PART 8 OF 8

Python Self-Learning Booklet 8: Modules, Files, and Exceptions

Welcome to the final session! In this booklet you will learn three powerful skills: how to **organize your code into modules**, how to **read and write files** (including JSON), and how to **handle errors gracefully** so your programs never crash unexpectedly.

📦 Modules

Split code into reusable files

📄 Files

Read and write text and JSON

🛡️ Exceptions

Handle errors without crashing

LEARNING GOALS

What You Will Be Able to Do

By the end of this session, you should be able to accomplish all of the following independently:

1

Create and use modules

Split programs into multiple `.py` files and import them.

2

Use all import styles

Single, multiple, whole module, and alias imports.

3

Read and write text files

Load data from disk and save results using `pathlib.Path`.

4

Work with JSON

Save and load structured data like dictionaries and lists.

5

Handle exceptions

Use `try/except/else` to manage bad input and missing files.

CONCEPT 1

Modules: Organizing Code into Files

A **module** is simply a `.py` file that another Python file can import. As your programs grow, modules keep things tidy, reusable, and easy to maintain.

Keep code tidy

Group related classes and functions in one place instead of one giant file.

Reuse code

Import the same module in many programs without copying and pasting.

Compose programs

Modules can import other modules, letting you build complex programs from small pieces.

CONCEPT 1 · MODULES

Creating Your First Module

Save a class inside a file called `car.py`. That file is now a module named `car`. Any other script can import from it.

```
# car.py
class Car:
    def __init__(self, make):
        self.make = make
```

The filename (without `.py`) becomes the module name. Keep module names lowercase with no spaces.

Now create a separate script and import the `Car` class from your new module:

```
from car import Car

my_car = Car("Audi")
print(my_car.make)   # Audi
```

What this does: `from car import Car` brings only the `Car` class into your file. You can use `Car(...)` directly without prefixing it with the module name.

CONCEPT 1 · MODULES

Four Ways to Import

Python gives you several import styles. Each has a different use case. Learn all four so you can read other people's code too.

1

Single class

`from car import Car`

Import one specific name. Use it directly.

2

Multiple classes

`from car import Car, ElectricCar`

Import several names, separated by commas.

3

Whole module

`import car`

Access everything via `car.Car(...)`. Avoids name conflicts.

4

Alias

`import electric_car as ec`

Create a short nickname for long module or class names.

CONCEPT 1 · MODULES

Importing Multiple Classes and Using Aliases

Multiple imports

```
from car import Car, ElectricCar

my_car = Car("Audi")
my_ev  = ElectricCar("BYD", 10000)
```

One `from ... import` line can list several names separated by commas.

Aliases

```
from electric_car import ElectricCar as EC

leaf = EC("Nissan", 40000)

# or alias the whole module
import electric_car as ec
car_obj = ec.ElectricCar("Tesla", 50000)
```

Use `as` to create a shorter name. Very handy for long class or module names.

CONCEPT 1 · MODULES

Modules Can Import Other Modules

One of the most powerful ideas: a module can import from another module. This lets you build class hierarchies across files.

```
# electric_car.py
from car import Car          # import Car from car.py

class ElectricCar(Car):
    def __init__(self, make, battery):
        super().__init__(make)   # runs Car.__init__
        self.battery = battery
```

What `super().__init__(make)` does

Calls the parent class `Car`'s constructor so `self.make` is set properly – even though the code lives in a different file.

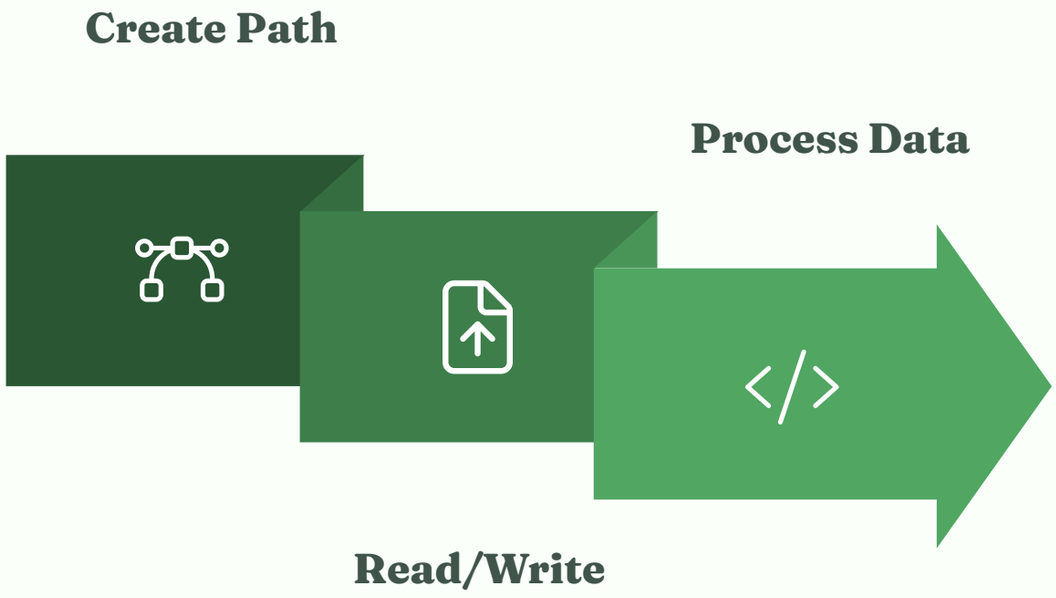
Result

`ElectricCar` inherits all of `Car`'s behaviour and adds a `battery` attribute on top.

CONCEPT 2

Reading and Writing Text Files

Programs become far more useful when they can load data from disk and save results. Python's `pathlib.Path` is the beginner-friendly way to work with files.



These three steps cover almost everything you need for basic file handling. The sections below walk through each operation with code examples.

CONCEPT 2 · FILES

Reading a File

Use `Path` from `pathlib` to point to a file, then call `read_text()` to load its entire contents into a string.

```
from pathlib import Path

path = Path('pi_digits.txt')
contents = path.read_text()
print(contents)
```

`Path('pi_digits.txt')` looks in the **same folder** as your script. If the file is somewhere else, you need to provide the full path.

To work line by line, strip trailing whitespace and split into a list:

```
contents = path.read_text()
contents = contents.rstrip()      # remove trailing \n
lines    = contents.splitlines()  # list of lines

for line in lines:
    print(line)
```

`rstrip()` removes trailing whitespace. `splitlines()` turns the text into a list like `["line1", "line2", ...]`.

CONCEPT 2 · FILES

Writing to a File

Single line

```
from pathlib import Path

path = Path('output.txt')
path.write_text("Hello from Python!\n")
```

Creates `output.txt` if it doesn't exist, or **overwrites** it if it does.

Multiple lines

```
from pathlib import Path

path = Path("lines.txt")

contents  = "Line 1\n"
contents += "Line 2\n"

path.write_text(contents)
```

Build one string with `\n` between lines, then write it all at once.

⚠️ **Important:** `write_text()` always **overwrites** the file. It does not append. To preserve old content, read it first, combine it with new text, then write the whole thing.

CONCEPT 3

JSON Files: Saving Structured Data

JSON (JavaScript Object Notation) is a text format that works perfectly with Python dictionaries and lists. It's the standard way to save and share structured data between programs.

json.dumps()

Converts a Python object (dict, list) **into** a JSON string so you can save it to a file.

json.loads()

Converts a JSON string **back into** a Python object so you can use it in your code.

Think of `dumps` as "dump to string" and `loads` as "load from string." Together they let you persist any Python data structure to disk.

CONCEPT 3 · JSON

Saving and Loading a Dictionary with JSON

Save to file

```
import json
from pathlib import Path

settings = {
    "language": "en",
    "theme": "dark"
}

path = Path("settings.json")
json_text = json.dumps(settings)
path.write_text(json_text)
```

Converts the dictionary to a JSON string, then writes it to `settings.json`.

Load from file

```
import json
from pathlib import Path

path = Path("settings.json")
json_text = path.read_text()
settings  = json.loads(json_text)

print(settings["language"])  # en
print(settings["theme"])     # dark
```

Reads the text back and converts it to a Python dictionary you can use normally.

CONCEPT 4

Exceptions: Handling Errors Gracefully

An **exception** is a runtime error that stops your program. Instead of letting it crash, you can *catch* the error and respond helpfully.

ZeroDivisionError

Dividing any number by zero.

ValueError

Converting non-numeric text to int() or float().

FileNotFoundError

Reading a file that doesn't exist on disk.

These three are the most common exceptions beginners encounter. The next cards show how to handle each one.

CONCEPT 4 · EXCEPTIONS

Basic `try/except`

Wrap risky code in a `try` block. If an error occurs, the matching `except` block runs instead of crashing the program.

Without exception handling ❌

```
print(5 / 0)
# ZeroDivisionError: division by zero
# Program stops here!
```

The program crashes and shows a confusing error message to the user.

With `try/except` ✅

```
try:
    print(5 / 0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Python tries the division. If it fails, the `except` block handles it – no crash.

CONCEPT 4 · EXCEPTIONS

The `else` Block

The optional `else` block runs **only if the `try` block succeeded** (no exception was raised). It keeps your "success" logic cleanly separated from error handling.

```
try:
    result = 10 / 2
except ZeroDivisionError:
    print("Error!")
else:
    print(result)   # 5.0  — only runs if no error occurred
```

1	2	3
try	**except**	**else**
Run risky code	Handle the error	Success path only

CONCEPT 4 · EXCEPTIONS

Handling Bad User Input (`ValueError`)

`input()` always returns a **string**. Calling `int()` on something like `"abc"` raises a `ValueError`. Always wrap user input conversions in `try/except`.

```
try:
    n = int(input("Enter a number: "))
except ValueError:
    print("That's not a number. Please try again.")
```

Rule of thumb: any time you convert `input()` to a number, wrap it in `try/except ValueError`. This is one of the most common patterns in beginner Python programs.

CONCEPT 4 · EXCEPTIONS

Handling Missing Files (`FileNotFoundError`)

If you try to read a file that doesn't exist, Python raises `FileNotFoundError`. Catch it to show a friendly message instead of a crash.

Basic handling

```
from pathlib import Path

path = Path("unknown.txt")

try:
    text = path.read_text()
except FileNotFoundError:
    print("File not found.")
```

With `else`

```
from pathlib import Path

path = Path("unknown.txt")

try:
    text = path.read_text()
except FileNotFoundError:
    print("File missing.")
else:
    print(text)   # only if read succeeded
```

The `else` version is preferred – it makes clear that printing the text only happens when the file loaded successfully.

CONCEPT 4 · EXCEPTIONS

Chaining Multiple `except` Blocks

One `try` block can have multiple `except` clauses – one for each type of error you want to handle differently.

```
try:
    number = int(input("Enter number: "))
    result = 10 / number
except ValueError:
    print("Please enter a valid integer.")
except ZeroDivisionError:
    print("Number cannot be zero.")
```

User types `abc`

`int("abc")` fails → `ValueError` is caught → prints integer message.

User types `0`

`10 / 0` fails → `ZeroDivisionError` is caught → prints zero message.

User types `2`

No error → result is `5.0`. Neither except block runs.

STEP-BY-STEP EXAMPLE

Putting It All Together: User Profile App

This example combines modules, files, JSON, and exceptions into one small but complete program. It asks for user details, validates input, saves a profile to disk, and loads it back safely.

01

Collect input

Ask the user for `name`, `country`, and `age` – handling invalid age with `ValueError`.

02

Build a dictionary

Store all three values in a Python dictionary called `profile`.

03

Save to JSON

Use `json.dumps()` and `write_text()` to persist the profile.

04

Load safely

Use `try/except FileNotFoundError` with `else` to read the file back.

05

Print summary

Display a friendly sentence using the loaded dictionary values.

STEP-BY-STEP EXAMPLE

The Full Code

```
import json
from pathlib import Path

# 1) Get user data safely (handle ValueError)
name    = input("Name: ")
country = input("Country: ")

try:
    age = int(input("Age: "))
except ValueError:
    print("Age must be a whole number. Using age = 0.")
    age = 0

profile = {"name": name, "age": age, "country": country}

# 2) Save to JSON
path      = Path("user_profile.json")
json_text = json.dumps(profile)
path.write_text(json_text)

print("Saved profile to user_profile.json")

# 3) Load back safely (handle FileNotFoundError)
try:
    loaded_text = path.read_text()
except FileNotFoundError:
    print("Could not load the profile file (missing).")
else:
    loaded_profile = json.loads(loaded_text)

    # 4) Use the loaded data
    print(f"{loaded_profile['name']} from "
          f"{loaded_profile['country']} is "
          f"{loaded_profile['age']} years old.")
```

STEP-BY-STEP EXAMPLE

How Each Part Works

1 Input collection

`name` and `country` are safe – they're already strings. `age` needs `int()`, which can fail, so it's wrapped in `try/except ValueError`.

2 Saving JSON

`json.dumps(profile)` converts the dictionary to a JSON string. `write_text()` saves it. If the file doesn't exist yet, Python creates it automatically.

3 Loading safely

The `try/except FileNotFoundError` guards the read. The `else` block only runs when the read succeeded – this is the clean, Pythonic pattern.

4 Using the data

After `json.loads()`, `loaded_profile` is a regular Python dictionary. Access values with square brackets: `loaded_profile['name']`.

COMMON MISTAKES

5 Mistakes to Avoid

These are the errors beginners make most often in this topic. Read them now – it will save you debugging time later.

❌ Forgetting the module prefix

If you use `import car`, you must write `car.Car(...)` – not just `Car(...)`. Only `from car import Car` lets you skip the prefix.

❌ Wrong file path

`Path("notes.txt")` looks in the **same folder** as your script. If the file is elsewhere, Python raises `FileNotFoundError`.

❌ Expecting `write_text()` to append

`write_text()` always **overwrites**. To preserve old content, read it first, combine it, then write the whole string.

❌ Doing math on raw input

`input()` returns a string. Always convert with `int()` or `float()` – and always wrap that conversion in `try/except ValueError`.

❌ Catching the wrong exception

Match the exception to the problem: `FileNotFoundError` for missing files, `ValueError` for bad number conversions. Don't use a bare `except:` – it hides bugs.

PRACTICE TASKS

Practice Task 1: Mini Module (Greeting)

This task practices creating a module and importing a function from it.

Step 1 — Create tools.py

```
def hello():
    print("Hello! Welcome to Python.")
```

Step 2 — Create main.py

```
from tools import hello

hello()
```

Run `main.py`. You should see the greeting printed. Both files must be in the **same folder**. Try also importing the whole module using `import tools` and calling `tools.hello()`.

PRACTICE TASKS

Practice Task 2: Math Tools Module

Create a module with two functions, then use them from a separate script with user input.

math_tools.py

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

def multiply(a, b):
    return a * b
```

main.py

```
from math_tools import add, multiply


try:
    x = float(input("First number: "))
    y = float(input("Second number: "))
except ValueError:
    print("Please enter valid numbers.")
else:
    print("Sum:", add(x, y))
    print("Product:", multiply(x, y))
```

Notice how `try/except ValueError` is used here too – good practice whenever you convert user input to a number!

PRACTICE TASKS

Practice Task 3: Read Lines from a Text File

Create a plain text file, then write a script that reads and prints it line by line.

Create `notes.txt`

Open any text editor and save a file with three lines, e.g. *"Python is fun."*, *"Files are useful."*, *"Modules keep code tidy."*

Write your script

```
from pathlib import Path

path     = Path("notes.txt")
contents = path.read_text()
lines    =
contents.rstrip().splitlines()

for line in lines:
    print(line)
```

Check the output

Each of your three lines should print on its own line, with no extra blank lines at the end. If you see an empty last line, check that `rstrip()` is applied.

PRACTICE TASKS

Practice Tasks 4 & 5: Writing Files and JSON

Task 4 — Write user sentences

Ask the user for three sentences using `input()`. Save them all to `sentences.txt`, each on its own line.

```
from pathlib import Path

s1 = input("Sentence 1: ")
s2 = input("Sentence 2: ")
s3 = input("Sentence 3: ")

contents  = s1 + "\n"
contents += s2 + "\n"
contents += s3 + "\n"

Path("sentences.txt").write_text(contents)
```

Task 5 — User profile JSON

Create a dictionary with `name`, `age`, `country`. Save it to `user_profile.json`. In a separate script, load it back and print a sentence.

```
# Script 1: save
import json
from pathlib import Path

profile = {"name": "Alex", "age": 20,
           "country": "Canada"}
path = Path("user_profile.json")
path.write_text(json.dumps(profile))

# Script 2: load
json_text = path.read_text()
p = json.loads(json_text)
print(f"{p['name']} is {p['age']}.")
```

EXPLORATION QUESTIONS

Think Deeper: Exploration Questions

These questions encourage you to experiment beyond the examples. Try each one in your code editor – the best way to learn is by testing your ideas.

1

Invalid JSON

What happens if you run your JSON loading code when `settings.json` is empty or contains corrupted text? What exception do you get? How would you handle it?

2

Import styles

Try `import car as c` versus `from car import Car` in the same small program. Which feels clearer to you? When might each style be preferred?

3

Validate age further

Modify the Step-by-Step Example so that if `age` is negative, the program prints a message and uses `0` instead. You don't need a `try/except` for this – an `if` statement after the `else` is enough.

SUMMARY

Session 8 Summary

You've covered a lot of ground. Here's everything in one place – use this as a quick reference when you're working on your own projects.

Modules

A `.py` file is a module. Import specific names, whole modules, or use aliases with `as`.

Text Files

`Path.read_text()` reads, `Path.write_text()` writes. Use `rstrip()` and `splitlines()` to process lines.

JSON

`json.dumps()` saves Python objects as text. `json.loads()` converts them back. Great for dictionaries and lists.

Exceptions

`try/except/else` catches `ValueError`, `ZeroDivisionError`, and `FileNotFoundError` before they crash your program.

SUMMARY

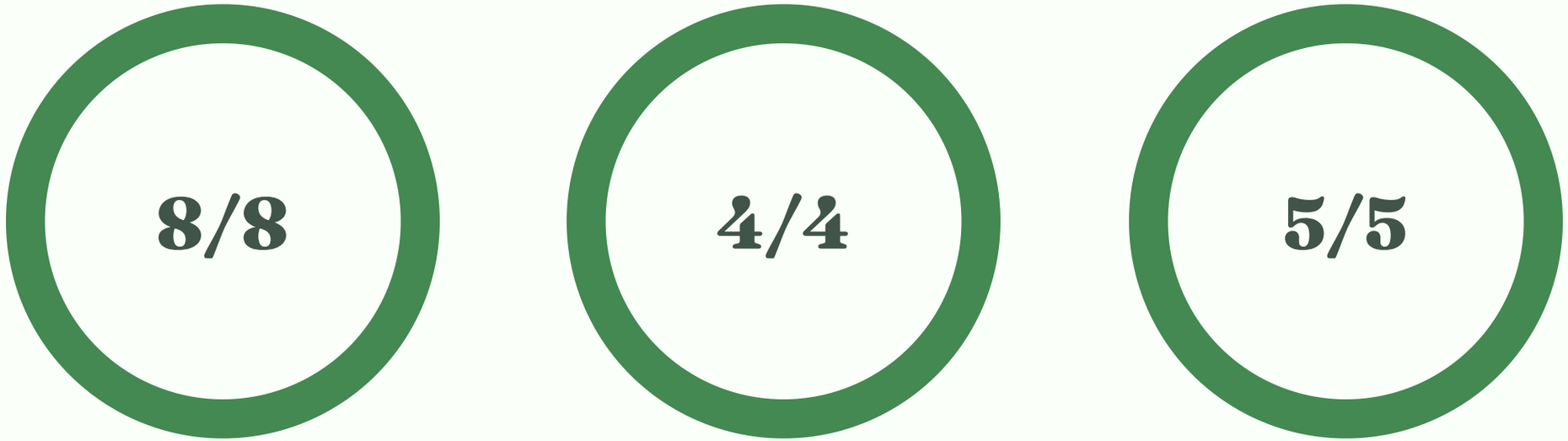
Key Syntax at a Glance

Syntax	What it does
`from car import Car`	Import one class from a module
`import car`	Import whole module; access via `car.Car(...)`
`import car as c`	Import whole module with alias `c`
`Path('file.txt').read_text()`	Read entire file as one string
`Path('file.txt').write_text(s)`	Write string `s` to file (overwrites)
`json.dumps(obj)`	Python object → JSON string
`json.loads(text)`	JSON string → Python object
`try: ... except ValueError:`	Catch bad number conversion
`try: ... except FileNotFoundError:`	Catch missing file error
`else:` after `except`	Runs only if no exception was raised

🎉 COMPLETE

Congratulations — You've Finished All 8 Sessions!

You have now completed the full Python Self-Learning Booklet series. From variables and loops all the way to modules, files, and exceptions – you've built a solid Python foundation.



Sessions complete

All booklets finished

Core concepts

Modules, files, JSON, exceptions

Practice tasks

Hands-on coding exercises

> Keep building. The best way to deepen your Python skills is to write real programs that solve real problems. Use modules to stay organized, files to persist data, and exceptions to make your programs robust. You're ready. 🐍