© Shehab Abdel-Salam

08 - Working with files

© Shehab Abdel-Salam

Working with files

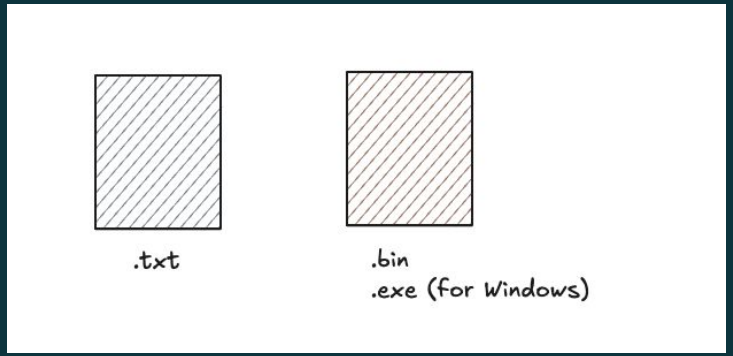
© Shehab Abdel-Salam

Working with files

A file is a collection of data stored in computer or other devices.

A file can be of many types (text, images, videos, executable code, etc.), but in general we have two main categories:

1. **Text files:** readable by humans.
2. **Binary files**: only readable by computers.



© Shehab Abdel-Salam

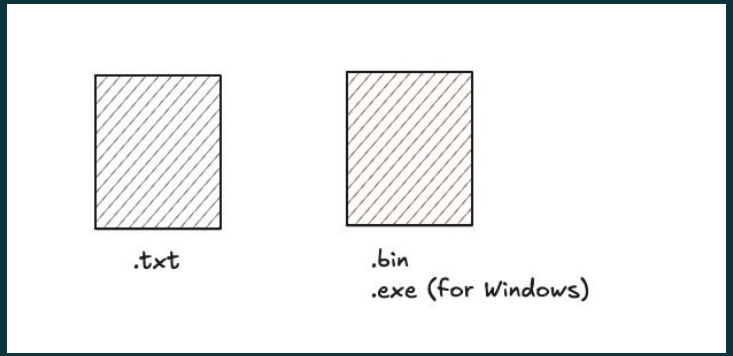
Working with files

A file is a collection of data stored in computer or other devices.

A file can be of many types (text, images, videos, executable code, etc.), but in general we have two main categories:

1. **Text files:** readable by humans.
2. **Binary files**: only readable by computers.

Python gives us easy ways to work with both.



© Shehab Abdel-Salam

Working with files

- Python provides built-in functions for **file management**.
- File management involves operations such as:
  - Reading
  - Writing, and
  - Appending data to files
- In Python, each operation is represented as a **mode**.

© Shehab Abdel-Salam

Opening and closing files

There are different modes for opening a file:

- `r` (read-only)
- `w` (write-only)
- `a` (append-only)
- `r+` (read+write)
- `w+` (write+read)
- `a+` (append+read)

Always close files after operations to free up resources

```
file = open("filename.txt", "r")
content = file.read()
file.close()




# A better approach is to use `with` keyword

with open("filename.txt", "r") as file:
     content = file.read()
```

© Shehab Abdel-Salam

Reading from files

open(filename, mode): Opens a file and returns a file object.

File Methods:

- `file.read()`
  - Reads the file content.
- `file.readline()`
  - Reads one entire line from the file.
- `file.readlines()`
  - Reads all the lines and returns them as a list of strings

```
with open("filename.txt", "r") as file:
    content = file.read()


with open("filename.txt", "r") as file:
    for line in file:
      print(line.strip())
```

© Shehab Abdel-Salam

Writing to files

File Methods:

- `file.write()`
  - Writes the string to the file.
- `file.writelines()`
  - Writes a list of strings to the file.
- `file.flush()`
  - Flushes the internal buffer, forcing the data to be written to the file.

```
with open("output.txt", "w") as file:
    file.write("Hello, world!\n")


with open("output.txt", "a") as file:
    file.write("Appending new data.\n")
```

© Shehab Abdel-Salam

Paths in Python

© Shehab Abdel-Salam

Paths in Python

A path is a string that represents the location of a file or directory in a file system.

Types of Paths:

- **Absolute Path:**
  - specifies the exact location of a file or directory from the root directory.
- **Relative Path:**
  - specifies the location of a file or directory relative to the current working directory.

© Shehab Abdel-Salam

Absolute paths

Provides an exact and fixed location independent of the current working directory.

Examples:

- Unix/Linux: `/home/user/documents/file.txt`
- Windows: `C:\Users\User\Document\file.txt`

© Shehab Abdel-Salam

Relative paths

Shorter and easier to manage in projects with nested directory structures.

Examples:

- `documents/file.txt`
- `../parent_directory/file.txt`

© Shehab Abdel-Salam

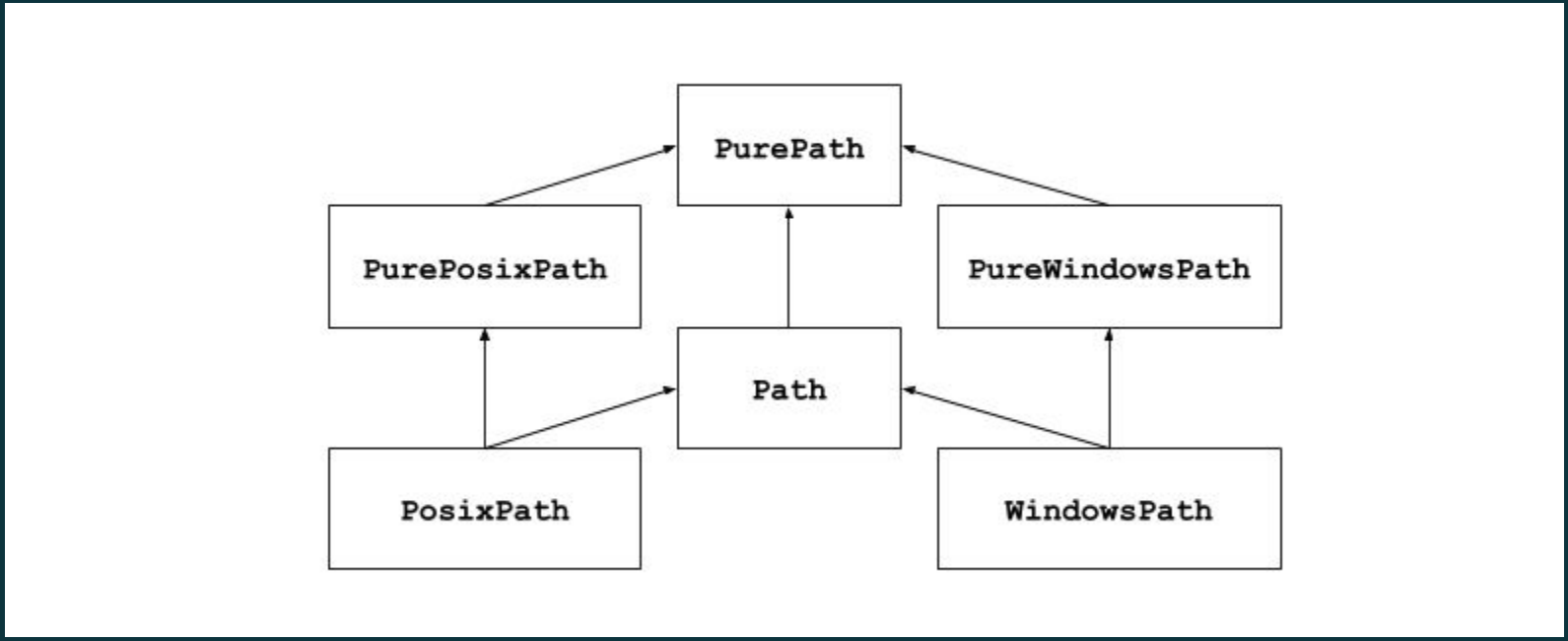
Basic Operations with Paths

Use Python's `**os.path**` module for cross-platform compatibility and path operations.

```
import os

# Joining paths using os.path.join()
path = os.path.join('directory', 'file.txt')

# Checking if a path exists if
os.path.exists(directory/file.txt'):
    print('File exists!')

# Getting current working directory
cwd = os.getcwd()
print('Current working directory:', cwd)
```

© Shehab Abdel-Salam

Object-oriented filesystem paths



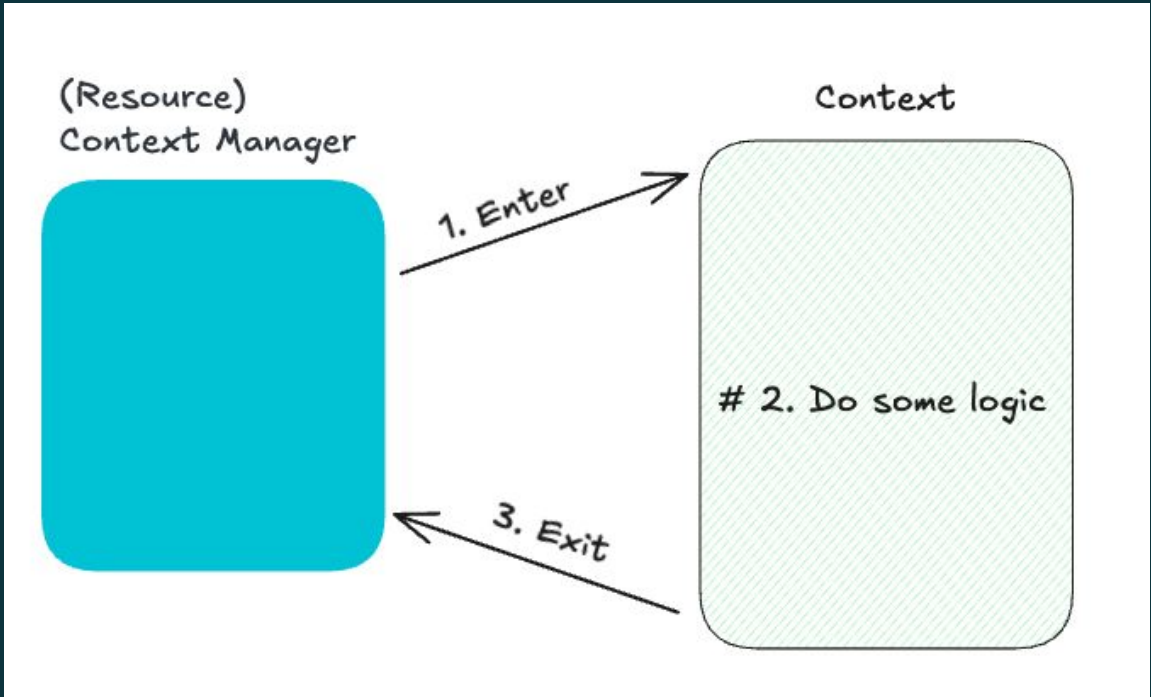
https://docs.python.org/3/library/pathlib.html

© Shehab Abdel-Salam

Context Managers

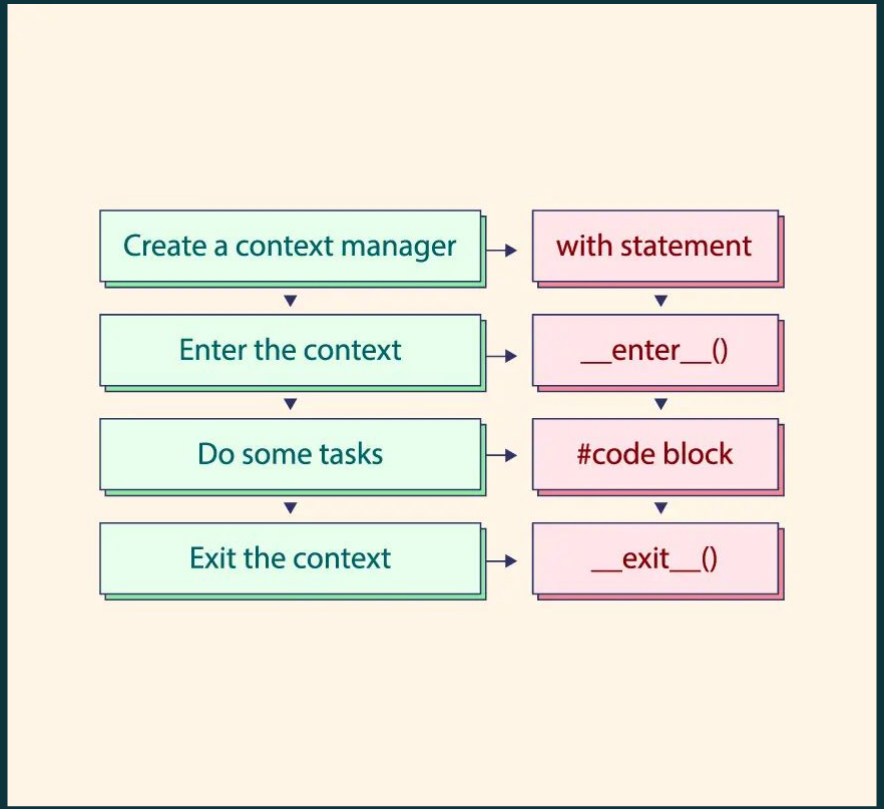
© Shehab Abdel-Salam

Context Managers



© Shehab Abdel-Salam

Context Managers



© Shehab Abdel-Salam

Context Managers

- Context managers provide a convenient way to manage resources and ensure they are properly cleaned up after use.

© Shehab Abdel-Salam

Context Managers

- Context managers provide a convenient way to manage resources and ensure they are properly cleaned up after use.
- Context managers help manage resources like files, network connections, or locks by automatically setting up and cleaning resources when entering and exiting a code block.

© Shehab Abdel-Salam

Context Managers

- Context managers provide a convenient way to manage resources and ensure they are properly cleaned up after use.
- Context managers help manage resources like files, network connections, or locks by automatically setting up and cleaning resources when entering and exiting a code block.
- Context managers improve code readability, simplify resource management, and ensure proper cleanup, reducing the risk of resource leaks.

© Shehab Abdel-Salam

Context Managers

Context managers can be implemented using:

1. Class-based approach

   Defining a class with

   `__enter__()` and `__exit__()` methods.

2. Function-based approach

   Using decorators **(@contextlib.contextmanager)**

```
class FileHandler:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()

# Usage:
with FileHandler('file.txt', 'r') as file:
    content = file.read()
# File is automatically closed at the end of the 'with' block
```

© Shehab Abdel-Salam

Conclusion

- Working with Files
  - Opening and closing files
  - Reading from files
  - Writing to files
- Types of Paths (Absolute vs Relative)
- Context Managers