CLI Tools, Logging & Advanced Python

Python Course · Module 11

ElnurBDa

Module 11 — Command-Line Tools & Advanced Python

In this module we:

- Build **real command-line interfaces (CLIs)** using `argparse`
- Replace `input()`-only scripts with tools that accept flags and subcommands
- Introduce **logging** instead of `print()` for serious scripts
- Explore a few **advanced Python features** you will see in real projects

Focus: make your scripts feel like **professional tools** and understand patterns you'll meet in open-source and production code.

From Scripts to Tools

So far many examples used:

```
name = input("Enter your name: ")
print("Hello", name)
```

This is great for learning, but less ideal when you want to:

- Automate tasks from other scripts or CI
- Run jobs from cron / Task Scheduler
- Reuse the same script with different options

Command-line tools use **arguments** and **flags** instead:

```
python greet.py --name Alice
```

We'll use argparse from the standard library to build these.

Basics of argparse

Pattern:

1. Create a parser
2. Define arguments
3. Parse `sys.argv` into a friendly object

```python
import argparse

def main():
    parser = argparse.ArgumentParser(
        description="Greet someone from the command line."
    )
    parser.add_argument("--name", "-n", required=True, help="Name to greet")

    args = parser.parse_args()

    print(f"Hello, {args.name}!")

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

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

```
usage: snippet.py [-h] --name NAME
snippet.py: error: the following arguments are required: --name/-n
```

Usage:

```
python greet.py --name Alice
python greet.py -n Bob
```

Try running with `--help` to see automatically generated help text.

Flags, Options & Positional Arguments

argparse supports:

- **Positional args** — required, order matters
- **Options / flags** — usually start with -- or -

Example:

```python
import argparse

def main():
    parser = argparse.ArgumentParser(description="Convert kilometers to miles.")
    parser.add_argument("kilometers", type=float, help="Distance in km")
    parser.add_argument(
        "--precision",
        "-p",
        type=int,
        default=2,
        help="Number of decimal places (default: 2)",
    )

    args = parser.parse_args()
    miles = args.kilometers * 0.621371
    print(f"{args.kilometers} km = {miles:.{args.precision}f} miles")

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

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

```
usage: snippet.py [-h] [--precision PRECISION] kilometers
snippet.py: error: the following arguments are required: kilometers
```

Usage:

```
python convert.py 10
python convert.py 10 --precision 4
```

———

Subcommands (Like git commit, git push)

Many tools have **subcommands**:

- `git status`, `git commit`, `git push`
- `docker run`, `docker ps`

We can model this with **sub-parsers**.

```python
import argparse

def cmd_add(args):
    print("Adding task:", args.title)

def cmd_list(args):
    print("Listing tasks; completed =", args.completed)

def build_parser():
    parser = argparse.ArgumentParser(prog="tasks", description="Simple task CLI")
    subparsers = parser.add_subparsers(dest="command", required=True)

    add_p = subparsers.add_parser("add", help="Add a new task")
    add_p.add_argument("title", help="Title of the task")
    add_p.set_defaults(func=cmd_add)

    list_p = subparsers.add_parser("list", help="List tasks")
    list_p.add_argument(
        "--completed",
        action="store_true",
        help="Show only completed tasks",
    )
    list_p.set_defaults(func=cmd_list)

    return parser

def main():
    parser = build_parser()
    args = parser.parse_args()
    args.func(args)  # dispatch to the right handler

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

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

```
usage: tasks [-h] {add,list} ...
tasks: error: the following arguments are required: command
```

Usage:

Connecting CLIs to Files & Databases

You can wire argparse into your existing **file** and **SQLite** code:

- tasks.py add → insert into SQLite (Module 7)
- tasks.py export --format csv → read rows and write CSV (Module 6)
- students.py deactivate --email student@example.com → update database row

Pattern:

- Each subcommand calls into a **function** in another module (db.py, services.py, etc.).
- CLI layer **parses arguments**, then passes them to core logic.

This keeps your business logic **testable** and your CLI thin.

Mini Task – Log Tracker CLI

Extend your **log tracker** project (Module 6) into a CLI:

- Create `log_cli.py` with `argparse`:
  - Subcommands: `add`, `list`, `clear`
  - Flags: `--user`, `--action`, maybe `--level` (INFO/WARN/ERROR)
- Internally, call functions from a separate module (e.g., `log_core.py`) that:
  - open the log file
  - append or read lines
  - handle errors gracefully

Goal: run commands like:

```
python log_cli.py add --user alice --action "login"
python log_cli.py list
python log_cli.py clear
```

Why Logging Beats print()

`print()` is fine while learning, but real programs use **logging**:

- Different **levels** (DEBUG, INFO, WARNING, ERROR, CRITICAL)
- Can write to **files**, not just the console
- Easier to turn on/off detailed output without changing code

Python's `logging` module is built-in and flexible.

Logging Basics

Simple pattern:

```
import logging

logging.basicConfig(
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(message)s",
)

logging.debug("This is a debug message")
logging.info("Starting application")
logging.warning("Low disk space")
logging.error("Something went wrong")
```

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

```
2025-11-28 18:34:25,012 [INFO] Starting application
2025-11-28 18:34:25,012 [WARNING] Low disk space
2025-11-28 18:34:25,012 [ERROR] Something went wrong
```

Notes:

- `basicConfig` sets a **global** configuration for the root logger
- By default, output goes to **stderr**
- Change `level=` to control how verbose logs are

Try changing `level=logging.DEBUG` to see all messages.

Log Levels in Practice

Common use:

- DEBUG – very detailed, for developers only
- INFO – high-level events (startup, shutdown, key actions)
- WARNING – something unexpected, but program continues
- ERROR – serious problem, part of the feature failed
- CRITICAL – very bad, program may exit

Example:

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def divide(a, b):
    logging.debug("divide called with a=%s, b=%s", a, b)
    if b == 0:
        logging.error("Attempted division by zero!")
        return None
    return a / b

result = divide(10, 0)
logging.info("Result is %s", result)
```

———————————— [finished] ————————————

```
DEBUG:root:divide called with a=10, b=0
ERROR:root:Attempted division by zero!
INFO:root:Result is None
```

Note the %s style placeholders – logging handles formatting lazily.

Logging to a File

Instead of writing only to the console, log to a **file**:

```python
import logging

logging.basicConfig(
    filename="app.log",
    level=logging.INFO,
    format="%(asctime)s [%(levelname)s] %(name)s: %(message)s",
)

logger = logging.getLogger("demo")

logger.info("Application started")
logger.warning("This is a warning")
logger.error("This is an error")
```

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

Open `app.log` in your editor to inspect messages.

Tips:

- Use `getLogger(__name__)` in real modules for better names
- Combine with security best practices (Module 10): **do not** log secrets

---

Mini Task – Add Logging to an Existing Project

Pick one earlier project:

- File log tracker (Module 6)
- SQLite score tracker (Module 7)
- Student Manager / To-Do backend (Module 9)

Add logging:

1. Configure `logging` in the main entry point.
2. Replace key `print()` calls with appropriate log levels.
3. Ensure that **no sensitive info** (passwords, tokens) is logged.
4. Trigger some errors to see how they appear in logs.

Advanced Python — Comprehensions Recap

You've seen list comprehensions. Quick recap:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
evens = [n for n in nums if n % 2 == 0]

print("squares:", squares)
print("evens:", evens)
```

——————————— [finished] ———————————

```
squares: [1, 4, 9, 16, 25]
evens: [2, 4]
```

Two more related tools:

- **Dict comprehensions**
- **Generator expressions**

We'll see both next.

Dict Comprehensions

Build dictionaries in one expression:

```
users = ["alice", "bob", "charlie"]

lengths = {name: len(name) for name in users}
print(lengths)
```

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

```
{'alice': 5, 'bob': 3, 'charlie': 7}
```

You can also transform existing dicts:

```
scores = {"alice": 90, "bob": 75, "charlie": 82}

passed = {name: s for name, s in scores.items() if s >= 80}
print(passed)
```

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

```
{'alice': 90, 'charlie': 82}
```

Use case: quick mappings, filtered views, basic data transformations.

Generator Expressions

List comprehensions materialize a list in memory. **Generator expressions** produce values **on demand**:

```
nums = range(1, 1_000_001)

total = sum(n * n for n in nums)
print("Sum of squares from 1 to 1,000,000:", total)
```

———————————— [finished] ————————————

```
Sum of squares from 1 to 1,000,000:
333333833333500000
```

Key differences:

- `[...]` → list comprehension (immediate list)
- `(...)` → generator expression (lazy)

Use generators when:

- The result is large
- You only need to **iterate once** or feed another function like `sum`, `max`

---

*args and **kwargs

Sometimes you don’t know in advance how many arguments a function needs.

- *args – extra **positional** arguments (tuple)
- **kwargs – extra **keyword** arguments (dict)

```
def debug_print(*args, **kwargs):
    print("ARGS:", args)
    print("KWARGS:", kwargs)

debug_print(1, 2, 3, name="alice", active=True)
```

———————————— [finished] ————————————

```
ARGS: (1, 2, 3)
KWARGS: {'name': 'alice', 'active': True}
```

Typical uses:

- Wrapper functions that pass arguments through
- Flexible APIs where some options are optional/rare

Default Parameters & Keyword-Only Arguments

You've seen simple defaults:

```
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")
greet("Bob", greeting="Hi")
```

———————— [finished] ————————

```
Hello, Alice!
Hi, Bob!
```

We can also force arguments to be **keyword-only** (for clarity):

```
def create_user(username, *, is_admin=False):
    print("Creating user:", username, "is_admin:", is_admin)

create_user("alice")
create_user("bob", is_admin=True)
# create_user("charlie", True)  # TypeError: must use keyword
```

———————— [finished] ————————

```
Creating user: alice is_admin: False
Creating user: bob is_admin: True
```

Use keyword-only args when positional use would be confusing.

Simple Decorators (Concept)

Decorators are a powerful pattern for **wrapping** functions with extra behavior.

Example: log when a function is called.

```python
import time

def log_calls(func):
    def wrapper(*args, **kwargs):
        print(f"[log_calls] calling {func.__name__} with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        print(f"[log_calls] {func.__name__} returned {result}")
        return result
    return wrapper

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

add(2, 3)
```

———————————————————— [finished] ————————————————————

```
[log_calls] calling add with args=(2, 3), kwargs={}
[log_calls] add returned 5
```

Pattern:

1. Decorator takes a function (`func`)
2. Returns a new function (`wrapper`) that adds behavior
3. `@decorator_name` syntax applies it to `add`

———

Timing Functions with a Decorator

Another common use: measure how long a function takes.

```python
import time

def timed(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        duration = (time.perf_counter() - start) * 1000
        print(f"{func.__name__} took {duration:.2f} ms")
        return result
    return wrapper

@timed
def slow_operation():
    time.sleep(0.2)

slow_operation()
```

———————————— [finished] ————————————

```
slow_operation took 200.08 ms
```

Real projects use decorators for:

- Logging
- Caching
- Authorization checks
- Retrying failed operations

Mini Project – CLI Admin Tool

Tie everything together by building a **CLI admin tool** for one of your apps:

- Example: Student Manager (`students` table in SQLite)
- Or: To-Do app tasks (Module 9)

Requirements:

1. Use `argparse` with subcommands like `list`, `create`, `deactivate`, `stats`.
2. Connect subcommands to database functions (reuse Module 7 patterns).
3. Use `logging` to record actions (who, what, when) to a file.
4. Use comprehensions / generator expressions for simple summaries (e.g., count active students).
5. Add at least one decorator (`@timed` or `@log_calls`) around a database operation.

You now have the pieces to turn small scripts into **serious tools** that are debuggable, observable, and maintainable.
