Lecture 17: Assertions & Exceptions

Comp 102

Forman Christian University

Recap

So far. . .

- Variables, strings, I/O
- Conditions, loops, functions
- Lists, tuples, 2D lists
- Searching, approximation

So far. . .

- Variables, strings, I/O
- Conditions, loops, functions
- Lists, tuples, 2D lists
- Searching, approximation

You can write programs that **work**.

But can they handle the **unexpected**?

Real-World Hook

Your phone's **Contacts** app lets you search for a name.

Real-World Hook

Your phone's **Contacts** app lets you search for a name.

- What if you search for a name that **doesn't exist**?

Real-World Hook

Your phone's **Contacts** app lets you search for a name.

- What if you search for a name that **doesn't exist**?
- What if you type a **number** where a name should go?

Real-World Hook

Your phone's **Contacts** app lets you search for a name.

- What if you search for a name that **doesn't exist**?
- What if you type a **number** where a name should go?
- What if the contact list is **empty**?

Real-World Hook

Your phone's **Contacts** app lets you search for a name.

- What if you search for a name that **doesn't exist**?
- What if you type a **number** where a name should go?
- What if the contact list is **empty**?

The app **doesn't crash** — it handles the problem!

Programs Crash!

Try each of these in Thonny's Shell:

```
>>> contacts = ['Ali', 'Fatima', 'Zain']
>>> contacts[10]

>>> int('hello')

>>> 10 / 0

>>> 'a' / 4
```

Programs Crash!

Try each of these in Thonny's Shell:

```
>>> contacts = ['Ali', 'Fatima', 'Zain']
>>> contacts[10]             # IndexError

>>> int('hello')

>>> 10 / 0

>>> 'a' / 4
```

Programs Crash!

Try each of these in Thonny's Shell:

```
>>> contacts = ['Ali', 'Fatima', 'Zain']
>>> contacts[10]                # IndexError

>>> int('hello')                # ValueError

>>> 10 / 0

>>> 'a' / 4
```

Programs Crash!

Try each of these in Thonny's Shell:

```
>>> contacts = ['Ali', 'Fatima', 'Zain']
>>> contacts[10]                    # IndexError

>>> int('hello')                    # ValueError

>>> 10 / 0                          # ZeroDivisionError

>>> 'a' / 4
```

Programs Crash!

Try each of these in Thonny's Shell:

```
>>> contacts = ['Ali', 'Fatima', 'Zain']
>>> contacts[10]                        # IndexError

>>> int('hello')                        # ValueError

>>> 10 / 0                              # ZeroDivisionError

>>> 'a' / 4                             # TypeError
```

Exceptions

Big Idea

An **exception** is Python's way of saying

"something went wrong, and I can't continue"

If uncaught, the program **crashes**.

Common Exception Types

Exception	When it Happens	Example
`IndexError`	Bad list index	`L[100]`
`ValueError`	Wrong value for conversion	`int('abc')`
`TypeError`	Wrong type in operation	`'a' / 4`
`ZeroDivisionError`	Division by zero	`10 / 0`
`NameError`	Undefined variable	`print(x)`

`try/except`: The Safety Net

Think of it like `if/else`:

```
if condition:
    # runs if True
else:
    # runs if False
```

```
try:
    # risky code
except:
    # runs if error
```

`try/except`: **The Safety Net**

Think of it like `if/else`:

```
if condition:
    # runs if True
else:
    # runs if False
```

```
try:
    # risky code
except:
    # runs if error
```

Exception **raised** by `try` block is **caught** by `except` block.

Program **does not crash**.

Example: sum_digits

Sum all digit characters in a string:

```python
def sum_digits(s):
    total = 0
    for ch in s:
        total += int(ch)
    return total
```

Example: `sum_digits`

Sum all digit characters in a string:

```python
def sum_digits(s):
    total = 0
    for ch in s:
        total += int(ch)
    return total
```

```python
sum_digits('1234')  # returns 10
sum_digits('12E4')  # ValueError!  CRASH
```

Example: sum_digits (fixed)

Wrap the risky line in try/except:

```python
def sum_digits(s):
    total = 0
    for ch in s:
        try:
            total += int(ch)
        except:
            pass  # skip non-digits
    return total
```

Example: `sum_digits` (fixed)

Wrap the risky line in `try/except`:

```
def sum_digits(s):
    total = 0
    for ch in s:
        try:
            total += int(ch)
        except:
            pass  # skip non-digits
    return total

sum_digits('12E4')  # returns 7, no crash!
```

You Try!

Write code that keeps asking the user for an integer until they enter a valid one.

You Try!

Write code that keeps asking the user for an integer until they enter a valid one.

Solution:

```python
while True:
    try:
        n = int(input("Enter an integer: "))
        break
    except:
        print("That's not an integer! Try again.")
```

Handling Specific Exceptions

Bare except catches **everything** — even bugs you want to see!

Better: name the **specific** exception.

```python
try:
    n = int(input("Enter a number: "))
    result = 100 / n
except ValueError:
    print("Not a valid number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
```

Handling Specific Exceptions

Bare except catches **everything** — even bugs you want to see!

Better: name the **specific** exception.

```
try:
    n = int(input("Enter a number: "))
    result = 100 / n
except ValueError:
    print("Not a valid number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
```

Demo: enter "abc" → ValueError
enter "0" → ZeroDivisionError
enter "5" → success

The `else` Clause

Code in `else` runs **only if no exception** was raised:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

The `else` Clause

Code in `else` runs **only if no exception** was raised:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Why not put it in the `try` block?

The else Clause

Code in else runs **only if no exception** was raised:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Why not put it in the try block?

- try block should contain **only** the risky code

The else Clause

Code in else runs **only if no exception** was raised:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Why not put it in the try block?

- try block should contain **only** the risky code
- else code won't be accidentally **caught** by except

The `else` Clause

Code in `else` runs **only if no exception** was raised:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Why not put it in the `try` block?

- `try` block should contain **only** the risky code
- `else` code won't be accidentally **caught** by `except`
- Makes it **clear** which code might fail vs. which runs on success

`else:` Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Input	Output
	— Crash!

else: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Input	Output
"abc"	Not a number!

— **Crash!**

else: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Input	Output
"abc"	Not a number!
"5"	Result: 20.0

— **Crash!**

else: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Input	Output
"abc"	Not a number!
"5"	Result: 20.0
"0"	ZeroDivisionError — **Crash!**

`else`: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
else:
    print(f"Result: {result}")
```

Input	Output
`"abc"`	`Not a number!`
`"5"`	`Result: 20.0`
`"0"`	`ZeroDivisionError` — **Crash!**

Key: `else` runs only when the `try` block finishes **without any error**.

The `finally` Clause

Code in `finally` runs **no matter what** — error or not:

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

The `finally` Clause

Code in `finally` runs **no matter what** — error or not:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

When is this useful?

The `finally` Clause

Code in `finally` runs **no matter what** — error or not:

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

When is this useful?

- **Cleanup** code that **must** run: closing files, freeing resources

The `finally` Clause

Code in `finally` runs **no matter what** — error or not:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

When is this useful?

- **Cleanup** code that **must** run: closing files, freeing resources
- Runs even if the exception is **not caught**

The `finally` Clause

Code in `finally` runs **no matter what** — error or not:

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

When is this useful?

- **Cleanup** code that **must** run: closing files, freeing resources
- Runs even if the exception is **not caught**
- Runs even if there is a `return` inside `try`

`finally`: Tracing

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

Input	Output

finally: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

Input	Output
"abc"	Not a number! Done!

`finally`: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

Input	Output
"abc"	Not a number! Done!
"5"	Done!

`finally`: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

Input	Output
"abc"	Not a number! Done!
"5"	Done!
"0"	Done! then crash

`finally`: Tracing

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
finally:
    print("Done!")
```

Input	Output
"abc"	Not a number! Done!
"5"	Done!
"0"	Done! then crash

Key: `finally` **always** runs — even when the program crashes.

Full Pattern

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Done!")
```

Full Pattern

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Done!")
```

try: risky code

Full Pattern

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Done!")
```

try: risky code

except: runs if error

Full Pattern

```
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Done!")
```

`try:` risky code

`except:` runs if error

`else:` runs if **no** error

Full Pattern

```python
try:
    n = int(input("Number: "))
    result = 100 / n
except ValueError:
    print("Not a number!")
except ZeroDivisionError:
    print("Can't divide by zero!")
else:
    print(f"Result: {result}")
finally:
    print("Done!")
```

`try`: risky code

`except`: runs if error

`else`: runs if **no** error

`finally`: **always** runs

You Try!

Write a function `safe_divide(a, b)` that returns a/b, or `None` if division fails.

```python
def safe_divide(a, b):
    """Return a/b, or None if
    division fails."""
    pass
```

You Try!

Write a function `safe_divide(a, b)` that returns a/b, or `None` if division fails.

```python
def safe_divide(a, b):
    """Return a/b, or None if
    division fails."""
    pass
```

Solution:

```python
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

You Try!

Write a function `safe_divide(a, b)` that returns a/b, or `None` if division fails.

```python
def safe_divide(a, b):
    """Return a/b, or None if
    division fails."""
    pass
```

Solution:

```python
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return None
```

Think: What does `safe_divide(10, 'a')` do?

Raising Exceptions & Assertions

Raising Your Own Exceptions

So far, *Python* raises exceptions automatically.

You can raise them yourself using `raise`:

```python
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError("Insufficient funds!")
    return balance - amount
```

Raising Your Own Exceptions

So far, *Python* raises exceptions automatically.

You can raise them yourself using `raise`:

```python
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError("Insufficient funds!")
    return balance - amount
```

```python
withdraw(100, 50)   # returns 50
withdraw(100, 200)  # ValueError!
```

Why Raise Exceptions?

Can't we just use `if/else` and `print`?

Why Raise Exceptions?

Can't we just use `if/else` and `print`?

- **Signal** the problem to the **caller**, not just the user
  - A `print` is easy to miss — an exception **cannot** be ignored

Why Raise Exceptions?

Can't we just use `if/else` and `print`?

- **Signal** the problem to the **caller**, not just the user
  - A `print` is easy to miss — an exception **cannot** be ignored
- **Separate** what went wrong from how to handle it
  - The function that *detects* the error may not know the right response
  - Let the *caller* decide: retry? show a message? log it?

Why Raise Exceptions?

Can't we just use `if/else` and `print`?

- **Signal** the problem to the **caller**, not just the user
  - A `print` is easy to miss — an exception **cannot** be ignored
- **Separate** what went wrong from how to handle it
  - The function that *detects* the error may not know the right response
  - Let the *caller* decide: retry? show a message? log it?
- **Stop bad data** from spreading through your program
  - Returning `-1` or `None` silently can cause bugs *later*
  - An exception **halts immediately** at the source of the problem

print vs raise

Bad: just print

```python
def withdraw(balance, amount):
    if amount > balance:
        print("Not enough funds")
        return balance
    return balance - amount
```

Good: raise

```python
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError(
            "Not enough funds")
    return balance - amount
```

print vs raise

Bad: just print

```
def withdraw(balance, amount):
    if amount > balance:
        print("Not enough funds")
        return balance
    return balance - amount

b = withdraw(100, 200)
print(b)  # 100 --- looks fine?!
```

Good: raise

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError(
            "Not enough funds")
    return balance - amount
```

print vs raise

Bad: just print

```
def withdraw(balance, amount):
    if amount > balance:
        print("Not enough funds")
        return balance
    return balance - amount

b = withdraw(100, 200)
print(b)  # 100 --- looks fine?!
```

Good: raise

```
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError(
            "Not enough funds")
    return balance - amount

b = withdraw(100, 200)
# ValueError --- can't ignore!
```

print **vs** raise

Bad: just print

```python
def withdraw(balance, amount):
    if amount > balance:
        print("Not enough funds")
        return balance
    return balance - amount

b = withdraw(100, 200)
print(b)  # 100 --- looks fine?!
```

Good: raise

```python
def withdraw(balance, amount):
    if amount > balance:
        raise ValueError(
            "Not enough funds")
    return balance - amount

b = withdraw(100, 200)
# ValueError --- can't ignore!
```

The `print` version **silently** returns wrong data. The `raise` version **forces** the caller to deal with the problem.

Example: `sum_digits` with `raise`

Instead of silently skipping bad characters, **announce** the problem:

```python
def sum_digits(s):
    total = 0
    for ch in s:
        if not ch.isdigit():
            raise ValueError(f"'{ch}' is not a digit!")
        total += int(ch)
    return total
```

Example: `sum_digits` with `raise`

Instead of silently skipping bad characters, **announce** the problem:

```python
def sum_digits(s):
    total = 0
    for ch in s:
        if not ch.isdigit():
            raise ValueError(f"'{ch}' is not a digit!")
        total += int(ch)
    return total
```

```python
sum_digits('12E4')
# ValueError:  'E' is not a digit!
```

You Try!

Write `pairwise_div(L1, L2)` that returns a new list where element `i` is `L1[i] / L2[i]`.

- `raise ValueError` if lists have different lengths
- `raise ZeroDivisionError` if any element of `L2` is 0

You Try!

Write `pairwise_div(L1, L2)` that returns a new list where element `i` is `L1[i] / L2[i]`.

- `raise ValueError` if lists have different lengths
- `raise ZeroDivisionError` if any element of L2 is 0

Solution:

```python
def pairwise_div(L1, L2):
    if len(L1) != len(L2):
        raise ValueError("Lists must be same length!")
    result = []
    for i in range(len(L1)):
        if L2[i] == 0:
            raise ZeroDivisionError(f"L2[{i}] is 0!")
        result.append(L1[i] / L2[i])
    return result
```

Assertions

Big Idea

An **assertion** is a claim about your program's state.

If the claim is **False**, the program halts immediately.

Assertions

Syntax:

```
assert <condition>, "error message"
```

If `condition` is **False** → raises `AssertionError`

Assertions

Syntax:

```
assert <condition>, "error message"
```

If `condition` is **False** → raises `AssertionError`

Examples:

```
assert len(s) != 0, "String is empty!"
assert b != 0, "Can't divide by zero!"
assert type(n) == int, "Must be an integer!"
```

Assertions: Examples

```python
def avg(grades):
    assert len(grades) != 0, "No grades!"
    return sum(grades) / len(grades)
```

```
avg([85, 90, 78])  # returns 84.33
avg([])            # AssertionError: No grades!
```

Assertions: Examples

```
def avg(grades):
    assert len(grades) != 0, "No grades!"
    return sum(grades) / len(grades)

avg([85, 90, 78])   # returns 84.33
avg([])             # AssertionError:  No grades!
```

```
def divide(a, b):
    assert b != 0, "b cannot be 0"
    return a / b

divide(10, 2)   # returns 5.0
divide(5, 0)    # AssertionError:  b cannot be 0
```

Assertions vs Exceptions

Assertions

"Am I (the programmer) right?"

- Used during **development**
- Catches **programmer** mistakes
- `assert condition`
- Program **halts** if False

Exceptions

"Did the user/world behave?"

- Used in **production**
- Handles **runtime** problems
- `try/except` or `raise`
- Program can **recover**

You Try!

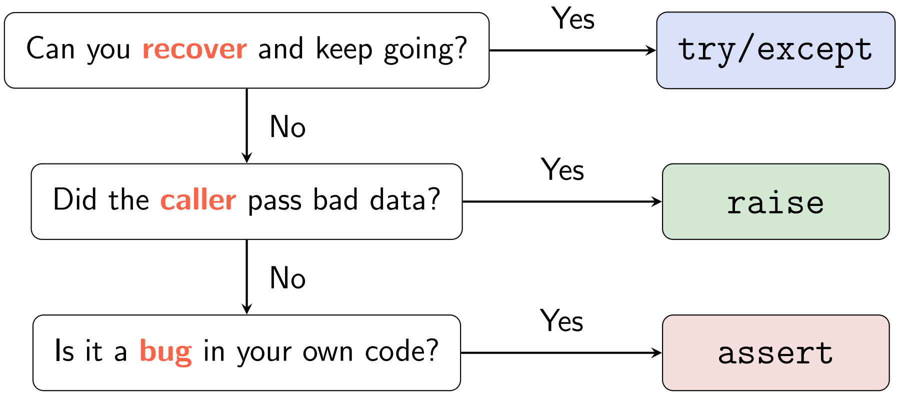
Add assert statements to pairwise_div:

```python
def pairwise_div(L1, L2):
    """ L1 and L2 are <non-empty> lists
        of <equal lengths> containing numbers.
    Returns pairwise division of L1 by L2.
    Raise ValueError if L2 contains 0. """
    # add assert lines here
```

You Try! Solution

```python
def pairwise_div(L1, L2):
    assert len(L1) > 0, "L1 is empty!"
    assert len(L2) > 0, "L2 is empty!"
    assert len(L1) == len(L2), "Different lengths!"
    result = []
    for i in range(len(L1)):
        if L2[i] == 0:
            raise ValueError(f"L2[{i}] is 0!")
        result.append(L1[i] / L2[i])
    return result
```

When to Use Each?


Can you **recover** and keep going?
Yes
`try/except`
No
Did the **caller** pass bad data?
Yes
`raise`
No
Is it a **bug** in your own code?
Yes
`assert`

Putting It All Together

Extended Example: Class Grades

A teacher has a class list. Each entry is a list of two parts:

- A list of first and last name
- A list of grades

```
class_list = [
  [['Ali', 'Khan'],      [85, 90, 78]],
  [['Fatima', 'Noor'],   [92, 88, 95]],
  [['Deadpool'],         []]
]
```

Extended Example: Class Grades

A teacher has a class list. Each entry is a list of two parts:

- A list of first and last name
- A list of grades

```
class_list = [
  [['Ali', 'Khan'],       [85, 90, 78]],
  [['Fatima', 'Noor'],    [92, 88, 95]],
  [['Deadpool'],          []]
]
```

Goal: compute the **average** grade for each student.

The Functions

```
def avg(grades):
    return sum(grades) / len(grades)

def get_stats(class_list):
    new_stats = []
    for stu in class_list:
        new_stats.append(
            [stu[0], stu[1], avg(stu[1])])
    return new_stats
```

The Functions

```python
def avg(grades):
    return sum(grades) / len(grades)

def get_stats(class_list):
    new_stats = []
    for stu in class_list:
        new_stats.append(
            [stu[0], stu[1], avg(stu[1])])
    return new_stats
```

`get_stats(class_list)` → **CRASH!**

`ZeroDivisionError`: Deadpool has **no grades** → `len([])` is 0

Option 1: `try/except`

Display a warning, keep going:

```python
def avg(grades):
    try:
        return sum(grades) / len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return None
```

Option 1: `try/except`

Display a warning, keep going:

```python
def avg(grades):
    try:
        return sum(grades) / len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
        return None
```

get_stats(class_list) →

```
[['Ali', 'Khan'],      [85, 90, 78], 84.33]
[['Fatima', 'Noor'],   [92, 88, 95], 91.67]
[['Deadpool'],         [],           None ]
```

Option 2: Guard Clause

Policy decision: no grades = zero average.

```
def avg(grades):
    if len(grades) == 0:
        return 0.0
    return sum(grades) / len(grades)
```

Option 2: Guard Clause

Policy decision: no grades = zero average.

```python
def avg(grades):
    if len(grades) == 0:
        return 0.0
    return sum(grades) / len(grades)
```

get_stats(class_list) →

```
[['Ali', 'Khan'],      [85, 90, 78], 84.33]
[['Fatima', 'Noor'],   [92, 88, 95], 91.67]
[['Deadpool'],         [],           0.0  ]
```

Option 3: Assert

Strictest: empty grades means a **bug upstream**.

```python
def avg(grades):
    assert len(grades) > 0, "No grades data!"
    return sum(grades) / len(grades)
```

Option 3: Assert

Strictest: empty grades means a **bug upstream**.

```python
def avg(grades):
    assert len(grades) > 0, "No grades data!"
    return sum(grades) / len(grades)
```

`get_stats(class_list)` →

```
AssertionError: No grades data!
```

Program **halts** — having no grades **should never happen**.

Comparing the Three Approaches

Approach	Behavior	Use When
`try/except`	Skip problem, continue	User-facing app; show friendly message
Guard clause	Return default value	Missing data has a sensible default
`assert`	Halt program	Bug in the code; should never happen

Comparing the Three Approaches

Approach	**Behavior**	**Use When**
`try/except`	Skip problem, continue	User-facing app; show friendly message
Guard clause	Return default value	Missing data has a sensible default
`assert`	Halt program	Bug in the code; should never happen

Which one is best? *It depends on the context!*

Summary

Summary

- **Exceptions** — Python's error reporting mechanism
  - Common: `ValueError`, `TypeError`, `IndexError`, `ZeroDivisionError`
  - `try/except` to catch; `raise` to throw
  - `else` for success; `finally` for cleanup
- **Assertions** — programmer's sanity check
  - `assert condition, "message"`
  - Halts on failure; for development-time checks
- **Rule of thumb:**
  - Exceptions for **runtime** problems
  - Assertions for **logic** bugs

Big Idea

Good programs don't just **work** —

— they fail **gracefully**.

Exceptions handle the unexpected.
Assertions verify the expected.

Questions?