**Logger and Python Packages**

* **Logger**

In Python, logging is an essential feature for tracking events that happen when some software runs. The logging module in Python is a standard module that provides a flexible framework for emitting log messages from Python programs.

Below is a basic guide to using the logging module:

**Basic Usage**

* 1. **Import the logging module:**
     + import logging
  2. **Configure the logging:**

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

datefmt='%Y-%m-%d %H:%M:%S')

* 1. **Create a logger:**
     + logger = logging.getLogger(\_\_name\_\_)
  2. **Generate log messages:**
     + logger.debug('This is a debug message')
     + logger.info('This is an info message')
     + logger.warning('This is a warning message')
     + logger.error('This is an error message')
     + logger.critical('This is a critical message')

### **Log Levels**

The logging module defines the following standard log levels:

1. **Logging.DEBUG:**

Detailed information, typically of interest only when diagnosing problems.

1. **Logging.INFO:**

Confirmation that things are working as expected.

1. **Logging.WARNING:**

An indication that something unexpected happened, or indicative of some problem in the near future (e.g., ‘disk space low’). The software is still working as expected.

1. **Logging.ERROR:**

Due to a more serious problem, the software has not been able to perform some functions.

1. **Logging.CRITICAL:**

A serious error indicating the program may be unable to continue running.

By using the **logging** module, you can create a flexible logging system that can be configured to output messages to various destinations and with different levels of detail.

**Example**

import logging

# Configure logging

logging.basicConfig(level=logging.DEBUG,

format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

datefmt='%Y-%m-%d %H:%M:%S')

# Create a logger

logger = logging.getLogger(\_\_name\_\_)

def divide(a, b):

try:

result = a / b

logger.info(f"Divided {a} by {b} successfully. Result: {result}")

return result

except ZeroDivisionError:

logger.error("Attempted to divide by zero")

return None

# Test the function and logging

divide(10, 2)

divide(10, 0)

**Python Packages**

* **OS Module:**

The os module provides a way of using operating system dependent functionality like reading or writing to the file system.

**Basic Usage**

* 1. **Import os:**

import os

* 1. **Environment Variables:**

# Get an environment variable

home = os.getenv('HOME')

# Set an environment variable

os.environ['MY\_VAR'] = 'my\_value'

* 1. **File and Directory Operations:**

# Get current working directory

current\_dir = os.getcwd()

# Change working directory

os.chdir('/path/to/directory')

# Create a new directory

os.makedirs('new\_directory/sub\_directory', exist\_ok=True)

# List contents of a directory

contents = os.listdir('.')

# Check if a path exists

os.path.exists('path/to/check')

# Check if it is a file or directory

os.path.isfile('path/to/file')

os.path.isdir('path/to/directory')

**Example**

import os

# Get the current working directory

current\_dir = os.getcwd()

print(f"Current directory: {current\_dir}")

# Create a new directory

new\_dir = os.path.join(current\_dir, 'new\_folder')

os.makedirs(new\_dir, exist\_ok=True)

# Create a new file in the new directory

new\_file\_path = os.path.join(new\_dir, 'new\_file.txt')

with open(new\_file\_path, 'w') as file:

file.write('Hello, os module!')

# Read the content of the new file

with open(new\_file\_path, 'r') as file:

content = file.read()

print(content)

# List contents of the new directory

contents = os.listdir(new\_dir)

print(f"Contents of {new\_dir}: {contents}")

* **Pathlib Module:**

pathlib offers an object-oriented approach to handling filesystem paths. It provides classes for different kinds of paths (e.g., PosixPath for Unix-like systems and WindowsPath for Windows).

**Basic Usage**

1. **Import pathlib:**

from pathlib import Path

1. **Creating Paths:**

# Creating a Path object

path = Path('example\_dir/example\_file.txt')

# Creating a Path object representing the home directory

home = Path.home()

1. **Basic Operations:**

# Check if a path exists

path.exists()

# Check if it is a file or directory

path.is\_file()

path.is\_dir()

# Create directories

new\_dir = Path('new\_directory/sub\_directory')

new\_dir.mkdir(parents=True, exist\_ok=True)

# Read text from a file

content = path.read\_text()

# Write text to a file

path.write\_text('Hello, World!')

1. **Iterating through directory contents:**

for file in Path('.').iterdir():

print(file)

**Example**

from pathlib import Path

# Create a Path object for the current directory

current\_dir = Path('.')

# Iterate through all files in the directory

for file in current\_dir.iterdir():

if file.is\_file():

print(f"File: {file.name}")

# Create a new directory

new\_dir = current\_dir / 'new\_folder'

new\_dir.mkdir(exist\_ok=True)

# Create a new file in the new directory

new\_file = new\_dir / 'new\_file.txt'

new\_file.write\_text('Hello, pathlib!')

# Read the content of the new file

print(new\_file.read\_text())

### **Comparison and Use Cases**

* **pathlib**:
  + Object-oriented and more intuitive.
  + Easier path manipulations.
  + Better suited for most file and directory operations in modern Python code.
* **os**:
  + Provides lower-level operating system functionality.
  + More comprehensive for certain tasks such as environment variable management and executing shell commands.
  + Useful for compatibility with older Python code.

Both modules can be used together depending on the task at hand. For most file path manipulations, pathlib is preferred, but for certain OS-specific tasks, the os module remains essential.

**Regular Expression**

**Basic Usage**

1. **Import the re module:**

import re

1. **Common Functions:**
   * **re.search(pattern, string):**

Searches for the first location where the regular expression pattern produces a match.

* + **re.match(pattern, string):**

Checks for a match only at the beginning of the string.

* + **re.findall(pattern, string):**

Finds all substrings where the pattern matches and returns them as a list.

* + **re.finditer(pattern, string):**

Finds all substrings where the pattern matches and returns them as an iterator.

* + **re.sub(pattern, repl, string):**

Replace the matches with a string.

**Example**

import re

pattern = r'\D\*' # Matches one or more digits

text = "There are 50 apples and 175 oranges."

# Search for the first occurrence

match = re.search(pattern, text)

if match:

print(f"Found a match: {match.group()}")

# Match only at the beginning

match = re.match(pattern, text)

if match:

print(f"Found a match at the beginning: {match.group()}")

else:

print("No match at the beginning.")

# Find all occurrences

matches = re.findall(pattern, text)

print(f"All matches: {matches}")

# Iterate over all matches

for match in re.finditer(pattern, text):

print(f"Match found: {match.group()} at position {match.start()} to {match.end()}")

pattern = r'\d+'

replacement = '#'

# Replace all matches

result = re.sub(pattern, replacement, text)

print(result)