallelism.
Complexity	May require **locks for shared memory**.	Simplifies concurrent task handling.

Threading vs Coroutines

Threading:

- Suitable for **blocking operations** when async libraries or support aren't available.
- Works well with traditional synchronous APIs (e.g., requests, sqlite3, socket).

Coroutines:

- **Preferred** when libraries provide **asynchronous support**.
- Ideal for **high-concurrency tasks** (e.g., handling 1000+ HTTP requests or concurrent database queries).

When Neither is Ideal:

- For **CPU-bound tasks** (e.g., image processing or mathematical computations), use **multiprocessing** or libraries like **concurrent.futures**.

Threading

```
import threading
import time

def download_file(file_name):
    print(f"Starting download: {file_name}")
    time.sleep(2)  # Simulate a time-consuming task
    print(f"Finished download: {file_name}")

# Create threads for downloading multiple files
files = ["file1.txt", "file2.txt", "file3.txt"]
threads = []

for file in files:
    thread = threading.Thread(target=download_file, args=(file,))
    threads.append(thread)
    thread.start()

# Wait for all threads to complete
for thread in threads:
    thread.join()

print("All downloads are complete.")
```

```
Starting download: file1.txt
Starting download: file2.txt
Starting download: file3.txt
Finished download: file1.txt
Finished download: file2.txt
Finished download: file3.txt
All downloads are complete.
```

- Each **file download is handled in a separate thread**, allowing downloads to run concurrently.
- **thread.start()** begins the thread, and **thread.join()** ensures the main thread waits for all threads to finish.

Coroutines

```
import asyncio

async def fetch_data(api_url):
    print(f"Fetching data from {api_url}...")
    await asyncio.sleep(2)  # Simulate network delay
    print(f"Data fetched from {api_url}.")

async def main():
    # Schedule multiple API calls concurrently
    tasks = [
        fetch_data("https://api1.example.com"),
        fetch_data("https://api2.example.com"),
        fetch_data("https://api3.example.com"),
    ]
    await asyncio.gather(*tasks)

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

```
Fetching data from https://api1.example.com...
Fetching data from https://api2.example.com...
Fetching data from https://api3.example.com...
Data fetched from https://api1.example.com.
Data fetched from https://api2.example.com.
Data fetched from https://api3.example.com.
```

1. Each **fetch_data** coroutine runs concurrently using **asyncio.gather()**.
2. The **await asyncio.sleep(2)** simulates an asynchronous I/O operation.

Combining Threading and Coroutines

```
import asyncio
import time
from concurrent.futures import ThreadPoolExecutor

def blocking_task(name):
    print(f"Blocking task {name} started...")
    time.sleep(3)  # Simulate a blocking operation
    print(f"Blocking task {name} finished.")

async def async_task(name):
    print(f"Async task {name} started...")
    await asyncio.sleep(2)  # Simulate a non-blocking operation
    print(f"Async task {name} finished.")

async def main():
    loop = asyncio.get_event_loop()
    with ThreadPoolExecutor() as executor:
        # Schedule blocking tasks in threads
        blocking_tasks = [
            loop.run_in_executor(executor, blocking_task, f"Thread-{i}")
            for i in range(2)
        ]

        # Schedule async tasks
        async_tasks = [
            async_task(f"Coroutine-{i}")
            for i in range(2)
        ]

        # Wait for all tasks to complete
        await asyncio.gather(*blocking_tasks, *async_tasks)

# Run the combined tasks
asyncio.run(main())
```

```
Blocking task Thread-0 started...
Blocking task Thread-1 started...
Async task Coroutine-0 started...
Async task Coroutine-1 started...
Async task Coroutine-0 finished.
Async task Coroutine-1 finished.
Blocking task Thread-0 finished.
Blocking task Thread-1 finished.
```

1. The **blocking_task** function runs in threads using **loop.run_in_executor()**.
2. The **async_task** function **runs as a coroutine**, allowing non-blocking execution.
3. Both types of **tasks are executed concurrently**.

REST API with Sanic

Sanic is a Python web framework designed for building **fast**, **asynchronous**, and **scalable** web applications and APIs. It's **built on Python's asyncio module** and provides native support for asynchronous request handling.

```
@app.get("/get-route")
async def get_handler(request):
    return text("This is a GET route.")

@app.post("/post-route")
async def post_handler(request):
    return text("This is a POST route.")
```

```
@app.route("/user/<name>")
async def greet_user(request, name):
    return text(f"Hello, {name}!")

@app.route("/product/<int:product_id>")
async def get_product(request, product_id):
    return text(f"Product ID: {product_id}")
```

```
from sanic import Sanic
from sanic.response import text

# Create the Sanic app
app = Sanic("MyFirstApp")

# Define a route
@app.route("/")
async def hello_world(request):
    return text("Hello, Sanic!")

# Run the app
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=8000)
```

Visit http://localhost:8000
Response: "Hello, Sanic!"

Database ORM with Tortoise

Tortoise ORM is an **asynchronous Object-Relational Mapping** (ORM) library designed to work with Python's **asyncio**. It provides a simple and efficient way to interact with databases while **leveraging Python's async capabilities**.

```
from tortoise.models import Model
from tortoise import fields

class User(Model):
    id = fields.IntField(pk=True)  # Primary Key
    name = fields.CharField(max_length=50)
    email = fields.CharField(max_length=100, unique=True)
    created_at = fields.DatetimeField(auto_now_add=True)

    def __str__(self):
        return self.name
```

Define Models: Models represent database tables. Each model corresponds to a table, and each attribute corresponds to a column.

```
from tortoise import Tortoise, run_async

async def init():
    await Tortoise.init(
        db_url="sqlite://db.sqlite3",  # Database URL
        modules={"models": ["__main__"]}  # Import path to your models
    )
    await Tortoise.generate_schemas()  # Create tables

# Run initialization
run_async(init())
```

Initialize Tortoise: Set up Tortoise ORM in the application.

```
@app.route("/users")
async def get_users(request):
        users = await User.all()
        return json([{"id": user.id, "name": user.name} for user in users])
```

https://github.com/helabenkhalfallah/art-bloom/tree/main

Tools

pip

pip is the **default package manager** for Python, used to install and manage Python libraries and dependencies from the Python Package Index (PyPI).

pipenv

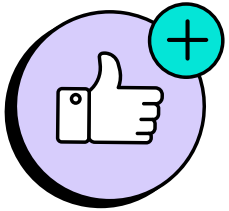
pipenv is a **dependency management tool** for Python that combines pip and virtualenv functionality. It automatically **creates and manages virtual environments and maintains a Pipfile for tracking dependencies**.

Pylint

pylint is a **static code analysis tool** for Python that checks for coding errors, enforces coding standards, and suggests improvements.

Pytest

pytest is a powerful **testing framework for Python,** supporting unit tests, functional tests, and complex test setups with fixtures and plugins.



Thanks!

Do you have any questions?

https://helabenkhalfallah.com/
https://helabenkhalfallah.medium.com/
https://x.com/b_k_hela