COL1000
Introduction to Programming

Priyanka Golia

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

```
  try:
     a = int(“hello”)
  except Exception:
    print(“caught error”)
  except ValueError:
    print(“caught value error”)
```

```
try:
    print("Start")
    raise ValueError("Invalid")
except Exception:
    print("Handled")
print("Done")
```

```
def check(n):
    assert n > 0, "n must be positive"
    print("All good")

try:
    check(-5)
except AssertionError as e:
    print("Caught:", e)
```

```
  try:
     a = int(“hello”)
  except Exception:
    print(“caught error”)
  except ValueError:
    print(“caught value error”)
```

caught error

```
try:
    print("Start")
    raise ValueError("Invalid")
except Exception:
    print("Handled")
print("Done")
```

```
def check(n):
    assert n > 0, "n must be positive"
    print("All good")

try:
    check(-5)
except AssertionError as e:
    print("Caught:", e)
```

```
try:
    a = int(“hello”)
except Exception:
  print(“caught error”)
except ValueError:
  print(“caught value error”)
```

Unreachable

caught error

```
try:
    print("Start")
    raise ValueError("Invalid")
except Exception:
    print("Handled")
print("Done")
```

```
def check(n):
    assert n > 0, "n must be positive"
    print("All good")

try:
    check(-5)
except AssertionError as e:
    print("Caught:", e)
```

```
try:
    a = int(“hello”)
except Exception:
  print(“caught error”)
except ValueError:
  print(“caught value error”)
```

Unreachable

caught error

```
try:
    print("Start")
    raise ValueError("Invalid")
except Exception:
    print("Handled")
print("Done")
```

```
def check(n):
    assert n > 0, "n must be positive"
    print("All good")

try:
    check(-5)
except AssertionError as e:
    print("Caught:", e)
```

Caught: n must be positive

```
try:
    a = int(“hello”)
except Exception:
  print(“caught error”)
except ValueError:
  print(“caught value error”)
```

Unreachable

caught error

```
try:
    print("Start")
    raise ValueError("Invalid")
except Exception:
    print("Handled")
print("Done")
```

Start
Handled
Done

```
def check(n):
    assert n > 0, "n must be positive"
    print("All good")

try:
    check(-5)
except AssertionError as e:
    print("Caught:", e)
```

Caught: n must be positive

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}"
```

```
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

```
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

Instead of import myfunc.
Now, we can directly call fact. No need to do “myfunc.fact”.

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

Instead of import myfunc.
Now, we can directly call fact. No need to do “myfunc.fact”.

If the condition fails, the test fails and you get an assertion error.

Concept	Purpose	Example
Specification	Describe expected behavior	“`add(a, b)` returns the sum of two integers.”
Assertion	Enforce expectations during execution	`assert a + b == expected`
Unit Test	Automate the process of verifying those expectations	Use `assert` or `pytest` to test automatically

From Sayan’s slides.

What is pytest?

Pytest is a popular testing framework in Python that:

- Automatically finds all files and functions that start with test_
- Runs them for you
- Gives a clear report of which tests passed or failed

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

Saved in test_myfunc.py

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 120

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

Saved in test_myfunc.py

```
============================= test session starts ==============================
platform darwin -- Python 3.13.1, pytest-8.3.4, pluggy-1.5.0
rootdir: /Users/priyanka/Desktop/col1000-lect
plugins: anyio-4.8.0
collected 2 items

test_myfunc.py ..                                                        [100%]

============================== 2 passed in 0.01s ===============================
```

2 passed — two functions started with test_

```
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 110

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

```python
def fact(n):
    """Return factorial of n."""
    assert n >= 0, "n should be non-negative"
    if n == 0 or n == 1:
        return 1
    return n * fact(n - 1)
```

Saved in myfunc.py

```python
from myfunc import fact

def test_fact_basic():
    assert fact(0) == 1
    assert fact(1) == 1
    assert fact(5) == 110

def test_fact_negative():
    try:
        fact(-1)
    except AssertionError:
        assert True
    else:
        assert False
```

```
============================= test session starts ==============================
platform darwin -- Python 3.13.1, pytest-8.3.4, pluggy-1.5.0
rootdir: /Users/priyanka/Desktop/col1000-lect
plugins: anyio-4.8.0
collected 2 items

test_myfunc.py F.                                                        [100%]

=================================== FAILURES ===================================
_____________________________ test_fact_basic ______________________________

    def test_fact_basic():
        assert fact(0) == 1
        assert fact(1) == 1
>       assert fact(5) == 110
E       assert 120 == 110
E        +  where 120 = fact(5)

test_myfunc.py:6: AssertionError
=========================== short test summary info ============================
FAILED test_myfunc.py::test_fact_basic - assert 120 == 110
========================= 1 failed, 1 passed in 0.02s ==========================
```

Concept	Meaning
Unit	A small part of your program (like a function)
Test Case	Code that checks whether the unit behaves as expected
Assertion	A statement that must be true for the test to pass
pytest	A tool that runs all your test cases automatically
Good Practice	Write tests as soon as you write your functions

Good Practice

Write one test per behavior (e.g., valid input, boundary case, error case)

Run pytest frequently as you code

Think of testing as part of coding, not an afterthought

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can’t avoid bugs — but you can learn to fix them efficiently.

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can’t avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

Specification — take two positive integers and compute their avg.

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can’t avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

Specification — take two positive integers and compute their avg.

```
avg(10,20)
```

What should be the output?

Debugging

What is debugging: when you write a program, bugs (errors or unexpected behavior) are inevitable.

Debugging means finding and fixing those bugs.

You can’t avoid bugs — but you can learn to fix them efficiently.

```
def avg(x, y):
    return x + y / 2
```

Specification — take two positive integers and compute their avg.

```
avg(10,20)
```

What should be the output?

It returns 20 instead of 15 — because division happens before additions.

Debugging with Print statement

The old way to do the debugging is to Print —
“when in doubt, print it out”

Debugging with Print statement

The old way to do the debugging is to Print —
“when in doubt, print it out”

```
def avg(x, y):
    print("x =", x, "y =", y)
    result = x + y / 2
    print("result =", result)
    return result
```

Debugging with Print statement

The old way to do the debugging is to Print —
“when in doubt, print it out”

```
def avg(x, y):
    print("x =", x, "y =", y)
    result = x + y / 2
    print("result =", result)
    return result
```

```
x = 10 y = 20
result = 20.0
```

Using Python's Built-in Debugger (pdb)

Instead of print(), we can pause the program and inspect variables interactively.

```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4     result = ( x +  y ) / 2
5     return result
6
7 print(avg(10,20))
```

Using Python's Built-in Debugger (pdb)

Instead of print(), we can pause the program and inspect variables interactively.

```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4     result = ( x +  y ) / 2
5     return result
6
7 print(avg(10,20))
```

To tell python that you would like to be in the debug mode.

Using Python's Built-in Debugger (pdb)

Instead of print(), we can pause the program and inspect variables interactively.

```
1 import pdb
2 pdb.set_trace()
3 def avg(x,y):
4     result = ( x +  y ) / 2
5     return result
6
7 print(avg(10,20))
```

To tell python that you would like to be in the debug mode.