From Functions to Small Projects

Python Course · Module 5.5

ElnurBDa

Module 5.5 — Function-Driven Mini Projects

Goals:

- Use functions and modules to build a **slightly larger script**
- Practice the `main()` pattern with imports
- Prepare to add **files, logging, and error handling** in Module 6

Think: “one step up” from tiny examples toward real tools.

Review — Project Layout with Modules

From Module 5:

```
project/
├── main.py
├── utils.py
└── math_ops.py
```

Imports:

```
from utils import greet
from math_ops import multiply
```

In this module we'll build similar structures with **clear responsibilities**.

Designing a Small CLI Tool

We'll keep using `input()` for now, but we'll **separate logic**:

Example: simple unit converter project:

```
converter_project/
├── main.py
└── converter.py
```

`converter.py`:

```
KM_IN_MILE = 1.60934

def km_to_miles(km: float) -> float:
    return km / KM_IN_MILE

def celsius_to_fahrenheit(c: float) -> float:
    return c * 9 / 5 + 32
```

———————————— [finished] ————————————

`main.py`:

```
from converter import km_to_miles, celsius_to_fahrenheit

def main():
    choice = input("Convert (k)m or (t)emperature? ").strip().lower()
    if choice == "k":
        km = float(input("Kilometers: "))
        print("Miles:", km_to_miles(km))
    elif choice == "t":
        c = float(input("Celsius: "))
        print("Fahrenheit:", celsius_to_fahrenheit(c))
    else:
        print("Unknown option")

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

———————————— [finished with error] ————————————

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presenterm6svXk1/snippet.py", line 1, in
<module>
```

Separating Pure Logic from I/O

Pure functions:

- Do not call `input()` or `print()`
- Only work with parameters and return values

This makes them:

- Easier to test
- Easier to reuse (CLI, web API, GUI)

In the previous example, all conversion logic is inside `converter.py`, while `main.py` only handles user interaction.

Module 6 will plug **file I/O** into a similar structure.

Mini Practice — Grade Utility

Design:

```
grades_project/
├── main.py
└── grades.py
```

`grades.py` should contain:

- `average(scores)`
- `letter_grade(score)` → `"A"`, `"B"`, ...
- `summarize(scores)` → dict with average, highest, lowest, pass_count

`main.py`:

- Reads a comma-separated list of numbers via `input()`
- Uses functions from `grades.py`
- Prints a short report

You will reuse this idea when reading grades from files in Module 6.

Introducing Simple Configuration

Before we start reading real config files, we can simulate configuration using a **separate module**:

```
settings_project/
├── config.py
└── main.py
```

config.py:

```
APP_NAME = "Demo App"
DEFAULT_LOG_LEVEL = "INFO"
MAX_ITEMS = 100
```

──────────────────── [finished] ────────────────────

main.py:

```
import config

def main():
    print("Starting", config.APP_NAME)
    print("Log level:", config.DEFAULT_LOG_LEVEL)
    print("Max items:", config.MAX_ITEMS)

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

──────────────── [finished with error] ─────────────────

```
Traceback (most recent call last):
  File "/tmp/nix-shell-61594-0/.presentermAnGYZc/snippet.py", line 1, in
<module>
    import config
ModuleNotFoundError: No module named 'config'
```

Later, Module 6/10 will replace some of these constants with **file- or env-based** configs.

Mini Project — Simple Menu-Driven App

Create `menu_project/`:

```
menu_project/
├── main.py
└── actions.py
```

`actions.py`:

- `def say_hello(name):` → prints greeting
- `def add_numbers(a, b):` → returns sum
- `def reverse_text(s):` → returns reversed string

`main.py`:

1. Shows a numbered menu: Hello, Add, Reverse, Quit.
2. Uses a loop + `if/elif` to call the right function in `actions.py`.
3. Keeps running until user chooses Quit.

This prepares you for more complex menus and CLIs in later modules.
