File handling in Python

File handling in Python allows you to create, read, write, append, and delete files. Here's a step-by-step guide to each of these operations with examples and explanations.

1. Creating a File

In Python, a file is created using the `open()` function. The function takes two main arguments:

- The name of the file to open or create.
- The mode in which you want to open the file.

Common modes include:

- `"w"`: Write mode (creates a new file if it doesn't exist or truncates it if it does).
- `"x"`: Exclusive creation (fails if the file already exists).

 # Example: Creating a file using "w" mode

```
file = open("example.txt", "w")
print("File created successfully.")
file.close()  # Always close the file after use
```

In this example, `example.txt` is created. If the file already exists, it will overwrite it. `file.close()` releases the resources.

2. Writing to a File

When writing to a file, you can use `"w"` or `"a"` mode:

- `"w"`: Write mode (overwrites the file).

- `"a"`: Append mode (adds data to the end of the file without deleting existing content).

Example with `"w"` mode:

```
# Example: Writing data to a file
file = open("example.txt", "w")
file.write("Hello, this is a sample text.")
print("Data written to file.")
file.close()
```

Here, `"Hello, this is a sample text."` is written to `example.txt`. If there was any existing content, it would be replaced.

Example with `"a"` mode:

```
# Example: Appending data to a file
file = open("example.txt", "a")
file.write("\nAppending this text to the file.")
print("Data appended to file.")
file.close()
```

This appends the text `"\nAppending this text to the file."` at the end of the existing content in `example.txt`.

3. Reading a File

Reading can be done with the `"r"` mode, which is the default mode for `open()`. The following methods are used:

- `read()`: Reads the entire file content.
- `readline()`: Reads one line at a time.
- `readlines()`: Reads all lines and returns them as a list.

 # Example: Reading data from a file

```
file = open("example.txt", "r")
content = file.read()  # Reads the entire content
print("File content:")
print(content)
file.close()
```

This will display the entire content of `example.txt`.

4. Appending to a File

Appending is done with `"a"` mode. When you use `"a"`, the file pointer is positioned at the end of the file, so new data is added there without modifying existing content.

 # Example: Appending additional content to a file

```
file = open("example.txt", "a")
file.write("\nAnother line added at the end.")
print("Additional content appended to file.")
file.close()
```

After running this code, `"Another line added at the end."` will be added to the end of `example.txt`.

5. Deleting a File

To delete a file, use the `os` module. Specifically, `os.remove()` deletes a specified file.

```
import os
```

 # Example: Deleting a file

```
file_path = "example.txt"
if os.path.exists(file_path):
    os.remove(file_path)
    print(f"{file_path} has been deleted.")
else:
    print("File does not exist.")
```

This code checks if `example.txt` exists. If it does, it deletes it; otherwise, it displays a message that the file does not exist.

with Statement: The with statement is used to open the file and ensures that it's automatically closed when the block of code inside the with statement is finished, making file handling more secure and less error-prone.

with open('List.txt', 'r') as file:

Summary of File Modes:

Mode	**Description**
`"r"`	Read (default mode, file must exist)
`"w"`	Write (creates or truncates the file)
`"a"`	Append (adds data to the end if the file exists)
`"x"`	Exclusive creation (fails if file exists)

Additional Tips on File Handling

1. **Always close files:** Use `file.close()` to release resources, or use the `with` statement for automatic closing:

```
with open("example.txt", "w") as file:
    file.write("Using with statement.")
```

2. **Using exceptions:** When dealing with files, it's good practice to handle potential errors with `try-except` blocks.

With these basics, you can create, read, write, append, and delete files in Python efficiently.

Exception Handling in Python

1. What is an Exception in Python?

In Python, an **exception** is an error that occurs during the execution of a program. When Python encounters a situation it cannot handle, it raises an exception. If the exception is not handled, it stops the program and shows an error message.

For example, dividing a number by zero or trying to open a file that does not exist will raise exceptions in Python.

2. Difference Between Syntax Error and Exception

- **Syntax Error:** This occurs when the Python interpreter detects incorrect syntax in the code. It's a compile-time error, meaning it prevents the code from running. It usually indicates a typo or a fundamental mistake in how the code is written.

- **Exception**: This occurs during program execution and represents an error that the program may be able to handle. Exceptions do not prevent the program from running if they are handled correctly.

Example of Syntax Error

#Syntax Error Example

print("Hello world" **#Missing closing parenthesis**

Output:

SyntaxError: unexpected EOF while parsing

Example of Exception

#Exception Example

a = 10 / 0 **#Division by zero is an error during execution**

Output:

ZeroDivisionError: division by zero

3. Try and Except Statement – Catching Exceptions

The `try` and `except` blocks allow you to handle exceptions gracefully, which means the program can continue running even if an error occurs.

#Example of Try and Except

```
try:
    a = 10 / 0     #Attempt to divide by zero
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Explanation:

1. The code in the `try` block is attempted.

2. If an exception is raised, the code in the `except` block is executed.

3. Here, the `ZeroDivisionError` is caught, and a message is printed instead of stopping the program.

Output:

You can't divide by zero!

#Example of Try and Except

```
a = [1, 2, 3, 4, 5]
try:
    print("Second element = ", a[1])
#Throws error since there are only 3 elements in array
    print ("fourth element = ", a[4])
except:
    print ("An error occurred")
```

4. Catching Specific Exceptions

Python allows you to catch specific exceptions if you know which types of errors to expect. This makes handling errors more precise.

#Example of Catching Specific Exception

```
try:
    num = int(input("Enter a number: "))  #User inputs a value
```

```
    result = 10 / num
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Explanation:

1. If the input is not an integer, a `ValueError` is raised.
2. If the input is zero, a `ZeroDivisionError` is raised.
3. Each error type has a specific `except` block to handle it.

Output (if input is "abc"):

Please enter a valid integer.

Output (if input is "0"):

You can't divide by zero!

#Example of Catching Specific Exception

```
def fun(a):
    if a < 4:
        # throws ZeroDivisionError for a = 3
        b = a/(a-3)
    #throws NameError if a >= 4
    print("Value of b = ", b)
try:
    fun(2)
    fun(4)
except ZeroDivisionError:
    print("ZeroDivisionError Occurred and Handled")
except NameError:
    print("NameError Occurred and Handled")
```

5. Try with Else Clause

The `else` clause is useful when you want to execute some code only if no exceptions were raised in the `try` block. It works like an additional safe zone after the `try` block.

#Example of Try with Else Clause

```
try:
    num = int(input("Enter a number: "))  # User inputs a value
    result = 10 / num
```

```
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("Please enter a valid integer.")
else:
    print("The division result is:", result)
```

Explanation:

1. If no exceptions occur, the `else` block is executed.

2. This is useful when the `try` block is successful, and you want to do additional actions.

Output (if input is "5"):

The division result is: 2.0

6. Finally Keyword in Python

The `finally` block executes regardless of whether an exception was raised or not. It's typically used to release resources, close files, or clean up actions that should happen no matter the outcome.

#Example of Finally Keyword

```
try:
    file = open("example.txt", "r")  # Try to open a non-existent file
except FileNotFoundError:
    print("File not found.")
```

```
finally:
    print("Executing finally block.")
```

Explanation:

1. The `finally` block runs whether an exception occurs or not.

2. Here, even if the file is not found, the message in the `finally` block is printed.

Output:

File not found.

Executing finally block.

7. Hierarchy of Built-in Exceptions

Python's built-in exceptions follow a hierarchy, with `BaseException` at the top. This allows for structured exception handling, where specific exceptions inherit from more general ones. Here’s a simplified version:

- **BaseException**
    - **Exception**
        - **ArithmeticError**
            - **ZeroDivisionError**
        - **LookupError**

- **IndexError**
- **KeyError**

- **ValueError**
- **TypeError**
- **FileNotFoundError**

Example of Exception Hierarchy

You can catch broader exceptions like `Exception` or more specific ones like `ZeroDivisionError`.

```
try:
    result = 10 / 0
except ArithmeticError:  # Catches any arithmetic-related error
    print("An arithmetic error occurred.")
except Exception:  # Catches any general exception
    print("Some error occurred.")
```

Explanation:

1. The `ZeroDivisionError` is a type of `ArithmeticError`.
2. This code will print the message for `ArithmeticError` because `ZeroDivisionError` is part of that category.

Output:

An arithmetic error occurred.

Packages in Python

In Python, **packages** are a way of structuring your code by grouping multiple modules into directories. A package is essentially a directory that contains a special file, `__init__.py`, which allows the directory to be imported as a module. By organizing code into packages, you can make it more modular, reusable, and easier to manage, especially in larger projects.

Let’s go through basic, medium, and advanced examples to understand packages better.

1. Simple Package for Basic Operations

Let's create a package called `math_operations` that contains modules for basic math operations.

Directory Structure:

math_operations/

├── __init__.py

├── addition.py

└── subtraction.py

Code for Each Module:

1. addition.py

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

2. subtraction.py

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

3. __init__.py (empty or used to import modules for easy access)

```
from .addition import add
from .subtraction import subtract
```

Using the Package

Now, in another Python file, you can use this package:

```
from math_operations import add, subtract
print(add(5, 3))        # Output: 8
print(subtract(5, 3))   # Output: 2
```

This demonstrates how a basic package can help organize simple functions.

---

2. Package with Multiple Modules and Nested Sub-Packages

For a more complex example, let's create a package called `geometry` that has modules for `area` and `perimeter` calculations. Each module will have a sub-package for different shapes.

Directory Structure:

```
geometry/
├── __init__.py
├── area/
│   ├── __init__.py
│   ├── circle.py
│   └── rectangle.py
└── perimeter/
    ├── __init__.py
    ├── circle.py
    └── rectangle.py
```

Code for Each Module:

1. area/circle.py

```
import math
def area_circle(radius):
    return math.pi * radius ** 2
```

2. area/rectangle.py

```
def area_rectangle(length, width):
    return length * width
```

3. perimeter/circle.py

```
import math
def perimeter_circle(radius):
    return 2 * math.pi * radius
```

4. perimeter/rectangle.py

```
def perimeter_rectangle(length, width):
    return 2 * (length + width)
```

5. __init__.py files (in `geometry/area`, `geometry/perimeter`, and `geometry`)

- geometry/area/__init__.py

  from .circle import area_circle

  from .rectangle import area_rectangle

- geometry/perimeter/__init__.py

  from .circle import perimeter_circle

  from .rectangle import perimeter_rectangle

- geometry/__init__.py

  from .area import area_circle, area_rectangle

  from .perimeter import perimeter_circle, perimeter_rectangle

Using the Package

You can now use this package as follows:

```
from geometry import area_circle, area_rectangle,
perimeter_circle, perimeter_rectangle

print(area_circle(5))

# Output: area of a circle with radius 5

print(area_rectangle(4, 6))

# Output: area of a rectangle 4x6

print(perimeter_circle(5))

# Output: perimeter of a circle with radius 5

print(perimeter_rectangle(4, 6))

# Output: perimeter of a rectangle 4x6.
```

Interfaces in Python

In Python, interfaces are abstract contracts that define a set of methods that a class must implement.

Python does not have a built-in interface keyword like Java, but it achieves similar functionality using abstract base classes (ABCs) from the `abc` module.

Key Features of Interfaces in Python:

1. An interface specifies method signatures but does not implement them.

2. A class implementing an interface must provide concrete implementations for all methods defined in the interface.

3. Python's `abc` module allows us to create interfaces using the `ABC` class and the `@abstractmethod` decorator.

Example: Interface in Python

Suppose we are building a payment processing system. We'll define an interface `PaymentProcessor` that outlines the required methods, and then implement it in multiple payment classes.

Step 1: Define the Interface

```
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount: float):
        """Process the payment of a given amount."""
        pass
```

- `PaymentProcessor` is an abstract base class.

- `@abstractmethod` marks `process_payment` as a method that subclasses must implement.

Step 2: Implement the Interface

```
class CreditCardPayment(PaymentProcessor):
    def process_payment(self, amount: float):
        print(f"Processing credit card payment of ${amount:.2f}")


class PayPalPayment(PaymentProcessor):
```

```
    def process_payment(self, amount: float):
        print(f"Processing PayPal payment of ${amount:.2f}")
```

- Both `CreditCardPayment` and `PayPalPayment` provide their own implementation of the `process_payment` method.
- If a subclass does not implement `process_payment`, Python will raise a `TypeError`.

Step 3: Use the Interface

```
def make_payment(processor: PaymentProcessor, amount: float):
    processor.process_payment(amount)

# Instantiate the payment processors
credit_card = CreditCardPayment()
paypal = PayPalPayment()

# Use them
make_payment(credit_card, 100.00)  # Processing credit card payment of $100.00
make_payment(paypal, 200.00)       # Processing PayPal payment of $200.00
```

Explanation:

1. Interface Definition: The `PaymentProcessor` interface ensures all payment classes have a `process_payment` method.

2. Implementation: Subclasses like `CreditCardPayment` and `PayPalPayment` implement the interface's method.

3. Polymorphism: The `make_payment` function works with any `PaymentProcessor` subclass, enabling flexible and scalable code.

Benefits of Interfaces:

- Encapsulation: Implementation details are hidden; only the contract is exposed.

- Flexibility: Makes it easier to swap out implementations without affecting other parts of the code.

- Code Maintainability: Promotes consistent structure across similar classes.

Threads in Python

- A thread is a separate path of execution that runs concurrently within a program.

- Python uses the threading module to work with threads, allowing developers to write concurrent code.

- Threads share the same memory space, making them lightweight compared to processes.

- They are suitable for I/O-bound tasks, such as file operations or network communications, but Python’s **Global Interpreter Lock (GIL)** limits their effectiveness for CPU-bound tasks.

Difference Between Multithreading and Multitasking

Aspect	Multithreading	Multitasking
Definition	Executing multiple threads within a single process.	Executing multiple independent processes simultaneously.
Memory Usage	Threads share the same memory space.	Processes have separate memory spaces.
Performance	Better for I/O-bound tasks.	Suitable for CPU-bound tasks.
Context Switching	Faster due to shared memory.	Slower because of separate memory allocations.
Overhead	Lightweight as threads use shared resources.	Higher overhead due to process management.

Thread Objects in Python

In Python, the threading.Thread class is used to create and manage threads.

Some key attributes and methods include:

Method/Attribute	Description
`start()`	Starts the thread and invokes the `run()` method.
`run()`	Defines the activity for the thread.
`join()`	Blocks the calling thread until the thread completes.
`is_alive()`	Returns `True` if the thread is still running.
`name`	Gets or sets the name of the thread.
`daemon`	Specifies if the thread runs in the background and stops when the main thread exits.

Functions and Constructor in the Thread class

Thread class Constructor

Following is the basic syntax of the Thread class constructor:

Thread(group=None, target=None, name=None, args=(), kwargs={})

The constructor allows many arguments, some of which are required while some are not. Let's see what they are:

- **group:** Should be **None**. It is reserved for future extension.
- **target:** This is the callable object or task to be invoked by the run() method.
- **name:** This is used to specify the thread name. By default, a unique name is generated following the format **Thread-N**, where **N** is a small decimal number.
- **args:** This is the argument **tuple** for the target invocation. We can provide values in it which can be used in the traget method. It's default value is empty, i.e. ()
- **kwargs:** This is keyword argument **dictionary** for the target invocation. This defaults to {}.

Ex 1 - Single Thread:

```
import threading

def print_hello():
    for i in range(3):
        print(f"Hello from thread{threading.current_thread().name}")
```

 # Create a thread

thread = threading.Thread(target=print_hello, name="HelloThread")

 # Start the thread

thread.start()

 # Wait for the thread to finish

thread.join()

print("Main thread execution completed.")

Explanation

1. Define a function **print_hello** to print a message.

2. Create a thread using **threading.Thread**, setting **target=print_hello** and giving it a name.

3. Start the thread with **start()** to execute the function concurrently.

4. Use **join()** to wait for the thread to complete.

5. The main thread resumes execution after the child thread finishes.

Output

Hello from thread HelloThread

Hello from thread HelloThread

Hello from thread HelloThread

Main thread execution completed.

Key Terms Recap

1. **Thread**: A small unit of a process that runs concurrently.
2. `threading` **module**: Provides tools to create and manage threads in Python.
3. `Thread` **class**: Represents an individual thread.
4. `start()`: Starts the thread and runs its target function.
5. `join()`: Waits for the thread to complete before continuing.
6. **Main thread**: The default thread where the program starts execution.
7. **F-string**: A way to format strings with embedded variables or expressions.

Ex – 1 Multiple Threads:

```
import threading
import time
def print_numbers(thread_name, count):
    for i in range(count):
        time.sleep(0.5)  # Simulate some work
        print(f"{thread_name} prints {i}")
```

```
# Create multiple threads
thread1 = threading.Thread(target=print_numbers, args=("Thread1", 5))
thread2 = threading.Thread(target=print_numbers, args=("Thread2", 3))

# Start threads
thread1.start()
thread2.start()

# Wait for both threads to complete
thread1.join()
thread2.join()

print("All threads completed.")
```

Explanation

1. Define a function `print_numbers` to print a sequence of numbers.

2. Create two threads with different arguments (`Thread1` and `Thread2`).

3. Start both threads using `start()`.

4. Use `join()` to ensure the main thread waits for both threads to finish.

5. Observe interleaved outputs due to concurrent execution.

Output

Thread1 prints 0

Thread2 prints 0

Thread1 prints 1

Thread2 prints 1

Thread1 prints 2

Thread2 prints 2

Thread1 prints 3

Thread1 prints 4

All threads completed.

Ex 2:

```
import threading # import the thread module
import time # import time module
def cal_sqre(num):# define the cal_sqre() function
    print(" Calculate the square root of the given number")
    for n in num:
```

```
        time.sleep(0.3)# at each iteration it waits for 0.3 time
        print(' Square is : ',n*n)

def cal_cube(num):# define the cal_cube() function
    print(" Calculate the cube of  the given number")
    for n in num:
        time.sleep(0.3)# at each iteration it waits for 0.3 time
        print(" Cube is : ",n*n*n)

arr = [4,5,6,7,2]# given array

t1=time.time()# get total time to execute the functions
cal_sqre(arr)# call cal_sqre() function
cal_cube(arr)# call cal_cube() function

print("Total time taken by threads is:",time.time()-
t1)# print the total time
```

Ex – 1 Thread Synchronization with Locks:

- **threading Lock -** Allow only one thread at a time to access a resource

```
import threading
class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()
    def increment(self):
        with self.lock:  # Acquire the lock
            local_value = self.value
            local_value += 1
            self.value = local_value
            print(f"Value after increment: {self.value}")
# Shared counter object
counter = Counter()
# Function for threads to increment the counter
def thread_task():
    for _ in range(3):
        counter.increment()
```

 # Create multiple threads

threads = [threading.Thread(target=thread_task) for _ in range(3)]

 # Start all threads

for thread in threads:

    thread.start()

 # Wait for all threads to complete

for thread in threads:

    thread.join()

print(f"Final counter value: {counter.value}")

Explanation

1. Define a `Counter` class with a `lock` to manage synchronized access to the shared counter.

2. The `increment` method acquires the lock using `with self.lock` to ensure only one thread modifies `self.value` at a time.

3. Create three threads to call `increment` concurrently.

4. Start and join all threads to ensure synchronization.

5. Without the lock, race conditions could cause incorrect counter values.

Output

Value after increment: 1

Value after increment: 2

Value after increment: 3

Value after increment: 4

Value after increment: 5

Value after increment: 6

Final counter value: 6

Key Takeaways

1. Threading is essential for concurrent programming in Python, especially for I/O-bound tasks.

2. Thread objects like `start()`, `join()`, and `Lock` ensure effective thread management and synchronization.

3. Use locks to prevent race conditions when multiple threads access shared resources.