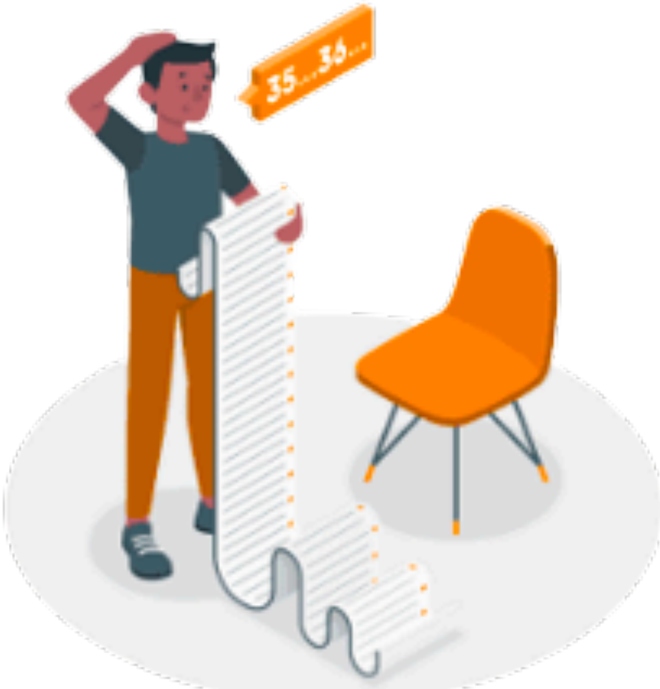
Recitation 3

CS210

Python Counter



Anamika Lochab

Print

Simples way to produce an output is using *print* statement

- It takes zero or more expressions separated by comma.
- Converts the expressions to string
- Writers the result to standard output

```
print("This is an example",1)
```

Reading Keyboard Input

Python provides built-in function to read a standard input (default keyboard)

```
user_input = input("Enter your input: ")
print(f"Received input is: {user_input}")
```

```
Enter your input: Hello, World!!
Received input is: Hello, World!!
```

File Objects

In python, access to files is provided through a file object, which is created by calling the builtin open function. Once such an object is created, a large number of methods are available for accessing the file, both for reading and writing.

Opening a file in python :

- The open function takes **between one and three arguments.**
- The first, which is the only required argument, is the **name of the file to open**.
- The second is a string which **describes the action(s)** that you plan to do to the file.
- The optional third argument specifies information about **how the file should be buffered**; *a value of 0 means no buffering, a value of 1 means line buffering, and any other positive value indicates the size of the buffer to be used.*

Arguments of *open* function:

Filename (Required)

- The first argument is the name of the file you wish to open.
- If the file is in the same directory as your Python script, you can just provide the filename. Otherwise, you need to provide the full path.
- Specify path using **forward slash** (/) `open('folder/subfolder/file.txt', 'r')`
- If you're on Windows, you might be tempted to use **backslashes**, but remember that backslash is also an escape character in Python strings. Therefore, you'll need to use double backslashes : `open('folder\\subfolder\\file.txt', 'r')`
- Python's os module provides a method ***os.path.join*** that automatically takes care of the slashes according to the operating system. This is a robust way to specify paths.

```
import os

path = os.path.join('folder', 'subfolder', 'file.txt')
open(path, 'r')
```

Arguments of *open* function:

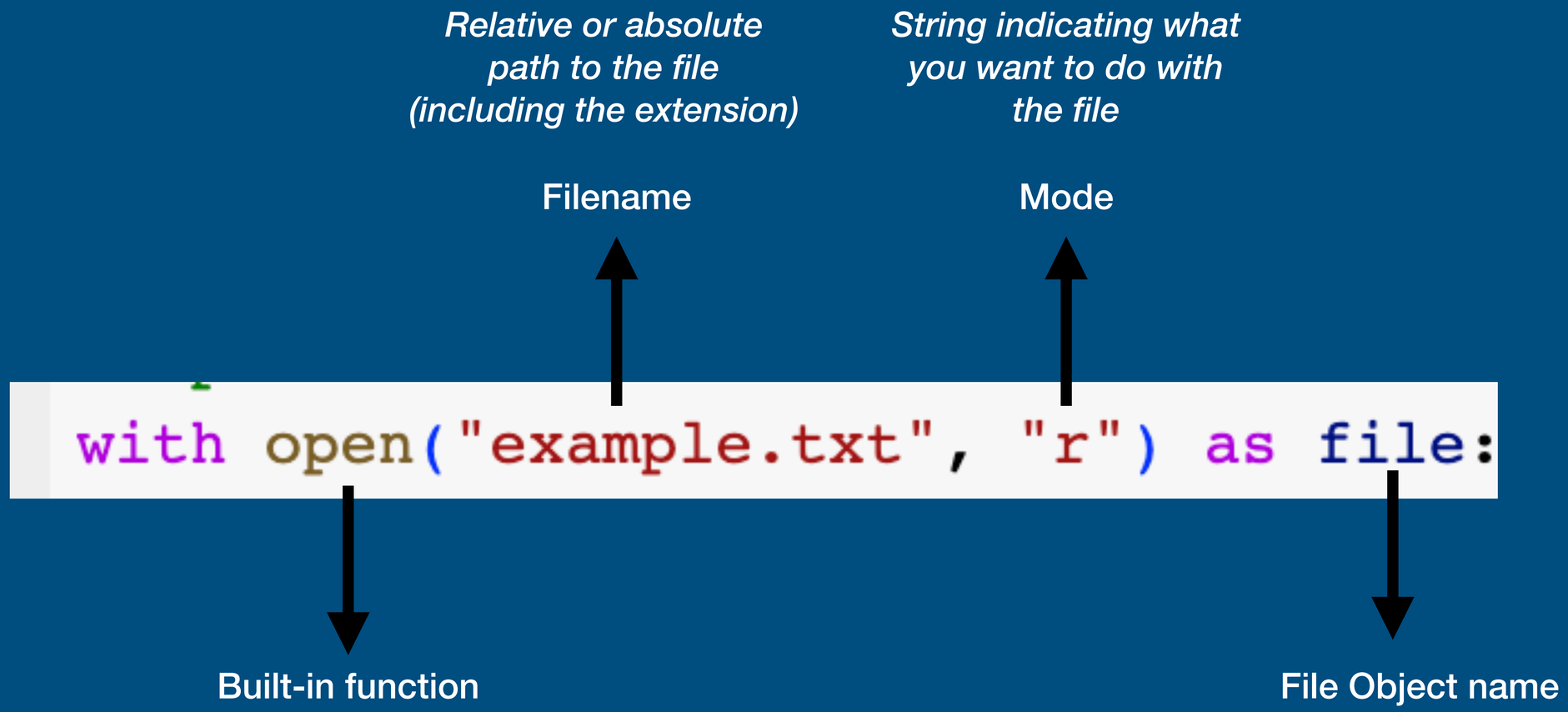
Mode (Optional)

- The second argument describes the action you intend to perform on the file. This is optional and defaults to "**r**", which means "**read**". Different modes are:

Modes	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode.
rb	Opens a file for reading only in binary format. The file pointer is placed at the beginning of the file. This is the default mode.
r+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
rb+	Opens a file for both reading and writing in binary format. The file pointer will be at the beginning of the file.
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
wb	Opens a file for writing only in binary format. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing.
w+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.

Arguments of *open* function:
Buffering (Optional)

- The third argument specifies how the file should be buffered:
  - 0 : No buffering (only allowed in binary mode)
  - 1: Line buffering (only usable in text mode)
  - Any positive integer: Sets buffer size (in bytes)
- **Advantages**: Buffering can significantly improve I/O performance by minimizing disk access and using memory operations, which are much faster.
- **Disadvantages**: In cases where immediate disk write is crucial (for example, for a critical log file), buffering can be risky as the data may be lost if the program crashes before flushing the buffer to disk.


Relative or absolute path to the file (including the extension)
String indicating what you want to do with the file
Filename
Mode
with open("example.txt", "r") as file:
Built-in function
File Object name

Specifying Encoding

```
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
```

- It specifies the **encoding to use** when reading or writing the file.
- This is especially important when working with **text files that may contain non-ASCII characters.**
- Some common encodings you might encounter are:
- **'utf-8'** : For reading UTF-8 encoded text.
- **'ascii'** : For reading basic ASCII text.
- **'iso-8859-1'**: For reading text in the ISO 8859-1 encoding, also known as Latin-1

Methods for reading data

Python provides several methods to read data from a file.

1. **read(size=-1)**: Reads and returns up to ***size*** bytes from the file. If the size argument is omitted or negative, the entire contents of the file will be read and returned.

```
with open('file.txt', 'r') as f:
    data = f.read()  # Reads the entire file
```

2. **readline(size=-1)**: Reads the next line from the file. *The* **size** *argument* is optional; if provided, at most ***size*** bytes will be read from the line

```
with open('file.txt', 'r') as f:
    line = f.readline()  # Reads the next line
```

3. **Iterating through the file object**: You can directly iterate through the file object to read line by line. This is memory-efficient for large files.

```
with open('file.txt', 'r') as f:
    for line in f:
        print(line, end='')
```

Closing a file

- The ***close()*** method of a *file* object flushes any unwritten information and closes the file object, after which no more writing can be done.
- Closing a file will free up the resources that were tied with the file

```
file = open("example.txt", "r")
# Perform file operations
file.close()
```

Using ***with*** Statements : Python offers a cleaner way to manage resources with ***with*** statements. The ***with*** statement ensures that the file is properly closed when the block of code is done executing, even if an error is thrown within the block. So, you don't have to explicitly call ***file.close()***

Writing to a file

- The **write()** method writes any string to an open file. It is important to note that Python strings can have binary data and not just text.
- The ***write()*** method **does not add a newline character ('\n')** to the end of the string.

```
# Method 1: Using 'with' statement (Automatically closes the file)
with open('example.txt', 'w') as file:
    file.write('Hello, world!\n')  # Writing a line to the file
    file.write('This is another line.\n')  # Writing another line

# Method 2: Traditional open and close (Manually close the file)
file = open('example.txt', 'w')  # Open file in write mode
file.write('Hello, world!\n')  # Writing a line to the file
file.write('This is another line.\n')  # Writing another line
file.close()  # Don't forget to close the file
```

File Attributes

Once a file is opened and you have one *file* object, you can get various information related to that file.

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file.

Discussion with .ipynb file

Python's Collections Library

- What is collections?

The ***collections*** library in Python is a built-in module that implements specialized container datatypes providing alternatives to Python's general-purpose built-in containers like *list*, *tuple*, and *dict*.

- Why collections?

While Python's built-in data structures are incredibly powerful, sometimes they aren't quite what you need. The ***collections*** library provides additional data structures to make your life easier.

Major Players of collections Library

- namedtuple
- deque
- Counter
- OrderedDict
- defaultdict

NumPy

NumPy stands for "Numerical Python"

- **Arrays**: The core functionality of NumPy revolves around arrays—n-dimensional, to be exact.
- **Vectorization**: Perform operations on entire arrays without using loops.
- **Broadcasting**: The term for how NumPy handles element-wise binary operations with different shapes.
- **Indexing and Slicing**: Unlike Python lists, NumPy arrays can be multi-dimensional, and you'll learn how to extract specific elements in various ways.

Discussion with .ipynb file