Writing and Using Driver Scripts in Python

1. Introduction: What Is a Driver Script?

A **driver script** (often called a "test driver" or "main driver") is a short Python program that exists **to run and test other pieces of code**, functions, classes, or modules, usually during development.

If your module is like a car engine, a driver script is the person turning the ignition and testing how the engine runs.

Driver scripts are indispensable in software development because they let you:

- Quickly check that a function behaves as expected.
- Test modules independently before integration.
- Produce reproducible examples and debugging runs.
- Serve as **entry points** for larger programs.

In essence, a driver script is *temporary glue code* that helps you **validate logic before building full applications**.

2. The Simplest Form: A Manual Function Test

Imagine you've written a small module called `math_utils.py` containing a few functions:

```
# math_utils.py
def add(a, b):
    return a + b

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

A driver script to test it might look like this:

```
# driver_math.py
import math_utils

print("Testing math_utils functions:")
print("2 + 3 =", math_utils.add(2, 3))
print("4 * 5 =", math_utils.multiply(4, 5))
```

Run it:

```
python driver_math.py
```

Output:

```
Testing math_utils functions:
2 + 3 = 5
4 * 5 = 20
```

This is the **simplest driver pattern**, you import the module you want to test, run its functions, and print or inspect the results.

3. Using `if __name__ == "__main__"`

A good practice in Python is to let your modules act both as importable libraries *and* as standalone drivers.

```
# math_utils.py
def add(a, b):
    return a + b

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

if __name__ == "__main__":
    # This block runs only when the script is executed directly
    print("Running math_utils driver tests...")
    print(add(2, 3))
    print(multiply(3, 7))
```

Now:

- Running `python math_utils.py` executes the test section.
- Importing it (`import math_utils`) does **not** execute that block.

This is one of the most common lightweight driver setups in Python.

4. Driver Scripts as Development Sandboxes

Driver scripts are often used as **sandboxes** to test and debug functionality before integration.

Example: id you are developing a text-cleaning function.

```
# text_tools.py
def clean_text(s):
    return s.strip().lower().replace(" ", "_")
```

Instead of building a whole program, create a small driver for interactive testing:

```
# driver_text_tools.py
```

```
from text_tools import clean_text

examples = ["  Hello World  ", "Python  Rocks", "   Mixed  CASE  "]

print("Testing clean_text:")
for text in examples:
    print(f"{text!r} -> {clean_text(text)!r}")
```

When run:

```
Testing clean_text:
'  Hello World  ' -> 'hello_world'
'Python  Rocks' -> 'python__rocks'
'   Mixed  CASE  ' -> 'mixed__case'
```

Driver scripts like this let you iterate fast, tweak logic, and see results immediately, especially useful before adding automated tests.

5. Parameterized Drivers

Instead of hard-coding inputs, you can make your driver accept command-line arguments using the **`argparse`** module. This makes the driver more reusable.

```
# driver_calculator.py
import argparse
from math_utils import add, multiply

parser = argparse.ArgumentParser(description="Test math_utils
operations.")
parser.add_argument("a", type=int, help="First number")
parser.add_argument("b", type=int, help="Second number")
parser.add_argument("--op", choices=["add", "multiply"],
default="add")

args = parser.parse_args()

if args.op == "add":
    result = add(args.a, args.b)
else:
    result = multiply(args.a, args.b)

print(f"Result: {result}")
```

Run it from the terminal:

```
python driver_calculator.py 3 4 --op multiply
```

Output:

```
Result: 12
```

This approach transforms your test driver into a mini command-line tool.

6. Structured Testing Drivers

Once your code grows, you will want more systematic testing than just print statements. Python provides the `unittest` framework (and third-party options like `pytest`).

Here's a **driver test suite** using `unittest`:

```
# test_math_utils.py
import unittest
from math_utils import add, multiply

class TestMathUtils(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertNotEqual(add(2, 3), 6)

    def test_multiply(self):
        self.assertEqual(multiply(4, 5), 20)
        self.assertTrue(multiply(2, 0) == 0)

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

Run it:

```
python test_math_utils.py
```

Output:

```
..
----------------------------------------------------------------------
-
Ran 2 tests in 0.000s
OK
```

This is still a **driver**, but now it is automated, it drives your code through multiple scenarios and asserts correctness.

7. Drivers for Modules with Side Effects

Sometimes functions interact with files, APIs, or databases. Drivers are perfect for validating those interactions safely.

Example: Testing file operations

```
# file_ops.py
def read_lines(path):
    with open(path, "r", encoding="utf-8") as f:
        return [line.strip() for line in f]
```

Driver:

```
# driver_file_ops.py
from file_ops import read_lines

try:
    lines = read_lines("example.txt")
    print(f"Read {len(lines)} lines:")
    for l in lines:
        print(">", l)
except FileNotFoundError:
    print("File not found, test skipped.")
```

You can use temporary files during testing to avoid altering real data.

8. Drivers for Class-Based Code

Drivers can test classes just as easily as functions.

```
# account.py
class Account:
    def __init__(self, name, balance=0):
        self.name = name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def __repr__(self):
        return f"{self.name}: ${self.balance}"

if __name__ == "__main__":
    acc = Account("Alice", 100)
    acc.deposit(50)
    print(acc)
```

Run it:

```
Alice: $150
```

In larger projects, you can move this test into a separate driver script (`driver_account.py`) that instantiates several accounts and checks behavior under different scenarios.

9. Logging vs. Printing in Drivers

For small tests, `print()` is fine.
For larger experiments, it's better to use the **`logging`** module so that results can be timestamped and saved.

```
# driver_logging_example.py
import logging
from math_utils import multiply

logging.basicConfig(filename="driver.log", level=logging.INFO)

for i in range(1, 5):
    result = multiply(i, i + 1)
    logging.info("multiply(%d, %d) = %d", i, i + 1, result)

print("Results saved to driver.log")
```

This approach makes your driver more professional and suitable for debugging production systems.

10. Drivers for Incremental Testing (Test Harnesses)

A **test harness** is a more advanced driver that automatically feeds many inputs and collects outputs.

Example, testing a factorial function:

```
# factorial.py
def factorial(n):
    if n < 0:
        raise ValueError("Negative input not allowed")
    return 1 if n in (0, 1) else n * factorial(n - 1)
```

Driver harness:

```
# driver_factorial.py
from factorial import factorial

test_values = [0, 1, 5, 10]
for v in test_values:
    print(f"{v}! = {factorial(v)}")

# Testing error case
try:
    factorial(-3)
except Exception as e:
    print("Handled error:", e)
```

Output:

```
0! = 1
1! = 1
5! = 120
10! = 3628800
Handled error: Negative input not allowed
```

This pattern lets you **validate edge cases** and document expected behavior.

11. Using Drivers to Benchmark Performance

Drivers can also measure performance using the `time` or `timeit` modules.

```
# driver_benchmark.py
import timeit
from math_utils import multiply

print("Benchmarking multiply(1234, 5678)...")

t = timeit.timeit(lambda: multiply(1234, 5678), number=1000000)
print(f"Completed 1,000,000 runs in {t:.3f} seconds")
```

This helps identify performance bottlenecks before deployment.

12. Organizing Driver Scripts

In a real project, you may have many drivers. Keep them organized:

```
project/
├── src/
│   ├── math_utils.py
│   ├── text_tools.py
├── drivers/
│   ├── driver_math.py
│   ├── driver_text_tools.py
│   ├── driver_benchmark.py
├── tests/
│   ├── test_math_utils.py
│   ├── test_text_tools.py
```

Advantages:

- Keeps production code clean.
- Lets you run drivers without affecting the main application.
- Makes it easier for others to reproduce tests.

13. From Drivers to Unit Tests

Driver scripts often evolve into formal **unit tests** once you finalize functionality.

A good practice:

1. Use a driver during development to explore behavior interactively.
2. When logic stabilizes, move tests into a structured framework (`unittest` or `pytest`).
3. Keep a few drivers for exploratory or integration testing.

For example, an exploratory driver that prints custom formatted results can later be converted into parameterized test cases.

14. Automating Drivers with a Main Controller

You can even have a **master driver** that runs several drivers automatically, useful for integration testing.

```
# run_all_drivers.py
import subprocess

scripts = [
    "driver_math.py",
    "driver_text_tools.py",
    "driver_factorial.py"
]

for s in scripts:
    print(f"Running {s}...")
    subprocess.run(["python", s])
    print("-" * 30)
```

This lets you trigger an entire suite of exploratory tests in one go.

15. Best Practices

Practice	Description
Keep them simple	Drivers are meant for testing, not production.
Use clear naming	Prefix with `driver_` or `test_` for clarity.
Document purpose	Add comments on what the driver tests.
Reset environment	Avoid leaving temporary files or states.
Handle exceptions	Always show clear error messages.
Log output	For complex systems, log to file for review.
Version control	Commit useful drivers; delete obsolete ones.

16. Common Pitfalls

Problem	Cause	Fix
Functions print nothing	Driver not calling them correctly	Add explicit calls and print outputs

Circular imports	Driver and module import each other	Keep drivers separate from production code
Failing silently	No error handling	Add try/except and logging
Hardcoded paths	Platform dependence	Use `os.path.join()`
Mixed responsibilities	Driver does too much	Keep each driver focused on one component

17. Summary

Driver scripts are simple but powerful tools that:

- Let you **test code quickly** before full integration.
- Serve as early prototypes and examples.
- Evolve naturally into automated test suites.

A good driver answers:

- What does this function do?
- What happens with invalid input?
- How fast does it run?
- Can I reproduce the output later?

18. Further Reading

- Python Docs: `unittest`
- Real Python: Writing and Running Tests
- Effective Python (Brett Slatkin): *Item 64 – Use Test Harnesses to Drive Development*