COL1000
Introduction to Programming

Priyanka Golia

Most (if not all) of the content is borrowed from Prof. Subodh Kumar's slides

```
try:
    raise ArithmeticError("Arithmetic
problem")
except Exception:
    print("Caught Exception")
except ArithmeticError:
    print("Caught ArithmeticError")
```

```
try:
   1/0
except ArithmeticError:
    print("Caught ArithmeticError")
except ZeroDivisionError:
    print("Caught ZeroDivisionError")
```

```
try:
    raise ArithmeticError("Arithmetic
problem")
except Exception:
    print("Caught Exception")
except ArithmeticError:
    print("Caught ArithmeticError")
```

Caught Exception

```
try:
   1/0
except ArithmeticError:
    print("Caught ArithmeticError")
except ZeroDivisionError:
    print("Caught ZeroDivisionError")
```

```
try:
    raise ArithmeticError("Arithmetic
problem")
except Exception:
    print("Caught Exception")
except ArithmeticError:
    print("Caught ArithmeticError")
```

Caught Exception

```
try:
   1/0
except ArithmeticError:
    print("Caught ArithmeticError")
except ZeroDivisionError:
    print("Caught ZeroDivisionError")
```

Caught ArithmeticError

```
try:
    raise ArithmeticError("Arithmetic
problem")
except Exception:
    print("Caught Exception")
except ArithmeticError:
    print("Caught ArithmeticError")
```

Caught Exception

```
try:
   1/0
except ArithmeticError:
    print("Caught ArithmeticError")
except ZeroDivisionError:
    print("Caught ZeroDivisionError")
```

Caught ArithmeticError

```
BaseException
└── Exception
    ├── ArithmeticError
    │   ├── OverflowError
    │   ├── ZeroDivisionError
    │   └── FloatingPointError
    ├── AssertionError
    ├── AttributeError
    ├── ImportError
    │   └── ModuleNotFoundError
    ├── NameError
    │   └── UnboundLocalError
    ├── TypeError
    ├── ValueError
    │   └── UnicodeError
```

Problem Specification

Assertions— Checking States

When we write:

```
assert x > y, "x should stay larger than y"
```

Python executes this like:

```
if not (x > y):
    raise AssertionError("x should stay larger than y")
```

Assertions— Checking States

When we write:

```
assert x > y, "x should stay larger than y"
```

Python executes this like:

```
if not (x > y):
    raise AssertionError("x should stay larger than y")
```

So yes — under the hood it raises an exception object, exactly like you could write manually.
But:
The purpose of “assertion” exception is not to recover or handle user-facing errors; It’s a signal that the program’s logic or assumptions have failed — the program has entered an invalid internal state. That’s why we usually don’t try/except around assertions. If an assertion fails, it means something is fundamentally wrong with the code.

Assertions— Checking States

When we write:

```
assert x > y, "x should stay larger than y"
```

Python executes this like:

assert = “Something that should never fail.”
try–except = “Something that might fail, and we’ll handle it.”

```
if not (x > y):
    raise AssertionError("x should stay larger than y")
```

So yes — under the hood it raises an exception object, exactly like you could write manually.
But:
The purpose of “assertion” exception is not to recover or handle user-facing errors; It’s a signal that the program’s logic or assumptions have failed — the program has entered an invalid internal state. That’s why we usually don’t try/except around assertions. If an assertion fails, it means something is fundamentally wrong with the code.

Assert vs if vs try-except

Situation	Recommended Tool	Why
You are checking your *own logic* (“b should never be o here”)	assert	To detect programming bugs
You are controlling *intended behavior* (“if b=o, return special value”)	if	Normal program logic
You are dealing with *unpredictable errors* (“user input may fail”)	try–except	Graceful recovery

Assertions— Checking States

```
def divide(a, b):
    assert b != 0, "Denominator must not be zero"  # before computation
    q = a / b
    assert abs(a - b*q) < 1e-6, "Result check failed"  # after computation
    return q
```

Assertions— Checking States

```
def divide(a, b):
    assert b != 0, "Denominator must not be zero"  # before computation
    q = a / b
    assert abs(a - b*q) < 1e-6, "Result check failed"  # after computation
    return q
```

First assertion checks the **input state**.

Second checks the **output state**.

Both together guarantee correctness.

From Assertions to Pre/Postconditions

Concept	Position	Meaning
Precondition	At the beginning	Must be true before execution starts
Invariant	During execution	Must remain true at each iteration
Postcondition	At the end	Must be true after execution finishes

All three are **assertions**, just at different program states.

From Assertions to Pre/Postconditions

```
def sqrt(x):
    assert x >= 0                # precondition
    y = x ** 0.5
    assert abs(y*y - x) < 1e-6   # postcondition
    return y
```

From Assertions to Pre/Postconditions

```
def sqrt(x):
    assert x >= 0                  # precondition
    y = x ** 0.5
    assert abs(y*y - x) < 1e-6  # postcondition
    return y
```

Concept	Meaning	Purpose
Precondition	A logical condition that **must be true before** the program (or function) starts running	Describes the assumptions or requirements on inputs
Postcondition	A logical condition that **must be true after** the program (or function) finishes (if it runs correctly)	Describes the guarantee provided by the program (or function) about outputs

From Assertions to Pre/Postconditions

Write precondition first, before implementing logic.

Think of boundary cases.

Use type annotations.

```
def avg (nums: list[float]) -> float:
        assert len(nums) > 0, “precondition failed, list must not be empty
         result = sum(nums) / len(nums)
         assert min(nums) <= result <= max(nums), “Postcondition failed”
```

Precondition —> caller’s responsibilities
Postcondition —> function’s guarantees.

Problem Decomposition

For real problems, specification and solution evolve together:

- Break problem into sub-problems.
- Each sub-problem has its own specification.
- Output of one becomes input to another.

Example:
In shortest-path algorithms:

Subproblem 1: “Is there an edge between u and v?” (decision)

Subproblem 2: “Find all reachable vertices.” (search)

Subproblem 3: “Find minimum total distance.” (optimization)

Problem Decomposition

For real problems, specification and solution evolve together:

- Break problem into sub-problems.
- Each sub-problem has its own specification.
- Output of one becomes input to another.

Example:
In shortest-path algorithms:

Subproblem 1: “Is there an edge between u and v?” (decision)

Subproblem 2: “Find all reachable vertices.” (search)

Subproblem 3: “Find minimum total distance.” (optimization)

This modular specification makes big systems verifiable part-by-part.

Fuzzy or Probabilistic Specifications

- Provide examples of input/output
  - Given partial I/O table, ‘fill up’ the rest
    - How do we know the filling is ‘good?’
- Solution is a sample from a probability distribution
  - Or, satisfies a specified property with a high probability
- Usually divided into concretely specified sub-problems
  - e.g., Apply statistical methods on data

Unit Testing

- **Unit testing** is the process of testing individual components (like functions) of a program to ensure that each part works correctly on its own.
- Unit tests are usually small, fast, and automatically executed. They help catch bugs early and ensure that later changes do not break existing functionality.

Each unit test:

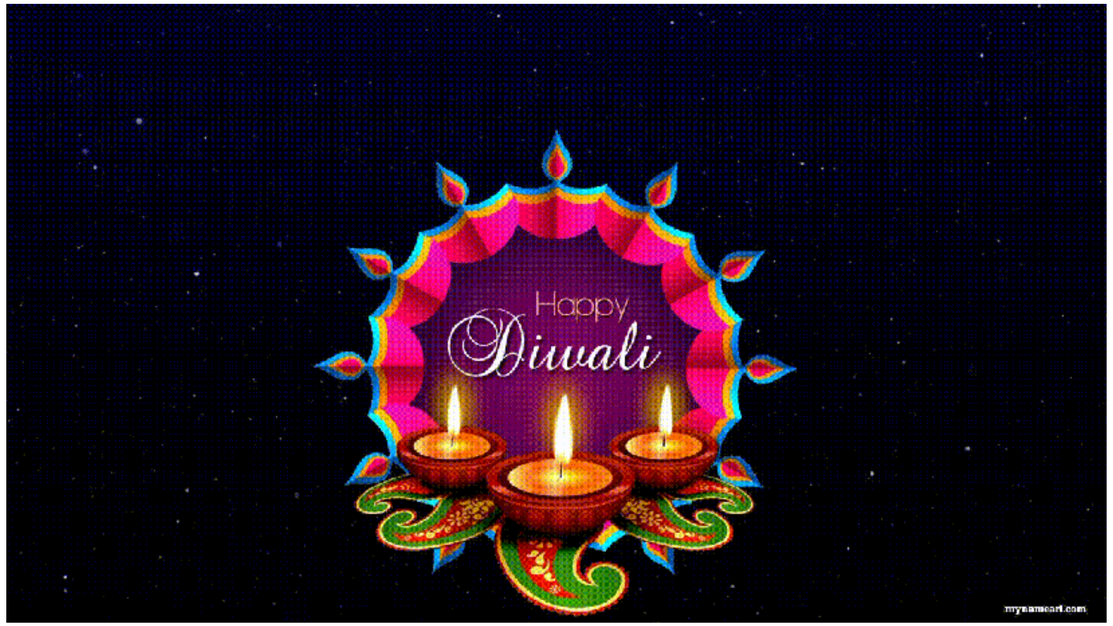
- Takes some input.
- Calls the function or method.
- Checks that the output matches the expected result (using **assertions**).

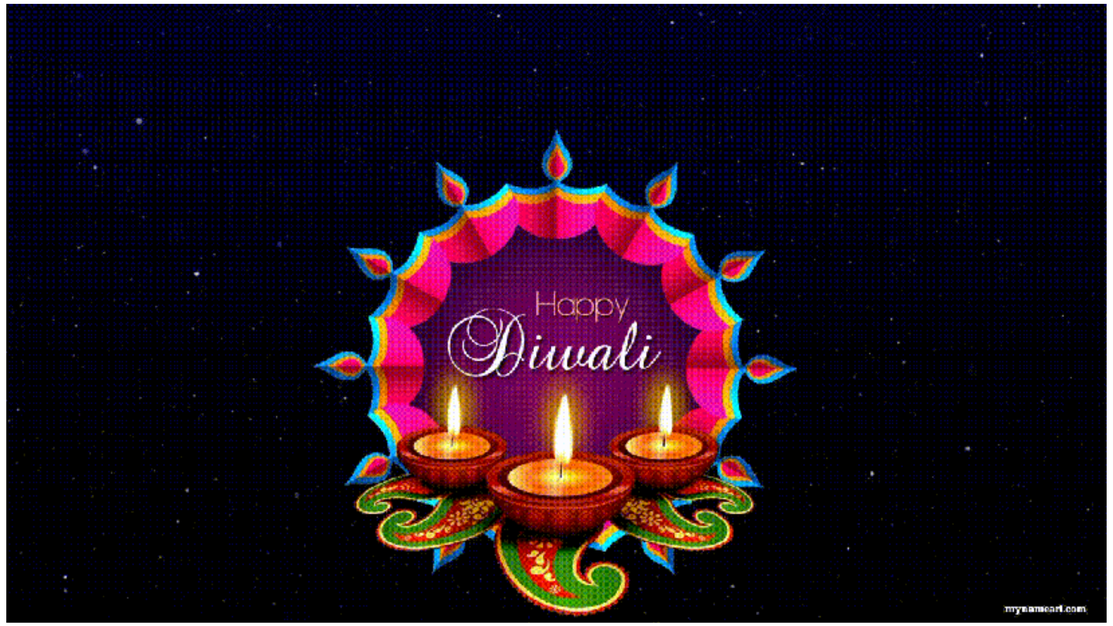
Unit Testing

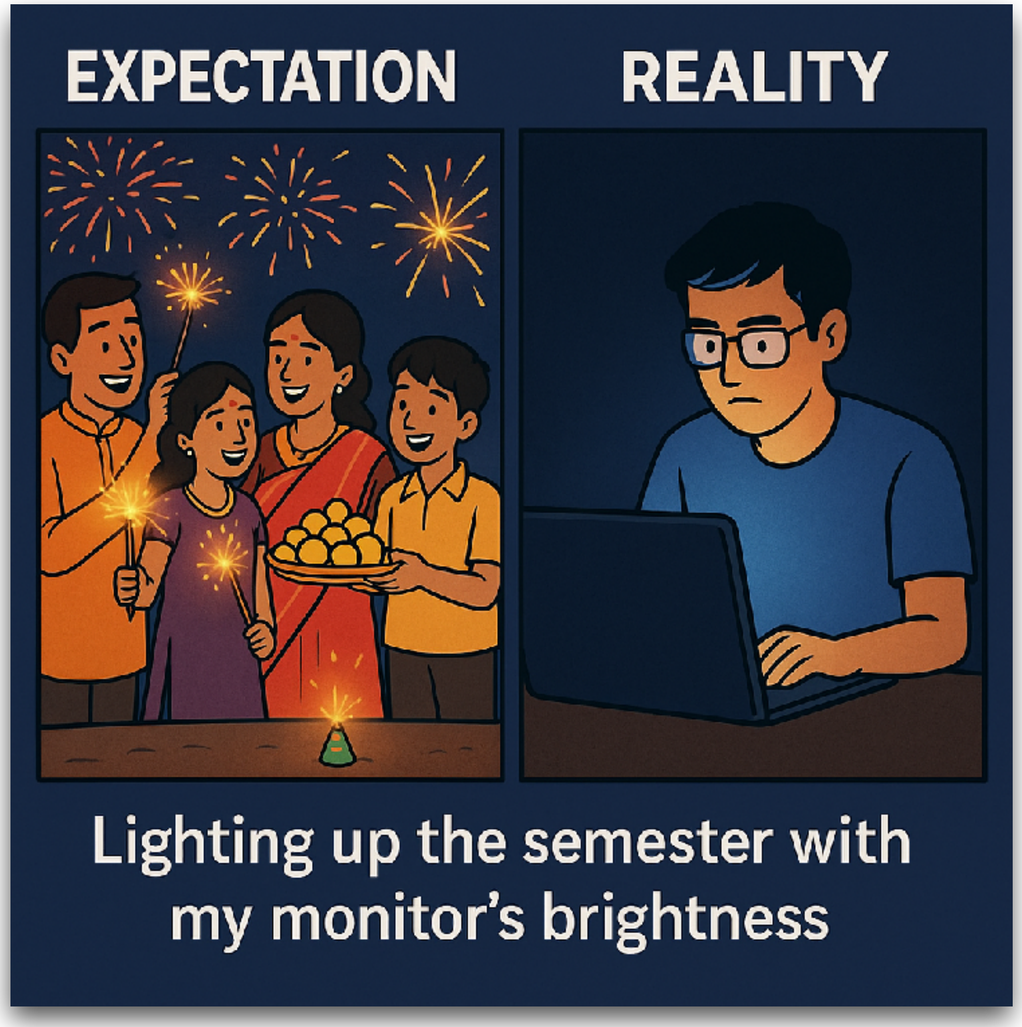
```
def myadd(a, b):
    return a + b
```

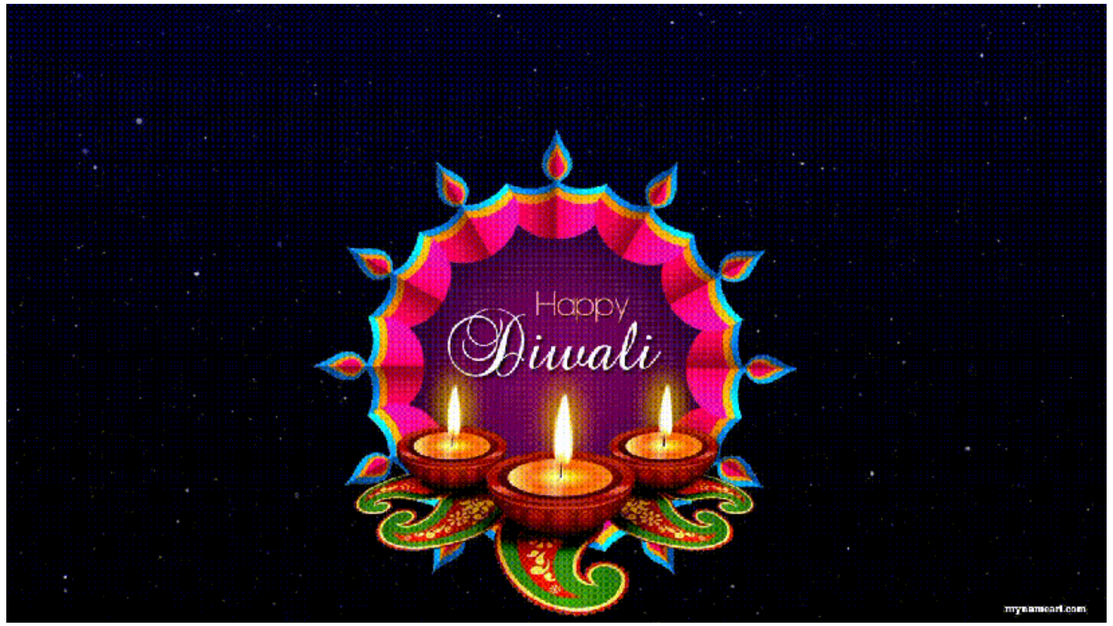
Say saved as “add.py”

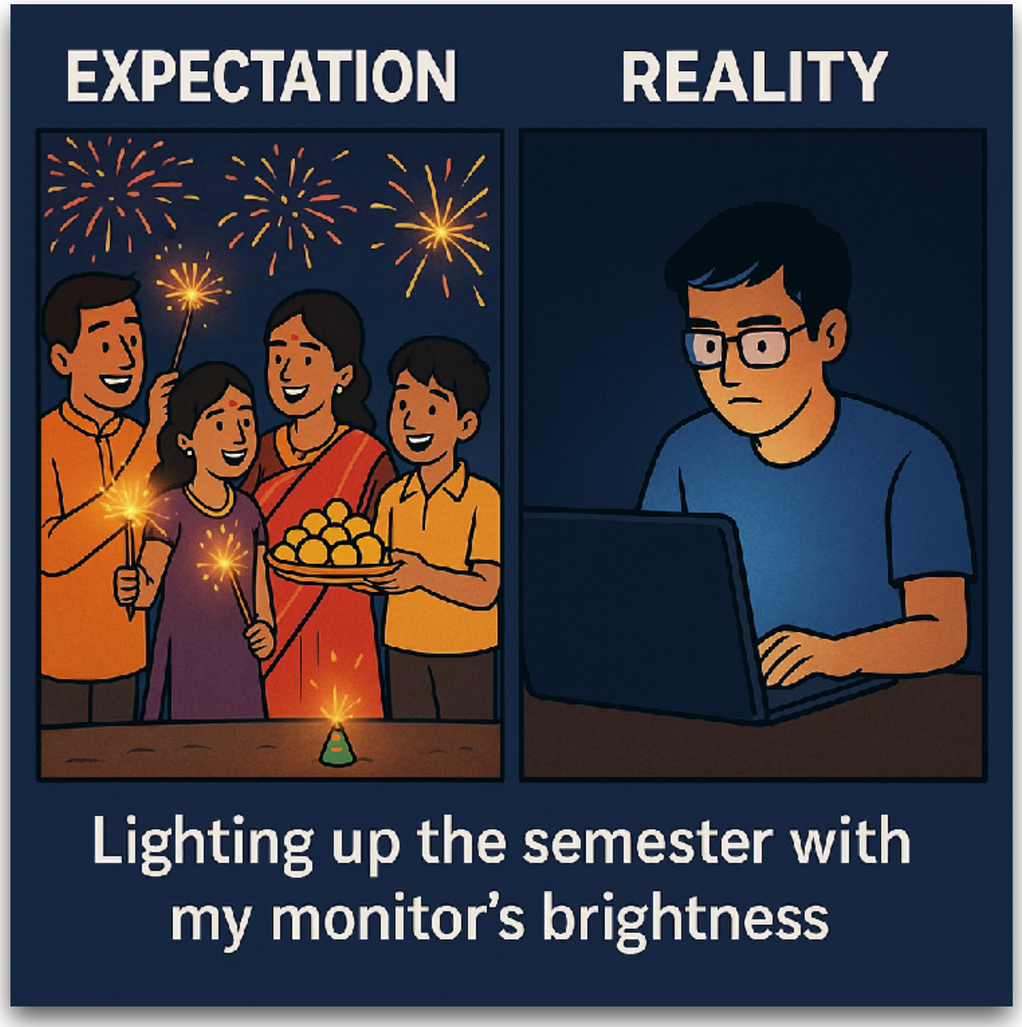
```
Import add
def test_add():
    result = add.myadd(2, 3)
    assert result == 5, f"Expected 5 but got {result}"
```


Happy
Diwali


Happy
Diwali


EXPECTATION
REALITY
Lighting up the semester with my monitor's brightness





No class on Tuesday 21st.
Class on Saturday 25th — 6 pm to 7 pm.