Standard Library in Python

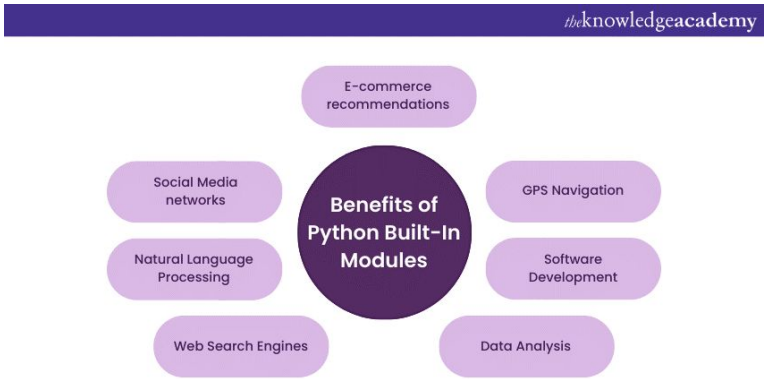
Built-in Modules

Content

Introduction

Python offers a comprehensive standard library with built-in modules designed to support a wide range of programming tasks without requiring external packages. This presentation will organize these modules into categories based on their functionality.

1. **File and Filesystem Operations**
   - Modules for interacting with files, directories and the file system.
2. **Mathematics and Numbers**
   - Tools for mathematical operations, number theory and numerical computation.
3. **Data Structures and Algorithms**
   - Modules providing built-in data types and useful algorithms.
4. **Text Processing and String Handling**
   - Utilities for manipulating text, parsing and formatting strings.
5. **Internet Protocols and Web**
   - Modules for handling network protocols, web access and internet data.
6. **Dates and Time**
   - Tools for managing and manipulating date and time values.
7. **Data Formats and Serialization**
   - Modules for reading/writing structured data like JSON, CSV and XML.
8. **Security and Cryptography**
   - Basic tools for hashing, encryption and secure communication.
9. **Utilities and Language Features**
   - General-purpose tools and modules that enhance Python`s functionality.
10. **Concurrency and Parallelism**
    - Modules that support multi-threading, multi-processing and asynchronous execution.
11. **Testing and Debugging**
    - Tools to test, debug and profile Python programs.
12. **Memory and Garbage Collection**
    - Modules for managing memory and interacting with the garbage collector.



File and Filesystem operations

1. **os** - **Interfaces with the operating system for file, process and environment management**.
   - Interacts with the operating system: file and directory manipulation (os.remove, os.rename, etc.).
   - Accesses environment variables, process management (os.environ, os.system).
   - Path manipulations (os.path).
2. **shutil - Performs high-level file and directory operations like copy and move.**
   - High-level file operations: copying, moving, removing directories and files.
   - shutil.copy, shutil.copytree, shutil.rmtree.
   - Disk usage stats: shutil.disk_usage.
3. **glob - Finds files using Unix-style wildcards patterns.**
   - Pattern matching for file paths (*.txt, data/*.csv).
   - Uses Unix-style wildcards.
   - glob.glob(), glob.iglob()
4. **pathlib - Provides object-oriented file system path operations**
   - Object-oriented approach to filesystem paths.
   - Path objects: easy joining, reading/writing, checking existence, etc.
   - Replaces much of os.path, glob, and parts of shutil.
5. **tempfile - Creates temporary files and directories securely**
   - Secure creation of temporary files and directories.
   - Functions: NamedTemporaryFile, TemporaryDirectory, mkstemp.

```python
# File and system Operations

''' OS module provides a way of using operating system dependent functionality like reading or writing to the file system, environment variables, etc.'''
import os

# Example 1: Listing files in current directory
files = os.listdir('.')  # Lists all files and folders in the current directory

# Example 2: Get environment variable
home = os.environ.get('HOME')  # Gets the HOME environment variable

# Example 3: Creating a directory
os.mkdir('new_folder')  # Creates a new folder named 'new_folder'

# Example 4: Remove a file
os.remove('old_file.txt')  # Deletes the file 'old_file.txt'



'''shutil module provides a higher-level interface for file operations, such as copying, moving, and deleting files and directories.'''

import shutil

# Example 1: Copy a file
shutil.copy('source.txt', 'backup.txt')  # Copies source.txt to backup.txt

# Example 2: Move a file
shutil.move('backup.txt', 'archive/backup.txt')  # Moves backup.txt to archive folder

# Example 3: Delete a directory
shutil.rmtree('old_folder')  # Recursively deletes 'old_folder' and all its contents

# Example 4: Get disk usage
```

```python
'''glob module provides a way to find all the pathnames matching a specified pattern according to the rules used by the Unix shell.'''
import glob

# Example 1: Get all .txt files in current directory
txt_files = glob.glob('*.txt')  # Returns a list of all .txt files

# Example 2: Recursive search
all_logs = glob.glob('**/*.log', recursive=True)  # Finds all .log files in all subdirectories


'''pathlib module provides an object-oriented approach to handle filesystem paths. It allows for easy manipulation of file paths and directories.'''

from pathlib import Path

# Example 1: Create a Path object and read text
path = Path('example.txt')
content = path.read_text()  # Reads the content of the file

# Example 2: Write to a file
path.write_text('Hello, World!')  # Overwrites file with this text

# Example 3: Check if file exists
exists = path.exists()  # Returns True if file exists

# Example 4: Join paths
data_path = Path('data') / 'input.csv'  # Creates a combined path object


'''tempfile module provides functions to create temporary files and directories. These files are automatically deleted when closed or when the program exits.'''

import tempfile

# Example 1: Temporary file with automatic cleanup
with tempfile.NamedTemporaryFile(delete=True) as tmp:
    tmp.write(b'Hello, temp file!')
    tmp.seek(0)
    print(tmp.read())  # Prints: b'Hello, temp file!'

# Example 2: Temporary directory
with tempfile.TemporaryDirectory() as tmpdir:
    print(f'Temporary directory created at: {tmpdir}')
    # Directory is deleted after the block ends
```

Mathematics and Numbers

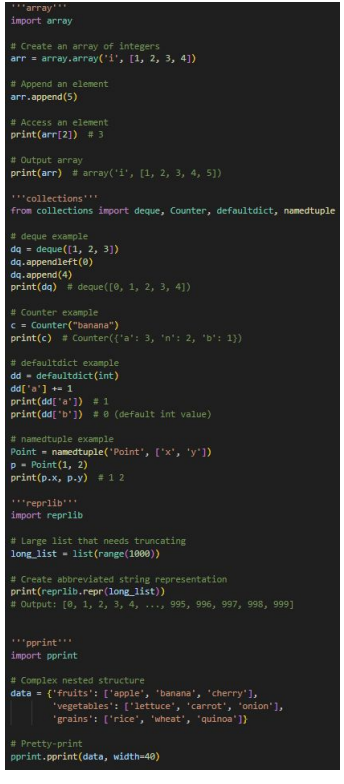
1. **math - Offers basic mathematical functions and constants**
   - Provides access to mathematical functions like sqrt(), log(), sin(), ceil(), etc.
   - Constants like math.pi, math.e.
2. **random - Generates pseudo-random numbers for simulations or sampling.**
   - Pseudo-random number generations.
   - Functions: random(), randint(a, b), choice(seq), shuffle(list), uniform(a, b).
3. **statistics - Calculates basic statistical measures.**
   - Basic statistical operations: mean, median, mode, stdev, variance
   - Works on sequences (lists, tuples).
4. **decimal - Performs precise decimal arithmetic with adjustable precision.**
   - Decimal floating-point arithmetic with user-definable precision.
   - Avoids binary floating-point issues (0.1 + 0.2 != 0.3).
5. **heapq - Implements a min-leap for efficient priority queue operations.**
   - Implements a min-heap queue using regular lists.
   - Functions: heappush(), heappop(), heapify().
6. **bisect - Provides fast binary search and insertion on sorted lists.**
   - Binary search operations on sorted lists.
   - Functions: bisect_left(), bisect_right(), insort().

```python
'''1. Math'''
import math
# Example 1: square root
print(math.sqrt(16))  # Output: 4.0
# Example 2: Trigonometric functions
print(math.sin(math.pi / 2))  # Output: 1.0
# Example 3: Ceiling and floor
print(math.ceil(2.3))  # Output: 3
print(math.floor(2.7))  # Output: 2
# Example 4: Logarithm
print(math.log(100, 10))  # Output: 2.0

''' 2. Random'''
import random
# Example 1: Random float between 0 and 1
print(random.random())  # e.g., 0.643...
# Example 2: Random integer between 1 and 10
print(random.randint(1, 10))  # e.g., 7
# Example 3: Choose a random element
print(random.choice(['apple', 'banana', 'cherry']))  # e.g., 'banana'
# Example 4: Shuffle a list
items = [1, 2, 3, 4, 5]
random.shuffle(items)
print(items)  # Shuffled list
# Example 5: Reproducibility with seed
random.seed(42)
print(random.random())  # Always the same if seed is fixed

'''3. statistics'''
import statistics
data = [1, 2, 2, 3, 4, 5]
# Example 1: Mean (average)
print(statistics.mean(data))  # 2.833...
# Example 2: Median
print(statistics.median(data))  # 2.5
# Example 3: Mode
print(statistics.mode(data))  # 2 (most frequent)
# Example 4: Standard deviation
print(statistics.stdev(data))  # e.g., 1.47
```

```python
'''4. decimal'''
from decimal import Decimal, getcontext
# Set precision
getcontext().prec = 5
# Example 1: Basic arithmetic
a = Decimal('0.1')
b = Decimal('0.2')
print(a + b)  # 0.3 (precise!)
# Example 2: Without Decimal
print(0.1 + 0.2)  # 0.30000000000000004 (imprecise due to binary float)

'''5. heapq'''
import heapq
# Example 1: Create a heap
numbers = [5, 3, 8, 1]
heapq.heapify(numbers)  # transforms list into a min-heap
print(numbers)  # [1, 3, 8, 5]
# Example 2: Add an element
heapq.heappush(numbers, 2)
print(numbers)  # [1, 2, 8, 5, 3]
# Example 3: Remove smallest element
print(heapq.heappop(numbers))  # 1

'''6. bisect'''
import bisect
# Must be a sorted list
scores = [10, 20, 30, 40, 50]
# Example 1: Find position to insert
index = bisect.bisect_left(scores, 25)  # Finds insertion point for 25
print(index)  # 2
# Example 2: Insert while maintaining order
bisect.insort(scores, 25)
print(scores)  # [10, 20, 25, 30, 40, 50]
```

Data Structures and Algorithms

1. ***array - Stores fixed-type numeric data more compactly than lists.***
   - Provides compact storage of basic data types (ints, floats).
   - More memory-efficient than lists for large numeric arrays
2. ***collections - Provides high-performance alternatives to built-in types.***
   - deque: Double-ended queue.
   - Counter: Multiset (counts occurrences).
   - defaultdict: Dictionary with default value.
   - OrderedDict (pre 3.7 relevance): Maintains insertion order.
   - namedtuple: Tuple with named fields.
3. ***reprlib - Safely creates abbreviated repr() strings for large objects***
   - Creates shortened repr() strings for large or deeply nested objects.
4. ***pprint - Nicely formats complex or nested data structures for readability.***
   - Pretty-prints nested structures like dictionaries or lists.
   - Can print to console or to a string.
5. ***enum - Defines symbolic constant values using enumerations.***
   - Defines named constant values with identity and comparison semantics.
   - Prevents magic numbers and improves readability.
6. ***functools - Offers functional tools like decorators and memoization.***
   - Higher-order functions: decorators, partial application, caching.
   - Key functions: lru_cache, partial, reduce, cmp_to_key.
7. ***itertools - Builds complex iterators for efficient looping and combination.***
   - Tools for creating and using iterators efficiently.
   - Common functions: chain, cycle, count, combinations, product, groupby.
8. ***operator - Provides functional access to Python's operators and object handling.***
   - Functional equivalents of operators (e.g., operator.add, operator.itemgetter, operator.attrgetter).
   - Improves performance and readability in high-order functions (e.g., sorted with key)



```
'''array'''
import array

# Create an array of integers
arr = array.array('i', [1, 2, 3, 4])

# Append an element
arr.append(5)

# Access an element
print(arr[2])  # 3

# Output array
print(arr)  # array('i', [1, 2, 3, 4, 5])

'''collections'''
from collections import deque, Counter, defaultdict, namedtuple

# deque example
dq = deque([1, 2, 3])
dq.appendleft(0)
dq.append(4)
print(dq)  # deque([0, 1, 2, 3, 4])

# Counter example
c = Counter("banana")
print(c)  # Counter({'a': 3, 'n': 2, 'b': 1})

# defaultdict example
dd = defaultdict(int)
dd['a'] += 1
print(dd['a'])  # 1
print(dd['b'])  # 0 (default int value)

# namedtuple example
Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)
print(p.x, p.y)  # 1 2

'''reprlib'''
import reprlib

# Large list that needs truncating
long_list = list(range(1000))

# Create abbreviated string representation
print(reprlib.repr(long_list))
# Output: [0, 1, 2, 3, 4, ..., 995, 996, 997, 998, 999]


'''pprint'''
import pprint

# Complex nested structure
data = {'fruits': ['apple', 'banana', 'cherry'],
        'vegetables': ['lettuce', 'carrot', 'onion'],
        'grains': ['rice', 'wheat', 'quinoa']}

# Pretty-print
pprint.pprint(data, width=40)
```

```
'''enum'''
from enum import Enum

class Status(Enum):
    PENDING = 1
    APPROVED = 2
    REJECTED = 3
# Usage
print(Status.APPROVED)      # Status.APPROVED
print(Status.APPROVED.name)  # 'APPROVED'
print(Status.APPROVED.value)  # 2


'''functools'''
from functools import lru_cache, partial, reduce

# lru_cache example
@lru_cache(maxsize=128)
def fib(n):
    if n < 2:
        return n
    return fib(n - 1) + fib(n - 2)

print(fib(10))  # 55

# partial example
def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
print(square(5))  # 25

# reduce example
nums = [1, 2, 3, 4]
product = reduce(lambda x, y: x * y, nums)
print(product)  # 24

'''itertools'''
import itertools

# chain: Flatten multiple iterables
a = [1, 2]
b = [3, 4]
print(list(itertools.chain(a, b)))  # [1, 2, 3, 4]

# count: Infinite counting (use with caution)
counter = itertools.count(start=10, step=2)
print(next(counter))  # 10
print(next(counter))  # 12

# combinations: All unique pairs
letters = ['A', 'B', 'C']
print(list(itertools.combinations(letters, 2)))
# [('A', 'B'), ('A', 'C'), ('B', 'C')]

# product: Cartesian product
colors = ['red', 'green']
sizes = ['S', 'M']
print(list(itertools.product(colors, sizes)))
# [('red', 'S'), ('red', 'M'), ('green', 'S'), ('green', 'M')]

# groupby: Group consecutive values
data = [('a', 1), ('a', 2), ('b', 3)]
for key, group in itertools.groupby(data, key=lambda x: x[0]):
    print(key, list(group))
# a [('a', 1), ('a', 2)]
# b [('b', 3)]
```

```
'''operator'''
import operator

# Basic operator functions
print(operator.add(2, 3))       # 5
print(operator.mul(4, 5))       # 20
print(operator.pow(2, 3))       # 8

# itemgetter: Used to extract items by index
data = [('a', 2), ('b', 1), ('c', 3)]
get_second = operator.itemgetter(1)
print(get_second(data[0]))  # 2

# Sort list of tuples by second item
sorted_data = sorted(data, key=operator.itemgetter(1))
print(sorted_data)  # [('b', 1), ('a', 2), ('c', 3)]

# attrgetter: Extracts object attributes
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

people = [Person('Alice', 30), Person('Bob', 25)]
sorted_people = sorted(people, key=operator.attrgetter('age'))
print([p.name for p in sorted_people])  # ['Bob', 'Alice']
```

Text Processing and String Handling

1. **re (Regular Expressions) - Enables pattern matching and manipulation using regular expressions.**
   - Pattern matching, searching, substitution.
   - Functions: search(), match(), findall(), sub(), compile().
2. **string (also includes Template) - Contains string constants and a simple safe substitution system.**
   - Constants like ascii_letters, digits, punctuation.
   - Template class for safe string substitution.
3. **textwrap - Wraps and formats text into nicely formatted blocks.**
   - Wraps and formats text to fit within a fixed width.
   - Functions: fill(), wrap(), dedent().
4. **unicodedata - Provides access to Unicode character properties and normalization**
   - Access to Unicode character properties (name, category, normalization).
   - Functions: name(), lookup(), normalize().
5. **difflib - Compares sequences to identify differences and similarities.**
   - Compares sequences to show differences.
   - Useful for text comparison, line diffs, fuzzy matching.
6. **stringprep - Prepares Unicode text for safe network transmission.**
   - Prepares Unicode text for network protocols (used in IDNA, SASL, etc.)
7. **html.parser - Parses HTML content using a simple event-driven parser.**
   - Parses HTML text (basic HTML structure extraction).
   - Subclass HTMLParser and override handler methods.

```
'''re'''
import re

text = "The rain in Spain"

# search: find first match anywhere
match = re.search(r"rain", text)
print(match.group())  # 'rain'

# match: only matches at the start
print(re.match(r"The", text))  # Match object

# findall: find all matching substrings
print(re.findall(r"\b\w{4}\b", text))  # ['rain', 'Spain']

# sub: replace text
print(re.sub(r"Spain", "France", text))  # 'The rain in France'

# compile: precompile pattern for reuse
pattern = re.compile(r"\b\w{4}\b")
print(pattern.findall(text))  # ['rain', 'Spain']

'''string and Template'''
import string
from string import Template

# Constants
print(string.ascii_letters)  # abc...XYZ
print(string.digits)         # 0123456789
print(string.punctuation)    # !"#$%&'()*+...

# Template usage (safe substitution)
template = Template("Hello, $name!")
print(template.substitute(name="Alice"))  # Hello, Alice!

# Safer with user input (missing keys raise KeyError with substitute but not with safe_substitute)
print(template.safe_substitute())  # Hello, $name!

'''textwrap'''
import textwrap

text = "Python is an amazing programming language that emphasizes code readability."

# Wrap text into a list of lines
wrapped = textwrap.wrap(text, width=30)
print(wrapped)

# Fill: wrap and join into a single string
print(textwrap.fill(text, width=30))

# Dedent: remove common leading whitespace
raw = """\
    This is indented text.
    So is this."""
print(textwrap.dedent(raw))
```

```
'''unicodedata'''
import unicodedata

char = 'Ñ'

# Unicode name
print(unicodedata.name(char))  # LATIN SMALL LETTER N WITH TILDE

# Normalize to composed form
s1 = 'Ñ'  # 'n' + combining tilde
s2 = unicodedata.normalize('NFC', s1)
print(s2 == 'Ñ')  # True

# Category
print(unicodedata.category('A'))  # 'Lu' (Letter, uppercase)

'''difflib'''
import difflib

text1 = "apple"
text2 = "applesauce"

# Show diff in unified format
diff = difflib.unified_diff([text1], [text2], lineterm='')
print('\n'.join(diff))

# Fuzzy ratio comparison
ratio = difflib.SequenceMatcher(None, text1, text2).ratio()
print(ratio)  # e.g., 0.615

# Close matches
names = ["banana", "bandana", "cabana"]
print(difflib.get_close_matches("banana", names))  # ['banana', 'bandana', 'cabana']

'''stringprep'''
import stringprep

# Check if character is in a prohibited list for networking input
print(stringprep.in_table_c12('\u0001'))  # True (control character)

# Rarely used directly in apps — used internally for IDNA, LDAP, etc.

'''html.parser'''
from html.parser import HTMLParser

class MyHTMLParser(HTMLParser):
    def handle_starttag(self, tag, attrs):
        print(f"Start tag: {tag}")

    def handle_data(self, data):
        print(f"Data: {data}")

parser = MyHTMLParser()
html = "<html><body><h1>Hello!</h1></body></html>"
parser.feed(html)
```

Internet Protocols and Web

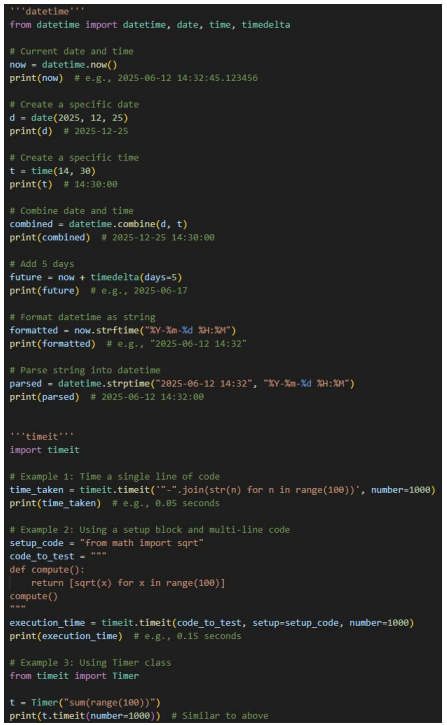
1. **urllib.request - Fetches URLs and handles HTTP requests natively.**
   - Performs HTTP(S) requests: downloading content, handling headers, proxies etc.
   - Functions: urlopen(), Request() and urlretrieve().
   - Basic for HTTP operations — no third-party dependencies.
2. **smtplib - Sends emails using SMTP protocol.**
   - Sends emails via SMTP.
   - Common methods: SMTP.sendemail(), SMTP.login(), SMTP.starttls().
3. **email - Constructs, parses, and handles email messages and attachments.**
   - Build or parse email messages (MIME, attachments, headers).
4. **json - Encodes and decodes data in JSON format.**
   - Encode and decode JSON data.
   - json.dumps(), json.loads(), json.dump(), json.load().
5. **xmlrpc.client - Calls remote methods over HTTP using XML-RPC protocol.**
   - Communicates with XML-RPC servers (remote procedure calls over HTTP).
   - Use ServerProxy to connect and call remote methods.
6. **xmlrpc.server - Hosts and exposes Python functions via XML-RPC**
   - Creates XML-RPC server endpoints.
   - Use SimpleXMLRPCServer and register_function().

```
'''urllib.request'''
import urllib.request

# Download and read a webpage
with urllib.request.urlopen('https://example.com') as response:
    html = response.read()
    print(html.decode())  # Outputs HTML content

# Download a file
urllib.request.urlretrieve('https://example.com/image.png', 'image.png')

# Basic for HTTP operations -- no third-party dependencies.

'''smtplib'''
import smtplib

# Send a simple email
sender = "you@example.com"
receiver = "friend@example.com"
message = """\
Subject: Hi There

This is a test email from Python."""

with smtplib.SMTP("smtp.example.com", 587) as server:
    server.starttls()  # Upgrade to secure connection
    server.login("you@example.com", "password")  # Login
    server.sendmail(sender, receiver, message)
# Works well with email.message for more complex email content.


'''email.message'''
from email.message import EmailMessage

# Construct an email message with a subject and body
msg = EmailMessage()
msg['Subject'] = "Meeting Reminder"
msg['From'] = "you@example.com"
msg['To'] = "colleague@example.com"
msg.set_content("Don't forget our meeting at 10am tomorrow.")

# Send it using smtplib
import smtplib
with smtplib.SMTP("smtp.example.com", 587) as server:
    server.starttls()
    server.login("you@example.com", "password")
    server.send_message(msg)
# You can also use msg.add_attachment() for files.
```

```
'''json'''
import json

# Dictionary to JSON string
data = {"name": "Alice", "age": 30}
json_str = json.dumps(data)
print(json_str)  # {"name": "Alice", "age": 30}

# JSON string to dictionary
parsed = json.loads(json_str)
print(parsed['name'])  # Alice

# Write JSON to file
with open('data.json', 'w') as f:
    json.dump(data, f)

# Read JSON from file
with open('data.json') as f:
    loaded = json.load(f)
    print(loaded)


'''xmlrpc.client'''
import xmlrpc.client

# Connect to an XML-RPC server
server = xmlrpc.client.ServerProxy("http://localhost:8000/")

# Call remote method
result = server.add(5, 3)
print(result)  # Should print 8 if server defines `add`

'''xmlrpc.server'''
from xmlrpc.server import SimpleXMLRPCServer

# Define a function to expose
def add(x, y):
    return x + y

# Create server and register the function
server = SimpleXMLRPCServer(("localhost", 8000))
server.register_function(add, "add")

print("Server running on port 8000...")
server.serve_forever()
#You can call this server using xmlrpc.client as shown above.
```

Dates and Time

1. **datetime - Manages dates, times and timedeltas with rich formatting and parsing.**
   - Work with dates, times and timestamps.
   - Main classes: datetime, date, time, timedelta.
   - Supports formatting (strftime) and parsing (strptime).
   - Timezone support exists but can be tricky – use zoneinfo (3.9+) or pytz for better control.
2. **timeit - Measures execution time of small code snippets for benchmarking.**
   - Measures execution time of small code snippets.
   - Useful for benchmarking and performance tuning.
   - Executes code many times for accurate timing.
   - More reliable than using time.time() for small operations



```
'''datetime'''
from datetime import datetime, date, time, timedelta

# Current date and time
now = datetime.now()
print(now)  # e.g., 2025-06-12 14:32:45.123456

# Create a specific date
d = date(2025, 12, 25)
print(d)  # 2025-12-25

# Create a specific time
t = time(14, 30)
print(t)  # 14:30:00

# Combine date and time
combined = datetime.combine(d, t)
print(combined)  # 2025-12-25 14:30:00

# Add 5 days
future = now + timedelta(days=5)
print(future)  # e.g., 2025-06-17

# Format datetime as string
formatted = now.strftime("%Y-%m-%d %H:%M")
print(formatted)  # e.g., "2025-06-12 14:32"

# Parse string into datetime
parsed = datetime.strptime("2025-06-12 14:32", "%Y-%m-%d %H:%M")
print(parsed)  # 2025-06-12 14:32:00


'''timeit'''
import timeit

# Example 1: Time a single line of code
time_taken = timeit.timeit('"-".join(str(n) for n in range(100))', number=1000)
print(time_taken)  # e.g., 0.05 seconds

# Example 2: Using a setup block and multi-line code
setup_code = "from math import sqrt"
code_to_test = """
def compute():
    return [sqrt(x) for x in range(100)]
compute()
"""
execution_time = timeit.timeit(code_to_test, setup=setup_code, number=1000)
print(execution_time)  # e.g., 0.15 seconds

# Example 3: Using Timer class
from timeit import Timer

t = Timer("sum(range(100))")
print(t.timeit(number=1000))  # Similar to above
```

Data Formats and Serialization

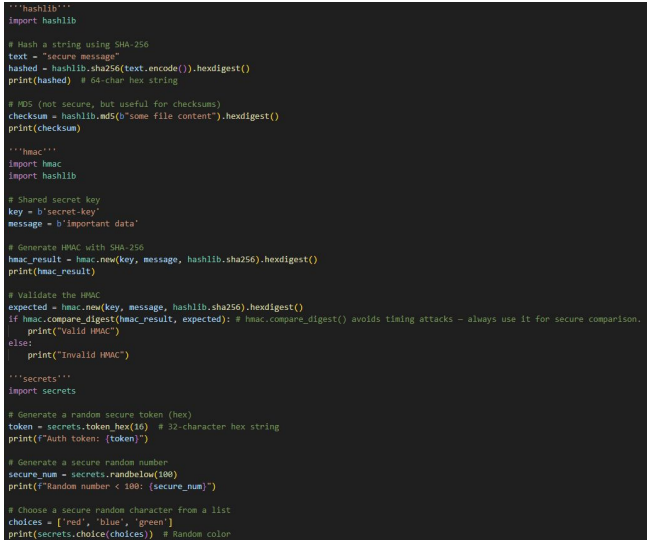
1. **csv - Reads and writes tabular data in CSV**
   - Reading and writing CSV (Comma-Separated Values) files.
   - Uses csv.reader, csv.writer and DictReader, DictWriter.
2. **sqlite3 - Provides embedded SQL database engine with SQL interface.**
   - Lightweight, embedded SQL database engine.
   - Use SQL queries with Python bindings via Connection and Cursor.
3. **zipfile - Reads and writes ZIP archive files.**
   - Create, read and extract ZIP archives.
   - Use ZipFile for context-managed operations.
4. **tarfile - Creates and extracts TAR archives, including compressed variants.**
   - Create and manipulate TAR archives (.tar, .tar.gz, etc).
   - Works similarly to zipfile but supports stream compression.
5. **zlib / bz2 / lzma - Compresses and decompresses data using DEFLATE algorithm, handles Bzip-2 compressed data streams and compresses data using the high-ratio LZMA/XZ format.**
   - Compress and decompress binary data
   a. zlib: DEFLATE compression (used in ZIP).
   b. bz2: Bzip2 compression.
   c. lzma: XZ compression (higher ratio, slower).

```
'''csv'''
import csv

# Writing to a CSV file
with open('people.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Name', 'Age'])
    writer.writerow(['Alice', 30])
    writer.writerow(['Bob', 25])

# Reading from a CSV file
with open('people.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)  # ['Name', 'Age'], then ['Alice', '30'], etc.

# Using DictWriter and DictReader
with open('people_dict.csv', 'w', newline='') as file:
    fieldnames = ['Name', 'Age']
    writer = csv.DictWriter(file, fieldnames=fieldnames)
    writer.writeheader()
    writer.writerow({'Name': 'Charlie', 'Age': 40})


'''sqlite3'''
import sqlite3

# Connect to a database (or create if it doesn't exist)
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Create a table
cursor.execute('CREATE TABLE IF NOT EXISTS users (id INTEGER PRIMARY KEY, name TEXT)')

# Insert a row
cursor.execute('INSERT INTO users (name) VALUES (?)', ('Alice',))
conn.commit()

# Query the data
cursor.execute('SELECT * FROM users')
print(cursor.fetchall())  # [(1, 'Alice')]

# Close the connection
conn.close()

'''zipfile'''
import zipfile

# Create a ZIP file
with zipfile.ZipFile('archive.zip', 'w') as zipf:
    zipf.write('people.csv')  # Adds file to the archive

# Extract ZIP file
with zipfile.ZipFile('archive.zip', 'r') as zipf:
    zipf.extractall('extracted_files')  # Extracts all contents
```

```
'''tarfile'''
import tarfile

# Create a TAR.GZ file
with tarfile.open('archive.tar.gz', 'w:gz') as tar:
    tar.add('people.csv')

# Extract TAR file
with tarfile.open('archive.tar.gz', 'r:gz') as tar:
    tar.extractall('extracted_tar')

'''zlib'''
import zlib

data = b'This is some data that needs to be compressed.'

# Compress data
compressed = zlib.compress(data)
print(compressed)

# Decompress data
original = zlib.decompress(compressed)
print(original.decode())  # 'This is some data that needs to be compressed.'

'''bz2'''
import bz2

data = b'Bzip2 is another compression method.'

# Compress
compressed = bz2.compress(data)
print(compressed)

# Decompress
print(bz2.decompress(compressed).decode())  # 'Bzip2 is another compression method.'

'''lzma'''
import lzma

data = b'LZMA offers very high compression ratio.'

# Compress
compressed = lzma.compress(data)

# Decompress
print(lzma.decompress(compressed).decode())  # 'LZMA offers very high compression ratio.'
```

Security and Cryptography

1. **hashlib - Generates hash digests like SHA-256 for data integrity.**
    - Provides hashing algorithms like SHA-256, SHA-1, MD5.
    - Useful for checksums, password hashing (not encryption!).
    - md5() and sha1() are **not secure** for cryptographic use.
    - Always encode strings to bytes before hashing.
2. **hmac - Computes secure message digests using shared keys.**
    - Hash-Based Message Authentication Code.
    - Ensures data integrity and authenticity using a shared secret and a hash function.
    - Safer than plain hashes for validating data or tokens.
3. **secrets - Generates cryptographically secure random numbers and tokens.**
    - Generates cryptographically strong random numbers and tokens.
    - Designed for things like passwords, tokens and API keys.
    - Use instead of random for anything security-sensitive.



```
'''hashlib'''
import hashlib

# Hash a string using SHA-256
text = "secure message"
hashed = hashlib.sha256(text.encode()).hexdigest()
print(hashed)  # 64-char hex string

# MD5 (not secure, but useful for checksums)
checksum = hashlib.md5(b"some file content").hexdigest()
print(checksum)

'''hmac'''
import hmac
import hashlib

# Shared secret key
key = b'secret-key'
message = b'important data'

# Generate HMAC with SHA-256
hmac_result = hmac.new(key, message, hashlib.sha256).hexdigest()
print(hmac_result)

# Validate the HMAC
expected = hmac.new(key, message, hashlib.sha256).hexdigest()
if hmac.compare_digest(hmac_result, expected): # hmac.compare_digest() avoids timing attacks — always use it for secure comparison.
    print("Valid HMAC")
else:
    print("Invalid HMAC")

'''secrets'''
import secrets

# Generate a random secure token (hex)
token = secrets.token_hex(16)  # 32-character hex string
print(f"Auth token: {token}")

# Generate a secure random number
secure_num = secrets.randbelow(100)
print(f"Random number < 100: {secure_num}")

# Choose a secure random character from a list
choices = ['red', 'blue', 'green']
print(secrets.choice(choices))  # Random color
```

Utilities and Language Features (1/2)

1. **sys - Accesses system-level variables, input/output and interpreter details.**
   - Access system-level functions and variables.
   - Useful for CLI tools and module path management.
2. **argparse - Parses command-line arguments and generates usage help.**
   - Parses command-line arguments.
   - Supports positional and optional arguments, help text, type checking.
3. **contextlib - Simplifies writing and managing context managers.**
   - Utilities for working with context managers (with statements).
   - Can replace try-finally blocks with cleaner code.
4. **abc - Creates abstract base classes to enforce method implementation.**
   - Defines abstract base classes (ABCs)
   - Enforces method implementation in subclasses.
5. **typing - Adds type hinting support for static analysis and tooling.**
   - Type hints for functions, classes, generics.
   - Enhances code quality and tool support (e.g., mypy).
6. **dataclasses - Automatically generates methods for classes that store data.**
   - Reduces boilerplate for classes storing data.
   - Auto-generates __init__, __repr__ and comparison methods.

```python
'''sys'''
import sys

# Command-line arguments
print(sys.argv)  # List of args passed to script (first is script name)

# Exit the program
if '--exit' in sys.argv:
    sys.exit("Exiting program...")

# Show Python version
print(sys.version)

# Modify import path at runtime
sys.path.append('/my/custom/modules')  # Adds new path for module search

'''argparse'''
import argparse

parser = argparse.ArgumentParser(description='Example CLI app')
parser.add_argument('--name', type=str, help='Your name')
args = parser.parse_args()

print(f"Hello, {args.name}!")

'''contextlib'''
from contextlib import contextmanager

# Custom context manager
@contextmanager
def open_file(name):
    f = open(name, 'w')
    try:
        yield f
    finally:
        f.close()

with open_file('sample.txt') as f:
    f.write("Hello with context!")
```

```python
'''abc'''
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return "Woof!"

# a = Animal()  # ❌ TypeError: Can't instantiate abstract class
d = Dog()
print(d.speak())  # "Woof!"


'''typing'''
from typing import List, Optional

def greet_all(names: List[str]) -> None:
    for name in names:
        print(f"Hello, {name}!")

def get_user(id: int) -> Optional[str]:
    return "Alice" if id == 1 else None

'''dataclasses'''
from dataclasses import dataclass

@dataclass
class Point:
    x: int
    y: int = 0  # default value

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

print(p1)  # Point(x=3, y=4)
print(p2)  # Point(x=3, y=0)
```

Utilities and Language Features (2/2)

1. **warnings - Issues runtime warnings to alert about potential issues.**
   - Issues runtime warnings to users.
   - Use warn() for deprecation, API change notices, etc.
   - Can be filtered or turned into exceptions.
2. **importlib - Dynamically imports and reloads Python modules.**
   - Programmatic module importing and reloading.
   - Allows dynamic imports – use with care in production.
3. **locale - Formats text and numbers according to cultural conventions.**
   - Format numbers, dates and strings according to locale settings.
   - Locale must be set explicitly (locale.setlocale()).
4. **gettext - Enables language translation for application localization.**
   - Provides internationalization (i18n) and localization (l10n) support.
   - Use _() as an alias for translated strings.
   - Needs proper setup to integrate in apps.
5. **codecs - Encodes and decodes text using various character encodings.**
   - Encodes and decodes files using different encoding (e.g., UTF-8, Latin-1).
   - open() in Python 3 handles encodings –codecs is mostly needed for legacy compatibility.

```
'''warnings'''
import warnings

# Emit a warning
warnings.warn("This feature is deprecated", DeprecationWarning)

'''importlib'''
import importlib

# Dynamically import a module
math_module = importlib.import_module('math')
print(math_module.sqrt(16))  # 4.0

# Reload a module (e.g., after code changes in development)
importlib.reload(math_module)

'''locale'''
import locale

# Set locale for number formatting (may vary by system)
locale.setlocale(locale.LC_ALL, '')  # Use system default locale

# Format number with grouping
num = 1234567.89
formatted = locale.format_string("%n", num, grouping=True)
print(formatted)  # e.g., '1,234,567.89' in en_US

# Get currency symbol
print(locale.localeconv()['currency_symbol'])

'''gettext'''
import gettext

# Set up translation
# Normally you'd use .mo files; here's a dummy fallback example
gettext.install('example', localedir='locale', names=['ngettext'])

# Mark strings for translation
print(_("Hello, world!"))  # Prints translated string if available

'''codecs'''
import codecs

# Write text with a specific encoding
with codecs.open('example_utf16.txt', 'w', encoding='utf-16') as f:
    f.write("This is UTF-16 encoded text.")

# Read it back
with codecs.open('example_utf16.txt', 'r', encoding='utf-16') as f:
    content = f.read()
    print(content)
```

Concurrency and Parallelism

1. **threading - Runs code concurrently using OS-level threads.**
   - Lightweight concurrent execution using OS-level threads.
   - Use thread, Lock, Event.
   - Python threads are subject to the **Global Interpreter Lock (GIL)** – useful for I/O bound tasks, not CPU-bound.
2. **concurrent.futures - Simplifies parallelism using thread or process pools.**
   - High-level API for asynchronous execution using threads or processes.
   - Executors: ThreadPoolExecutor, ProcessPoolExecutor.
   - Easy to parallelize with submit() and map()
3. **multiprocessing - Executes tasks in separate processes for true parallelism.**
   - True parallelism using multiple processes (bypasses GIL).
   - Heavier than threads – avoid for quick/lightweight tasks.
   - Requires if __name__ == "__main__:" block on Windows.
4. **asyncio - Handles asynchronous I/O using coroutines and event loops.**
   - Asynchronous I/O using async/await.
   - Run coroutines concurrently with asyncio.run() or asyncio.gather().
   - Not parallel execution – it's **cooperative multitasking.**
5. **queue - Manages thread-safe queues for inter-thread communication.**
   - Thread-safe FIFO/LIFO/priority queues.
   - Great for producer-consumer problems.
   - Works with threading, not multiprocessing (use multiprocessing.Queue instead).

```
'''threading'''
import threading
import time

def worker():
    print("Thread starting")
    time.sleep(1)
    print("Thread finished")

# Create and start a thread
t = threading.Thread(target=worker)
t.start()
t.join()  # Wait for thread to finish
# Use for I/O bound tasks like downloading files, reading sockets, etc.

'''concurrent.futures'''
# Threadpool Executor
from concurrent.futures import ThreadPoolExecutor

def square(n):
    return n * n

with ThreadPoolExecutor(max_workers=3) as executor:
    results = executor.map(square, [1, 2, 3, 4])
    print(list(results))  # [1, 4, 9, 16]

#ProcessPool Executor
from concurrent.futures import ProcessPoolExecutor

# For CPU-bound tasks
with ProcessPoolExecutor() as executor:
    results = executor.map(square, range(5))
    print(list(results))  # [0, 1, 4, 9, 16]
# ThreadPoolExecutor for I/O, ProcesPoolExecutor for CPU-intensive work

'''multiprocessing'''
from multiprocessing import Process

def compute():
    print("Running in a separate process")

if __name__ == "__main__":
    p = Process(target=compute)
    p.start()
    p.join()
#This runs in a separate Python process - True parallel execution.
```

```
'''asyncio'''
import asyncio

async def task(name):
    print(f"{name} started")
    await asyncio.sleep(1)
    print(f"{name} done")

async def main():
    await asyncio.gather(task("Task A"), task("Task B"))

asyncio.run(main())
# Great for async web requests, socker programming or streaming.

'''queue'''
import queue
import threading
import time

q = queue.Queue()

def producer():
    for i in range(5):
        print(f"Producing {i}")
        q.put(i)
        time.sleep(0.5)

def consumer():
    while True:
        item = q.get()
        print(f"Consuming {item}")
        q.task_done()

# Start threads
threading.Thread(target=producer).start()
threading.Thread(target=consumer, daemon=True).start()

q.join()  # Wait for all items to be processed
# Safely shares data between threads. Use for producer-consumer patterns.
```

Testing and Debugging

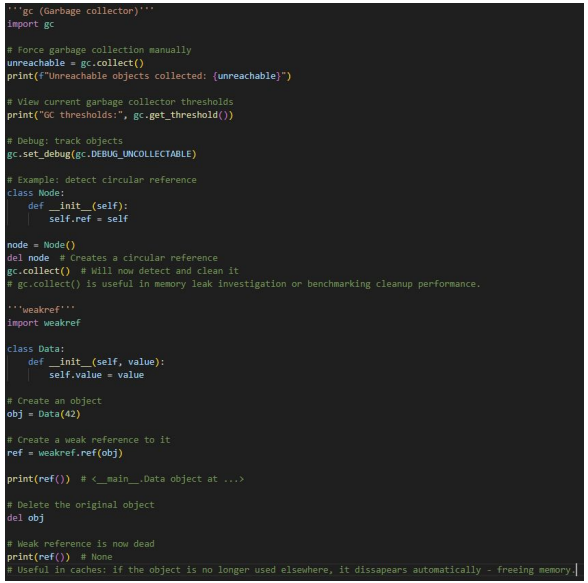
1. **unittest - Provides a framework for writing and running structured test cases.**
   - Python's built-in testing framework (inspired by JUnit).
   - Supports setup/teardown, assertions, test discovery.
2. **doctest - Validates code examples embedded in docstrings.**
   - Finds and runs examples embedded in docstrings.
   - Treats code in documentation as actual testable Python.
3. **trace - Traces program execution and tracks line-by-line coverage.**
   - Tracks and reports Python program execution.
   - Useful for code coverage and debugging.
4. **pdb - Enables interactive step-by-step debugging in the console.**
   - Interactive debugger (step-by-step execution, breakpoints).
   - Text-based – ideal for terminal usage.
5. **inspect - Retrieves metadata like signatures and source code of Python objects.**
   - Introspects functions, classes, modules.
   - Use to get function signatures, source code, argument names, etc.
6. **traceback - Prints and formats stack traces for debugging and error reporting.**
   - Extracts, formats and prints stack traces.
   - Use in logging or custom error handling.
7. **cProfile, profile, pstats - Profiles execution time of functions to find performance bottlenecks / Sorts and analyzes output from Python profiling tools.**
   - Profiling tools to measure performance (time per function call).
   - Use pstats to sort and filter profile reports.
   - Helps identify bottlenecks and optimize code.

```
'''unittest'''
import unittest

# Function to test
def add(a, b):
    return a + b

# Test case
class TestMath(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 3), 5)

# Run tests if this file is executed
if __name__ == '__main__':
    unittest.main()
# Run with python test_file.py or python -m unittest tests.py

'''doctest'''
def multiply(x, y):
    """
    Multiplies two numbers.

    >>> multiply(2, 3)
    6
    >>> multiply(-1, 5)
    -5
    """
    return x * y

if __name__ == "__main__":
    import doctest
    doctest.testmod()
# Automatically checks that the docstring examples run and return expected values.

'''trace'''
import trace

# Trace execution of a script
tracer = trace.Trace(trace=True, count=False)
tracer.run('print("Tracing this line")')
# Outputs each line of code as it executes -- useful for debugging flow.

'''pdb'''
import pdb

def calculate():
    a = 10
    b = 5
    pdb.set_trace()  # Enter interactive debugging
    c = a + b
    print(c)

calculate()
# When the debugger pauses, you can type commands.
```

```
'''inspect'''
import inspect

def greet(name: str) -> str:
    """Greets the user."""
    return f"Hello, {name}"

# Get function signature
print(inspect.signature(greet))  # (name: str) -> str

# Get source code
print(inspect.getsource(greet))
# Helps with dynamic introspection, decorators and debugging.

'''traceback'''
import traceback

try:
    1 / 0
except Exception as e:
    print("Error captured:")
    traceback.print_exc()
# Outputs full stack trace - often used in logging for detailed crash reports.


'''cProfile, profile, pstats'''
import cProfile

def run():
    total = 0
    for i in range(1000):
        total += i ** 2
    return total

cProfile.run('run()')

import pstats

# Load and analyze a saved profile
stats = pstats.Stats('profile_output.prof')
stats.sort_stats('time').print_stats(10)
# Use cProfile.run('run()', 'profile_output.prof') to save profile data.
```

Memory and Garbage Collection

1. **gc - Controls and monitors Python's garbage collector for memory management.**
    - Interface to the **garbage collector** – which reclaims unused memory.
    - Python uses reference counting and cyclic GC.
    - You can enable/disable automatic collection manually.
    - Useful for debugging memory leaks or tuning performance.
2. **weakref - References objects without preventing their garbage collection.**
    - Allows referencing an object **without increasing its reference count.**
    - Useful for caches, observers or memory-sensitive mappings.
    - When the object is deleted, the weak reference return None.
    - Avoids memory leaks in certain data structures.



```
'''gc (Garbage collector)'''
import gc

# Force garbage collection manually
unreachable = gc.collect()
print(f"Unreachable objects collected: {unreachable}")

# View current garbage collector thresholds
print("GC thresholds:", gc.get_threshold())

# Debug: track objects
gc.set_debug(gc.DEBUG_UNCOLLECTABLE)

# Example: detect circular reference
class Node:
    def __init__(self):
        self.ref = self

node = Node()
del node  # Creates a circular reference
gc.collect()  # Will now detect and clean it
# gc.collect() is useful in memory leak investigation or benchmarking cleanup performance.

'''weakref'''
import weakref

class Data:
    def __init__(self, value):
        self.value = value

# Create an object
obj = Data(42)

# Create a weak reference to it
ref = weakref.ref(obj)

print(ref())  # <__main__.Data object at ...>

# Delete the original object
del obj

# Weak reference is now dead
print(ref())  # None
# Useful in caches: if the object is no longer used elsewhere, it dissapears automatically - freeing memory
```

Glossary

- **csv** – Reads and writes tabular data using the CSV format.
- **sqlite3** – Embeds a lightweight SQL database engine for structured data.
- **zipfile** – Reads and writes .zip archive files.
- **tarfile** – Handles .tar archives with optional compression (.gz, .bz2, etc.).
- **zlib** – Compresses and decompresses binary data using DEFLATE algorithm.
- **bz2** – Compresses data using the Bzip2 compression algorithm.
- **lzma** – Provides LZMA/XZ compression for high compression ratios.
  hashlib – Generates hash digests (e.g., SHA-256) for data verification.
- **hmac** – Computes secure message digests using a secret key and hash function.
- **secrets** – Generates cryptographically secure random numbers and tokens.
- **sys** – Provides system-specific functionality like argument handling and path control.
- **argparse** – Parses command-line arguments and auto-generates help messages.
- **contextlib** – Simplifies creation and management of context managers.
- **abc** – Defines abstract base classes and enforces method implementation.
- **typing** – Supports type hinting and static type checking.
- **dataclasses** – Reduces boilerplate for data-holding classes using a decorator.
- **warnings** – Issues runtime warnings for deprecated or risky behavior.
- **gc** – Interfaces with Python's garbage collector for memory management.
- **weakref** – Creates references to objects that don't prevent garbage collection.
- **importlib** – Dynamically imports and reloads modules at runtime.
- **locale** – Formats numbers, dates, and text according to local conventions.
- **gettext** – Adds internationalization/localization support to applications.
- **codecs** – Handles encoding and decoding text in various formats (e.g., UTF-8, UTF-16).
  threading – Runs code concurrently using threads (best for I/O-bound tasks).
- **concurrent.futures** – Provides a high-level API for thread/process pools.
- **multiprocessing** – Executes code in separate processes for real parallelism.
- **asyncio** – Manages asynchronous I/O with async/await coroutines.
- **queue** – Provides thread-safe FIFO, LIFO, and priority queues.
- **unittest** – Organizes and runs structured unit tests using a test framework.
- **doctest** – Tests code examples embedded in docstrings.
- **trace** – Tracks execution and line coverage of code.
- **pdb** – Enables interactive debugging with step-by-step execution.
- **inspect** – Retrieves information about functions, classes, and modules.
- **traceback** – Prints and formats exception tracebacks.
- **cProfile / profile** – Profiles code to measure execution time of function calls.
- **pstats** – Analyzes and sorts profiling results from profile or cProfile.
- **datetime** – Manages dates, times, and timedeltas with formatting and parsing support.
- **timeit** – Measures execution time of small code snippets accurately.

Learn More & Explore

- ***Official Documentation***
  - https://docs.python.org/3/library/ the official and most comprehensive resource. Includes usage, examples, and API references for every module.
- ***Interactive Learning Platforms***
  - https://pythontutor.com/ – Vizualizez code execution step-by-step, great for understanding control flow, data structures and recursion.
- ***Tutorials & Guides***
  - https://www.geeksforgeeks.org/python-modules/ – Explains common modules with examples and problems to practice.
- ***Community Support***
  - https://stackoverflow.com/questions/tagged/python – Ask and search thousands of questions related to any standard module
  - https://www.reddit.com/r/learnpython/ – Community-driven Q&A advice and code review for learners and devs.


THANK YOU FOR WATCHING
MY PRESENTATION!
I HOPE YOU LIKED IT :D


THANKS FOR WATCHING
OUR PRESENTATION