COL1000
Introduction to Programming

Priyanka Golia

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

Is it a tail recursion or not?

```
def f(n):
    if n <= 1:
        return n
    return f(n-10) + f(n/10)
```

```
def risky_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b is zero")
    return a / b

def wrapper(x):
    try:
        return risky_div(10, x)
    except ZeroDivisionError:
        print("Caught in wrapper")

for v in [5, 0, 2]:
    print("Result:", wrapper(v))
```

How many times is the function “count” called? Write the complete output.

```
def count(n):
    if n == 0:
        return
    print("Enter:", n)
    count(n-1)
    print("Exit:", n)

count(3)
```

Is it a tail recursion or not?

```
def f(n):
    if n <= 1:
        return n
    return f(n-10) + f(n/10)
```

No!

```
def risky_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b is zero")
    return a / b

def wrapper(x):
    try:
        return risky_div(10, x)
    except ZeroDivisionError:
        print("Caught in wrapper")

for v in [5, 0, 2]:
    print("Result:", wrapper(v))
```

How many times is the function “count” called? Write the complete output.

```
def count(n):
    if n == 0:
        return
    print("Enter:", n)
    count(n-1)
    print("Exit:", n)

count(3)
```

Is it a tail recursion or not?

```
def f(n):
    if n <= 1:
        return n
    return f(n-10) + f(n/10)
```

No!

```
def risky_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b is zero")
    return a / b

def wrapper(x):
    try:
        return risky_div(10, x)
    except ZeroDivisionError:
        print("Caught in wrapper")

for v in [5, 0, 2]:
    print("Result:", wrapper(v))
```

How many times is the function “count” called? Write the complete output.

```
def count(n):
    if n == 0:
        return
    print("Enter:", n)
    count(n-1)
    print("Exit:", n)

count(3)
```

4

Is it a tail recursion or not?

```
def f(n):
    if n <= 1:
        return n
    return f(n-10) + f(n/10)
```

No!

```
def risky_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b is zero")
    return a / b

def wrapper(x):
    try:
        return risky_div(10, x)
    except ZeroDivisionError:
        print("Caught in wrapper")

for v in [5, 0, 2]:
    print("Result:", wrapper(v))
```

How many times is the function “count” called? Write the complete output.

```
def count(n):
    if n == 0:
        return
    print("Enter:", n)
    count(n-1)
    print("Exit:", n)

count(3)
```

4

Enter: 3
Enter:2
Enter:1
Exit:1
Exit:2
Exit:3

Is it a tail recursion or not?

```
def f(n):
    if n <= 1:
        return n
    return f(n-10) + f(n/10)
```

No!

```
def risky_div(a, b):
    if b == 0:
        raise ZeroDivisionError("b is zero")
    return a / b

def wrapper(x):
    try:
        return risky_div(10, x)
    except ZeroDivisionError:
        print("Caught in wrapper")

for v in [5, 0, 2]:
    print("Result:", wrapper(v))
```

```
Result: 2.0
Caught in wrapper
Result: None
Result: 5.0
```

How many times is the function “count” called? Write the complete output.

```
def count(n):
    if n == 0:
        return
    print("Enter:", n)
    count(n-1)
    print("Exit:", n)

count(3)
```

4

```
Enter: 3
Enter:2
Enter:1
Exit:1
Exit:2
Exit:3
```

Problem Specification

What is a problem specification ?

A problem specification tells:

What inputs are expected (what kind of data the program expects)

What outputs are produced (what it will produce)

The relationship between inputs and outputs

What is a problem specification ?

A problem specification tells:

What inputs are expected (what kind of data the program expects)

What outputs are produced (what it will produce)

The relationship between inputs and outputs

Specification is how we tell the computer what problem we are solving — precisely, unambiguously, and completely.

Write a program to find the larger number?

Bad specification.

Write a program to find the larger number?

Why bad:
Doesn’t say what kind of numbers (integers? floats?).
Doesn’t say what happens if both are equal.

Bad specification.

Write a program to find the larger number?

Why bad:
Doesn’t say what kind of numbers (integers? floats?).
Doesn’t say what happens if both are equal.

Good specification.

Input: Two integers a and b

Output: An integer m such that m == a or m == b, and m >= a and m >= b

If a == b, the function returns either one.

Compute factorial of a number.

Bad specification.

Compute factorial of a number.

Why bad:
Doesn’t say what range of numbers is allowed?
Doesn’t say what kind of numbers (integers? floats?)?
Doesn’t say what happens if input is zero

Bad specification.

Compute factorial of a number.

Why bad:
Doesn’t say what range of numbers is allowed?
Doesn’t say what kind of numbers (integers? floats?)?
Doesn’t say what happens if input is zero

Good specification.

Input: An integer $n \geq 0$
Output: An integer f satisfying $f = 1 \times 2 \times \ldots \times n$
Special case: If $n = 0$, return 1.

Problem Specification

A **specification** is a **mathematically precise statement** of the problem.

It should describe *what* the program must accomplish, not *how* it should do it.

Given an answer, it should be possible to **verify** whether it is **correct**.

Verification means applying a *logical condition* that returns True if and only if the answer is correct.

Problem Specification

A **specification** is a **mathematically precise statement** of the problem.
It should describe *what* the program must accomplish, not *how* it should do it.

Given an answer, it should be possible to **verify** whether it is **correct**.
Verification means applying a *logical condition* that returns True if and only if the answer is correct.

Sometimes the "solution" is **a set or list of answers** (e.g., all divisors, all prime numbers ≤ N).
Then we must check:

1. **Soundness (correctness):** Every produced answer satisfies the condition.
2. **Completeness:** Every valid answer according to the condition appears in the output.

The logical condition should be **computable** for every possible instance of the output, hence for every possible input.

Term	Meaning	Example
Correctness (Soundness)	Every produced answer satisfies the specification	Every output factor divides n
Completeness	Every valid answer appears in output	All divisors of n are printed

Print all integers x in the range 1 to 8 (8 including), such that$8 \% x == 0$

Term	Meaning	Example
Correctness (Soundness)	Every produced answer satisfies the specification	Every output factor divides n
Completeness	Every valid answer appears in output	All divisors of n are printed

Print all integers x in the range 1 to 8 (8 including), such that$8 \% x == 0$

Output = {1,2,8} -> it is sound (correct) but incomplete.

Term	Meaning	Example
Correctness (Soundness)	Every produced answer satisfies the specification	Every output factor divides n
Completeness	Every valid answer appears in output	All divisors of n are printed

Print all integers x in the range 1 to 8 (8 including), such that$8 \% x == 0$

Output = {1,2,8} -> it is sound (correct) but incomplete.

Output = {1,2,4,8} -> it is sound and complete.

“What Makes a Specification Good?”

Unambiguous: Every valid input maps to exactly one interpretation of the task.

Verifiable: We can check if an answer satisfies it.

Computable: The condition can be evaluated by a program.

Correct: Reflects the intended problem (no false goals).

Complete: Covers all input and output cases.

Program Correctness ?

```
—> 1
a = 10
b = 2
—> 2
c = a / b
—> 3
```

Start: a,b undefined
Midway: a=10,b=2,c undefined
End: a=10,b=2,c=5

Program Correctness ?

```
—> 1
a = 10
b = 2
—> 2
c = a / b
—> 3
```

Start: a,b undefined
Midway: a=10,b=2,c undefined
End: a=10,b=2,c=5

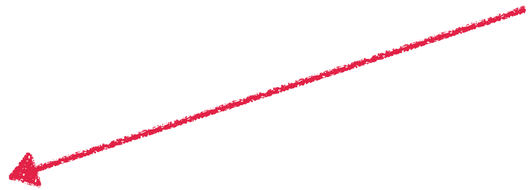
Correctness means the program stays in **valid states** — no inconsistent values — from start to end.

Program Correctness ?

```
—> 1
a = 10
b = 2
—> 2
c = a / b
—> 3
```

Start: a,b undefined
Midway: a=10,b=2,c undefined
End: a=10,b=2,c=5

Correctness means the program stays in **valid states** — no inconsistent values — from start to end.



A program state is a snapshot of all variable values at a given point.

Assertions— Checking States

An **assertion** is a logical condition that must hold at a certain point in the code.

Assertions— Checking States

An **assertion** is a logical condition that must hold at a certain point in the code.

```
x = 10
assert x > 0, "x must be positive"
y = 5
assert x > y, "x should stay larger than y"
```

Assertions— Checking States

An **assertion** is a logical condition that must hold at a certain point in the code.

```
x = 10
assert x > 0, "x must be positive"
y = 5
assert x > y, "x should stay larger than y"
```

assert <condition>, <optional_message>
where:
<condition> is any Boolean expression (something that evaluates to True or False), and
<optional_message> is any expression (often a string) that will be displayed if the assertion fails.

Assertions— Checking States

An **assertion** is a logical condition that must hold at a certain point in the code.

```
x = 10
assert x > 0, "x must be positive"
y = 5
assert x > y, "x should stay larger than y"
```

assert <condition>, <optional_message>
where:
<condition> is any Boolean expression (something that evaluates to True or False), and
<optional_message> is any expression (often a string) that will be displayed if the assertion fails.

If an assertion fails, the program reached an *illegal* state.

Assertions are like checkpoints — ensuring we're still on track.

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.