Chapter 4

Organizing Files and Debugging Programs

4.1 Organizing Files

Topics Covered: The shutil Module, Walking a Directory Tree, Compressing Files with the zipfile Module, Project: Renaming Files with American-Style Dates to European-Style Dates, Project: Backing Up a Folder into a ZIP File,

4.1.1 Organizing Files

Organizing files in Python involves managing files and directories (folders) on your computer's file system. Python provides several built-in modules to interact with the file system, making it easy to create, delete, move, and modify files and directories. The primary modules used for organizing files in Python are **os and shutil**.

os module

The os module provides a way to interact with the operating system's file system. It offers various functions to manipulate files and directories. Some of the commonly used functions include:

- **os.path.exists(path)**: Checks if a file or directory exists at the specified path.
- **os.path.isfile(path)**: Checks if the given path is a file.
- **os.path.isdir(path)**: Checks if the given path is a directory.
- **os.listdir(path):** Returns a list of files and directories in the specified directory.
- **os.mkdir(path):** Creates a new directory.
- **os.makedirs(path):** Creates intermediate directories if they don't exist while creating a new directory.
- **os.rename(src, dst):** Renames a file or directory from src to dst.
- **os.remove(path):** Removes (deletes) a file.
- **os.rmdir(path):** Removes an empty directory.
- **os.unlink(path):** Removes a file (alternative to os.remove()).

os.removedirs(path): Removes a directory and any empty parent directories.

Here's an example of creating a directory and a file using os module:

```
import os
# Create a directory
os.mkdir("my_directory")
# Create a file inside the directory
with open("my_directory/my_file.txt", "w") as file:
   file.write("Hello, this is a sample file.")
```

4.1.2 The Shutil Module

The shutil (shell utility) module in Python provides a higher-level interface for file operations. It extends the functionality of the os module and simplifies tasks like copying, moving, and removing files and directories. Let's explore some of the commonly used functions in the shutil module with examples:

Copying files

The shutil.copy() function is used to copy a file from the source to the destination location. It preserves the file's permissions but doesn't copy the metadata like timestamps.

Syntax: shutil.copy(src_file, dst_file)

Example : Copying file from one folder to the other

```
import shutil, os
os.chdir('C:\\Users\\thyagu\\')
shutil.copy('C:\\Users\\thyagu\\spam1.txt',
'C:\\Users\\thyagu\\delicious')
#Output
'C:\\Users\\thyagu\\delicious'
```

Example : Copying the contents of one file to the other

```
import shutil
shutil.copy('C:\\Users\\thyagu\\eggs.txt', 'C:\\Users\\thyagu\\eggs2.txt')

#Output
'C:\\Users\\thyagu\\eggs2.txt'
```

Copying directories: The shutil.copytree() function is used to recursively copy a directory and its contents from the source to the destination.

Example: Copy a directory and its contents

```
import shutil, os
os.chdir('C:\\Users\\thyagu\\18CS55\\')
shutil.copytree('C:\\Users\\thyagu\\18CS55\\bacon','C:\\Users\\thyagu\
\18CS55\\bacon_backup')
```

#Output:

```
'C:\\Users\\thyagu\\18CS55\\bacon_backup'
```

Moving files or directories: The shutil.move() function is used to move a file or a directory (and its contents) from the source to the destination.

Example

```
import shutil
shutil.move('C:\\Users\\thyagu\\one.txt','C:\\Users\\thyagu\\eggs')
```

#Output :

```
'C:\\Users\\thyagu\\eggs'
```

The destination path can also specify a filename.

Example

```
import shutil
shutil.move('C:\\Users\\thyagu\\one1.txt','C:\\Users\\thyagu\\eggs1\\newone.
txt')
```

#Output:

```
'C:\\Users\\thyagu\\18CS55\\eggs'
```

Removing directories: The shutil.rmtree() function is used to recursively remove a directory and all its contents. Be careful when using this function, as it permanently deletes data.

# Example: Remove a directory and its contents

```
import shutil
directory_to_remove = "folder_to_remove"
shutil.rmtree(directory_to_remove)
```

These are some of the commonly used functions in the shutil module. Remember to handle file and directory paths carefully and make sure you have proper permissions to perform these operations. Always take precautions and backup important data before performing any file operations.

4.1.3 Walking a Directory Tree

Walking a directory tree in Python refers to the process of traversing through a directory and its subdirectories to perform operations on files or directories found within. The os module provides a function called **os.walk()** that allows you to walk through a directory tree easily. It returns a generator that yields a tuple of the current directory's path, a list of subdirectories, and a list of files in that directory.

The basic syntax of **os.walk() is as follows**

```
import os
for root, directories, files in os.walk(top, topdown=True):
   # Do something with 'root' (current directory path)
   # Do something with 'directories' (list of subdirectories in 'root')
   # Do something with 'files' (list of files in 'root')
```

Example : Let's see an example of how to walk through a directory tree and print the names of all the files and directories found:

```
import os
for folderName, subfolders, filenames in os.walk('C:\\Users\\thyagu\\18CS55'):
   print('The current folder is ' + folderName)
   for subfolder in subfolders:
      print('SUBFOLDER OF ' + folderName + ': ' + subfolder)
   for filename in filenames:
       print('FILE INSIDE ' + folderName + ': '+ filename)
   print('')
```

# Output

```
The current folder is C:\Users\thyagu\18CS55
SUBFOLDER OF C:\Users\thyagu\18CS55: .ipynb_checkpoints
SUBFOLDER OF C:\Users\thyagu\18CS55: bacon
SUBFOLDER OF C:\Users\thyagu\18CS55: bacon_backup
SUBFOLDER OF C:\Users\thyagu\18CS55: eggs1
SUBFOLDER OF C:\Users\thyagu\18CS55: __pycache__
FILE INSIDE C:\Users\thyagu\18CS55: bacon.txt
FILE INSIDE C:\Users\thyagu\18CS55: capitalsquiz1.txt
FILE INSIDE C:\Users\thyagu\18CS55: capitalsquiz10.txt
FILE INSIDE C:\Users\thyagu\18CS55: capitalsquiz11.txt
FILE INSIDE C:\Users\thyagu\18CS55: capitalsquiz14.txt
```

4.1.4 Compressing Files with zipfile Module

Compressing files in Python can be easily achieved using the zipfile module, which provides functionalities for creating, reading, and extracting ZIP archives. The **zipfile** module is part of the Python standard library, so you don't need to install any additional packages to use it.

Here's a step-by-step guide on how to compress files using the zipfile module with examples:

Import the zipfile module

```
import zipfile
```

Create a new ZIP file and add files to it:
To compress files, you need to first create a new ZIP archive and then add the files you want to include in the compressed form. You can do this using the ZipFile class from the zipfile module.

```
# List of files you want to compress
files_to_compress = ['file1.txt', 'file2.txt', 'folder/file3.txt']

# Name of the ZIP archive you want to create
zip_file_name = 'compressed_files.zip'

# Create a new ZIP file and open it in write mode
with zipfile.ZipFile(zip_file_name, 'w') as zipf:
    for file in files_to_compress:
        # Add the file to the ZIP archive
        zipf.write(file)
```

In this example, we specified three files to compress: 'file1.txt', 'file2.txt', and 'folder/file3.txt'. The ZipFile class is used to create a new ZIP file named 'compressed_files.zip'. The files are then added to the ZIP archive using the write() method.

Example

```
import zipfile
newZip = zipfile.ZipFile('C:\\Users\\thyagu\\18CS55\\new.zip', 'w')
```

```
newZip.write('C:\\Users\\thyagu\\18CS55\\spamX.txt',
compress_type=zipfile.ZIP_DEFLATED)
newZip.close()
```

Compressing an entire directory: If you want to compress an entire directory and its contents, you can use the write() method with the arcname parameter.

```
import os

# Name of the directory to compress
directory_to_compress = 'my_directory'

# Name of the ZIP archive you want to create
zip_file_name = 'compressed_directory.zip'

# Create a new ZIP file and open it in write mode
with zipfile.ZipFile(zip_file_name, 'w') as zipf:
   # Walk through the directory and its subdirectories
   for folder, subfolders, files in os.walk(directory_to_compress):
      for file in files:
          # Get the full path of the file
         file_path = os.path.join(folder, file)
          # Compress the file, preserving the folder structure
         zipf.write(file_path, os.path.relpath(file_path, directory_to_compress))
```

In this example, we use os.walk() to iterate through the directory 'my_directory' and its subdirectories. Each file is added to the ZIP archive, and we use os.path.relpath() to preserve the directory structure within the ZIP file.

Reading Zip File

Reading a ZIP file in Python involves using the zipfile module to open the archive, extract its contents, and work with the files inside it. Let's go through a step-by-step explanation with an example:

Suppose you have a ZIP file named example.zip, and it contains two text files: file1.txt and file2.txt. Here's how you can read the contents of the ZIP file:

Import the zipfile module

```
import zipfile
```

Specify the ZIP file name:

```
zip_file_name = 'example.zip'
```

Open the ZIP file in read mode:

```
# Open the ZIP file in read mode
with zipfile.ZipFile(zip_file_name, 'r') as zipf:
   # Code to read the contents of the ZIP file goes here

Get the list of file names in the ZIP archive:
with zipfile.ZipFile(zip_file_name, 'r') as zipf:
   file_list = zipf.namelist()
print("Files in the ZIP archive:")
print(file_list)
```

Output:

```
Files in the ZIP archive:
['file1.txt', 'file2.txt']
```

The namelist() method returns a list of all the file names in the ZIP archive.

Read the contents of individual files:

```
with zipfile.ZipFile(zip_file_name, 'r') as zipf:
   for file_name in file_list:
      # Read the content of each file
      with zipf.open(file_name) as file_in_zip:
         content = file_in_zip.read()

          # Convert the content to a string for text files (optional)
          content_str = content.decode()

          # Print or process the content
         print(f"Contents of {file_name}:")
         print(content_str)
         print("=" * 40)
```

# Output:

```
Contents of file1.txt:
This is the content of file1.txt.

========================================
Contents of file2.txt:
This is the content of file2.txt.

========================================
```

In this example, we use the open() method of the ZipFile object to read the content of each file inside the ZIP archive. The read() method returns the file data as bytes. If the files are text-based, you can convert the bytes to a string using the .decode() method (as shown in the example above).

Close the ZIP file: Since we are using the with statement to open the ZIP file, it will automatically be closed at the end of the block.

Otherwise one can use close() function as follows: **exampleZip.close()**

4.1.5 Extracting files from a ZIP archive in Python can be done using the zipfile module.

The zipfile module provides the ZipFile class, which allows you to open and extract files from a ZIP archive. Here's how you can extract files:

Example

```
import zipfile, os
exampleZip = zipfile.ZipFile('C:\\Users\\thyagu\\18CS55\\example.zip')
exampleZip.extractall()
```

The extractall() method extracts all files and directories from the ZIP archive to the current working directory.

Extract specific files from the ZIP archive: If you want to extract specific files or directories, you can use the extract() method as illustrated below:

```
exampleZip.extract('spam.txt')
#Output
'C:\\Users\\thyagu\\18CS55\\spam.txt'
```

If you want to extract files to specific directories, you can use the extract() method as illustrated below:

```
exampleZip.extract('spam.txt', 'C:\\some\\new\\folders')
```

4.1.6 Project : Renaming Files with American – style Dates to European – Style Dates

To rename files with American-style dates to European-style dates, we need to identify the files with date patterns in the American format (MM-DD-YYYY) and then rename them to the European format (DD-MM-YYYY). We'll use regular expressions to find the date patterns and perform the renaming. Here's a Python program to achieve this:

```
import os
import re
from datetime import datetime

def american_to_european_date(date_str):
    # Convert American-style date (MM-DD-YYYY) to European-style date
(DD-MM-YYYY)
    date_obj = datetime.strptime(date_str, '%m-%d-%Y')
    return date_obj.strftime('%d-%m-%Y')

def rename_files_with_dates(directory_path):
    # Regular expression to match American-style dates (MM-DD-YYYY)
    date_pattern = r'\b(\d{2}-\d{2}-\d{4})\b'

    for root, dirs, files in os.walk(directory_path):
        for filename in files:
            file_path = os.path.join(root, filename)
            date_match = re.search(date_pattern, filename)

            if date_match:
                american_date = date_match.group(1)
                european_date = american_to_european_date(american_date)
                new_filename = filename.replace(american_date, european_date)

                # Rename the file
                os.rename(file_path, os.path.join(root, new_filename))
                print(f"Renamed '{filename}' to '{new_filename}'")

# Replace 'your_directory_path' with the path to the directory containing
the files
directory_path = 'your_directory_path'
rename_files_with_dates(directory_path)
```

4.1.7 Project: Backing up a Folder into a ZIP File

```
import os
import zipfile

def backup_folder_to_zip(folder_path, zip_file_name):
    # Check if the folder exists
    if not os.path.exists(folder_path):
        print(f"Error: Folder '{folder_path}' does not exist.")
        return

    # Create a new ZIP file
    with zipfile.ZipFile(zip_file_name, 'w') as zipf:
        # Walk through the folder and its subdirectories
        for foldername, subfolders, filenames in os.walk(folder_path):
            for filename in filenames:
                # Get the full path of the file
                file_path = os.path.join(foldername, filename)
                # Add the file to the ZIP archive
                zipf.write(file_path, os.path.relpath(file_path, folder_path))

    print(f"Backup successful. Folder '{folder_path}' has been backed up to
'{zip_file_name}'.")

# Replace 'your_folder_path' with the path to the folder you want to back
up
folder_to_backup = 'your_folder_path'

# Replace 'backup_folder.zip' with the desired name for the ZIP file
backup_file_name = 'backup_folder.zip'

backup_folder_to_zip(folder_to_backup, backup_file_name)
```

4.2 Debugging

Topics Covered: Raising Exceptions, Getting the Traceback as a String, Assertions, Logging, IDLE"s Debugger.

4.2.1 Introduction

Debugging is the process of identifying and resolving errors, defects, or issues in software programs or computer systems. It is an essential part of the software development lifecycle and involves locating and fixing problems that prevent the program from functioning correctly or as intended.

Common debugging tools include integrated development environments (IDEs) with debugging features, code profilers, and log analyzers. Additionally, developers may use techniques like step-by-step execution, adding temporary print statements (often called "print debugging"), or employing specialized debugging libraries.
In Python, debugging refers to the process of identifying and resolving errors or bugs in Python code. Python provides several tools and techniques to help developers debug their programs effectively.

Some of the commonly used debugging methods in Python include:

Print Statements: Inserting print statements in the code to display the values of variables and messages at different points in the program to track its flow and identify issues.

assert Statement: Using Python's assert statement to add assertions to your code, which checks if certain conditions are true during runtime and raises an AssertionError if not.

pdb (Python Debugger): The built-in interactive debugger in Python, accessed via the pdb module. It allows you to pause code execution, inspect variables, and step through the code line-by-line.

Logging: Utilizing Python's logging module to create log messages with varying levels of severity, enabling you to trace the program's execution and identify where issues occur.

Exception Handling: Using try, except, and finally blocks to catch and handle exceptions that may occur during the execution of your code, preventing program crashes and providing informative error messages.

IDE Debugging: Integrated Development Environments (IDEs) like PyCharm, Visual Studio Code, and others offer built-in debugging features. These IDEs allow you to set breakpoints, inspect variables, and step through your code while it's running.

External Debugging Tools: Exploring external tools and libraries like pdb++, ipdb, and py-spy, which offer advanced debugging capabilities, enhanced interfaces, and profiling for running Python programs.

traceback Module: Using Python's traceback module to obtain and manipulate the traceback information of exceptions as strings.

Debugging Decorators: Creating custom debugging decorators to wrap functions and print information about function calls, arguments, and return values.

Visual Debugging: Some libraries like PdbV or qdb allow for visual debugging, offering a more interactive and graphical debugging experience.

Code Linters: Using code linters like pylint or flake8 to statically analyze the code and identify potential bugs, style violations, and other issues.

Unit Testing: Writing unit tests with Python's unittest or third-party libraries like pytest to validate the behavior of individual code units and catch errors early.

Profiling: Utilizing Python's built-in cProfile or external profiling tools like line_profiler and memory_profiler to analyze the performance of code and identify bottlenecks.

Debugger Launchers: Some IDEs and tools provide debugger launchers that facilitate running scripts in debugging mode directly from the command line.

Raise Statement: Manually using the raise statement to raise exceptions at specific points in the code for debugging purposes and to test error-handling mechanisms.

4.2.2 Raising Exceptions

Raising exceptions in Python allows you to intentionally trigger and handle errors programmatically. It allows you to signal that something unexpected or erroneous has occurred in your code and provides a way
to handle such situations gracefully.

Python raises an exception whenever it tries to execute invalid code.
Exceptions are raised with a raise statement.

In Python, you can raise exceptions using the raise statement. When you raise an exception, you can include an optional error message or an instance of an exception class to provide more information about the error. Here's the basic syntax of the raise statement:

```
raise [ExceptionType("Error message")]
```

In code, a raise statement consists of the following:

- The **raise** keyword
- A call to the **Exception()** function
- A string with a helpful **error message** passed to the Exception() function

For **example,** enter the following into the interactive shell.

raise Exception('This is the unspecific error message.')

# Output

```
---------------------------------------------------------------------------
Exception                        Traceback (most recent call last)
Cell In[1], line 1
----> 1 raise Exception('This is the unspecific error message.')

Exception: This is the unspecific error message.
```

Example: Let's go through another example to illustrate how to raise exceptions in Python:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

try:
    numerator = 10
    denominator = 0

    result = divide(numerator, denominator)
    print("Result:", result)

except ValueError as e:
    print("An error occurred:", e)
```

In the above example, we have a function called **divide(a, b)**, which takes two arguments, a and b, and returns the result of dividing a by b. However, before performing the division, we check if b is equal to zero. If b is zero, we raise a ValueError with the message "**Cannot divide by zero**."

In the try block, we call the divide function with numerator set to 10 and denominator set to 0. Since the denominator is zero, the divide function will raise a ValueError.

The except block catches the raised ValueError, and we print the error message provided in the exception ("Cannot divide by zero.").

When you run this code, you'll see the following output:

An error occurred: Cannot divide by zero.

The raised exception allows us to handle the error in a controlled manner, preventing the program from crashing and providing useful information about what went wrong. Raising exceptions is particularly useful when you want to validate input data, enforce certain conditions, or handle exceptional cases in your code.

Another Example

```
def printBox(symbol, width, height):
    if len(symbol) != 1:
        raise Exception('Symbol must be a single character string.')
    if width <= 2:
        raise Exception('Width must be greater than 2.')
    if height <= 2:
        raise Exception('Height must be greater than 2.')

    # Printing the top edge of the box
    print(symbol * width)

    # Printing the sides of the box
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)

    # Printing the bottom edge of the box
    print(symbol * width)

# Testing the function with different arguments
```

```
for sym, w, h in (('*', 4, 4), ('O', 20, 5), ('x', 1, 3), ('ZZ', 3, 3)):
    try:
        printBox(sym, w, h)
    except Exception as err:
        print('An exception happened: ' + str(err))
```

Explanation

- The function **printBox(symbol, width, height)** is defined to print a box made of the specified symbol character, with the given width and height.
- It first checks the validity of the symbol, ensuring it is a single-character string using the condition **len(symbol) != 1**. If the symbol is not a single character, it raises an Exception with the message "Symbol must be a single character string."
- Next, it checks the validity of the width and height inputs, ensuring both are greater than 2. If any of them are not valid, it raises respective exceptions with appropriate error messages.
- If all the input validations pass, the function proceeds to print the box pattern. It starts by printing a line of symbol characters with a length of width, representing the top edge of the box.
- Next, the function uses a loop to print height - 2 lines, which make up the sides of the box. Each side consists of the symbol character, followed by spaces (equal to width - 2), and again the symbol character.
- Finally, the function prints another line of symbol characters with a length of width, representing the bottom edge of the box.
- The program tests the printBox function with different sets of arguments using the for loop. For each call, it wraps the function call in a try-except block to catch any raised exceptions.
- The output of the program will be printed for each set of arguments. It will display the box made of symbols, and if any exceptions occur during execution, the error messages will be printed accordingly.

#Output:

```
****
*  *
*  *
****
oooooooooooooooooooo
o          o
o          o
o          o
oooooooooooooooooooo
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

4.2.3 Getting the Traceback as a string

The traceback can be obtained it as a string by calling traceback.format_exc(). Instead of crashing the program right when an exception occurs, the traceback information can be written to a log file and keep the program running as illustrated below:

```
import traceback
try:
   raise Exception('This is the error message.')
except:
   errorFile = open('Log_ErrorInfo.txt', 'w')
   errorFile.write(traceback.format_exc())
   errorFile.close()
   print('The traceback info was written to Log_ErrorInfo.txt.')
```

#Output:

```
The traceback info was written to Log_ErrorInfo.txt.
```

4.2.4 Assertions

Assertions in Python are statements used to check if a given condition holds true during the execution of a program. They are primarily used for debugging and development purposes to detect programming errors and logic flaws early in the code. When an assertion fails (the condition evaluates to False), Python raises an AssertionError exception.

The syntax for an assertion is as follows:

```
assert condition, message
```

Here, condition is the expression that should be True, and message is an optional string that can be used to provide additional information about the assertion.

In code, an assert statement is composed of the following components:

- The **assert** keyword
- A **condition** (that is, an expression that evaluates to True or False)
- A **comma**
- A **string** to display when the condition is False

Let's see an example to understand how assertions work:

```
def divide(a, b):
    assert b != 0, "Denominator cannot be zero."
    result = a / b
    return result

try:
    numerator = 10
    denominator = 0

    result = divide(numerator, denominator)
    print("Result:", result)

except AssertionError as e:
    print("An assertion error occurred:", e)
```

In this example, we have a function called **divide(a, b)**, which takes two arguments, a and b, and returns the result of dividing a by b. Before performing the division, we have an assertion assert **b != 0**, "**Denominator cannot be zero**.", which checks if b is not equal to zero.

Now, let's consider two scenarios:

When denominator is not zero

If the denominator is not zero, the assertion passes (evaluates to True), and the function proceeds to perform the division. The result is printed as expected.

When denominator is zero

If the denominator is zero, the assertion fails (evaluates to False). Python raises an AssertionError exception with the message "Denominator cannot be zero." Since we have a try-except block, the except block catches the exception, and we print the error message provided in the assertion.

When you run this code with a zero denominator, the output will be**:**

```
An assertion error occurred: Denominator cannot be zero.
```

The use of assertions is most suitable for situations where you want to check if certain conditions are true during runtime, such as verifying the validity of function arguments, input data, or intermediate values during debugging. However, it's essential to use assertions judiciously and primarily for debugging purposes, as they can be disabled globally in Python using the -O (optimize) command-line switch, potentially affecting the behavior of the program.

# Example: Let us consider another example as illustrated below

```
Door = 'open'
assert Door == 'open', 'The house doors need to be "open".'
Door = 'I am sorry, Dave. I am afraid I cant do that.'
assert Door == 'open', 'The pod bay doors need to be "open".'
```

In the given code, there are two assert statements used to check the value of the variable Door. An assert statement is used to test conditions that should be true during the execution of the code. If the condition evaluates to False, Python raises an AssertionError with an optional error message.

Let's break down the code and understand the output:

```
Door = 'open'
assert Door == 'open', 'The house doors need to be "open".'
```

In this part of the code, the variable Door is assigned the value 'open'. The first assert statement checks if Door is equal to 'open'. Since the condition is True, the assertion passes, and there is no error raised.

```
Door = 'I am sorry, Dave. I am afraid I cant do that.'
assert Door == 'open', 'The pod bay doors need to be "open".'
```

In this part of the code, the variable Door is re-assigned a different value 'I am sorry, Dave. I am afraid I can’t do that.'. The second assert statement checks if Door is equal to 'open'. However, the condition is False because Door is now equal to 'I am sorry, Dave. I am afraid I can’t do that.'. This causes the assert statement to raise an AssertionError with the error message 'The pod bay doors need to be "open".'.

Output:

```
---------------------------------------------------------------------------
AssertionError                        Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_5972\3107636341.py in <module>
     2 assert Door == 'open', 'The house doors need to be "open".'
     3 Door = 'I am sorry, Dave. I am afraid I cant do that.'
----> 4 assert Door == 'open', 'The pod bay doors need to be "open".'
AssertionError: The pod bay doors need to be "open".
```

As we can see in the output, the AssertionError is raised with the provided error message 'The pod bay doors need to be "open".', indicating that the value of Door is not as expected (i.e., not equal to 'open').

Another Example ***Using an Assertion in a Traffic Light Simulation***

Let's create a simple traffic light simulation using assertions in Python. In this simulation, we'll model a basic traffic light with three states: red, yellow, and green. The light will change from red to green, then to yellow, and back to red in a loop.

We'll use assertions to ensure that the traffic light is always in a valid state and that the transitions follow the expected sequence.

```
def change_state(current_state):
        if current_state == "RED":
            return "GREEN"
        elif current_state == "GREEN":
            return "YELLOW"
        elif current_state == "YELLOW":
            return "RED"
def simulate_traffic_light(num_iterations):
        traffic_light = {'ns': 'GREEN', 'ew': 'RED'}
        for _ in range(num_iterations):
            ns_state = traffic_light['ns']
            ew_state = traffic_light['ew']
            # Assertions to check the validity of traffic light states.
            assert ns_state in ["RED", "GREEN", "YELLOW"], "Invalid state:
        " + ns_state
            assert ew_state in ["RED", "GREEN", "YELLOW"], "Invalid
        state: " + ew_state
            print("Current states: NS - {}, EW - {}".format(ns_state,
        ew_state))
            traffic_light['ns'] = change_state(ns_state)
            traffic_light['ew'] = change_state(ew_state)
num_iterations = 10
simulate_traffic_light(num_iterations)
```

4.2.5 Disabling Assertions

In Python, you can disable assertions globally by using the -O (optimize) command-line switch or the PYTHONOPTIMIZE environment variable. When you run the Python interpreter with the -O switch, all assertions in the code will be ignored, and they will not have any effect on the program's execution. This is useful when you want to disable assertions in a production environment for performance reasons.

Here's how to disable assertions using the -O switch:

```
python -O your_script.py
```

Alternatively, you can set the PYTHONOPTIMIZE environment variable to any non-empty value before running the Python script:
export PYTHONOPTIMIZE=1

```
python your_script.py
```

If assertions are disabled, any assert statements in the code will be ignored, and the program will run without raising any AssertionError regardless of the conditions specified in the assertions.

4.2.6 Logging

Logging is a way to understand what is happening in your program and in what order it is happening. Python’s logging module makes it easy to create a record of custom messages.

Using the logging Module

To configure the logging module to exhibit log messages on your screen during the execution of your program, duplicate the following code snippet at the beginning of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

Example

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
(levelname)s-  (message) s')
logging.debug('Start of program')
def factorial(n):
   logging.debug('Start of factorial(%s%%)' % (n))
   total = 1
   for i in range(1, n + 1):
      total *= i
      logging.debug('i is ' + str(i) + ', total is ' + str(total))
   logging.debug('End of factorial(%s%%)' % (n))
   return total

print(factorial(5))
logging.debug('End of program')
```

# Output

```
2023-03-05 11:51:12,779 - DEBUG- Start of program
2023-03-05 11:51:12,795 - DEBUG- Start of factorial(5%)
2023-03-05 11:51:12,795 - DEBUG- i is 1, total is 1
2023-03-05 11:51:12,810 - DEBUG- i is 2, total is 2
2023-03-05 11:51:12,810 - DEBUG- i is 3, total is 6
2023-03-05 11:51:12,810 - DEBUG- i is 4, total is 24
2023-03-05 11:51:12,810 - DEBUG- i is 5, total is 120
2023-03-05 11:51:12,810 - DEBUG- End of factorial(5%)
2023-03-05 11:51:12,810 - DEBUG- End of program
```

4.2.7 Logging Levels

In Python's logging module, log messages can be categorized into different levels based on their importance and severity. These log levels help control which messages are displayed or recorded based on the current logging configuration. There are **five standard logging levels**, listed here in increasing order of severity:

Debug: The lowest level of severity. These log messages are used for debugging and are typically used during development to track the program's flow, variable values, and other detailed information. They are usually not shown in production environments.

Info: This level provides informational messages that indicate the program's progress or certain events. INFO messages are less detailed than DEBUG messages but provide relevant information about the program's execution. They are suitable for production environments to give insights into the program's operations.

Warning: Indicates a potential issue or a situation that could lead to an error but does not halt the program's execution. Warnings are used to alert developers to potential problems that might need attention.

Error: Indicates an error that occurred during program execution, but the program can continue running. Errors are used to report issues that might impact the program's functionality.

Critical: The highest level of severity. Indicates a critical error or problem that may prevent the program from continuing its execution. CRITICAL messages typically prompt immediate action to address the issue.

When configuring the logging module, you can set the logging level, and only messages with a severity level equal to or higher than the configured level will be logged. For example, setting the logging level to INFO will log messages of level INFO, WARNING, ERROR, and CRITICAL, but not DEBUG messages.

Table 4.1:

Level	Logging Function	Description
DEBUG	logging.debug()	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	logging.info()	Used to record information on general events in your program or confirm that things are working at their point in the program.

WARNING	logging.warning()	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	logging.error()	Used to record an error that caused the program to fail to do something.
CRITICAL	logging.critical()	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Table Source : Automating the Boring stuff by AlSwigart [https://automatet =heb oringstuff.com/]

Here's an example of setting the logging level:

```
import logging

# Set the logging level to INFO
logging.basicConfig(level=logging.INFO)

# Log messages
logging.debug("This will not be displayed.")
logging.info("This is an informational message.")
logging.warning("This is a warning.")
logging.error("This is an error.")
logging.critical("This is a critical message.")
```

In this example, only the messages with levels INFO, WARNING, ERROR, and CRITICAL will be displayed or recorded, while the DEBUG message will be ignored because its severity is lower than the configured logging level (INFO).

4.2.8 Logging to a File

Using logging. basicConfig() function, the log messages can be written to a text file as illustrated below:

```
import logging
logging.basicConfig(filename='logx.txt', level=logging.DEBUG,
            format='%(asctime)s - %(levelname)s - %(message)s')
```

Logging to a file in Python involves directing log messages to a file instead of printing them to the console. This is useful for recording logs over time and for persisting them for later analysis. The logging module in Python provides built-in support for logging to files. Here's an example:

```
import logging
# Configure logging to a file
logging. basicConfig(
   level=logging.INFO,
   format='%(asctime)s - %(levelname)s - %(message)s',
   filename='example.log',
   filemode='w'
)

# Log messages
logging.debug("This is a debug message.")
logging.info("This is an informational message.")
logging.warning("This is a warning.")
logging.error("This is an error.")
logging.critical("This is a critical message.")
```

In this example, we configure the logging module to write log messages to a file named example.log. The basicConfig function is used to set up the logging configuration.

4.2.9 Parameters used in basicConfig

level=logging.INFO: This sets the logging level to INFO. It means only messages with a severity level of INFO or higher will be recorded in the file.

format='%(asctime)s - %(levelname)s - %(message)s': This specifies the format of the log messages. In this case, we include the timestamp, log level, and message in each log entry.

filename='example.log': This sets the name of the log file. The logs will be written to this file.

filemode='w': This sets the file mode to 'write'. It means that the log file will be created anew each time the program runs, and any existing contents in the file will be cleared. If you want to append new logs to an existing file, you can use 'a' instead.

After configuring logging, we log messages at different levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) using logging.debug, logging.info, logging.warning, logging.error, and logging.critical respectively. These log messages will be written to the example.log file in the specified format.

When you run the code, you will see that the log file example.log is created, and the log messages are written to the file. Each line in the log file will contain the timestamp, log level, and the logged message, making it easier to analyze and debug the program's behavior over time.

4.2.10 IDLE’s Debugger

IDLE, the Integrated Development and Learning Environment for Python, includes a built-in debugger that allows developers to interactively debug their Python code. The debugger helps identify and resolve issues in the code by allowing step-by-step execution, examining variable values, setting breakpoints, and more. Let's see how to use IDLE's Debugger with an example:

Suppose we have the following Python code in a file named example.py:

```
def factorial(n):
   if n == 0:
      return 1
   else:
      return n * factorial(n - 1)
result = factorial(5)
print("Factorial of 5 is:", result)
```

To use IDLE's Debugger:

Open IDLE and load the example.py script.

Go to the "Debug" menu and select "Debugger" or use the keyboard shortcut **Ctrl+F5**.

The Debugger window will open, showing the source code with a line highlighted in yellow. This indicates the next line to be executed.

Click on the "**Step**" button or use the keyboard shortcut F7 to execute the current highlighted line.

You will see that the debugger stops at the factorial(5) function call. Hover over the variable n to see its current value (which is 5).

Click on the "**Step**" button again, and you will enter the factorial function.

The debugger stops at the if statement. Hover over the variable n to see its value (5).

Click on the "**Step**" button again, and you will execute the return n * factorial(n - 1) line.

The debugger will recursively call the factorial function with n-1. Continue clicking on the "**Step**" button to see how the recursion works.

The debugger will stop at the base case when n becomes 0. Hover over the variable n to see its value (0).
Click on the "**Step**" button again, and the factorial function returns 1.

The **debugger** will go back to the previous call of the factorial function, where n is 1, and calculate 1 * factorial(0).

Finally, the **debugger** will return to the main script, and the result variable will hold the value of the factorial of 5.

Using IDLE's Debugger, you can visualize the flow of execution, examine variable values at each step, and understand how your code behaves. It's a powerful tool for finding and fixing issues in your Python programs.