Time module in python

The `time` module in Python provides various time-related functions. It allows you to work with time in terms of **seconds**, **delays**, and **date/time formatting**.

Commonly Used Functions in `time` Module:

Function	Description
`time.time()`	Returns the current time in seconds since the epoch (Unix time)
`time.sleep(seconds)`	Suspends execution for the given number of seconds
`time.localtime([secs])`	Converts seconds since epoch to a time tuple in local time
`time.gmtime([secs])`	Converts seconds since epoch to UTC time
`time.strftime(format,t)`	Converts a time tuple to a string as per format

Example Usage:

```
import time

# Current time in seconds
print("Epoch time:", time.time())

# Delay for 2 seconds
print("Wait for 2 seconds...")
time.sleep(2)
print("Resumed!")

# Local time
local = time.localtime()
print("Local time tuple:", local)
print("Formatted:", time.strftime("%Y-%m-%d %H:%M:%S", local))

# Convert string to time
time_str = "2025-07-16 11:00:00"
parsed = time.strptime(time_str, "%Y-%m-%d %H:%M:%S")
print("Parsed time tuple:", parsed)

# Readable time from timestamp
print("CTime:", time.ctime())
```

Notes:

- **Epoch**: Time counted from 00:00:00 UTC on 1 January 1970.
- `sleep()` is useful for **delays**, animations, and **rate limiting**.
- `strftime()` and `strptime()` use formatting codes like:
    - `%Y` = Year, `%m` = Month, `%d` = Day
    - `%H` = Hour, `%M` = Minute, `%S` = Second

Command-Line Utility in python

In Python, a **Command-Line Utility** is a script or program that runs from the terminal/command prompt and accepts **arguments** and **options** like a regular system command.

Benefits:

- Automates tasks (e.g., file renaming, data processing)
- Can be packaged and installed like Linux commands.
- Works well for scripting and devops.

Simple CLI Utility Using `argparse`

We'll make a utility that **adds or subtracts two numbers** via command line.

Step 1: Code (*calc.py*)

```
import argparse

# Create parser
parser = argparse.ArgumentParser(description="Simple
calculator utility")

# Add arguments
parser.add_argument("num1", type=float, help="First number")
parser.add_argument("num2", type=float, help="Second number")
parser.add_argument("--operation", "-o", choices=["add",
"sub"], default="add", help="Operation to perform")

# Parse arguments
args = parser.parse_args()

# Perform operation
if args.operation == "add":
    result = args.num1 + args.num2
else:
    result = args.num1 - args.num2

print("Result:", result)
```

Step 2: Run in Command Line

```
python calc.py 10 5 --operation add
# Output: Result: 15.0

python calc.py 10 5 -o sub
# Output: Result: 5.0
```

Notes:

- argparse.ArgumentParser() sets up help, usage, etc.
- --operation is an **optional argument** with a default.

- You can add more features like multiplication, logging, file inputs, etc.

Walrus operator in python

The **Walrus Operator (:=)** in Python is used for **assignment expressions**. It was introduced in **Python 3.8**.

It lets you **assign a value to a variable as part of an expression** — like inside an `if`, `while`, list comprehension, etc.

Example 1: Traditional vs Walrus

Without Walrus

```
value = input("Enter something: ")
if value != "":
    print("You entered:", value)
```

With Walrus

```
if (value := input("Enter something: ")) != "":
    print("You entered:", value)
```

Example 2: **`while`** loop

Old Way

```
line = input("Enter text (blank to stop): ")
while line != "":
    print("Line:", line)
    line = input("Enter text (blank to stop): ")
```

With Walrus

```
while (line := input("Enter text (blank to stop): ")) != "":
    print("Line:", line)
```

Caution:

- Works only in Python **3.8+**
- Avoid **overusing** it where it reduces code readability
- Don't use it for **multi-variable assignments**

When to Use:

- Inside `if` / `while` conditions to avoid repetition
- In comprehensions where filtering and assigning are both needed
- When you want to keep code **clean and DRY**

Shutil module in python

The `shutil` module in Python provides **high-level file operations**, such as copying, moving, archiving, and removing files or directories.
It's part of the **standard library**, so you don't need to install anything.

Common Functions in shutil Module:

Function	Description
shutil.copy(src, dst)	Copies a file (only contents, not metadata)
shutil.copy2(src, dst)	Like copy but also copies metadata
shutil.copytree(src, dst)	Recursively copies an entire directory tree
shutil.move(src, dst)	Moves a file or directory
shutil.rmtree(path)	Recursively deletes a directory tree

Example Code:

```
import shutil
import os

# Copy a file
shutil.copy("source.txt", "backup.txt")

# Copy file with metadata
shutil.copy2("source.txt", "backup_with_meta.txt")

# Move a file
shutil.move("backup.txt", "folder1/backup.txt")

# Delete a directory
shutil.rmtree("old_folder")

# Create a ZIP archive of a folder
shutil.make_archive("my_backup", "zip", "my_folder")

# Extract archive
shutil.unpack_archive("my_backup.zip", "restored_folder")

# Get disk usage
usage = shutil.disk_usage("/")
print(f"Total: {usage.total}, Used: {usage.used}, Free:
{usage.free}")
```

Tips:

- copy() works only on **files**, not directories.
- Be careful with rmtree() — it deletes everything without confirmation!
- make_archive() can save in formats: "zip", "tar", etc.

Python random Module

The random module in Python is used to generate **random numbers** and perform **random operations**, such as selecting random elements from a list, shuffling, or generating random data for simulations and games.

You need to import it before using:

```
import random
```

Common Functions in `random` Module

Function	**Description**	**Example**
`random.random()`	Returns a random float number between **0.0 and 1.0**	`random.random()` → 0.7345
`random.randint(a, b)`	Returns a random **integer** between **a and b** (both included)	`random.randint(1, 10)` → 7
`random.randrange(start, stop[, step])`	Returns a random number from a given range (like `range()`)	`random.randrange(0, 10, 2)` → 8
`random.choice(sequence)`	Returns a **random element** from a list, tuple, or string	`random.choice(['apple', 'banana', 'cherry'])` → `'banana'`
`random.choices(sequence, k=n)`	Returns a **list of n random elements** (with replacement)	`random.choices([1, 2, 3], k=2)` → `[2, 2]`
`random.sample(sequence, k=n)`	Returns **n unique random elements** (without replacement)	`random.sample([1, 2, 3, 4], k=2)` → `[3, 1]`
`random.shuffle(list)`	**Shuffles** the elements of a list in place (changes original order)	`random.shuffle(my_list)`

1. Random integer

```
import random
num = random.randint(1, 100)
print(num)
```

2. Random choice from list

```
fruits = ['apple', 'banana', 'mango', 'cherry']
print(random.choice(fruits))
```

3. Shuffle list

```
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)
```

4. random.randrange(start, stop, step)

```
print(random.randrange(0, 10, 2))  # even numbers between 0-10
```

What is a Generator

A **generator** is a function that uses the `yield` keyword instead of `return`. It "yields" values one at a time and **pauses** after each `yield`, resuming from where it left off.

Example of a Generator Function

```
def count_up_to(n):
    i = 1
    while i <= n:
        yield i
        i += 1

gen = count_up_to(5)
for number in gen:
    print(number)
```

Output:

```
1
2
3
4
5
```

How is it Different from return?

return	**yield**
Ends the function completely	Pauses and saves the state
Returns a single value	Returns a generator object
Cannot be used for iteration	Can be iterated (like using `for`)

Generator vs List Example

```
# List version (stores all numbers)
def squares_list(n):
    return [i*i for i in range(n)]

# Generator version (generates one at a time)
```

```
def squares_gen(n):
    for i in range(n):
        yield i*i
print(squares_list(5))       # [0, 1, 4, 9, 16]
print(list(squares_gen(5))) # [0, 1, 4, 9, 16]
```

Benefits of Generators

- **Memory-efficient**: Doesn't load everything in memory
- **Lazy evaluation**: Values generated on the fly
- **Useful for streaming data or infinite sequences**

Important Notes

- You can iterate only once over a generator
- Use next(generator) to manually get the next value
- Raise Stop Iteration automatically when done

What is Function Caching?

Save the result of a function when it's called with a certain input, and if it's called again with the same input, return the saved result instead of calculating again.

- Save time
- Improve performance
- Avoid repeating heavy calculations

Real-Life Analogy

Imagine you ask your friend:
"What is 5 squared?" — He calculates and says **25**.

Now, if you ask him again:
"What is 5 squared?" — He says:
"I already told you — it's 25!" — No need to calculate again.

How to Do Function Caching in Python?

Python provides a **built-in decorator** called `lru_cache` from the `functools` module.

```
from functools import lru_cache
import time

@lru_cache(maxsize=None)  # No limit on number of cached calls
def square(n):
    time.sleep(4)
    print(f"Calculating square of {n}")
    return n * n
```

Example Usage:

```
print(square(5))  # First time — calculates and prints
print(square(5))  # Second time — uses cache, no calculation
```

Output:

```
Calculating square of 5
25
25
```

Notice: “Calculating…” only prints once — because second time it’s using **cached result**.

Regular Expression

A **Regular Expression** (or **RegEx**) is a pattern used to **search, match, or replace** strings.

Think of it as a **smart search tool** that can find patterns — like phone numbers, emails, dates, or specific words in a string.

Python’s RegEx Module: `re`

To use regular expressions in Python, you must import the built-in `re` module:

```
import re
```

Basic Functions in `re` Module

Function	**Description**
`re.search()`	Search for the pattern in the string
`re.match()`	Match pattern **only at the beginning**
`re.findall()`	Find **all matches** and return a list
`re.sub()`	Replace matched patterns with something
`re.compile()`	Compile a pattern for reuse

Example 1: `re.search()`

```
import re

text = "My phone number is 123-456-7890"
result = re.search(r'\d{3}-\d{3}-\d{4}', text)
print(result.group())  # Output: 123-456-7890
```

`\d` means digit, `{3}` means exactly 3 times

RegEx Patterns (Cheat Sheet)

Symbol	**Meaning**	**Example**	**Matches**
`.`	Any character	`a.c`	`abc, a1c`
`\d`	Any digit (0-9)	`\d\d`	`12, 45`
`\D`	Non-digit	`\D+`	`abc, -+`
`\w`	Word character (a-z, A-Z, 0-9, _)	`\w+`	`hello, abc123`

Symbol	Meaning	Example	Matches
\s	Whitespace	\s+	space, tab, newline
^	Starts with	^Hi	Hi there
$	Ends with	end$	The end
[]	Set of characters	[aeiou]	a, e, i...
`	`	OR operator	`cat
*	0 or more	lo*	l, lo, loo...
+	1 or more	lo+	lo, loo
{n}	Exactly n times	\d{3}	123, 456

Example 2: re.findall()

```
text = "My numbers are 123 and 456"
matches = re.findall(r'\d+', text)
print(matches)  # ['123', '456']
```

Example 3: re.sub() for Replace:

```
text = "My email is test123@gmail.com"
updated = re.sub(r'\w+@\w+\.\w+', 'hidden@email.com', text)
print(updated)
```

Example 4: re.match() vs re.search():

```
re.match(r'Hello', 'Hello World')   # Match at beginning
re.search(r'World', 'Hello World')  # Found anywhere
```

Using re.compile() (for reuse):

```
pattern = re.compile(r'\d+')
print(pattern.findall("I have 2 apples and 10 bananas"))
```

Use Cases of RegEx:

- Validating email or phone number
- Extracting data from text or HTML
- Replacing sensitive info (e.g., mask Aadhaar)
- Log file analysis, scraping, etc.

Quick Email Matching Example

```
email = "My email is neeraj.kath@gmail.com"
match = re.search(r'[a-zA-Z0-9._]+@[a-z]+\.[a-z]+', email)
print(match.group())  # neeraj.kath@gmail.com
```

What is asyncio

asyncio is a **Python module for writing asynchronous programs** — especially useful when you're doing I/O-bound tasks like:

- Reading/writing files
- Making API calls

- Waiting for a timer
- Network/socket programming

Instead of **waiting** for a task to finish, it **lets other tasks run** — increasing efficiency.

Real-Life Analogy

Imagine you're cooking 3 dishes:

- One needs boiling water
- One needs frying
- One needs baking

Instead of waiting for each step to finish, you:

- Start boiling → switch to frying → switch to baking → and rotate between them.

That’s **asynchronous multitasking**.

Traditional (Synchronous) vs Async

Synchronous Code:

```
import time

def task1():
    time.sleep(2)
    print("Task 1 done")

def task2():
    time.sleep(2)
    print("Task 2 done")

task1()
task2()
```

Total time: 4 seconds

Asynchronous Code using `asyncio`:

```
import asyncio

async def task1():
    await asyncio.sleep(2)
    print("Task 1 done")

async def task2():
    await asyncio.sleep(2)
    print("Task 2 done")

async def main():
    await asyncio.gather(task1(), task2())

asyncio.run(main())
```

Total time: 2 seconds (both tasks sleep *in parallel*)

Key Keywords and Concepts

Keyword	Meaning
`async def`	Defines an **asynchronous function**
`await`	Tells Python to **pause here** and run something else
`asyncio.run()`	Runs the top-level coroutine
`asyncio.sleep(n)`	Non-blocking sleep (like `time.sleep` but async)
`asyncio.gather()`	Run multiple async tasks together

Real-World Example: Simulate API Calls

```
import asyncio

async def fetch_data(site):
    print(f"Fetching from {site}")
    await asyncio.sleep(1)  # simulate network delay
    print(f"Done fetching {site}")

async def main():
    sites = ['Google', 'YouTube', 'Facebook']
    tasks = [fetch_data(site) for site in sites]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

Output:

```
Fetching from Google
Fetching from YouTube
Fetching from Facebook
Done fetching Google
Done fetching YouTube
Done fetching Facebook
```

Runs **in parallel**, total time ≈ 1 second.

Async vs Threading

Feature	Asyncio	Threading
Use case	I/O-bound	CPU or I/O-bound
Memory	Lightweight	Heavier
Speed	Faster switching	Slower context switching
Complexity	Cleaner, scalable	Needs locks and care

When to Use Asyncio?

- Calling APIs or doing network operations
- Waiting for user input
- Reading/writing files (non-blocking libs like aiofiles)
- Building fast web apps (like FastAPI)

Multithreading in Python

Multithreading is a technique where **multiple threads** run **concurrently** within a single process. It helps in performing **multiple tasks simultaneously**, especially useful in **I/O-bound** operations.

For example:

- Reading/writing files while handling user input.
- Downloading multiple files at the same time.

Key Concepts:

Term	**Explanation**
Thread	A lightweight unit of a process that can run in parallel.
Main Thread	The thread in which the Python program starts.
Child Thread	Threads created by the main thread for other tasks.

How to Use Multithreading in Python

Python provides a built-in `threading` module.

Example:

```
import threading
import time

def display():
    for i in range(5):
        print(f"Thread running: {i}")
        time.sleep(1)

# Creating thread
t = threading.Thread(target=display)

# Starting thread
t.start()

# Main thread continues
for i in range(5):
    print(f"Main thread: {i}")
    time.sleep(1)
```

Important Methods:

Method	**Description**
`start()`	Starts the thread.
`join()`	Waits for the thread to complete.
`is_alive()`	Checks if thread is still running.

Why Not for CPU-bound Tasks?

Due to **GIL (Global Interpreter Lock)** in Python, only one thread executes at a time in CPU-bound operations. So, for **CPU-bound** tasks, prefer **multiprocessing** instead of multithreading.

Use Cases:

- Web scraping multiple sites at once.
- File downloads.
- Concurrent I/O operations (API calls, file reads).

What is Multiprocessing?

Multiprocessing is a technique where **multiple processes** run in parallel, each with its own Python interpreter and memory space.
It is used to **bypass the Global Interpreter Lock (GIL)** and is ideal for **CPU-bound tasks** like:

- Image processing
- Heavy calculations
- Data analysis

Key Concepts:

Term	**Description**
Process	An independent unit of execution with its own memory.
Main Process	The process in which Python script starts.
Child Process	New processes started by the main process using `multiprocessing` module.

Example:

```
from multiprocessing import Process

def calculate_square(numbers):
    print("Squares:")
    for n in numbers:
        print(f"{n}^2 = {n*n}")

def calculate_cube(numbers):
    print("Cubes:")
    for n in numbers:
        print(f"{n}^3 = {n*n*n}")
```

```
if __name__ == "__main__":
    nums = [2, 3, 4, 5]

    # Create processes
    p1 = Process(target=calculate_square, args=(nums,))
    p2 = Process(target=calculate_cube, args=(nums,))

    # Start processes
    p1.start()
    p2.start()

    # Wait for processes to finish
    p1.join()
    p2.join()

    print("Done with multiprocessing!")
```

Output (order may vary):

```
Squares:
2^2 = 4
3^2 = 9
4^2 = 16
5^2 = 25

Cubes:
2^3 = 8
3^3 = 27
4^3 = 64
5^3 = 125
Done with multiprocessing!
```

Important Methods:

Method	**Description**
`start()`	Starts the process.
`join()`	Waits for the process to finish.
`is_alive()`	Checks if the process is still running.

Comparison: Multithreading vs Multiprocessing:

Feature	**Multithreading**	**Multiprocessing**
Ideal for	I/O-bound tasks	CPU-bound tasks

Feature	**Multithreading**	**Multiprocessing**
Memory	Shared	Separate memory space
Speed	Slower for CPU tasks	Faster for CPU tasks
GIL Affected?	Yes	No

Additional Concepts:

Process Pooling (for multiple tasks):

```
from multiprocessing import Pool

def square(n):
    return n * n

with Pool(processes=4) as pool:
    result = pool.map(square, [1, 2, 3, 4])
    print(result)  # Output: [1, 4, 9, 16]
```

Sharing Data (Optional):
Use `Value`, `Array`, or `Queue` from `multiprocessing` to share data between processes.

Summary:

Feature	**Details**
Module	`multiprocessing`
Best for	CPU-bound tasks
Independent memory?	Yes
GIL limitation?	No
Common tools	`Process`, `Pool`, `Queue`, `Lock`

Notes:

- `Process(target=func, args=(...))` — allows passing functions and their arguments to separate processes.
- `start()` — begins execution of each process.
- `join()` — waits for process to complete.
- Output order is **not guaranteed** as processes run **independently** and **simultaneously**.