Python Input/Output

Raul P. Pelaez

February 10, 2026

Contents

1 Introduction

In this lesson, we will learn how to handle input and output operations in Python. We will cover how to parse command-line arguments using the `argparse` module, how to read and write files using built-in Python functions, and how to work with different file formats commonly used in scientific computing.

Info

Command-line arguments
The parameters passed to a terminal command that appear after the command name are called command-line arguments. For example, in the command:

```
cd some_folder
```

`some_folder` is a command-line argument passed to the `cd` command.

2 Importing code from other files

First, let me tell you about a detail of `import` statements. Lets imagine we have two files in the **same folder**, `script.py` and `library.py`. Say that `library.py` contains the following code:

```python
# In library.py
def greet(name):
    print(f"Hello, {name}!")
```

If we want to use functions defined in `library.py` from `script.py`, we can import the library using the `import` statement:

```python
# In script.py
import library

library.greet("Alice")
```

Output

```
Hello, Alice!
```

When running `script.py`, Python will look for a file named `library.py` in the same folder and import it.

Advice

Importing from other folders
Python has a hierarchy of places to look for imported modules. By default, it looks in the current folder and in the standard library. Then, it will also look in the folders specified in the `PYTHONPATH` environment variable, which `conda` sets to include the `site-packages` folder of the current environment.

In Python, importing another script entails executing all the code in the imported script. This means that if `library.py` contains some code that is not inside a function, that code will be executed when we import `library.py`. Lets create a new library, `library2.py`, to include some print statements outside of the function and see what happens when you import it from `script.py`.

```
# In library2.py
def greet(name):
    print(f"Hello, {name}!")

print("This will be printed when we import library.py")
```

Now try to run `script.py` and see the output. You should see the print statement from `library2.py` in the output of `script.py`.

```
import library2
library2.greet("Alice")
```

Output

```
This will be printed when we import library.py
Hello, Alice!
```

Sometimes, we want to have some code in a script that should only be executed when the script is run directly, and not when it is imported. To achieve this, we can use the following idiom:

```
# In library3.py
def greet(name):
    print(f"Hello, {name}!")
if __name__ == '__main__':
    # Code here will only be executed when the script is run directly
    print("This will only be printed when we run script.py directly")
```

Now, if we import `library3.py` from `script.py`, we will not see the print statement from `library3.py` in the output of `script.py`.

```
import library3
library3.greet("Alice")
```

Output

```
Hello, Alice!
```

But if we run `library3.py` directly, we will see the print statement from `library3.py` in the output.

```
python library3.py
```

Output

```
This will only be printed when we run script.py directly
```

Info

Dunder variables
Dunder variables (short for "double underscore" variables) are special builtin variables in Python that have names starting and ending with double underscores. They are used to define special behavior in classes and modules. The `__name__` variable is a dunder variable that holds the name of the current module. When a module is run directly, `__name__` is set to `__main__`, which allows us to check if a script is being run directly or imported as a module.

Advice

From now on, try to structure your code in a way that the main logic of your program is inside a function, and use the `if __name__ == '__main__':` idiom to call that function. By working this way, you will make the components of your code reusable in other scripts. More importantly, you will make your code *expectable*, since other people reading your code will be looking for this kind of structure.

3 Parsing arguments to your code: Argparse

The `argparse` module is Python's recommended command-line parsing module. It makes it easy to write user-friendly command-line interfaces where users can specify arguments and options when running your scripts. With it, we can create scripts that we can run from the terminal and pass parameters to them, for instance:

```
python my_script.py --some_parameter 1 --another_parameter "hello"
```

To use argparse, you need to:

- Create an `ArgumentParser` object
- Add arguments using `add_argument()`
- Parse the arguments using `parse_args()`

Example

```python
# In argparse_example.py
import argparse

def parse_arguments():
    parser = argparse.ArgumentParser(description='A simple script that greets the user')
    parser.add_argument('--name')
    return parser.parse_args()

def main():
    args = parse_arguments()
    print(f"Hello, {args.name}!")

if __name__ == '__main__':
    main()
```

You can run this script with:

```
python argparse_example.py --name "Alice"
```

Output

```
Hello, Alice!
```

The `parse_args()` method returns a special dictionary, in which the options we have configured appears as members. In the example above, adding the argument `name` makes it possible to write `args.name` on the object returned by `parse_args()`.

Advice

Argparse functions have an extensive list of arguments that can be used to customize the behavior of the argument parser. For instance, you can specify the type of the argument, a default value, a help message, and more.

```
# The program will fail if the argument is not provided
parser.add_argument('--input', required=True, help='Input file path')
```

Info

You get help for free

When you use `argparse`, you get a help message for free. If you run your script with the `--help` flag (even if you have not added such an argument), you will see a message that describes the arguments that your script accepts, along with the help messages you have provided for each argument.

```
python argparse_example.py --help
```

Output

```
usage: argparse_example.py [-h] [--name NAME]

A simple script that greets the user

options:
  -h, --help   show this help message and exit
  --name NAME
```

4 Dealing with files in pure Python

While Pandas provides convenient functions for handling tabular data files, sometimes we need to load non-tabular data, or we need to perform more complex file operations. In these cases, we can use Python's built-in file operations.

Info

Basic File Operations

Python uses the built-in open() function to work with files. The basic syntax is:

```
# Reading a file
with open('filename.txt', 'r') as file:
    content = file.read()
# Writing to a file
with open('output.txt', 'w') as file:
    file.write('Hello, World!')
```

`r` stands for read mode, `w` for write mode, and `a` for append mode. The default mode is `r`.

Advice

Always use context managers
The `with` statement ensures proper file handling by automatically closing the file when you're done, even if an error occurs. This is much safer than manually opening and closing files:

```
# Good practice:

with open('file.txt', 'r') as file:
    content = file.read()
# Avoid this:

file = open('file.txt', 'r')
content = file.read()
file.close()  # Might never execute if an error occurs
```

Info

The `with` statement
The `with` statement is a context manager that allows you to manage resources, such as file streams, in a way that ensures they are properly cleaned up after use. When you use `with`, it automatically handles the opening and closing of the file, even if an error occurs. This makes your code more robust and less prone to resource leaks.
The object returned by `open` will be initialized (the file will be opened) when the block of code inside the `with` statement is entered, and it will be cleaned up (the file will be closed) when the block of code is exited, regardless of whether an exception was raised or not.

Info

Reading line by line
We can process a file line by line using a for loop.

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

Advice

Notice how `file` here is an iterable object that we can loop over. We can do this because it follows the iterator protocol, remember that in Python, anything that "behaves like" an iterable can be used in a for loop. The file object understands "being looped over" as "reading the file line by line".

5 Relevant file formats

When working with scientific data in Python, you'll encounter various file formats, each with its own advantages and use cases.

Info

These are NumPy-specific file formats:

- .npy: Stores a single NumPy array
- .npz: Stores multiple NumPy arrays in a compressed format

```
import numpy as np
# Save array to .npy file

data = np.array([1, 2, 3, 4, 5])
np.save('data.npy', data)
# Load array from .npy file
loaded_data = np.load('data.npy')
# Save multiple arrays to .npz file
x = np.array([1, 2, 3])
y = np.array([4, 5, 6])
np.savez('arrays.npz', x=x, y=y)
# Load from .npz file
arrays = np.load('arrays.npz')
x_loaded = arrays['x']
y_loaded = arrays['y']
```

Advice

The choice of format can significantly impact your program's performance:

```
import numpy as np
import time
# Create a large array
data = np.random.rand(1000000)

t0 = time.time()
np.save('data.npy', data)
print(f"NPY save time: {time.time() - t0:.4f}s")

t0 = time.time()
np.savetxt('data.txt', data)
print(f"TXT save time: {time.time() - t0:.4f}s")
```

Output

```
NPY save time: 0.0023s
TXT save time: 0.6974s
```

Info

Compressing files

Compressing files can save disk space and reduce transfer times. Python has built-in support for various compression formats, including .zip and .tar.gz. Here's how to compress and decompress files using the `zipfile` module:

```
import zipfile
# Compress a file
with zipfile.ZipFile('data.zip', 'w') as zipf:
    zipf.write('data.npy')
# Decompress a file
with zipfile.ZipFile('data.zip', 'r') as zipf:
    zipf.extractall('data_folder')
```

6 Exercises

6.1 CLI Log Filter

Goal

We will write a Python script that reads a text log file and outputs only the lines that match a user-provided keyword. The script will support writing the result to a new file, or printing to the terminal.
At the end, your script should be callable like this:

```
$ python log_filter.py --input "app.log" --keyword "ERROR" --output "errors.log"
ERROR 2026-02-10T08:01:12Z auth: invalid password for user=alice ip=203.0.113.7
ERROR 2026-02-10T08:02:04Z api: 401 unauthorized id=req_8f1a user=alice
ERROR 2026-02-10T08:04:21Z worker: smtp timeout job_id=job_1029 host=smtp01 ms=5000
```

You have an example `app.log` file in BB. The script should print the lines in the input file that specifically **start** with the keyword.

hint

- Use regular expressions to match the keyword in each line.
- The regexp metacharacter `^` means "starts with".
- Make this script a true "program" by structuring it as a set of functions and using the `if __name__ == '__main__':` guard.

Milestone

Open the terminal, activate the conda environment, and navigate to your work folder. Use the `code` command to open the current folder and then create a new Python script named `log_filter.py`. Download the example `app.log` file and use `mv` to move it to the current folder.

hint

- Remember that `~` is the home folder and `.` is the current one.
- In Mac, the file will be in `~/Downloads`, while in Windows WSL, you will find it in something like `/mnt/c/Users/your_name/Downloads`

Milestone

In `log_filter.py`, start by creating and configuring an `ArgParse` instance with the desired options. Write a program that just shows the parameters to the terminal in order to test it. Using `argparse`, add the following arguments:

- `--input` (required): input log file path
- `--keyword` (required): keyword to match
- `--output` (optional): output file path, print to terminal if not present.

hint

Use the optional function arguments in `add_argument` to specify the type, required status and help documentation string for each parameter.

Milestone

In `log_filter.py`, implement a function `filter_lines(lines, keyword)` that returns only the lines starting with the keyword (case-sensitive).

Advanced milestone

Add a `--ignore-case` flag. If present, matching should be case-insensitive.
Add a `--count` flag. If present, print only the number of matching lines. If `--output` is also provided, still write the matching lines to the output file.
Add a `--head X` argument that prints only the first X matches.

hint

For case-insensitive matching, convert both the line and the keyword to lowercase before checking containment.