Python Modules, packages and File Handling.

Lesson Plan

PW SKILLS

Prerequisites:

- Basic understanding of Python programming language.
- Familiarity with functions, variables, and control flow in Python.

1. Introduction to Modules:

Definition:

A module in Python is a file containing Python code. This code can consist of functions, classes, and variables. Modules serve as a way to organize Python code into reusable units. They help in better code management and reusability.

Importance of Modular Programming:

Modular programming is a software design technique where functionality is divided into small, independent, and interchangeable modules. This approach offers several benefits such as code reusability, easier maintenance, and better organization. In Python, modules play a crucial role in achieving modularity.

Creating a Simple Module:

```
# example_module.py
def greet(name):
    print("Hello, " + name)

def add(x, y):
    return x + y

pi = 3.14159
```

import Statement:

The import statement in Python is used to import modules into your code. It allows you to access the functionality defined in the module. There are several ways to import modules, including importing the entire module, importing specific functions or variables, and aliasing imports.

```
# importing the entire module
import example_module

# importing specific functions or variables
from example_module import greet, pi

# aliasing imports
import example_module as ex
```

Hierarchy of Module Import:

When Python imports a module, it searches for the module in a specific order:

1. The current directory.
2. The directories listed in the PYTHONPATH environment variable.
3. The standard library directories.

Examples of Built-in Modules:

Python comes with a rich standard library that includes many useful built-in modules. Some common examples include math, random, os, sys, and datetime.

2. Creating and Using Modules:

Importing Modules:

We should understand different ways of importing modules:

- Using the import statement to import the entire module.
- Using from ... import ... to import specific functions or variables.
- Aliasing imported modules or functions to avoid name conflicts.

Explaining the __name__ Variable:

The __name__ variable in Python holds the name of the current module. It is useful for determining whether a module is being run as a standalone program or being imported into another module.

```
# example_module.py

def greet(name):
    print("Hello, " + name)

if __name__ == "__main__":
    greet("World")
```

In this example, if the module is run directly (python example_module.py), it will execute the code inside the if __name__ == "__main__": block. If it is imported into another module, that block will not execute.

3. Introduction to Packages:

Definition:

A package in Python is a directory containing one or more modules, along with an __init__.py file. This file can be empty or can contain initialization code for the package. Packages provide a hierarchical structure for organizing and distributing Python code.

Comparison between Modules and Packages:
While modules are individual Python files containing code, packages are directories containing multiple modules. Packages offer a more organized and structured way of managing code, especially in large projects where codebase can become complex.

Package Structure:
A Python package typically has the following structure:

```
my_package/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init__.py
        submodule1.py
        submodule2.py
```

The __init__.py files indicate to Python that the directories should be treated as packages. These files can be empty, or they can contain initialization code that runs when the package is imported.

Examples of Popular Python Packages:

Let's take the example of the requests package, which is widely used for making HTTP requests in Python:

pip install requests

Once installed, you can import and use the requests module in your Python code:

import requests

response = requests.get("https://www.example.com")
print(response.status_code)

In this example, requests is a package, and get() is a function from the requests module used to make an HTTP GET request.

4. Creating and Using Packages:

Example of creating and using a package

Let's create a simple package called my_package with two modules: module1.py and module2.py.

```
my_package/
    __init__.py
    module1.py
    module2.py
```

module1.py

```
def greet(name):
    print(f"Hello, {name}!")

def square(x):
    return x ** 2
```

module2.py

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

init.py

```
# Empty __init__.py file
```

Now, to use this package, you can import it in another Python script:
from my_package

```
import module1, module2

module1.greet("Alice")
print(module1.square(5))

print(module2.add(3, 4))
print(module2.subtract(8, 5))
```

Importing Packages:

To import modules from packages, you can use absolute or relative imports:# Absolute import
import my_package.module1

```
# Relative import
from . import module2

For nested packages, you can use dot notation:

from my_package.subpackage import submodule1
```

File Handling

1. Introduction to File Handling:

Definition:
File handling in Python refers to operations performed on files, including reading from and writing to files. Files can be of different types such as text files (containing readable characters) or binary files (containing encoded data).

Importance of File Handling:
File handling is essential for many real-world applications where data needs to be stored, retrieved, and manipulated. It allows Python programs to interact with files on the system, enabling tasks like data storage, logging, and configuration management.

File Modes:
File modes determine how a file is opened and whether it can be read from, written to, or both. Common file modes include:

- 'r': Open file for reading (default).
- 'w': Open file for writing. Creates a new file or overwrites existing content.
- 'a': Open file for appending. Adds new content to the end of the file.
- 'b': Open file in binary mode (e.g., 'rb', 'wb', 'ab').

Opening and Closing Files:

To open a file, Python provides the open() function. It takes two arguments: the file path and the mode. It returns a file object that can be used to perform file operations. It's important to close files after usage to release system resources.

```
# Open a file in read mode
file = open('example.txt', 'r')
# Perform operations on the file
# Close the file
file.close()
```

File Pointers:

A file object maintains a current position called the file pointer, which points to the next byte to be read or written. When a file is opened, the file pointer is initially positioned at the beginning of the file. It moves as data is read from or written to the file.

2. Reading from Files:

Reading Entire File Contents:

To read the entire contents of a file, you can use the read() method. It reads the file from the current file pointer position until the end of the file.

```
file = open('example.txt', 'r')
content = file.read()
print(content)
file.close()
```

Reading Line by Line:

The readline() method reads a single line from the file, while the readlines() method reads all lines and returns them as a list.

```
file = open('example.txt', 'r')
line1 = file.readline()
print(line1)
lines = file.readlines()
print(lines)
file.close()
```

Iterating Over File Objects:

You can directly iterate over a file object in a for loop to read the file line by line.

```
file = open('example.txt', 'r')
for line in file:
    print(line)
file.close()
```

Handling File Exceptions:

File operations can raise exceptions like FileNotFoundError if the file does not exist or PermissionError if the file cannot be accessed due to permission issues. It's important to handle these exceptions gracefully.

```
try:
    file = open('example.txt', 'r')
    content = file.read()
    print(content)
    file.close()
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
```

3. Writing to Files:

Writing to Files:

To write data to a file, you can use the write() method. It writes the specified string to the file at the current file pointer position.

```
file = open('example.txt', 'w')
file.write("Hello, world!\n")
file.write("This is a new line.")
file.close()
```

Writing Multiple Lines:

The writelines() method writes a list of strings to the file. Each string is written as a separate line.

```
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
file = open('example.txt', 'w')
file.writelines(lines)
file.close()
```

Appending to Files:

To append data to the end of a file, you can open the file in append mode ('a'). It ensures that new data is added to the existing content without overwriting it.

```
file = open('example.txt', 'a')
file.write("This is appended content.")
file.close()
```

File Buffering and Flushing:

File buffering refers to the process of temporarily storing data in memory before writing it to the file. File objects are automatically buffered for efficiency. You can manually flush the buffer using the flush() method to ensure that all buffered data is written to the file.

```
file = open('example.txt', 'w')
file.write("This is some data.")
file.flush()  # Flush buffer to ensure data is written
immediately
file.close()
```

Best Practices for Error Handling and File Closing:

It's good practice to use try-except-finally blocks for error handling when working with files. This ensures that files are closed even if an error occurs during file operations.

```
try:
    file = open('example.txt', 'r')
    content = file.read()
    print(content)
except FileNotFoundError:
    print("File not found!")
finally:
    file.close()
```

4. Working with File Paths and Directories:

Introduction to File Paths:

File paths represent the location of files or directories on the filesystem. They can be absolute (starting from the root directory) or relative (starting from the current directory).

import os

```
# Joining file paths using os.path.join()
path = os.path.join('folder', 'file.txt')
print(path)
```

Using the os Module for File Operations:

The os module provides various functions for working with files and directories, such as os.listdir() to list directory contents and os.path.exists() to check if a file or directory exists.

```
import os

# List contents of a directory
contents = os.listdir('.')
print(contents)

# Check if a file exists
print(os.path.exists('example.txt'))
```

Creating, Renaming, and Deleting Files and Directories:

The os module also provides functions for creating, renaming, and deleting files and directories.

```
import os

# Create a directory
os.mkdir('new_folder')

# Rename a file
os.rename('old_file.txt', 'new_file.txt')

# Delete a file
os.remove('file_to_delete.txt')

# Delete an empty directory
os.rmdir('empty_directory')
```

Handling File Path Manipulations:

The os.path module provides functions for manipulating file paths, such as checking if a path is a file (os.path.isfile()) or a directory (os.path.isdir()).

```
import os

# Check if a path is a file
print(os.path.isfile('example.txt'))

# Check if a path is a directory
print(os.path.isdir('folder'))
```