Free Python Standard Library How-to Cheatsheet for Data Engineers

A printable cheatsheet to help you use the right Python standard library for your data pipeline. How-to style and copy-pastable code

Joseph Machado

2025-08-10

Data IO (reading & writing data)

- Represent file & folders location with `pathlib.Path`

```
from pathlib import Path

file_name = 'your_file.csv'                                    ①

file_path = Path(file_name)

print(file_path.absolute())                                    ②
print(file_path.name)                                          ③
```

① Use your file name
② Prints full file path
③ Prints the file name

- Create and copy files with `pathlib.Path.touch`

```
from pathlib import Path

new_file = Path("some_file.csv")
new_file.touch()                                               ①

new_file.copy(Path("./some_new_file.csv"))                     ②
```

① Create the file
② Make a copy of `new_file` at `some_new_file.csv`

- Identify files that match a pattern with `pathlib.Path.glob`

```
# Find files, glob returns and iterator
list(Path(".").glob('*.py'))                                    ①
list(Path(".").glob('*/*.py'))                                  ②
list(Path(".").glob('**/.py'))                                  ③
```

① Find any Python file in the current directory
② Find any Python file in the directory one level down
③ Find any Python file in the directory at any level down

- Read and write to a file with `open`

```
from pathlib import Path
file_name = 'your_file.csv'                                     ①


file_path = Path(file_name)

with open(file_path) as f:
  for line in f:                                                ②
    print(line)

with open(file_path) as f:
  file_contents = f.readlines()                                 ③

print(file_contents)
```

① Replace with your file name
② Print one line at a time
③ Read all lines into a list of strings

The `r` indicates read mode; the key modes are:

1. 'r': open for reading (default)
2. 'w': open for writing, truncating the file first
3. 'a': open for writing, appending to the end of the file if it exists
4. '+': open for updating (reading and writing)

- Read csv file with each row as a list of strings using `csv.reader`

```
from pathlib import Path
import csv

file_name = 'your_file.csv'                                    ①
file_path = Path(file_name)

with open(file_path, 'r') as f:
  csvreader = csv.reader(f)
  for line in csvreader:                                       ②
    print(line)
```

① Replace with your file name
② Read the CSV file as a list of strings

- Write to a csv file using `csv.writer`

```
from pathlib import Path
import csv

file_name = 'your_file_to_write_to.csv'                        ①

file_path = Path(file_name)

with open(file_path, 'w') as f:
  csvwriter = csv.writer(f)
  for line in [[1, 'emp_1'], [2, 'emp_2']]:                    ②
    csvwriter.writerow(line)
```

① Replace with your file name to write the data into
② Read the CSV file as a list of strings

- Read csv files with each row as a dict with `csv.DictReader`

```
from pathlib import Path
file_name = 'your_file.csv'                                    ①

file_path = Path(file_name)

with open(file_path, 'r') as f:
  dict_reader = csv.DictReader(f)
  for line in dict_reader:
    print(line)                                                ②
```

① Replace with your file name
② Each line will be represented as a dict, with the key as column name and value as column value

- Write a list of dict as a csv file with `csv.DictWriter`

```
from pathlib import Path
import csv
file_name = 'your_file_to_write_to.csv'                                    ①

file_path = Path(file_name)

fake_data = [{'id': 1, 'name': 'emp_1'}, {'id': 2, 'name': 'emp_2'}]
with open(file_path, 'w+') as f:
  writer = csv.DictWriter(f,fieldnames=['id', 'name'])
  writer.writeheader()                                                     ②
  for line in fake_data:
    writer.writerow(line)                                                  ③
```

① Replace with your file to write the dict data into
② Write headers to the file (i.e., Column Names)
③ Write the dict as csv data

- Create a temporary file with `tempfile.TemporaryFile`

```
import tempfile

with tempfile.TemporaryFile(mode='w+') as tf:
  tf.write("header\n")                                                     ①
  tf.write("data\n")

  tf.seek(0)                                                               ②
  print(tf.read())

# TemporaryFile is removed                                                ③
```

① Write data into the temp file. The `\n` indicates a new line (Python will use the right newline character for your OS)
② We need to go to the start of the file with `seek`, else the tf.read will be empty
③ TemporaryFile is removed

- Create a temporary file that can be referenced by name with `tempfile.NamedTemporaryFile`

```
import tempfile

with tempfile.NamedTemporaryFile(mode='w+') as tf:
  tf.write("header\n")
  tf.write("data\n")
  print(tf.name)                                                    ①

# NamedTemporaryFile is removed                                   ②
```

① Print the NamedTemporaryFile's name
② NamedTemporaryFile is removed

- Pass input arguments to a Python script with `argparse`

Listing 1 `fake_file_operator.py`

```
import pathlib
import argparse

parser = argparse.ArgumentParser(                                   ①
                    prog='ProgramName',
                    description='What the program does')

parser.add_argument(
  '-f',                                                             ②
  '--filename',
  default='some_file.csv',                                          ③
  type=pathlib.Path                                                 ④
)

parser.add_argument(
  '-op',
  '--operation',
  required=True,                                                    ⑤
  choices=['overwrite', 'append', 'truncate', 'delete']             ⑥
)

args = parser.parse_args()                                          ⑦

print(f"File name is {args.filename} and operation is {args.operation}")
```

① Initialize the `ArgumentParser` to define what to expect

② This argument can be passed in either as `-f` or as `--filename`
③ This argument will have a default in case the user does not provide a value for this
④ This argument should represent a valid file or folder path
⑤ This argument is required; calling the Python script without this argument will raise an exception
⑥ This argument will only accept one of these allowed choices
⑦ The inputs provided will be validated against the settings defined above with `add_argument`, and you will now have access to the arguments passed to the script

- Run a Python script with input arguments as shown below

```
python fake_file_operator.py -f some_file_path.csv -op delete
```

Data Structures

- Represent an ordered list of items with `list`

```
# list
sample_list = ['a', 10, '100']                                    ①

print(sample_list[2])                                             ②

sample_list.append('b')                                           ③
second_list = ['c', 'd']

sample_list.extend(second_list)                                   ④
print(sample_list)

del(sample_list[2])                                               ⑤
```

① A list can be made up of values of different types
② Access elements in a list using `[index]`, index starts at 0
③ Add elements to the end of a list using `append`
④ Add the elements of `second_list` to `sample_list`
⑤ Delete the element at index 2, which is "100"

- Need to access data with an ID? Use dict

```
# dict
sample_dict = {1: 'emp_1', 2: 'emp_2'}                                    ①

# get
sample_dict[1]
sample_dict[3]                                                            ②
sample_dict.get(3)                                                        ③
sample_dict.get(3, 10)                                                    ④

from collections import defaultdict
d = defaultdict(list)
d[1] = ['emp1']
print(d[3])                                                               ⑤
```

① Define a dict as `{key1: value1, key2: value2, ...}`
② This will raise a `KeyNotFound` exception, since 3 is not a key of sample_dict
③ Exception-safe way to get a value from a dict. Since 3 is not a key, you will get none
④ Similar to the above line, but with a fallback default of 10
⑤ This will print an empty list. Since we used `defaultdict(list)`.

- Representing a unique set of values? Use a set

```
# Set
uniq_set = set()                                                          ①
uniq_set.add(10)
uniq_set.add(10)
uniq_set.add(10)
uniq_set.add(10)

print(uniq_set)                                                           ②

second_uniq_set = set()
second_uniq_set.add(10)
second_uniq_set.add(20)
second_uniq_set.add(30)

uniq_set.intersection(second_uniq_set)                                    ③
uniq_set.difference(second_uniq_set)                                      ④

second_uniq_set.difference(uniq_set)                                      ⑤
```

① Define a set data structure with `set`

② The `uniq_set` variable will be 10, as duplicates are not added to the set
③ `Intersect` operation will find the elements in both the sets, which will be 10
④ `Difference` operation will find elements in uniq_set but not in second_uniq_set, which will be None
⑤ This `Difference` will find elements in second_uniq_set but not in uniq_set, which will be 20 and 30

- Represent complex data with `dataclass`

```
from dataclasses import asdict, dataclass, field

@dataclass                                                          ①
class Person:
  name: str
  friends: list[str] = field(default_factory=list)

  def get_friends(self):                                            ②
    print(f"{self.name}s friends are {(',').join(self.friends)}")

p = Person('person_name_1', ['person_2', 'person_10'])
print(p.name, p.friends)
p.get_friends()

print(asdict(p))                                                    ③
```

① Define a dataclass with the `@dataclass` decorator
② Create functions to transform your data
③ This will convert the dataclass object into a dict of keys and values

> **!** Important
>
> - Some of Python's data structures and Python Objects are reference copies. Use the `copy.deepcopy` to create a separate copy.

- Use the `copy` module to make actual copies of Python data structures

```
from copy import deepcopy

sample_dict = {1: 'emp_1', 2: 'emp_2'}
sample_dict_2 = sample_dict
sample_dict_2[1] = 'employee_1'
print(sample_dict[1])                                               ①
```

```
sample_dict = {1: 'emp_1', 2: 'emp_2'}
sample_dict_2 = deepcopy(sample_dict)
sample_dict_2[1] = 'employee_1'
print(sample_dict[1])                                                    ②

@dataclass
class Person:
  name: str
  friends: list[str] = field(default_factory=list)

p = Person('person_name_1', ['person_2', 'person_10'])
p2 = p
p2.name = 'person_name_2'
print(p.name)                                                            ③

from copy import deepcopy
p = Person('person_name_1', ['person_2', 'person_10'])
p2 = deepcopy(p)
p2.name = 'person_name_2'
print(p.name)                                                            ④
```

① This will be `employee_1`
② This will be `emp_1`
③ This will be `person_name_2`
④ This will be `person_name_1`

- Use `Decimal` to represent precise decimal numbers; Python defaults to floats, which are imprecise.

```
# Float v Decimal
print(1.1 + 2.2)                                                         ①

from decimal import Decimal, getcontext
getcontext().prec = 4                                                    ②
print(Decimal(1.1) + Decimal(2.2))                                       ③
```

① This will print a value with a long list of 0s and some decimals
② Set precision for `Decimal` types
③ This will print `3.3000`

- Use `datetime` module to represent and manipulate date & time

```python
from datetime import datetime
print(datetime.now())                                                    ①

date_str = "2025-10-14"
date_obj = datetime.strptime(date_str, "%Y-%m-%d")                       ②
print(date_obj)  # 2025-10-14 00:00:00

formatted = date_obj.strftime("%B %d, %Y")                               ③
print(formatted)  # "October 14, 2025"
```

① Prints the current date and time of your machine
② `strptime` stands for String Parse Time, which converts a string to time. The input string should match the format we have specified
③ `strftime` stands for String Format Time. This converts a datetime object into a string with the specified format

> **Note**
>
> The format for various datetime representations can be seen here, in the official docs

- Use `datetime.timedelta` to calculate time differences

```python
from datetime import datetime
date_str_1 = "2025-10-14"
date_obj_1 = datetime.strptime(date_str_1, "%Y-%m-%d")

date_str_2 = "2025-10-16"
date_obj_2 = datetime.strptime(date_str_2, "%Y-%m-%d")

# Calculate the time delta
time_delta = date_obj_2 - date_obj_1                                     ①

print(f"Time delta: {time_delta}")                                       ②
print(f"Days: {time_delta.days}")                                        ③
print(f"Total seconds: {time_delta.total_seconds()}")
print(f"Hours: {time_delta.total_seconds() / 3600}")
```

① The difference of datetime objects is of the `datetime.timedelta` type
② Prints the time delta
③ The `datetime.timedelta` has methods to print the difference in various time units

> **Tip**
>
> JSON is a combination of lists & dicts that is a standard for communication between systems ref.

- Use `json.dumps` function to convert a dict to a json string (aka serialization)

```
import json

data = {
    "name": "Alice",
    "age": 30,
    "hobbies": ["reading", "coding"],
    "active": True
}

json_string = json.dumps(data)                                    ①
print(json_string)
```

① Convert the dict into a string

- Use `json.loads` function to convert a json string into a dict (aka deserialization)

```
json_string = '{"name": "Bob", "age": 25, "active": true}'

data = json.loads(json_string)                                    ①
print(type(data))                                                 ②
```

① Convert a string (with the proper json format) into a dict
② This will show that the data is of dict type

Transformations

- Create single-line functions called `lambda` functions

```
def square(x):
    return x ** 2

square_lambda = lambda x: x ** 2                                  ①

print("Regular function:", square(5))
```

```
print("Lambda function:", square_lambda(5))

add = lambda x, y: x + y                                                    ②
print(f"\nLambda add: {add(3, 7)}")

multiply = lambda x, y, z: x * y * z
print(f"Lambda multiply: {multiply(2, 3, 4)}")
```

① Define a lambda function as `lambda input1, input2,..: some_operation`
② Define a lambda function with multiple input parameters

- Apply a function to every element with `map`

```
def square(x):
    return x ** 2

numbers = [1, 2, 3, 4, 5]
squared = map(square, numbers)                                              ①

print("Original numbers:", numbers)
print("Squared (map object):", squared)                                     ②
print("Squared (as list):", list(squared))                                  ③
```

① Apply the `square` function to every element in `numbers`
② The `square` function is only executed when we need to write out the output
③ Here, we force `map` to run by converting it to a list

- Filter elements with `filter`

```
def is_even(x):
    return x % 2 == 0

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = filter(is_even, numbers)                                            ①

print("Original numbers:", numbers)
print("Evens (filter object):", evens)
print("Evens (as list):", list(evens))                                      ②
```

① Only keep the elements in numbers that return true when `is_even` is called on it
② `filter` will not be executed until needed. Here we force it by converting the result to a `list`

- Use `min`, `max`, or `sum` to get the min, max, & sum of elements.

Note: Sum only works with numerical data.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

print("Numbers:", numbers)
print("Minimum:", min(numbers))  # Output: 1
print("Maximum:", max(numbers))  # Output: 9
print("Sum:", sum(numbers))      # Output: 39
```

- Use custom logic to define `min` or `may` with the `key` parameter

```
students = [
    {"name": "Alice", "score": 85, "age": 20},
    {"name": "Bob", "score": 92, "age": 22},
    {"name": "Charlie", "score": 78, "age": 21},
    {"name": "David", "score": 95, "age": 19}
]

lowest_scorer = min(students, key=lambda s: s["score"])                    ①
highest_scorer = max(students, key=lambda s: s["score"])

print(f"Lowest scoring student: {lowest_scorer}")

print(f"Highest scoring student: {highest_scorer}")
```

① Sort based on the score of the dict

- Sort using the `sorted` function.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
sorted_numbers = sorted(numbers)                                           ①

print("Original:", numbers)
print("Sorted:", sorted_numbers)
```

① Returns a new sorted data structure

- Sort with custom logic using the `key` input parameter.

```
words = ["banana", "apple", "cherry", "date"]
print("Sorted words:", sorted(words))                                    ①
print("Sorted by length:", sorted(words, key=len))                       ②

students = [
    {"name": "Alice", "score": 85},
    {"name": "Bob", "score": 92},
    {"name": "Charlie", "score": 78},
    {"name": "David", "score": 88}
]

sorted_students = sorted(students, key=lambda s: s["score"])             ③
print("Sorted by score:")
for student in sorted_students:
    print(f"{student['name']}: {student['score']}")
```

① Sorted based on alphabetical order
② Sorted based on the length of the word
③ Sorted based on score

- Sort in place using the `sort` function.

```
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
numbers.sort()                                                           ①
print("After sort():")
print("Original:", numbers)
```

① `sort` updates `numbers` in place with the sorted elements

- Find if any or all of the values in an iterable are truthy (not empty or None or False) with the `any` & `all` functions, respectively

```
your_values_list = [True, True, False, 0, None, 100]

any(your_values_list)                                                    ①
all(your_values_list)                                                    ②
```

① This will check if any of the elements in the list can be considered a true (ie, not None, empty, or 0). This will be True.
② This will check if all of the elements in the list can be considered true (ie, not None, empty, or 0). This will be False.

- Combine multiple lists into a list of tuples with `zip`

```
names = ["Alice", "Bob", "Charlie"]
ages = [25, 30, 35]
cities = ["NYC", "LA", "Chicago"]

print("Zip three lists:", list(zip(names, ages, cities)))                    ①
```

① Prints a list of tuples. Each tuple has a name, an age, and a city.

- Loop through elements in a list using `for element in list:`

```
fruits = ["apple", "banana", "cherry"]
print("Loop through list:")
for fruit in fruits:
    print(f"  {fruit}")
```

- Loop through key, values in a dict using `for key, value in dict.items():`

```
person = {"name": "Alice", "age": 25, "city": "NYC"}
for key, value in person.items():                                            ①
    print(f"  {key}: {value}")

# keys() and values()                                                        ②
# person.keys() to get only keys and person.values() to get only values
```

① `dict.items()` will return a list of key, value pairs, like `[(key1, value1), (key2, value2), ...]`
② Use `dict.keys()` and `dict.values()` to get a list of keys and values in the dict, respectively

- Use `enumerate` to get the index and element when looping

```
fruits = ["apple", "banana", "cherry"]
print("Loop with enumerate():")
for index, fruit in enumerate(fruits):
    print(f"  Index {index}: {fruit}")
```

- Write single-line loops using `comprehensions`

```
numbers = []
for i in range(5):
    numbers.append(i)
print("Traditional loop:", numbers)

numbers = [i for i in range(5)]                                    ①
print("List comprehension:", numbers)

words = ["hello", "world", "python"]
uppercase = [word.upper() for word in words]                       ②
print("Uppercase:", uppercase)
```

① `range(5)` with return numbers 0 - 4, and we use list comprehension to store them as a list
② Applying `str.upper()` for every word in `words` list

- Single line loops using `comprehensions with if..else`

```
numbers = [1, 2, 3, 4, 5, 6]
labels = ["even" if n % 2 == 0 else "odd" for n in numbers]        ①
print("Labels:", labels)
```

① Use `if..else` along with list comprehension

- Dictionary comprehension is similar to list comprehension, but with `{key: value}`

```
squares_dict = {i: i ** 2 for i in range(5)}
print("Dict comprehension:", squares_dict)
```

- Only process data as needed with `generators`

```
def count_up_to(n):
    """ Generator that counts from 1 to n"""
    i = 1
    while i <= n:
        yield i                                                    ②
        i += 1                                                     ④

print("Generator function:")
for num in count_up_to(5):                                         ①
    print(f"  {num}")                                              ③
```

① The `count_up_to` function is called

② The `count_up_to` function returns 1 and moves control to the caller
③ The `num` is printed, and `count_up_to` is called again
④ The logic continues from `i += 1` and so on, until the while condition fails

Check multiple conditions in order with `if..elif..else`

```
age = 18

if age < 13:                                                    ①
    print("You are a child")
elif age < 18:
    print("You are a teenager")
elif age < 65:
    print("You are an adult")
else:
    print("You are a senior")
```

① Conditions are checked in the order they are specified

- Check if a regex pattern exists in a string with the `re.search` function

```
import re

text = "My phone number is 123-456-7890"

if re.search(r"\d{3}-\d{3}-\d{4}", text):                       ①
    print("Phone number found!")
else:
    print("No phone number found")
```

① Search for the given regex pattern and return true if found

- Extract string matching a regex pattern with the `re.search` function

```
text = "Contact me at john@example.com"

match = re.search(r"\w+@\w+\.\w+", text)                        ①
if match:
    print(f"\nEmail found: {match.group()}")
    print(f"Position: {match.start()} to {match.end()}")
```

① Search for a given regex pattern and return the first matching substring if found, else None

- Find all strings that match a given pattern with `re.findall`

```
text = "Contact alice@example.com or bob@test.org for info"

emails = re.findall(r"\w+@\w+\.\w+", text)                                    (1)
print(f"\nAll emails: {emails}")
```

(1) Search for a given regex pattern and return a list of all the matching substrings if found, else []

- Replace regex patterns with `re.sub`

```
text = "My phone is 123-456-7890"

result = re.sub(r"\d{3}-\d{3}-\d{4}", "[REDACTED]", text)                     (1)
print(f"\nOriginal: {text}")
print(f"Replaced: {result}")
```

(1) Replace the specified regex pattern with `[REDACTED]` in the text variable

- Split a string into a list of strings with `split`

```
text = "apple,banana,cherry"
result = text.split(",")                                                      (1)
print(f"split(): {result}")
```

(1) Specify the character using which to split the string; here, it is a comma

- Replace character(s) of a string with the `replace` function.

```
text = "I love cats"
result = text.replace("cats", "dogs")                                         (1)
print(f"\nreplace(): {result}")
```

(1) Replace cats with dogs. Note that this can be done with the regex module, but this is simpler.

- Use `f-strings` to create a string interlaced with variables and functions inside `{}`

```
name = "Alice"
age = 25
expenses = 19.99
print(f"f-string: Hello, {name}! You are {age} years old. and your expenses
are ${expenses:.2f} and twice your age is {age * 2}")
```

- Ensure your code can handle errors and clean up with `try/except/finally/else` blocks

```
def divide_numbers(a, b):
    try:
        result = a / b
    except ZeroDivisionError:                                    ①
        print("Error: Cannot divide by zero")
        return None
    except TypeError:
        print("Error: Please provide numbers")
        return None
    else:                                                        ②
        print(f"Success! {a} / {b} = {result}")
        return result
    finally:                                                     ③
        print("Division attempt completed\n")

# Test cases
print("Case 1: Valid division")
divide_numbers(10, 2)

print("Case 2: Division by zero")
divide_numbers(10, 0)

print("Case 3: Invalid type")
divide_numbers(10, "abc")
```

① Code block to execute for specific types of exceptions

② Runs only if NO exception occurred
③ Always runs, whether error or not

Maintenance

- Access environment variables `os.getenv`

```
import os
os.environ['HOME']                                               ①
os.getenv('HOME', 'SOME_DEFAULT_PATH')                           ②
```

① This will fail when no HOME environment variable is set
② This will print `SOME_DEFAULT_PATH` if you do not have the `HOME` environment variable

- Log what is happening in your code with `logging`

Listing 2 `sample_log.py`

```
# add file name
import logging

logger = logging.getLogger(__name__)

logger.debug("Some debug info")
logger.info("Some info")
logger.warning("Some warning")
logger.error("Some error")
logger.critical("Some critical")
```

By default, only warning and above messages are logged. But you can run it with another level, as shown below.

```
python sample_log.py  --log=DEBUG
```

- Use `contextlib.contextmanager` to handle opening and closing connections to external systems in one place.

```
import sqlite3
from contextlib import contextmanager

@contextmanager
def sqlite_connection(db_path):                                    ①
    conn = sqlite3.connect(db_path)
    try:
        yield conn                                                 ③
        conn.commit()                                              ④
    except Exception:
        conn.rollback()                                            ⑤
        raise
    finally:
        conn.close()
```

```
with sqlite_connection('database.db') as conn:                    ②
    cursor = conn.cursor()
    cursor.execute('SELECT * FROM table_name')
    results = cursor.fetchall()
# with exit                                                       ④
```

① Handling commits, rollbacks, and re-using this function using the with
② When code enters the `with` block, the `sqlite_connection` is called
③ The `sqlite_connection` function responds with an open connection `conn`
④ Upon exiting the `with` block, control is passed back to `sqlite_connection`
⑤ The connection is closed or rolled back if there are exceptions

- Ensure that you are using the correct data type with `mypy`

Listing 3 `sample_type_check.py`

```
def get_some_data():
  return {1: 'a', 2: 'b', 3: 'c'}

data = get_some_data()
data['a']                                                         ①
```

① The key 'a' is not valid and will raise a KeyNotFound exception

```
1 python sample_type_check.py
```

However, if you had used types and a type checker, you would have been able to catch this issue before running your code.

Listing 4 `sample_type_check.py`

```
def get_some_data() -> dict[int, str]:
  return {1: 'a', 2: 'b', 3: 'c'}

data = get_some_data()
data['a']
```

```
# Check types with mypy before running the code
mypy sample_type_check.py                                                    ①
```

① mypy will catch the error and show `sample_type_check.py: error: Invalid index type "str" for "dict[int, str]"; expected type "int" [index] Found 1 error in 1 file (checked 1 source file)`

The error can be fixed before running the code.

Testing

- Ensure that your code does what it's supposed to do with `unittest`

Listing 5 `sample_test_file.py`

```
import unittest

def add(a, b):
    return a + b

# Unit tests
class TestAdd(unittest.TestCase):

    def test_add_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)                                       ①

    def test_add_negative_numbers(self):
        self.assertEqual(add(-1, -1), -2)


if __name__ == "__main__":
    unittest.main()
```

① `assert` checks if the value you get from your function is what you expect it to be

Run the tests as shown below:

```
python sample_test_file.py
```

Environment Management

- Setup a python project with `uv`

```
# Create project
uv init my-project # <1>
cd my-project

uv python pin 3.14 # <2>

uv add requests pandas mypy # <3>
```

① Create a project called `my-project`
② Define the Python version to be used
③ Add libraries that you need

- Create a virtual env for your Python project with `uv`

```
# You should have created a project first (see the above tip)
uv venv # <1>
source .venv/bin/activate # <2>
.venv\Scripts\activate # <2>

python main.py # <3>

deactivate # <4>
```

① Create a virtual environment with `uv`
② Activate the virtual environment (use `source ..` for Unix, `.venv\Scripts..` for Windows)
③ Run `main.py` inside the virtual environment
④ Deactivate the virtual environment when done

- Remove unnecessary packages with `remove`

```
uv remove requests
```

- Create a single script with the library it needs using `uv add --script`

```
# Create script
uv init --script sample_data_proc.py --python 3.14 # <1>

uv add --script sample_data_proc.py 'request' 'polars' 'pytest' # <2>
```

① Create an individual Python script with a specific Python version
② Define the libraries for this specific Python script

The created `sample_data_proc.py` will look like shown below.

Listing 6 `sample_data_proc.py`

```
# /// script
# requires-python = ">=3.14"                                    ①
# dependencies = [                                              ②
# "polars",
# "pytest",
# "request",
# ]
# ///


def main() -> None:
    print("Hello from sample_data_proc.py!")


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

① Defines the Python version. Autogenerated when creating the script with `uv`
② Defines the libraries. Autogenerated when adding libraries