Python Unit-2

What is a file?

- Files are named locations on disk to store information
- They are used to store data permanently.
- Data is stored in non-volatile memory.
- We can retrieve data whenever required.

Types of files?

1. **Text files:-**
    - Stores data in the form of characters. It is used to store data and strings.
    - Ex: txt
2. **Binary files:-**
    - Stores data in the form of bytes(group of 8 bits)
    - Ex: Audio, video, Image

What is file handling?

- Opening a file.
- Performing some operations on it.
- Closing a file

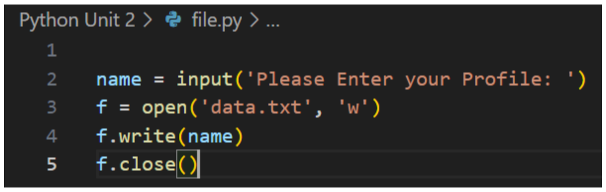
Memory

1. Random Access memory / Volatile memory
    - A computer storage that temporarily holds data being used or processed.
2. Non Volatile Memory
    - File store in this memory

Data Storage

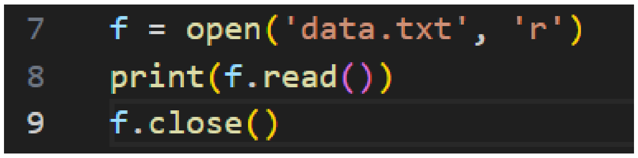
1. File Handling
2. Database

Example:



Python Unit 2 > file.py > ...

```python

name = input('Please Enter your Profile: ')
f = open('data.txt', 'w')
f.write(name)
f.close()
```



```python
f = open('data.txt', 'r')
print(f.read())
f.close()
```

Q. File handling vs Database

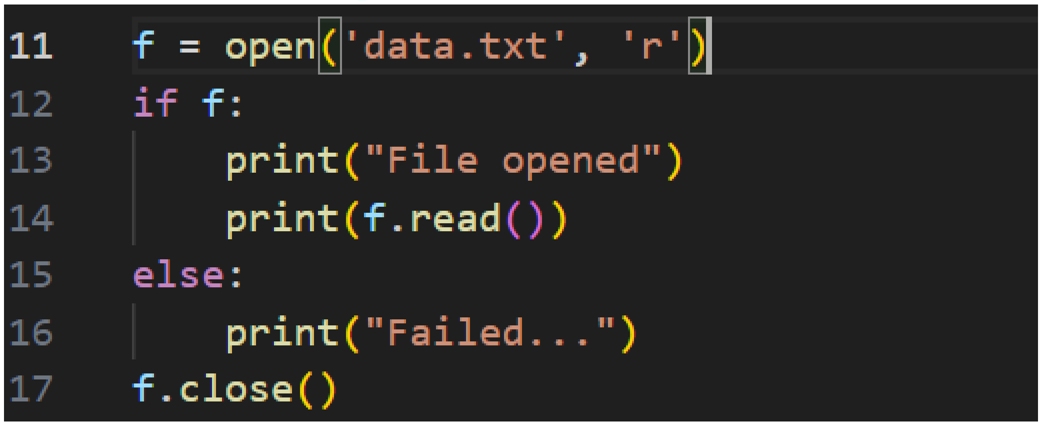
Opening file:-

- Python provides an in-built function **open()** to open a file.
- **Syntax:-**

  f = open(filename,mode='r',buffering, encoding=None, errors=None, newline=None,   closefd=True)

  f = open(filename,mode='r')
  filename :- file to be accessed.
  mode :- purpose of opening file. Ex: read, write
  f :- file handler, file pointer

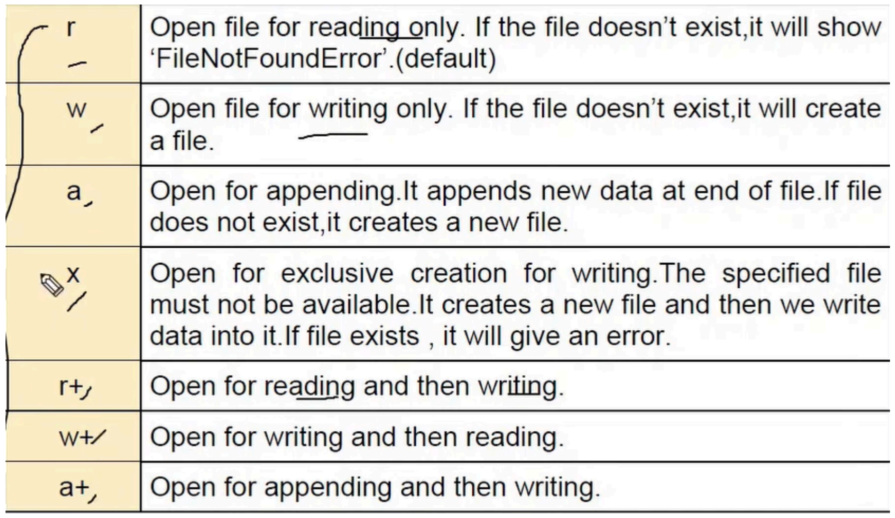
  Example:



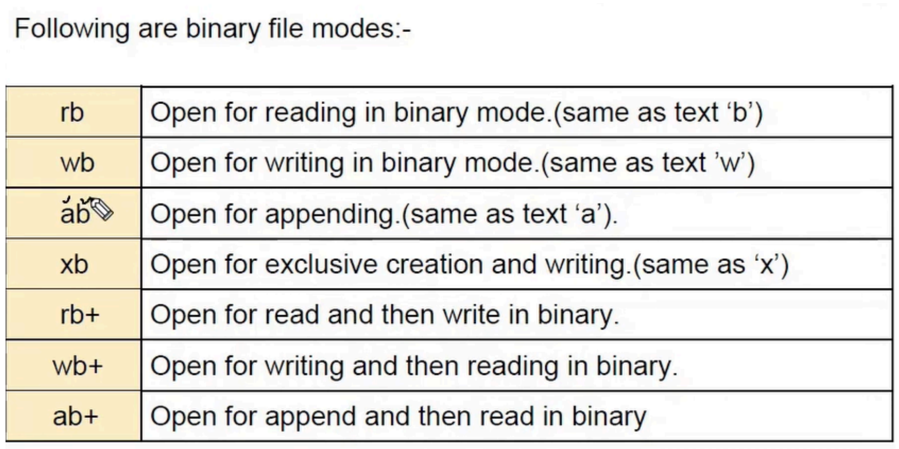
```python
f = open('data.txt', 'r')
if f:
    print("File opened")
    print(f.read())
else:
    print("Failed...")
f.close()
```

Mode of opening file:-

- when you open a file for operations, you have to specify mode. mode specifies the purpose of opening a file. There are two types of modes
  1. **Text modes**
     - when you open a file in text mode, python treats it's content as "str" type.

- When you get data from a text mode file, python first decodes the raw bytes using either a platform dependent encoding or specified encoding
- 

r	Open file for reading only. If the file doesn't exist,it will show 'FileNotFoundError'.(default)
w	Open file for writing only. If the file doesn't exist,it will create a file.
a	Open for appending.It appends new data at end of file.If file does not exist,it creates a new file.
x	Open for exclusive creation for writing.The specified file must not be available.It creates a new file and then we write data into it.If file exists , it will give an error.
r+	Open for reading and then writing.
w+	Open for writing and then reading.
a+	Open for appending and then writing.

2. **Binary modes**
   - When you open a file for binary mode, python uses the data without any encoding. Binary mode file reflects the raw data directly in the file
   - Python treats file contents as "bytes" type. These modes are used while dealing with non-text files like images, audios, videos etc.
   - 

Following are binary file modes:-

rb	Open for reading in binary mode.(same as text 'b')
wb	Open for writing in binary mode.(same as text 'w')
ab	Open for appending.(same as text 'a').
xb	Open for exclusive creation and writing.(same as 'x')
rb+	Open for read and then write in binary.
wb+	Open for writing and then reading in binary.
ab+	Open for append and then read in binary

Buffering:-

- Positive Integer value used to set buffer size for file.
- In text mode, buffer size should be 1 Byte or more than 1.
- In binary mode, buffer size can be 0.

- Default size:- 4096-8192 bytes

Encoding:-

- Encoding type used to decode and encode files.
- Should be used in text mode only.
- Default value depends on the OS.
- For windows :- cp125R

Errors:-

- Represents how encoding and decoding errors are to be handled.
- Cannot be used in binary mode.
- Some standard values are :- strict, ignore, replace etc

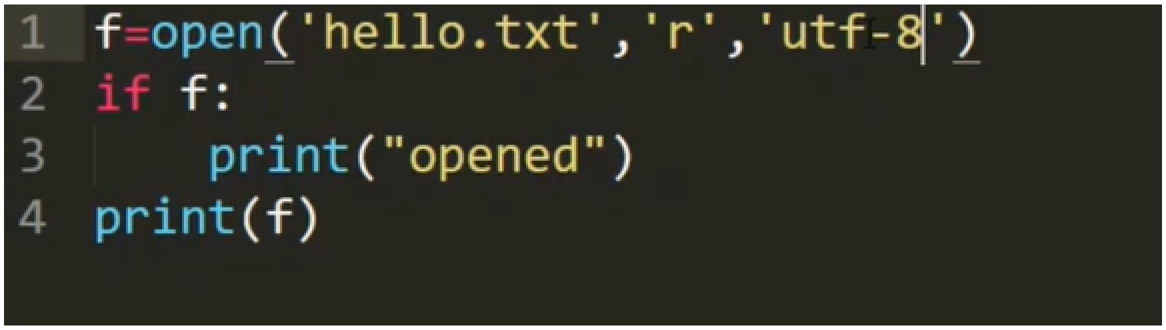
Newline:-

- It can be \n, \r, \r\n

Closefd:-

What is a File Descriptor:

- A file descriptor is a low-level handle assigned by the operating system to represent an open file. It is an integer that uniquely identifies an open file in a program
- **`closefd=True`** (default)
- **`closefd=false`**
  This is less common and typically used in more advanced scenarios, such as when integrating with other systems or libraries that require direct manipulation of file descriptors.



```
f=open('hello.txt','r','utf-8')
if f:
    print("opened")
print(f)
```

Closing file:-

1. **How to close a file?**
    - close() :- function used to close a file.
2. **What happens if we do not close a file?**

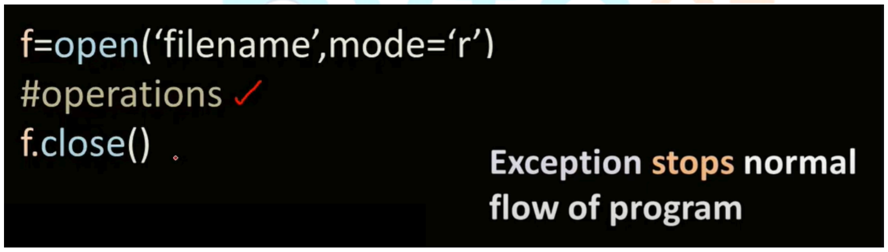
    Data will corrupt

    Memory wastage

- **What happens when we close a file?:-**
- File object is deleted from memory and the file is no longer accessible unless we open it again.
- After program execution, the python garbage collector will destroy the file object and close the file automatically.
- **Note:-** Don't rely on garbage collectors.

Ways of closing files:-

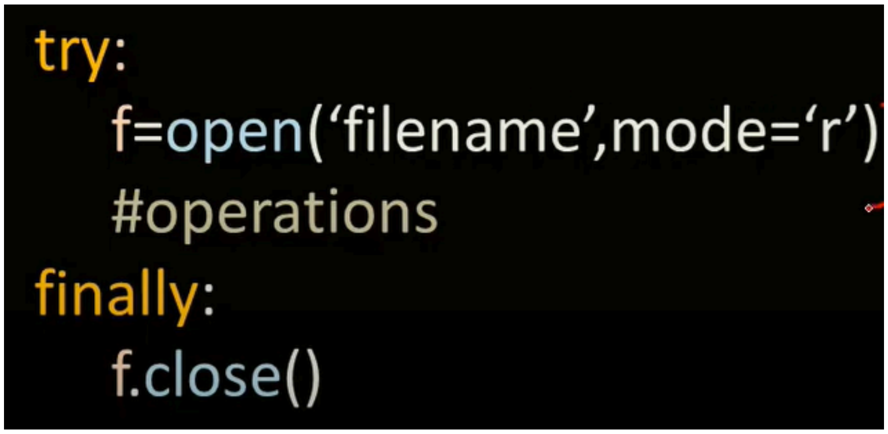
1. Normal way



```
f=open('filename',mode='r')
#operations
f.close()
```

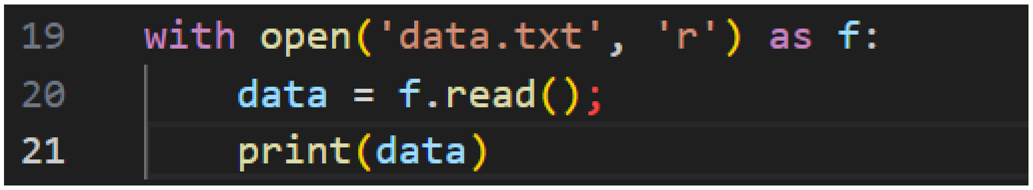
Exception stops normal flow of program

2. Using exception handling



```
try:
    f=open('filename',mode='r')
    #operations
finally:
    f.close()
```

3. with statement



```python
with open('data.txt', 'r') as f:
    data = f.read();
    print(data)
```

File Objects:

- In Python, files are opened using the **`open()`** function, which returns a file object. This object provides methods and attributes to interact with the file

- f = open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True)
- f is object

File Built-in Function (`open()`)

The **`open()`** function in Python is used to open a file. This function returns a file object, which can then be used to read from, write to, or perform other operations on the file.

- **Syntax**: **`open(filename, mode)`**
  - **filename**: The name of the file you want to open.
  - **mode**: The mode in which you want to open the file. Common modes include:
    - **`'r'`** for reading (default).
    - **`'w'`** for writing (will overwrite the file if it exists).
    - **`'a'`** for appending (will add to the end of the file).
    - **`'b'`** for binary mode (e.g., **`'rb'`** or **`'wb'`** for reading or writing binary files).

File Built-in Methods(File Object Method)

- Once you have opened a file using the open() function, you get a file object. This file object has several built-in methods that allow you to interact with the file.
- Here are some common file object methods:
- **read(size=-1):** Reads the entire file or a specified number of bytes if the size is provided. Example:

```python
file = open("example.txt", "r")
content = file.read(10)  # Reads the first 10 characters
print(content)
file.close()
```

- **write(string):** Writes a string to the file.

```python
file = open("example.txt", "w")
file.write("Hello, World!")
file.close()
```

- **`readline()`:** Reads a single line from the file.

```python
file = open("example.txt", "r")
line = file.readline()
print(line)
file.close()
```

- **`readlines()`:** Reads all lines in the file and returns them as a list.

```python
file = open("example.txt", "r")
lines = file.readlines()
print(lines)
file.close()
```

- **`close()`:** Closes the file, freeing up any system resources tied to it.

```python
file = open("example.txt", "r")
file.close()
```

- **`read(size=-1)`, `readline(size=-1)`, `readlines(hint=-1)`, `write(string)`, `writelines(lines)`, `close()`, `flush()`, `seek(offset, whence=0)`, `tell()`, `truncate(size=None)`, `fileno()`, `isatty()`, `readable()`, `writable()`, `seekable()`, `detach()`, `readinto(b)`, `closefd()`, `__enter__()`, `__exit__(exc_type, exc_val, exc_tb)`.**

File Built-in Attributes:-

- **name:-** has name of specified file.

- **mode:-** mode of specified file.
- **closed:-** has boolean value. Shows file closed or not.
- **encoding:-** has encoding name.
- **Syntax:-**
  - File_object.variable_name

```
file = open('example.txt', 'r')
print(file.name)  # Prints the name of the file
print(file.mode)  # Prints the mode in which the file is opened
print(file.closed)  # Prints False as the file is open
file.close()
print(file.closed)  # Prints True as the file is now closed
```

Standard Files

Standard files are a set of predefined file streams that are automatically opened when a program starts. These streams allow a program to interact with the input and output devices in a structured way.

In Python, the operating system provides **three standard file streams**:

1. **Standard Input (`sys.stdin`)**: This stream is used to receive input data from the user. By default, it takes input from the keyboard.

```
import sys

print("Enter your name: ")
name = sys.stdin.readline().strip()  # Reading input from standard input
print(f"Hello, {name}!")
```

2. **Standard Output (`sys.stdout`)**: This stream is used to send output data to the user. By default, it prints data to the terminal or console.

```
import sys

sys.stdout.write("This is an output message.\n")
```

3. **Standard Error (`sys.stderr`)**: This stream is used to display error messages. By default, it sends error output to the terminal or console, but it is separate from standard output.

```python
import sys

sys.stderr.write("This is an error message.\n")
```

Why Are Standard Files Important?

Standard files make it easy for programs to communicate with users and other systems. Here's why they're important:

1. **Handling Input and Output Easily**:
   - **Standard Input** lets programs get information from the user, like typing in a name or number.
   - **Standard Output** lets programs show information to the user, like printing a message on the screen.
   - **Standard Error** helps programs report problems separately from regular messages, so you can see what went wrong.
2. **Separating Regular Output from Errors**:
   - By using different streams for normal output (`stdout`) and error messages (`stderr`), you can handle problems more efficiently. For example, error messages won't get mixed up with regular results, making it easier to fix issues.
3. **Flexibility**:
   - You can **redirect** these streams. For example, instead of showing output on the screen, you can save it to a file. This is useful when you want to save logs or automate tasks without needing constant user input.

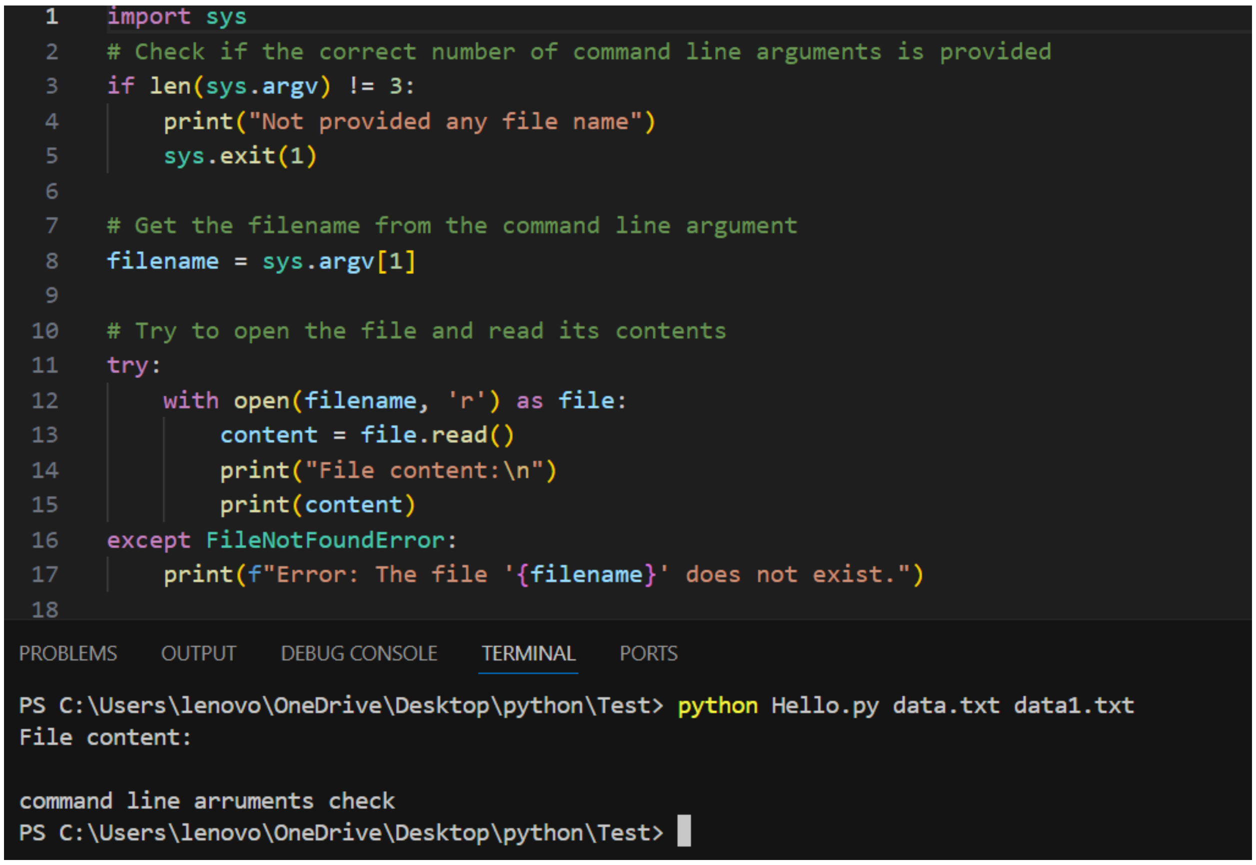
Real Application example: a simple log analyzer

Why This is Useful:

- **Organized Logging**: By separating standard output and errors, the application can keep a clear record of what happened (normal messages) and what went wrong (error messages).
- **Automation**: This kind of setup is often used in automated systems where logs are reviewed later, so the program can run without constant monitoring.
- **Error Tracking**: You can track and troubleshoot errors without cluttering normal application output.

What are Command Line Arguments?

- Command line arguments are extra values that you pass to a script when you run it from the terminal. These values can be filenames, paths, or other data the program needs.
- Command line arguments are useful in file handling because you can pass file names as arguments, allowing the script to work with different files without needing to modify the code.
- In Python, you can access command line arguments using the `sys.argv` list from the `sys` module.



```python
import sys
# Check if the correct number of command line arguments is provided
if len(sys.argv) != 3:
    print("Not provided any file name")
    sys.exit(1)

# Get the filename from the command line argument
filename = sys.argv[1]

# Try to open the file and read its contents
try:
    with open(filename, 'r') as file:
        content = file.read()
        print("File content:\n")
        print(content)
except FileNotFoundError:
    print(f"Error: The file '{filename}' does not exist.")

```

```
PS C:\Users\lenovo\OneDrive\Desktop\python\Test> python Hello.py data.txt data1.txt
File content:

command line arruments check
PS C:\Users\lenovo\OneDrive\Desktop\python\Test>
```

File System Concepts

- **Files**: These are individual documents or pieces of data stored on the disk (e.g., `.txt`, `.csv`, `.jpg` files).
- **Directories (Folders)**: These are containers that hold files and can also hold other directories, creating a hierarchical structure.

- ## Key Python Modules for File System Handling

- Python provides modules such as `os` and `shutil` to interact with the file system. Here's a breakdown of the key functionality:

1. **`os` module**: This module provides a way of interacting with the operating system, including creating, renaming, and deleting files and directories.
2. **`os.path` module**: This module offers utilities to work with file paths, check if files exist, and determine file types (e.g., directories vs. files).
3. **`shutil` module**: This module is useful for higher-level file operations, such as copying and moving files.

- **Common File System Operations**
  1. **Creating Directories**: Use `os.mkdir()` to create a directory.
  2. **Checking if a Directory Exists**: Use `os.path.exists()` to check if a directory exists.
  3. **Removing Directories**: Use `os.rmdir()` to remove an empty directory.
  4. **Creating/Opening Files**: Use the `open()` function to create or open a file.
  5. **Renaming Files**: Use `os.rename()` to rename a file.
  6. **Deleting Files**: Use `os.remove()` to delete a file.

File Execution

File execution means running a Python script that performs a set of actions, such as reading from, writing to, or manipulating files. Python scripts (files with `.py` extension) are executed by the Python interpreter.

There are two main ways to execute Python files:

- **Direct Execution**
- **Script as a Module**

Persistent Storage

Persistent storage refers to saving data in such a way that it remains available even after the program has stopped running. This is different from in-memory storage, where data is lost when the program terminates.

Persistent Storage Modules in Python

Python offers several built-in modules for persistent storage, with each suited for different use cases. Here are some key modules to cover:

1. **`pickle` Module**
2. **`shelve` Module**
3. **`sqlite3` Module**

pickle Module

The pickle module is used to serialize and deserialize Python objects. Serialization is the process of converting a Python object into a format that can be saved to a file. Deserialization is the reverse, where the file is read and converted back into a Python object.

Example: Using pickle for Persistent Storage

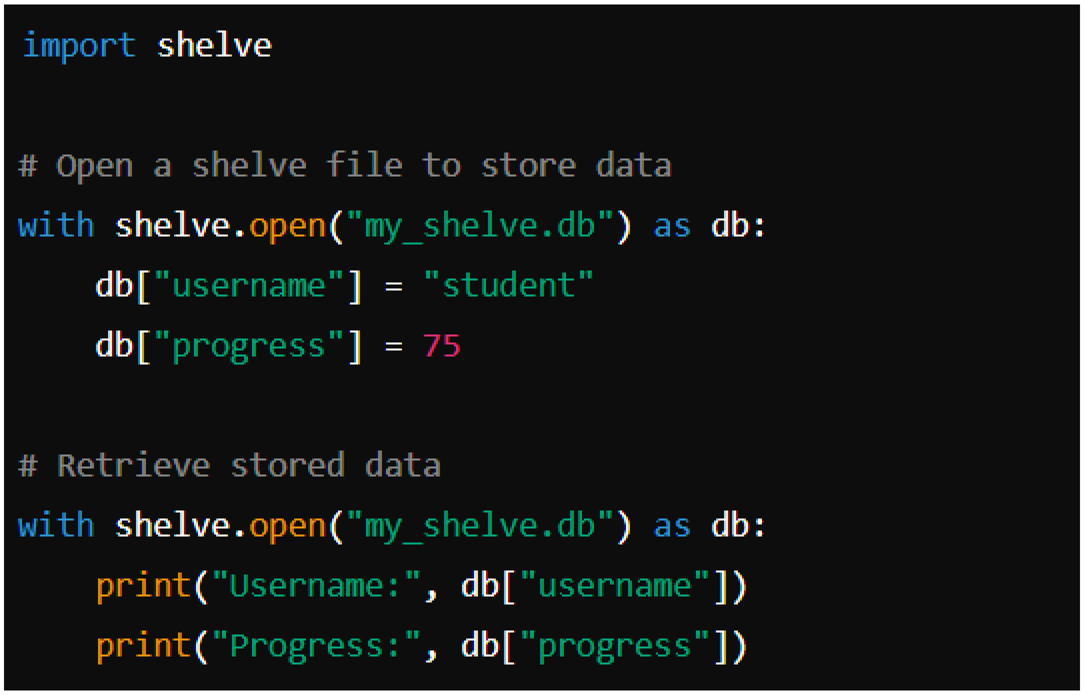
```python
import pickle

# Sample data to serialize (could be a list, dictionary, etc.)
data = {"name": "Alice", "age": 25, "city": "New York"}

# Serialize and save data to a file
with open("data.pkl", "wb") as file:
    pickle.dump(data, file)

# Deserialize and load data from the file
with open("data.pkl", "rb") as file:
    loaded_data = pickle.load(file)

print(loaded_data)  # Output: {'name': 'Alice', 'age': 25, 'city': 'New York'}
```

- **Usage**: pickle is useful for saving Python objects like lists, dictionaries, and custom objects to a file for later use.

4. shelve Module

The shelve module builds on pickle and provides a simple key-value store. It behaves like a dictionary, but the data is stored in a file, allowing for persistent storage across program runs.

Example: Using shelve for Persistent Storage



```python
import shelve

# Open a shelve file to store data
with shelve.open("my_shelve.db") as db:
    db["username"] = "student"
    db["progress"] = 75

# Retrieve stored data
with shelve.open("my_shelve.db") as db:
    print("Username:", db["username"])
    print("Progress:", db["progress"])
```

- **Usage**: `shelve` is great for simple persistent storage of Python objects that you can access using keys, like a dictionary.

5. `sqlite3` Module

`sqlite3` is a lightweight database module that stores data in a relational database. It is ideal for more structured data and when you need to perform complex queries.

Example: Using `sqlite3` for Persistent Storage

```python
import sqlite3

# Connect to the database (creates the file if it doesn't exist)
conn = sqlite3.connect("students.db")

# Create a cursor object to interact with the database
cur = conn.cursor()

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

# Insert data
cur.execute("INSERT INTO students (name, grade) VALUES ('Alice', 90)")
cur.execute("INSERT INTO students (name, grade) VALUES ('Bob', 85)")

# Commit the changes
conn.commit()

# Retrieve and display data
cur.execute("SELECT * FROM students")
rows = cur.fetchall()
for row in rows:
    print(row)

# Close the connection
conn.close()
```

- **Usage**: `sqlite3` is used for more complex and structured data storage, such as saving user information or app data in a relational format

Related Modules

1. os Module

The `os` module provides functions to interact with the operating system, including file and directory operations. Key functions related to file handling:

- **`os.getcwd()`**: Gets the current working directory.

- **`os.chdir(path)`**: Changes the current working directory to `path`.
- **`os.listdir(path)`**: Lists all the files and directories in the specified `path`.
- **`os.remove(filename)`**: Deletes the file with the given `filename`.
- **`os.rename(old_name, new_name)`**: Renames a file or directory.

2. shutil Module

The `shutil` module provides higher-level operations on files and directories, such as copying, moving, or deleting them.

- **`shutil.copy(src, dst)`**: Copies a file from `src` to `dst`.
- **`shutil.move(src, dst)`**: Moves a file or directory from `src` to `dst`.
- **`shutil.rmtree(path)`**: Recursively deletes a directory tree.

3. pathlib Module

The `pathlib` module offers an object-oriented approach to file handling and path manipulation. It is a modern replacement for `os.path`.

- **`Path.exists()`**: Checks if a file or directory exists.
- **`Path.is_file()`**: Checks if the path points to a file.
- **`Path.is_dir()`**: Checks if the path points to a directory.
- **`Path.mkdir()`**: Creates a new directory.

4. json Module

For handling data in JSON format, which is commonly used for storing data in files, you can use the `json` module.

- **`json.dump(obj, file)`**: Writes an object as a JSON string to a file.
- **`json.load(file)`**: Reads a JSON object from a file.

5. csv Module

For handling CSV files, which are commonly used for storing tabular data, you can use the `csv` module.

- **`csv.reader(file)`**: Reads data from a CSV file.
- **`csv.writer(file)`**: Writes data to a CSV file.

Exception

What Are Exceptions in Python?

Definition:
An exception is an event that disrupts the normal flow of a program's execution. When an error occurs, Python generates an exception, and unless the exception is handled, the program will crash.

Types of Errors:

1. **Syntax Errors**: Mistakes in the structure of the code, such as missing a colon or parentheses.
   - Example: `if x == 5 print("Hello")`
2. **Exceptions**: These occur during the execution of a program, like dividing by zero or trying to access a non-existent file.

Handling Exceptions:

- **try block**: The code that may potentially cause an error goes inside the `try` block.
- **except block**: When an exception occurs, the code inside the corresponding `except` block is executed.
- **Multiple Exceptions**: You can handle multiple types of exceptions with different `except` blocks.
- **else block**: This runs if no exceptions occur.
- **finally block**: This runs no matter what, whether an exception occurs or not. It's used for cleanup actions like closing files.

```
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
    print("That's not a number.")
else:
    print("The result is:", result)
finally:
    print("This will always be printed.")
```

Raising Exceptions:

You can also intentionally raise exceptions using the `raise` keyword.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be at least 18.")
    return age

try:
    age = int(input("Enter your age: "))
    check_age(age)
except ValueError as e:
    print(e)
```

Detecting Exceptions

When you suspect that a particular block of code might raise an error, you wrap it inside a `try` block. The `try` block helps you **detect** if an exception occurs.

Handling Exceptions

After detecting an exception, we can handle it using the `except` block. The goal is to prevent the program from crashing and handle errors in a controlled way.

Best Practices in Exception Handling:

- **Use specific exceptions**: Catch specific exceptions like `ValueError` or `ZeroDivisionError` instead of using a general `except` block. This makes your code easier to debug and maintain.
- **Use the `finally` block**: Use this block for tasks that should run whether an exception occurs or not, such as closing files or cleaning up resources.

Context Management

Context management allows you to properly manage resources (e.g., files, connections) by ensuring they are automatically cleaned up (e.g., closed) after their use, even if an exception occurs.

Python's context management is often done using the `with` statement, which simplifies resource management and ensures that resources are released, avoiding memory leaks or locking issues.

Using Context Managers with the `with` Statement

The `with` statement is used with objects that support context management. The most common example is file handling.

Context Management and Exceptions

When using context management with the `with` statement, the context manager ensures that necessary cleanup (like closing the file) is performed, even if an exception is raised inside the block.

```python
try:
    with open("example.txt", "r") as file:
        data = file.read()
        print(data / 2)  # Intentional error to cause an exception
except Exception as e:
    print(f"An error occurred: {e}")
```

Creating a Custom Context Manager

You can create your own context manager by defining a class that implements the special methods `__enter__` and `__exit__`.

Benefits of Context Management:

1. **Automatic Cleanup**: Resources like files, network connections, or database connections are automatically cleaned up after use.
2. **Simpler Code**: You don't have to manually close resources in `finally` blocks.
3. **Exception Safety**: Context managers ensure that cleanup is performed even if an exception is raised.

Exceptions as Strings

In Python, exceptions are objects that belong to various exception classes like `ValueError`, `TypeError`, etc. However, these exception objects carry useful information in the form of messages (error descriptions) that we can convert to strings.

Extracting Exception Messages

When an exception is raised, you can capture it using the `except` block and access its message as a string by converting the exception object to a string.

```python
try:
    number = int(input("Enter a number: "))
    result = 10 / number
except ZeroDivisionError as e:
    print("An exception occurred:", str(e))  # Convert the exception to a string
```

`repr()` vs. `str()` for Exceptions

In addition to `str()`, Python has a `repr()` function that gives a more detailed representation of the exception object, often including the exception type and message.

Practical Use Cases

- **Logging Errors**: When logging exceptions, you may want to convert them to strings to make the error messages more readable.
- **User-Friendly Messages**: By converting exceptions to strings, you can provide user-friendly error messages while hiding technical details.

What are Assertions?

An **assertion** is a debugging aid that tests a condition. If the condition evaluates to `True`, the program continues running normally. If the condition evaluates to `False`, Python raises an `AssertionError` exception, which can be used to catch bugs early in the development process.

When to Use Assertions

Assertions are typically used during development to enforce certain conditions. They are helpful when you want to ensure that the internal logic of your code is functioning as expected.

```python
try:
    x = 3
    assert x > 5, "x should be greater than 5"
except AssertionError as e:
    print(f"Assertion failed: {e}")
```

Assertions vs. Exceptions

- **Assertions**: Used to test conditions that should logically never happen. They are a debugging tool and are generally turned off in production code.
- **Assertions** are for catching bugs in your code during development.
- **When to Use**: Use assertions during development to catch programming mistakes or logic errors that should not occur. If something goes wrong, it means there's a bug in your code.

- **Exceptions**: Used for handling errors that may occur during normal program execution (e.g., file not found, invalid input).
- **When to Use**: Use exceptions to handle things that might go wrong during runtime and that you expect might happen occasionally.
- **Exceptions** are for handling unexpected events or errors during the normal operation of your program.

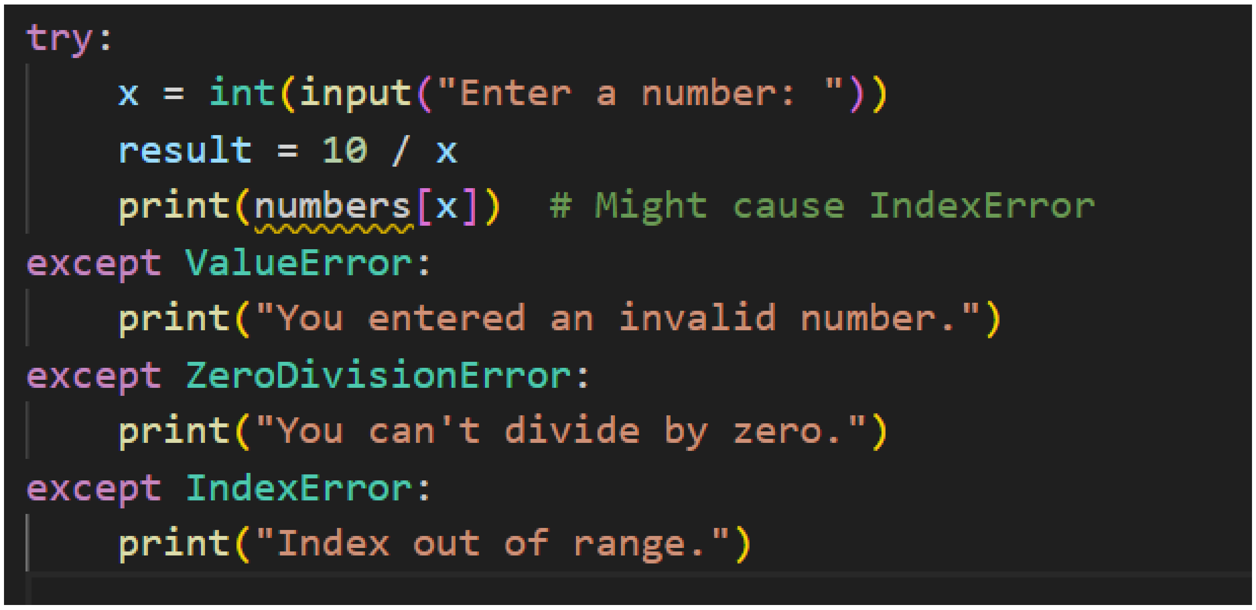
What are Standard Exceptions?

Standard exceptions are predefined error types in Python that handle common error conditions. These exceptions are part of Python's built-in library and cover a wide range of error scenarios such as dividing by zero, accessing undefined variables, invalid operations on data types, and more.

Common Standard Exceptions

Here are some of the most commonly encountered standard exceptions in Python, along with explanations and examples:

1. `ValueError`:
    - **When it Occurs**: Raised when a function receives an argument of the right type but an inappropriate value.
2. `TypeError`:
    - **When it Occurs**: Raised when an operation or function is applied to an object of an inappropriate type.
3. `IndexError`:
    - **When it Occurs**: Raised when you try to access an index in a list that is out of range.
4. `KeyError`:
    - **When it Occurs**: Raised when you try to access a key in a dictionary that doesn't exist.
5. `ZeroDivisionError`:
    - **When it Occurs**: Raised when you try to divide by zero.
6. `AttributeError`:
    - **When it Occurs**: Raised when an attribute reference or assignment fails.
7. `FileNotFoundError`:
    - **When it Occurs**: Raised when trying to open a file that doesn't exist.
8. `NameError`:
    - **When it Occurs**: Raised when a local or global name is not found.



```
try:
    x = int(input("Enter a number: "))
    result = 10 / x
    print(numbers[x])  # Might cause IndexError
except ValueError:
    print("You entered an invalid number.")
except ZeroDivisionError:
    print("You can't divide by zero.")
except IndexError:
    print("Index out of range.")
```

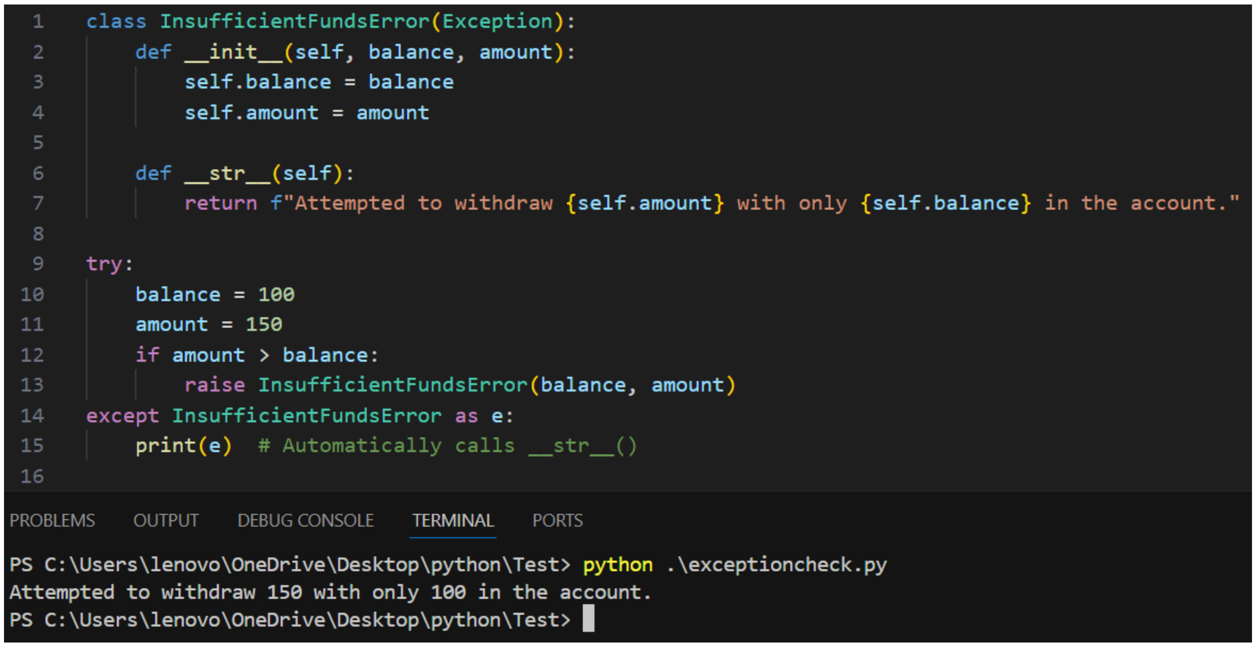
Create Custom Exceptions

- **Tailored Error Handling**: Custom exceptions allow you to represent specific types of errors in your code that aren't covered by standard exceptions.
- **Code Clarity**: They can make your code more understandable by clearly indicating the type of error that occurred.

Steps to Create a Custom Exception

Custom exceptions are created by defining a new class that inherits from Python's built-in `Exception` class or one of its subclasses.

1. We define a class `InsufficientFundsError` that inherits from `Exception`.
2. The custom exception is raised using `raise`, and we provide a custom error message.
3. The custom exception is caught using `except`, just like a built-in exception.



```python
class InsufficientFundsError(Exception):
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount

    def __str__(self):
        return f"Attempted to withdraw {self.amount} with only {self.balance} in the account."

try:
    balance = 100
    amount = 150
    if amount > balance:
        raise InsufficientFundsError(balance, amount)
except InsufficientFundsError as e:
    print(e)  # Automatically calls __str__()

```

```
PS C:\Users\lenovo\OneDrive\Desktop\python\Test> python .\exceptioncheck.py
Attempted to withdraw 150 with only 100 in the account.
PS C:\Users\lenovo\OneDrive\Desktop\python\Test>
```

- `__init__`: This is the constructor method called when you create an object. It initializes the object's attributes.
- `self`: Refers to the current instance of the class and allows access to the instance's attributes and methods.
- `__str__`: Defines how the object should be represented as a string, which is what you see when you print the object.

When to Use Custom Exceptions

- **Specific Error Conditions**: Use custom exceptions when you want to represent specific error conditions that are not covered by the built-in exceptions.
- **Reusability**: Custom exceptions can be reused across multiple modules or parts of the program where similar errors might occur.

Why Exceptions Exist in Python:

1. **Error Handling**: Exceptions allow you to manage errors without crashing the program.
   - *Example*: Handling division by zero error.
2. **Separation of Concerns**: Exceptions separate error-handling logic from the core business logic.
   - *Example*: Separating input validation from main logic.
3. **Maintainability**: Exceptions centralize error handling, making code easier to maintain.
   - *Example*: Catching different types of errors in one place.
4. **Predefined Exceptions**: Python provides built-in exceptions to handle common errors.
   - *Example*: Handling a `TypeError` when performing an invalid operation.

5. **Custom Error Messages**: Exceptions let you create detailed, user-friendly error messages.
    - *Example*: Using a custom exception message.
6. **Fault Tolerance**: Exceptions allow programs to continue running despite errors.
    - *Example*: Logging an error and continuing execution.
7. **Cleaner Code**: Exceptions reduce the need for multiple `if-else` checks, resulting in cleaner code.
    - *Example*: Handling file opening without redundant checks.

Why Exceptions at All?

1. **Inevitability of Errors**: Errors are bound to happen; exceptions help handle them gracefully.
    - *Example*: Catching wrong input type.
2. **Avoiding Program Crashes:** Exceptions prevent programs from crashing when errors occur.
    - ***Example*:** Preventing a crash when dividing by zero.
3. **Graceful Recovery**: Exceptions allow programs to recover from errors and continue running.
    - *Example*: Handling a missing file and prompting for another one.
4. **Separation of Error Handling from Logic**: Exceptions keep error handling separate from core business logic.
    - *Example*: Separating logic from error handling when converting input.
5. **Promotes Robustness and Reliability**: Exceptions make your code more reliable by anticipating failure points.
    - *Example*: Handling network issues in a web app.
6. **Improving User Experience**: Exceptions provide informative messages and help users correct mistakes.
    - *Example*: Explaining why a divide-by-zero operation failed.
7. **Encouraging Defensive Programming**: Exceptions allow developers to anticipate problems and handle them proactively.
    - *Example*: Handling data corruption gracefully.

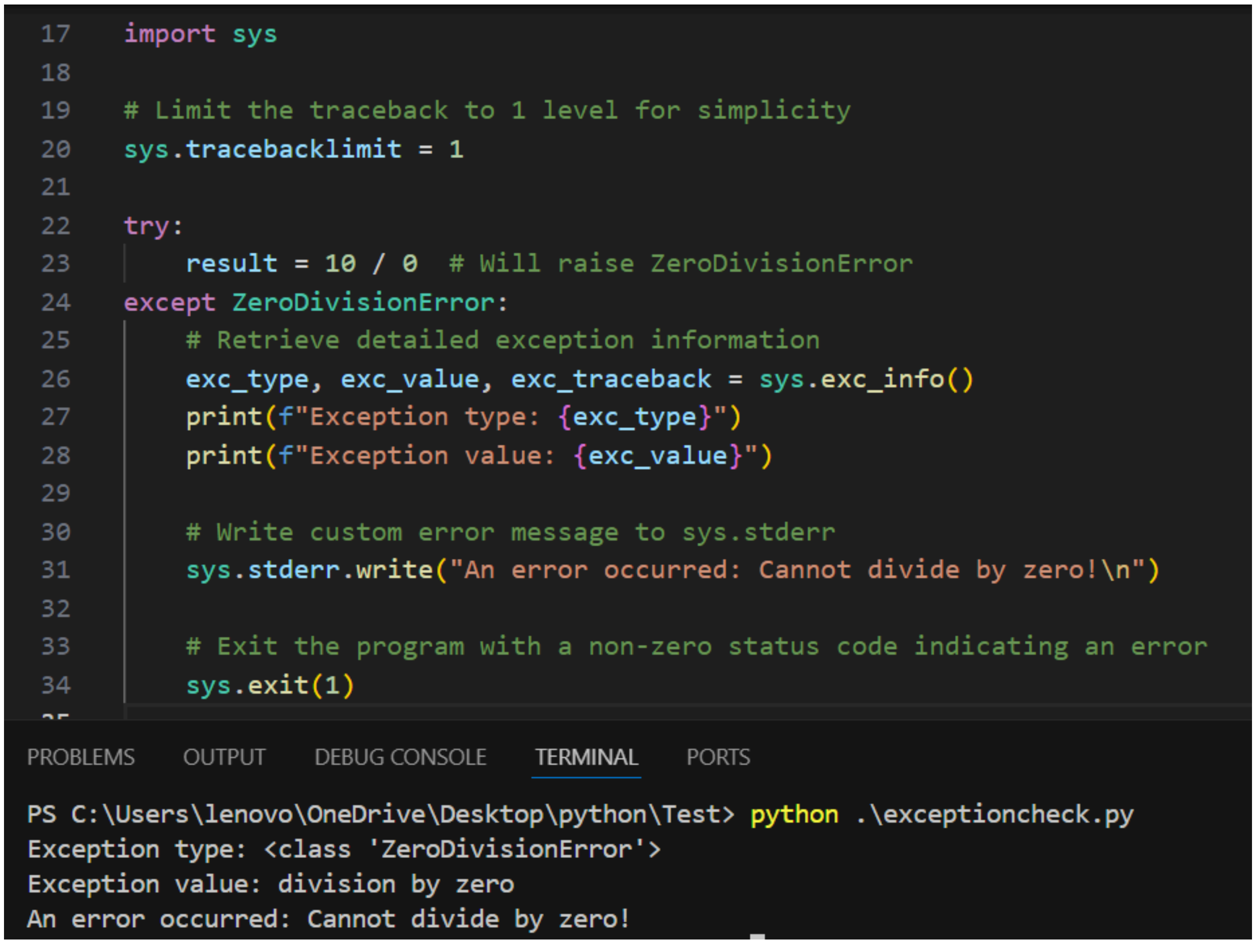
Exceptions and the `sys` Module

Accessing System-Specific Information:

- ○ The `sys` module provides functions and variables to interact with the Python interpreter. One of its key features in exception handling is to access detailed error information when an exception occurs.

2. **`sys.exc_info()`**: Retrieves detailed information (type, value, traceback) about the most recent exception.
3. **`sys.exit()`**: Exits the program with an optional status code, often used during fatal errors.

4. `sys.stderr`: Writes error messages to the standard error stream instead of normal output.
5. `sys.tracebacklimit`: Limits the depth of traceback information shown when an exception occurs.



```python
import sys

# Limit the traceback to 1 level for simplicity
sys.tracebacklimit = 1

try:
    result = 10 / 0  # Will raise ZeroDivisionError
except ZeroDivisionError:
    # Retrieve detailed exception information
    exc_type, exc_value, exc_traceback = sys.exc_info()
    print(f"Exception type: {exc_type}")
    print(f"Exception value: {exc_value}")

    # Write custom error message to sys.stderr
    sys.stderr.write("An error occurred: Cannot divide by zero!\n")

    # Exit the program with a non-zero status code indicating an error
    sys.exit(1)
```

```
PS C:\Users\lenovo\OneDrive\Desktop\python\Test> python .\exceptioncheck.py
Exception type: <class 'ZeroDivisionError'>
Exception value: division by zero
An error occurred: Cannot divide by zero!
```

Related Modules

1. sys Module:

- **Purpose**: Provides functions and variables that can be used to manipulate different parts of the Python runtime environment, especially useful in exception handling.
- **Real-World Example**: Imagine you're developing a command-line tool that processes user input from a file. If the file is missing or an error occurs, you want the program to exit gracefully and provide a meaningful error message. You can use the `sys` module to achieve this.
- **Key Functions**:

  - `sys.exc_info()`: Retrieves information about the current exception (type, value, and traceback).
  - `sys.exit()`: Exits the program with an optional status code.

`traceback` Module:

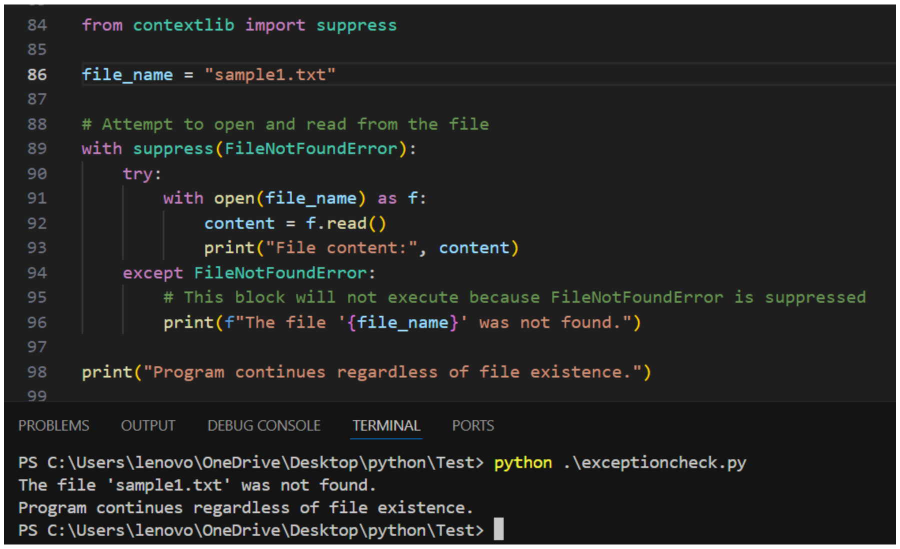
- **Purpose**: Provides utilities to extract, format, and print stack traces of exceptions. It's useful for logging detailed error messages.
- **Real-World Example**: Imagine you are working on a large web application, and one of your APIs is crashing. Using the `traceback` module, you can print out detailed error logs to help identify exactly which part of your code is causing the problem.
- **Key Functions**:
  - `traceback.format_exc()`: Returns a string representation of the traceback.
  - `traceback.print_exc()`: Prints the traceback to standard output.

`logging` Module:

- **Purpose**: Allows you to log error messages, warnings, or any other information, which can be helpful for debugging and maintaining logs of exceptions. Instead of printing error messages to the screen, you can log them to a file or a server for later analysis.
- **Real-World Example**: Suppose you're running a payment processing system. If something goes wrong (like a network error), you want to log the error in a file so you can investigate later, even after the system has been running for a while.
- **Key Functions**:
  - `logging.error()`: Logs error-level messages, usually for handling exceptions.

`contextlib` Module:

- **Purpose**: Provides utilities for working with context managers, which can be useful for managing resources and handling exceptions.
- **Real-World Example**: Imagine you're writing a script to clean up files on a computer. Some files may be locked or inaccessible, but you want to skip those files and continue cleaning up the others without crashing the program
- **Key Functions**:
  - `contextlib.suppress()`: Suppresses specified exceptions.

- 

```python
from contextlib import suppress

file_name = "sample1.txt"

# Attempt to open and read from the file
with suppress(FileNotFoundError):
    try:
        with open(file_name) as f:
            content = f.read()
            print("File content:", content)
    except FileNotFoundError:
        # This block will not execute because FileNotFoundError is suppressed
        print(f"The file '{file_name}' was not found.")

print("Program continues regardless of file existence.")
```

```
PS C:\Users\lenovo\OneDrive\Desktop\python\Test> python .\exceptioncheck.py
The file 'sample1.txt' was not found.
Program continues regardless of file existence.
PS C:\Users\lenovo\OneDrive\Desktop\python\Test>
```

`warnings` Module:

- **Purpose**: Allows you to issue warnings, which are less severe than exceptions but can notify the user of potential issues.
- **Real-World Example**: Let's say you are developing software that uses an old feature of Python, which will be removed in the future. You can issue a warning to let users know they should update their code to use a newer feature.
- **Key Functions**:
    - `warnings.warn()`: Issues a warning, which can be caught or displayed to the user.

Modules

Modules and Files

- **Definition**: A module in Python is a file containing Python code. Modules help in organizing and reusing code. For instance, `math.py` could be a module containing functions like `sqrt()` and `pow()`.
- **Creating a Module**: To create a module, simply write your Python code in a file with a `.py` extension. For example, `mymodule.py` could contain:

Namespaces

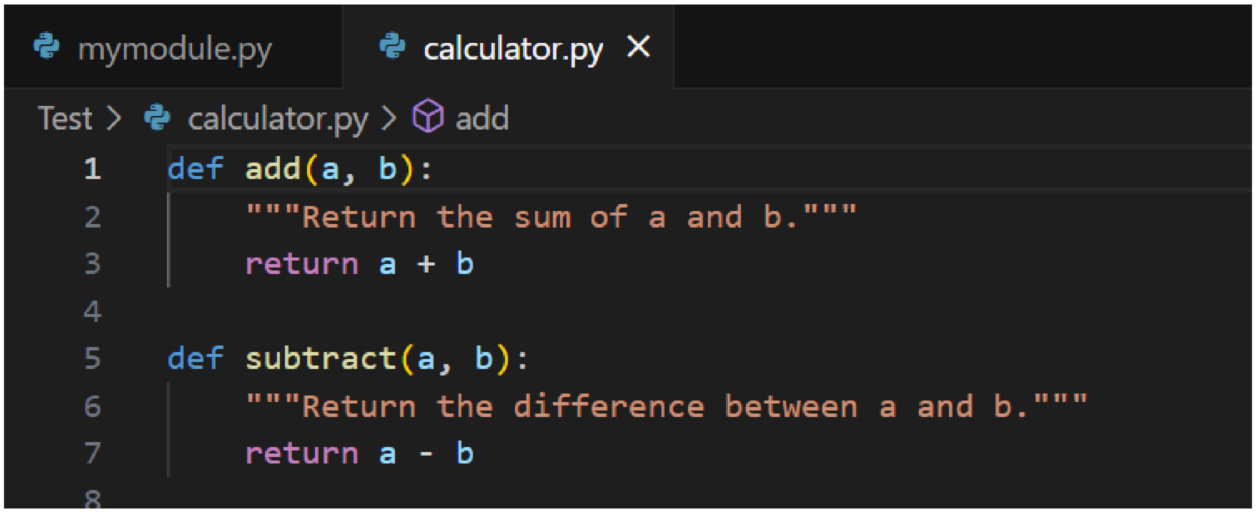
- **Definition**: A namespace is a mapping from names to objects. Namespaces ensure that names are unique and can be used without conflict. Each module creates its own namespace.
- **Usage**: When you import a module, you access its functions or variables using dot notation.
- **Rules:**
    1. **Unique Names**: Names within a namespace must be unique; the last defined name overrides any previous definitions.
    2. **Scope**: Namespaces define the scope of variables, functions, and classes.
    3. **Hierarchy**: Python namespaces are hierarchical, with local namespaces nested inside global namespaces.
    4. **Accessing Names**: Use dot notation to access names in a module or object namespace (e.g., `module.name`).
    5. **Importing Modules**: Importing a module creates a new namespace for that module's contents.
    6. **Avoiding Conflicts**: Use descriptive and unique names to prevent naming conflicts.
    7. **Import Conventions**: Import specific attributes or use aliases to manage namespaces effectively.
    8. **Module Organization**: Organize code into modules and packages for clear and manageable namespaces.
    9. **Namespace Pollution**: Avoid `from module import *` to prevent namespace pollution.
    10. **Built-in Names**: Avoid using names that conflict with Python's built-in names.

Importing Modules

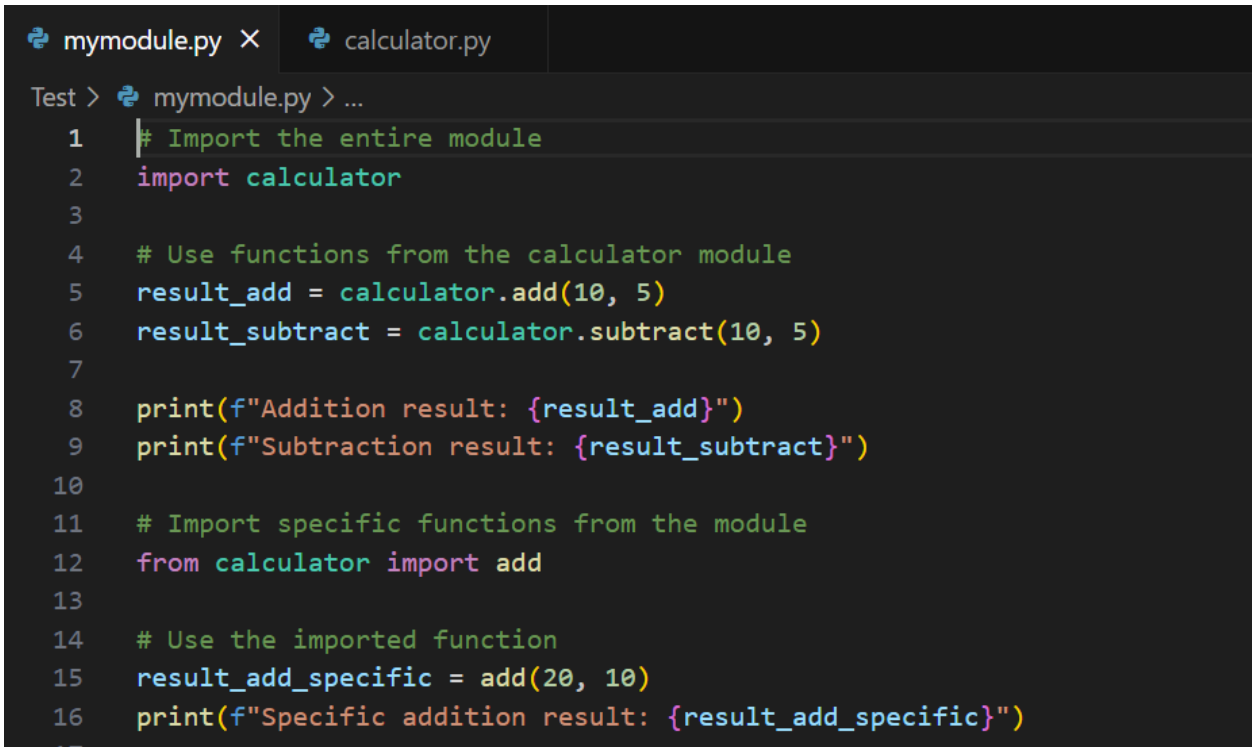
- **Basic Import**: Use the `import` statement to include a module in your script
- **Import with Alias**: You can give a module an alias to shorten its name:

Importing Module Attributes

- **Importing Specific Functions/Variables**: You can import specific items from a module to avoid clutter
- **Importing All Attributes**: You can import all attributes from a module using `*`, but this is generally discouraged as it can lead to confusion



```python
def add(a, b):
    """Return the sum of a and b."""
    return a + b

def subtract(a, b):
    """Return the difference between a and b."""
    return a - b
```



```python
# Import the entire module
import calculator

# Use functions from the calculator module
result_add = calculator.add(10, 5)
result_subtract = calculator.subtract(10, 5)

print(f"Addition result: {result_add}")
print(f"Subtraction result: {result_subtract}")

# Import specific functions from the module
from calculator import add

# Use the imported function
result_add_specific = add(20, 10)
print(f"Specific addition result: {result_add_specific}")
```

Module built-in function

`__import__()`

- **Description**: This is a built-in function that allows dynamic import of a module. While rarely used directly, it allows importing a module by name as a string, which can be helpful in situations where the module name is not known until runtime.

`dir()`

- **Description**: This function returns a list of all attributes (functions, variables, etc.) in a module. It's useful for inspecting the contents of a module.

`globals()`

- **Description**: This function returns a dictionary of the current global namespace. It’s useful for inspecting or modifying the global variables and modules in a script.

`locals()`

- **Description**: This function returns a dictionary of the current local namespace, which includes variables defined within a function or a block of code.

`help()`

- **Description**: This function provides a built-in help system, displaying documentation for a module or any other Python object. It is particularly useful for exploring modules and learning about their functionalities.

`importlib.import_module()`

- **Description**: This function is part of the `importlib` module and allows dynamic importing of modules. It is more commonly used in modern code than `__import__()`.
-

What is a Package?

- **Definition**: A package in Python is a directory containing multiple modules and a special `__init__.py` file. Packages allow you to organize your code into hierarchical structures and make it easier to manage complex projects.
- **Analogy**: Think of a package as a folder that contains related files (modules), just like a folder in your computer might contain related documents.
- `__init__.py`: This special file is required to treat the directory as a package. It can be empty or used to initialize the package.

Sub-Packages

- **Definition**: A sub-package is a package within another package, allowing for deeper organization of your code. This can be helpful for very large projects.

`__init__.py` file

- **Definition**: In Python 2.x and early Python 3.x, the presence of an `__init__.py` file in a directory was required to make Python treat that directory as a package. Without this file, the directory was not considered a package, and its contents couldn't be imported.
- **Modern Python**: In more recent versions of Python (3.3 and later), the `__init__.py` file is no longer strictly required for a directory to be recognized as a package. However, it is still widely used for initialization and customization purposes.
- **Purpose**: When you import a package or a module within a package, Python executes the code inside the `__init__.py` file. This allows you to set up package-level variables, import other modules, or execute initialization code.
- **Simplified Access**: You can use the `__init__.py` file to import specific modules or functions at the package level, making them directly accessible when the package is imported.

Other features of modules

1. **Module Caching**
    - **Explanation**: When a module is imported, Python caches it in memory. This means that subsequent imports of the same module within the same program won't reload the module but will use the cached version. This improves performance by avoiding unnecessary disk I/O.
2. **Private Members in Modules**
    - **Explanation**: You can define private members (functions or variables) in a module by prefixing their names with an underscore (`_`). These members won't be imported when using `from module import *`.
3. **Docstrings in Modules**
    - **Explanation**: Just like functions and classes, modules can have docstrings. This documentation is written at the beginning of the module and can be accessed using `help()` or `module.__doc__`.
4. **Built-in and Standard Library Modules**
    - **Explanation**: Python has a rich set of built-in modules (like `sys`, `os`, and `math`) and a standard library that includes modules for tasks such as file handling, HTTP requests, and regular expressions. Encourage students to explore the standard library.
5. **Third-Party Modules**
    - **Explanation**: Python has a vast ecosystem of third-party modules that can be installed using `pip`.

byteXL