PASADENA CITY COLLEGE

CIS 112

Intro to Programming Using Python

Module 6 Part 1

Agenda for the Day!

- communication:
  - os,
  - io,
  - time

Quick Note on the Final Exam

Threading

Threading is a technique used to achieve parallelism in software applications. It allows multiple threads of execution to run concurrently within a single process. Each thread can perform a separate task while sharing the same memory space, making it possible to perform multiple operations at once.

Key Concepts in Threading

1. Thread:
   - A thread is the smallest unit of execution within a process. Each thread runs in the context of the process and shares the process's resources, such as memory and file handles.
2. Main Thread:
   - The main thread is the initial thread that starts when a program begins. It can create additional threads for concurrent execution.
3. Concurrency vs. Parallelism:
   - Concurrency refers to the ability to handle multiple tasks at once but not necessarily simultaneously. It often involves interleaving the execution of tasks.
   - Parallelism involves executing multiple tasks simultaneously, which can be achieved on multi-core processors.
4. Multithreading:
   - Multithreading involves creating multiple threads within a single process. Each thread runs independently but can communicate and share data with other threads.

Blocking

Blocking occurs when a thread is unable to proceed with its execution because it is waiting for some condition to be met or for an operation to complete. During this waiting period, the thread is considered "blocked" and does not consume CPU resources.

Common Blocking Operations

1. I/O Operations:
   - Reading from or writing to files, network sockets, or other I/O devices can block a thread until the operation is complete.
2. Synchronization Primitives:
   - Using synchronization mechanisms like locks, semaphores, and condition variables can cause threads to block while waiting for a resource to become available.
3. Sleeping:
   - Deliberately pausing a thread's execution using functions like time.sleep() causes the thread to block for a specified duration.

The Inmates are Running the Asylum!

- In addition to accessing files directly, we sometimes need to access root operating system functions from within our program
  - The **OS** library is a powerful toolkit to run a variety of operating system level functions directly from a python program
- We can use it to create custom file paths, access file metadata, remove files from hard-drives etc.

I'm comin' out of the lake and I've been doin' just fine, gotta gotta get out because I wanna walk.
Started out as a fish, how did it end up like this?
I was only a fish, I was only a fish.

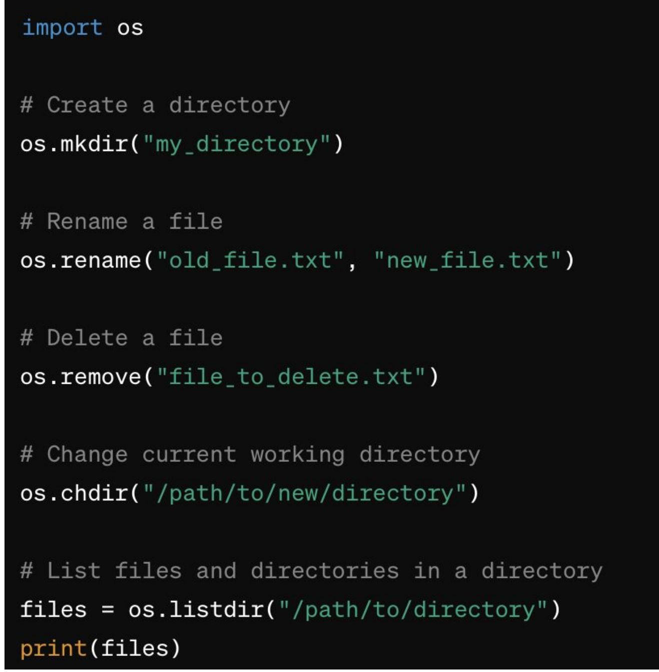
OS Module

OS me!

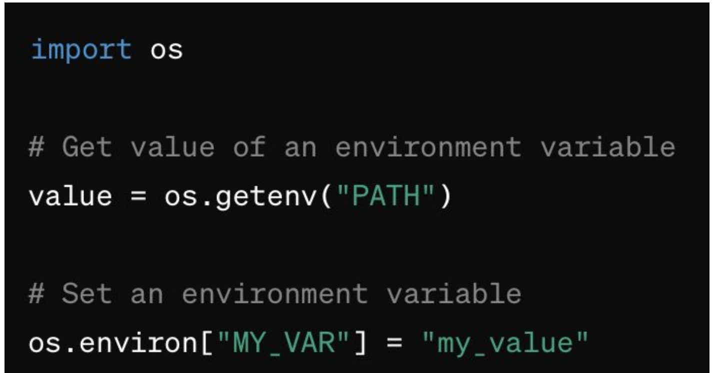
- The OS module in Python provides functions for interacting with the operating system.
- OS comes under Python’s standard utility modules. This module provides a portable way of using operating system-dependent functionality.
- The *os* and *os.path* modules include many functions to interact with the file system.

Let’s see some examples of OS in action!

1. Working with Files and Directories:
   - Creating, renaming, and deleting directories/files.
   - Changing current working directory.
   - Listing files and directories in a directory.
2. Environment Variables:
   - Accessing and modifying environment variables.



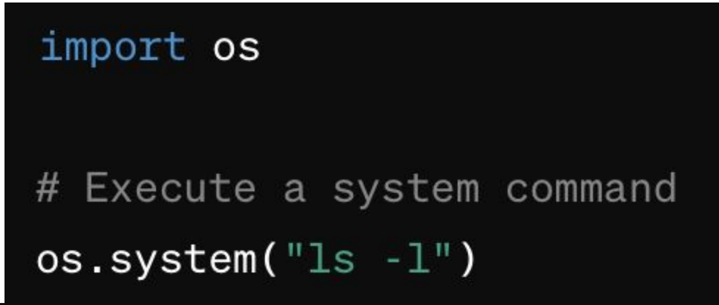
```python
import os

# Create a directory
os.mkdir("my_directory")

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

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

# Change current working directory
os.chdir("/path/to/new/directory")

# List files and directories in a directory
files = os.listdir("/path/to/directory")
print(files)
```



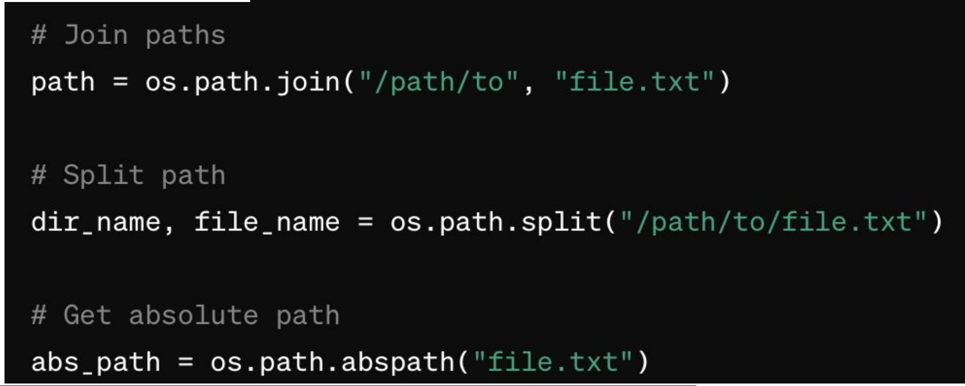
```python
import os

# Get value of an environment variable
value = os.getenv("PATH")

# Set an environment variable
os.environ["MY_VAR"] = "my_value"
```

More examples of OS in action!

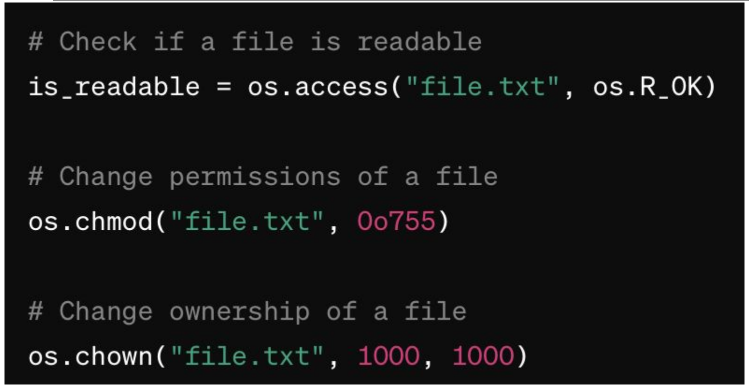
3. Executing System Commands:
   - Running system commands from Python.
4. Path Manipulation:
   - Joining and splitting paths.
   - Getting the absolute path.
5. Permissions and Ownership:
   - Checking permissions and ownership of files.
   - Changing permissions and ownership.



```python
import os

# Execute a system command
os.system("ls -l")
```



```python
# Join paths
path = os.path.join("/path/to", "file.txt")

# Split path
dir_name, file_name = os.path.split("/path/to/file.txt")

# Get absolute path
abs_path = os.path.abspath("file.txt")
```



```python
# Check if a file is readable
is_readable = os.access("file.txt", os.R_OK)

# Change permissions of a file
os.chmod("file.txt", 0o755)

# Change ownership of a file
os.chown("file.txt", 1000, 1000)
```

10

What is IO

- The io module is a powerful and flexible tool for handling **input** and **output** operations in Python, whether you're working with **files, strings,** or **network connections.**
- It abstracts away many of the complexities of I/O operations, making it easier to write robust and portable code.

What are Streams in Python?

In Python, a **stream** is an abstract concept representing a source or destination of data that can be read from or written to in a sequential manner. Streams are used for handling input and output operations, enabling you to work with data that might be coming from or going to various sources such as files, network connections, or in-memory buffers.

Types of Streams

1. Input Streams: Used to read data from a source (e.g., reading from a file, reading user input, reading data from a network socket).
2. Output Streams: Used to write data to a destination (e.g., writing to a file, sending data over a network, printing to the console).

Key Concepts

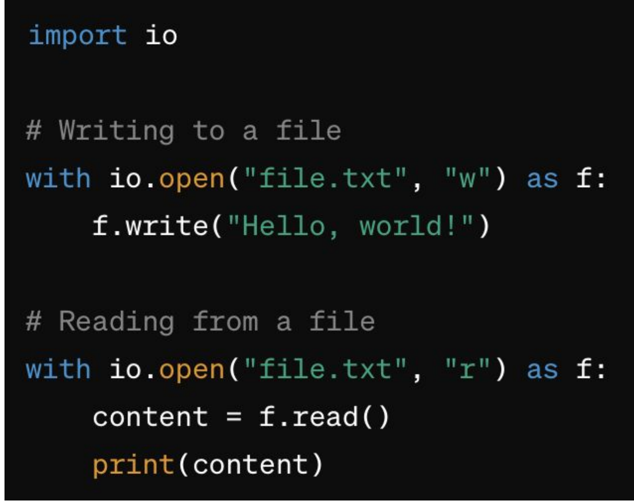
1. Buffering:
   - Buffering refers to the temporary storage of data to improve performance. For instance, when you write data to a file, it may be stored in a buffer before being actually written to disk.
   - Buffered streams are typically faster than raw streams because they reduce the number of I/O operations by combining many small operations into fewer large ones.
2. Sequential Access:
   - Streams provide sequential access to data, meaning you read or write data in a specific order, from start to end. This is in contrast to random access, where you can move directly to any part of the data.
3. File-Like Objects:
   - Streams in Python are often referred to as file-like objects because they support methods similar to file objects, such as read(), write(), seek(), and close().

More about IO!

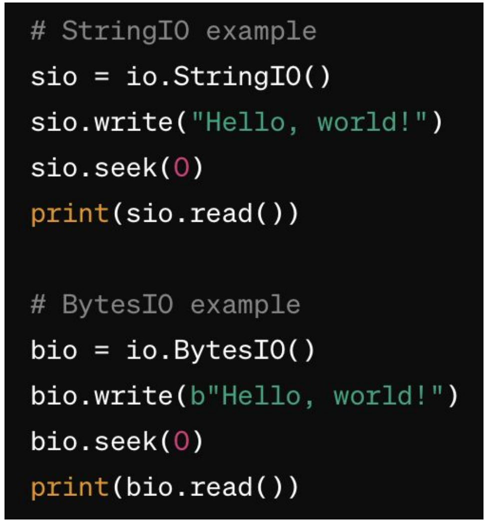
1. Base Classes:
   - IOBase: This is the abstract base class for all I/O classes in Python. It provides common methods and properties for working with streams, such as read(), write(), seek(), tell(), flush(), etc.
2. File I/O:
   - FileIO: This class provides a low-level interface for reading and writing raw byte data to and from files.
3. Buffering:
   - The io module provides buffering functionality for input and output streams to improve performance. Buffered I/O operations reduce the number of system calls by batching read and write operations.
4. Text Handling:
   - The io module handles text data encoding and decoding transparently. It supports various text encodings, such as ASCII, UTF-8, UTF-16, etc.
   - Text-based I/O classes automatically handle encoding and decoding when reading from or writing to files.
5. Pipes and Subprocesses:
   - The io module provides support for communicating with subprocesses using pipes. You can create pipe objects for reading from and writing to subprocesses' standard input, output, and error streams.
6. Wrapper Classes:
   - The io module includes wrapper classes that provide additional functionality on top of basic I/O operations. For example, TextWrapper wraps long lines of text to fit within a specified width.
7. Compatibility:
   - The io module provides a consistent interface for I/O operations across different types of streams, making it easier to write code that works with files, strings, and other data sources interchangeably.

Examples!

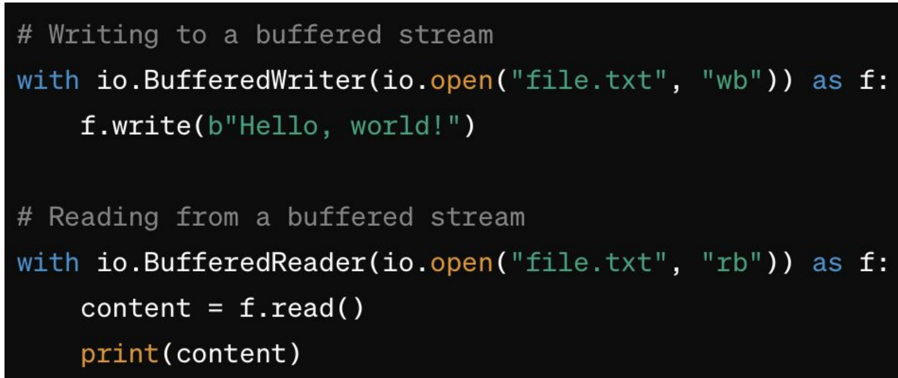
1. Reading and Writing to Files:
   - Reading from and writing to files using different modes.
2. StringIO and BytesIO:
   - Creating in-memory file-like objects for string and bytes data.
3. Working with Buffered Streams:
   - Reading and writing data using buffered streams.



```python
import io

# Writing to a file
with io.open("file.txt", "w") as f:
    f.write("Hello, world!")

# Reading from a file
with io.open("file.txt", "r") as f:
    content = f.read()
    print(content)
```



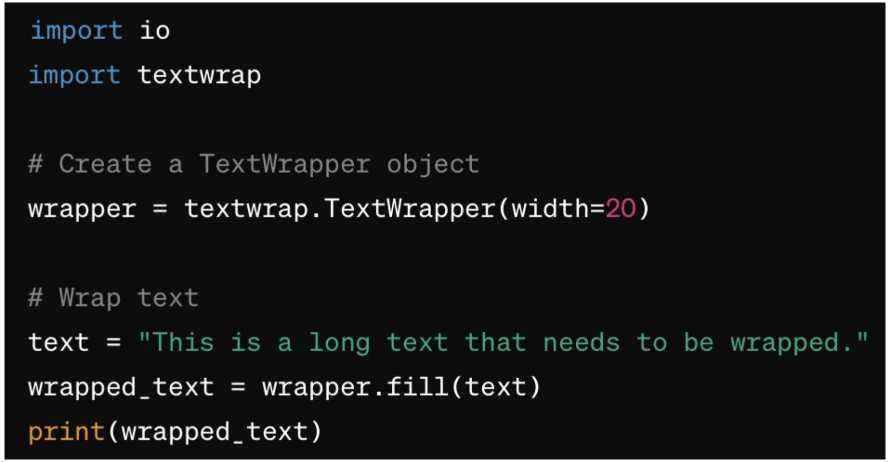
```python
# StringIO example
sio = io.StringIO()
sio.write("Hello, world!")
sio.seek(0)
print(sio.read())

# BytesIO example
bio = io.BytesIO()
bio.write(b"Hello, world!")
bio.seek(0)
print(bio.read())
```



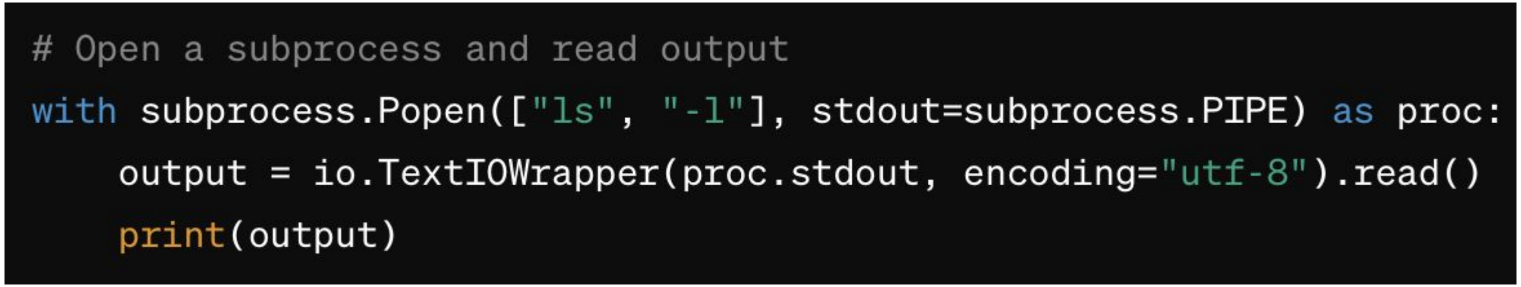
```python
# Writing to a buffered stream
with io.BufferedWriter(io.open("file.txt", "wb")) as f:
    f.write(b"Hello, world!")

# Reading from a buffered stream
with io.BufferedReader(io.open("file.txt", "rb")) as f:
    content = f.read()
    print(content)
```

More examples

4. TextWrapper:
   - Wrapping long lines of text.
5. Working with Pipes:
   - Communicating with subprocesses using pipes.



```python
import io
import textwrap

# Create a TextWrapper object
wrapper = textwrap.TextWrapper(width=20)

# Wrap text
text = "This is a long text that needs to be wrapped."
wrapped_text = wrapper.fill(text)
print(wrapped_text)
```

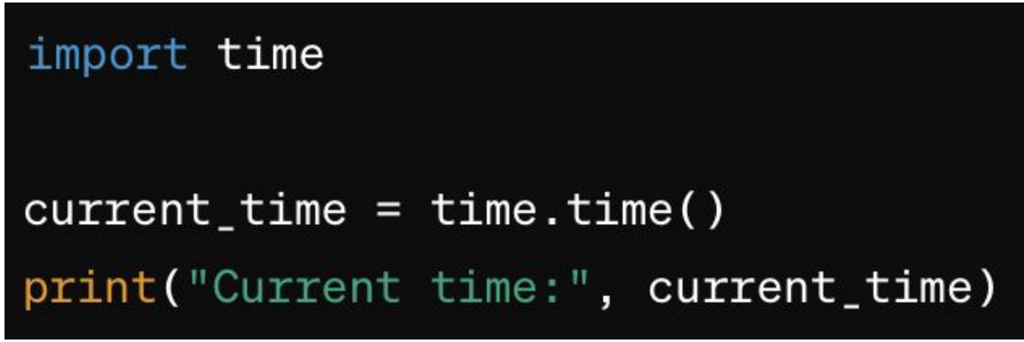


```python
# Open a subprocess and read output
with subprocess.Popen(["ls", "-l"], stdout=subprocess.PIPE) as proc:
    output = io.TextIOWrapper(proc.stdout, encoding="utf-8").read()
    print(output)
```

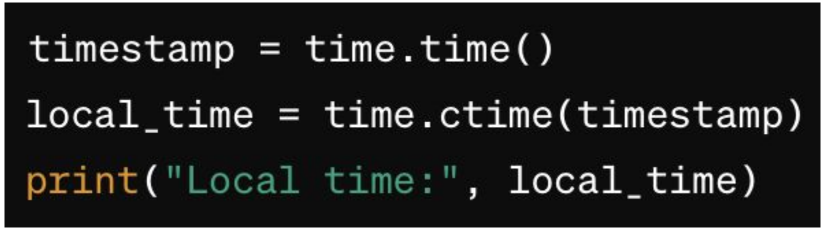
Time

How do we use this thing!?

1. Time Access:
   - The time() function returns the current system time in seconds since the epoch (January 1, 1970, 00:00:00 UTC). This timestamp is often used for measuring elapsed time.
2. Time Conversion:
   - The ctime() function converts a timestamp to a human-readable string representing the local time.



```python
import time

current_time = time.time()
print("Current time:", current_time)
```

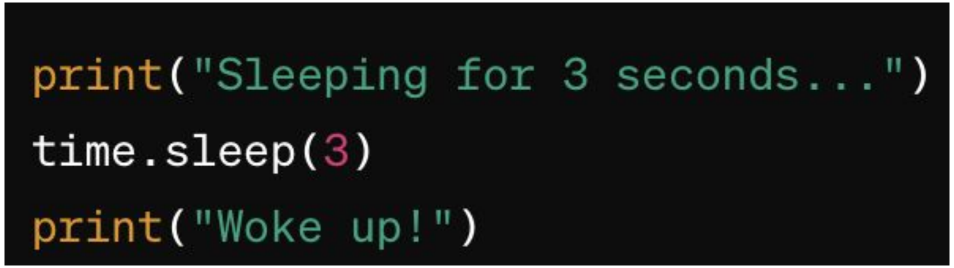


```python
timestamp = time.time()
local_time = time.ctime(timestamp)
print("Local time:", local_time)
```

Give me more!

3. Sleep:

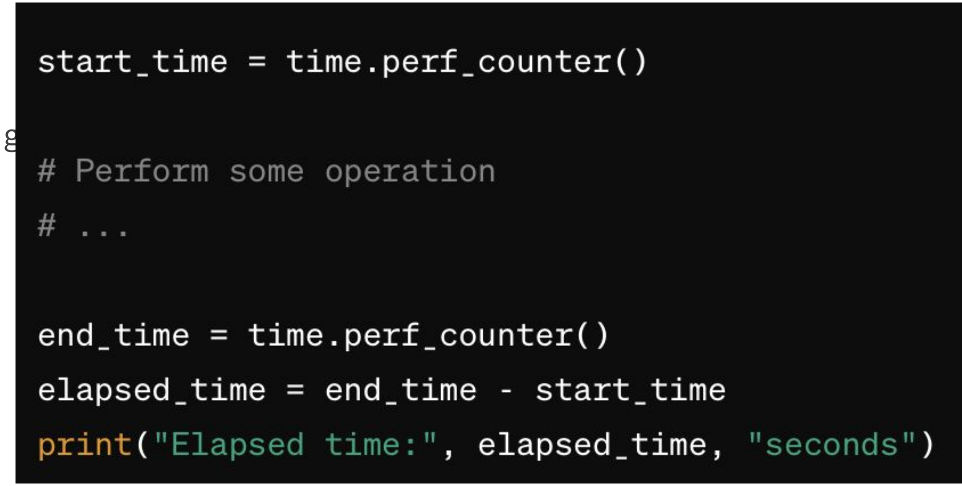
- The sleep() function suspends execution of the current thread for a specified number of seconds.



```python
print("Sleeping for 3 seconds...")
time.sleep(3)
print("Woke up!")
```

4. Performance Measurement:

- The perf_counter() function provides a high-resolution timer for measuring the elapsed time between two points in your code. This is useful for performance profiling.



```python
start_time = time.perf_counter()

# Perform some operation
# ...

end_time = time.perf_counter()
elapsed_time = end_time - start_time
print("Elapsed time:", elapsed_time, "seconds")
```