FILE, EXCEPTION HANDLING AND OOP

Content

User defined Modules and Packages in Python

Files: File manipulations

File and Directory related methods

Python Exception Handling

OOPs Concepts –

- Class and Objects
- Constructors
- Data hiding
- Data Abstraction
- Inheritance

NAMESPACES AND SCOPE IN PYTHON

In function

Namespaces and Scope in Python

A namespace in Python refers to the mapping between variable names and their values. There are different types of namespaces based on scope:

Types of Scope:

- **Local Scope:** Variables declared inside a function are part of the local namespace and exist only within that function.
- **Global Scope:** Variables declared outside any function belong to the global namespace and can be accessed inside functions if needed.
- **Built-in Scope:** Python's built-in functions and variables (e.g., print, len) exist in this scope.
- **Enclosing Scope (for Nested Functions):** If a function is inside another function, the outer function's variables can be accessed by the inner function.

Example

```
a = 10  # Global variable

def func():
    a = 5  # Local variable (different from global 'a')
    print("Inside function:", a)

func()
print("Outside function:", a)
```

Modifying Global Variables in a Function

- By default, functions **cannot** modify global variables unless they use the global keyword.
- **Example using global**

```
a = 10

def modify_global():

    global a # Declaring that we are modifying the global 'a’

    a = 20

modify_global()

print(a) # Output: 20
```

- Using global is discouraged because it can make debugging harder. Instead, using **function arguments and return values** is preferred.

FILES AND THE OPERATING SYSTEM IN PYTHON

Files and the Operating System in Python

Python provides high-level tools like pandas.read_csv to read data files easily.

Understanding file handling basics is important for efficient data processing.

Python’s open() function is used to open files in different modes.

Opening and Reading Files

- **Opening a File**
  - open(path): Opens a file in read-only (r) mode by default.

```
path = './programs/SAMPLE.txt'

f = open(path)

Open a file in read mode

file = open("example.txt", "r")


# Read content

content = file.read()

print(content)

Output:

have a nice day

# Close the file

file.close()
```

Reading Files

- Iterate through lines in a file:

```
for line in f:
    pass          # Process each line
```

- Read all lines into a list:

```
lines = [x.rstrip() for x in open(path)]
```

Output:

```
['Sueña el rico en su riqueza,',
 'que más cuidados le ofrece;',
 '',
 'sueña el pobre que padece',
 'su miseria y su pobreza;',
 '']
```

•Use **iteration (for line in f:)** for large files.
•Use **list comprehension (lines = [...])** when you need process all lines at once and memory is not a concern.

Closing a File

- **Important**: Always close files after use to free resources.

```
f.close()
```

- **Using with Statement**

```
with open(path) as f:
    lines = [x.rstrip() for x in f]
# output:
have a nice day
```

Reading File Content

- **Using read() Method**
  - Reads a specific number of characters:

```
f = open("example.txt", "r")
```

```
f.read(10)
```

Output:

```
'have a nic'
```

Reading in Binary Mode

- To read a file in binary mode, use 'rb':

```
f2 = open(path, 'rb')

f2.read(10)

Output:

b'Sue\xc3\xb1a el '
```

Checking File Position

- **tell(): Returns the current file position**

```
f.tell() # Output: 11
```

- **seek(pos): Moves the file pointer to a specific position**

```
f.seek(3)
```

```
f.read(1) # Output: 'ñ'
```

- **Closing files:**

```
f.close()
```

```
f2.close()
```

Writing to Files

- To write text to a file:

```
with open('tmp.txt', 'w') as handle:
  handle.writelines(x for x in open(path) if len(x) > 1)
```

- **Reading the modified file:**

```
with open('tmp.txt') as f: lines = f.readlines()
```

- **Output:**

```
['Sueña el rico en su riqueza,\n',
'que más cuidados le ofrece;\n',
'sueña el pobre que padece\n']
```

Important File Methods

Method	Description
read([size])	Reads file content, with an optional size argument
readlines([size])	Returns a list of lines from the file, with an optional size argument
write(str)	Writes a string to the file
writelines(strings)	Writes a sequence of strings to the file
close()	Closes the file handle
flush()	Flushes the internal I/O buffer to disk
seek(pos)	Moves the file pointer to the specified position
tell()	Returns the current file position
closed	Returns True if the file is closed

Introduction to Errors and Exceptions

- Errors and exceptions occur during the execution of a program. Handling them properly is essential to ensure program robustness and avoid crashes.
- **Types of Errors:**

1. Syntax Errors:

1. Occur when Python cannot interpret code due to incorrect syntax.
2. Example

```
print("Hello  # Missing closing quote
```

output:

```
SyntaxError: EOL while scanning string literal
```

2. Exceptions:

- Occur during execution even if the syntax is correct.
- Example:

```
print(10 / 0)  # Division by zero
```

- Output

```
ZeroDivisionError: division by zero
```

Handling Exceptions using try-except

- We use a try-except block to catch and handle exceptions gracefully.
- **Example:** Handling ValueError when converting a string to float.

```
def attempt_float(x):
    try:
        return float(x)  # Try to convert x to float
    except ValueError:
        return x  # Return input if conversion fails

print(attempt_float("1.23"))   # 1.23
print(attempt_float("hello"))  # 'hello'
```

Catching Multiple Exceptions

- We can catch multiple exception types by specifying them in a tuple.
- **Example:** Handling TypeError and ValueError together.

```
def attempt_float(x):
    try:
        return float(x)
    except (TypeError, ValueError) as e:
        print(f"Error: {e}")  # Print the error message
        return None  # Return None to indicate failure

print(attempt_float((1, 2)))  # Error message + None
print(attempt_float("abc"))   # Error message + None
print(attempt_float("1.23"))  # 1.23 (Valid conversion)
```

4. Using finally and else Blocks

- **finally block:** Executes whether or not an exception occurs.
- **else block:** Executes only if the try block succeeds.
- **Example:** Ensuring file closure

```
try:
    f = open("example.txt", "w")
    f.write("Hello, World!")
except:
    print("Error occurred")
else:
    print("Write successful")
finally:
    f.close()  # Always executes
```

Debugging with IPython

- **%xmode magic command**: Controls traceback verbosity (Plain, Context, Verbose).
- **%debug or %pdb magic commands**: Allow interactive debugging after an error.
- **Example:** Assertion failure in IPython:

```
assert (5 + 6 == 10), "Assertion failed!”
```

```
Output:
```

```
AssertionError: Assertion failed!
```

Best Practices for Exception Handling

Always handle specific exceptions rather than using a general except: block.

Use logging to track exceptions instead of printing error messages.

Avoid suppressing exceptions that indicate critical failures.

Close resources (files, database connections) using finally or context managers (with statement).

Provide meaningful error messages to help debugging.

Example: Using logging instead of print

```
import logging


def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        logging.error("Attempted to divide by zero")
        return None
print(divide(10, 0))  # Logs an error
```