
## Function Caching in Python:

**In Python, you can implement function caching using various techniques. Caching helps to store the results of expensive function calls and retrieve them when the same inputs occur again, thereby saving computation time. Here are a few common ways to implement function caching in Python:**

1. **Using a Dictionary: You can use a dictionary to store the results of function calls with their respective arguments as keys.**

```python
def cached_function(func):
    cache = {}

    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return wrapper

# Usage
@cached_function
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))  # This call will be cached
```

2. **Using `functools.lru_cache`: Python's standard library provides the `functools.lru_cache` decorator, which implements a least-recently-used caching mechanism.**

```python
from functools import lru_cache

@lru_cache(maxsize=None)  # None means unlimited cache size
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))  # This call will be cached
```

3. **Using Memoization: This is a technique where you explicitly store computed results in a function's closure.**

```python
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 1:
        return n
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]

print(fibonacci(10))  # This call will be cached
```

**Each of these methods has its own advantages and use cases. Choose the one that best fits your requirements.**


## CODE:

```
def cached_function(func):
    cache = {}

    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]

    return wrapper

# Example function to be cached
def add(x, y):
    print("Calculating...")
    return x + y

# Decorate the function with caching
cached_add = cached_function(add)

# Test the cached function
print(cached_add(3, 4))  # Output: Calculating... 7
print(cached_add(3, 4))  # Output: 7 (no 'Calculating...'
printed, result fetched from cache)
```

## Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\40.py
Calculating...
7
7
```


# Handling Exceptions with `try`, `except`, `else`, and `finally` in Python

**In Python, the `try`, `except`, `else`, and `finally` blocks provide a powerful mechanism for handling exceptions and executing cleanup code. Let's explore each of these blocks and their roles within exception handling.**

## 1. `try` Block:

**The `try` block is used to wrap the code that might raise an exception. Any code within this block that raises an exception will be caught by the subsequent `except` block(s).**

```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

## 2. `except` Block:

**The `except` block catches exceptions raised in the corresponding `try` block. You can specify the type of exception you want to catch, or catch all exceptions using a generic `except` block.**

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except:
    print("An error occurred!")
```


## 3. `else` Block:

**The `else` block is executed if no exceptions are raised in the `try` block. It is often used to place code that should run only if the `try` block executes successfully.**

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful. Result:", result)
```

## 4. `finally` Block:

**The `finally` block is always executed, regardless of whether an exception occurred or not. It is typically used for cleanup actions, such as closing files or releasing resources.**

```
try:
    file = open("example.txt", "r")
    # Perform operations on the file
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()  # Ensure file is always closed
```

## Using `else` and `finally` Together:


**You can combine `else` and `finally` blocks to execute cleanup code after successful execution of the `try` block.**

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Cannot divide by zero!")
else:
    print("Division successful. Result:", result)
finally:
    print("Cleaning up resources...")
```

## CODE:

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("Cannot divide by zero!")
    else:
        print("Division successful. Result:", result)
    finally:
        print("Cleaning up resources...")

# Test the divide function
divide(10, 2)
divide(10, 0)
```

## Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\40_1.py
Division successful. Result: 5.0
Cleaning up resources...
Cannot divide by zero!
Cleaning up resources...
```


## Coroutines In Python:

**Coroutines are a powerful feature in Python that allow for cooperative multitasking, enabling functions to pause execution while retaining their state and later resume from where they left off. Introduced in Python 3.5, coroutines are implemented using the `async` and `await` keywords, and they form the basis of asynchronous programming in Python.**

## Understanding Coroutines

**At their core, coroutines are functions that can pause execution at specific points using the `await` keyword and can be resumed later. Unlike traditional functions, coroutines are not executed all at once; instead, they can yield control back to the event loop, allowing other coroutines to run in the meantime.**

## Syntax of Coroutines

**To define a coroutine, you use the `async def` syntax:**

```
async def my_coroutine():
    # Coroutine body
    await some_async_function()
```

**Here, `async def` declares `my_coroutine()` as a coroutine function. Within the function body, you can use the `await` keyword to pause execution until an asynchronous operation, such as an I/O operation or another coroutine, completes.**


## Executing Coroutines

**Coroutines are typically executed within an event loop, which schedules and manages their execution. The `asyncio` module in Python provides an event loop implementation for asynchronous programming. Here's how you can execute a coroutine within an event loop:**

```
import asyncio

async def main():
    await my_coroutine()

# Create and run the event loop
asyncio.run(main())
```

**In this example, `main()` is a coroutine function that calls `my_coroutine()` using `await`. The `asyncio.run()` function is then used to run the event loop and execute the coroutine.**

## Benefits of Coroutines

- **Concurrency: Coroutines enable concurrent execution of multiple tasks within a single thread, making efficient use of system resources.**
- **Asynchronous I/O: Coroutines are well-suited for asynchronous I/O operations, such as reading from files, making network requests, or interacting with databases, without blocking the main thread.**


- **Stateful Execution: Coroutines can retain their state between successive invocations, allowing for more complex and flexible control flow compared to regular functions.**

## CODE:

```python
import asyncio

# Define a coroutine function
async def my_coroutine():
    print("Coroutine is starting...")
    # Simulate some asynchronous operation
    await asyncio.sleep(2)
    print("Coroutine is resuming after 2 seconds...")
    return "Coroutine finished"

# Create and run the event loop
async def main():
    print("Starting main function...")
    result = await my_coroutine()
    print("Result:", result)

asyncio.run(main())
```

## Output:

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\40_2.py
Starting main function...
Coroutine is starting...
Coroutine is resuming after 2 seconds...
Result: Coroutine finished
```


## OS Module:

The `os` module in Python provides a platform-independent way of interacting with the operating system. It allows you to perform various operating system-related tasks, such as navigating the file system, manipulating environment variables, and executing system commands.

### *1. File and Directory Operations*

The `os` module enables you to work with files and directories on the file system. You can create, delete, rename, and traverse directories, as well as check file properties such as existence, size, and permissions.

```
import os

# Check if a file exists
if os.path.exists("myfile.txt"):
    print("File exists")

# Create a new directory
os.mkdir("my_directory")

# List files in a directory
files = os.listdir("my_directory")
print("Files in directory:", files)
```


## 2. Environment Variables

You can access and modify environment variables using the `os` module. This allows you to interact with the environment in which your Python script is running, accessing information such as system paths, user settings, and configuration variables.

```
import os

# Get the value of an environment variable
path = os.getenv("PATH")
print("System PATH:", path)

# Set a new environment variable
os.environ["NEW_VAR"] = "value"
```

## 3. Process Management

The `os` module provides functions for interacting with processes, such as spawning new processes, retrieving process IDs, and terminating processes.

```
import os

# Execute a system command
os.system("ls -l")

# Get the process ID of the current Python process
pid = os.getpid()
print("Process ID:", pid)
```


## 4. Platform Information

You can retrieve information about the underlying operating system using the `os` module. This includes details such as the name of the operating system, the machine's hostname, and system-specific constants.

```
import os

# Get the name of the operating system
os_name = os.name
print("Operating System:", os_name)

# Get the hostname of the machine
hostname = os.uname().nodename
print("Hostname:", hostname)
```

## 5. File Path Manipulation

The `os.path` submodule provides functions for manipulating file paths in a platform-independent manner. This includes joining and splitting path components, checking file extensions, and retrieving directory names

```
import os

# Join path components
full_path = os.path.join("my_directory", "myfile.txt")
print("Full path:", full_path)

# Get the directory name from a path
dirname = os.path.dirname(full_path)
print("Directory name:", dirname)
```


**CODE:**

```
import os

# Check if a file exists
if os.path.exists("myfile.txt"):
    print("File exists")
else:
    print("File does not exist")

# Create a new directory
os.mkdir("my_directory")

# List files in a directory
files = os.listdir("my_directory")
print("Files in directory:", files)

# Get the value of an environment variable
path = os.getenv("PATH")
print("System PATH:", path)

# Set a new environment variable
os.environ["NEW_VAR"] = "value"
print("New environment variable set")

# Execute a system command
os.system("ls -l")

# Get the process ID of the current Python process
pid = os.getpid()
print("Process ID:", pid)

# Get the name of the operating system
os_name = os.name
print("Operating System:", os_name)

# Get the hostname of the machine
hostname = os.uname().nodename
```


```
print("Hostname:", hostname)

# Join path components
full_path = os.path.join("my_directory", "myfile.txt")
print("Full path:", full_path)

# Get the directory name from a path
dirname = os.path.dirname(full_path)
print("Directory name:", dirname)
```

**Output:**

```
C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts\python.exe C:\Users\saksh\PycharmProjects\pythonProject\40_3.py
File does not exist
Files in directory: []
System PATH: C:\Users\saksh\PycharmProjects\pythonProject\.venv\Scripts;C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Windows\System32\WindowsPowerSh
New environment variable set

Host Name:                 ATTACKER
OS Name:                   Microsoft Windows 11 Pro
OS Version:                10.0.22631 N/A Build 22631
OS Manufacturer:           Microsoft Corporation
OS Configuration:          Standalone Workstation
OS Build Type:             Multiprocessor Free
Registered Owner:          sakshamcore@outlook.com
Registered Organization:
```