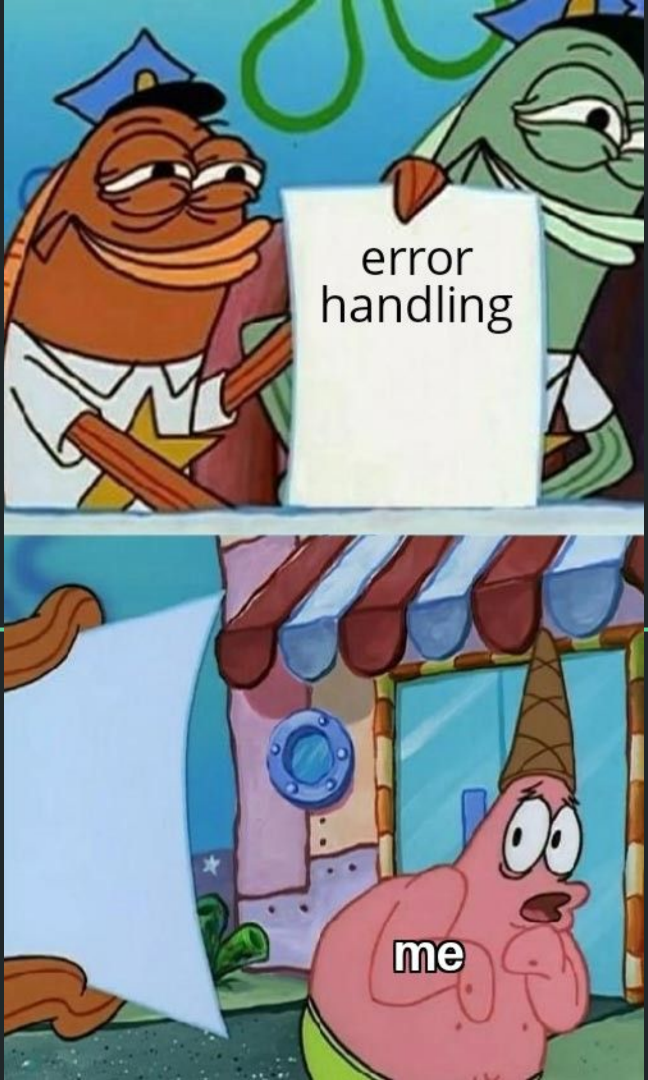
Error handling



Objectives for today

- Identify and handle common Python exceptions
- Use try/except and raise to write defensive, self-documenting functions
- Write and interpret unit tests using assert
- Apply error handling to real data workflows

You’ll encounter various types of errors

- **Syntax**: language rules broken
    - E.g., quotes missing, incorrect indentation
- **Runtime**: unable to execute
    - E.g., zero division error, or an unrecognized variable
- **Semantic/Logical**: unexpected output, e.g.:

```
>>> name = “Alice”
>>> print(“Hello name”)
>>> Hello name
```

Code can run without error, but still not be doing what you intend!

For a full list of possible errors:
https://www.tutorialsteacher.com/python/error-types-in-python

Exceptions are errors that occur while a Python program is running — as opposed to syntax errors, which Python catches before the code even runs

- `ZeroDivisionError` — dividing by zero
- `TypeError` — wrong data type (e.g. passing a string where a number is expected)
- `IndexError` — accessing a list position that doesn't exist
- `KeyError` — looking up a dictionary (or Pandas column) key that doesn't exist
- `FileNotFoundError` — reading a file that doesn't exist

`try/except` blocks let you *catch* these exceptions and respond gracefully instead of crashing

Different ways to handle error catching

Option #1: Messages to the user & breaking the code

```
if something
    print('This isn't working.')
    break
```

💡 An **`if`** block is best when you can and should check a condition *before* attempting an operation. It's proactive — you're validating inputs or state in advance.

For example, checking whether a DNA string contains only valid bases before computing GC content is a natural **`if`** situation.

Different ways to handle error catching

Option #1: Messages to the user & breaking the code

Option #2: try/except

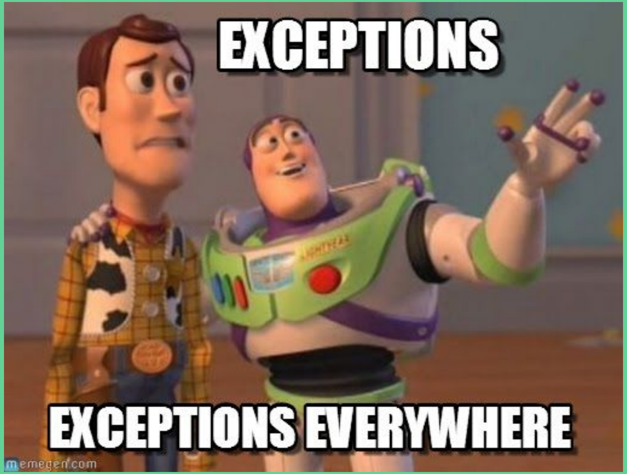
`try` a certain operation, `except` do something else

💡 **`try/except`** is better when the failure would come from actually *attempting* the operation, especially when that operation depends on something outside your control — a file that may not exist, user input, etc.

For example, you can't always know ahead of time whether **`pd.read_csv(filepath)`** will succeed, and writing an **`if`** check robust enough to cover every failure mode would be more complex than just trying it and catching what goes wrong.

If you can write a simple, readable condition that catches the problem *before* it happens, use **`if`**.

If the problem only reveals itself when you *try* something, use **`try/except`**.



Unit tests

- Trying a **known example** with a function and asserting that it gives the expected result.
- You *do not* need to use `unittests` (a specific package to implement this)

keyword

condition you're checking

statement that prints if it fails

```
assert sum([1, 2, 3]) == 6, "Should be 6"
```

https://realpython.com/python-testing/;
https://www.dataquest.io/blog/unit-tests-python/

Defensive code: code that anticipates things going wrong

The goal is to make your code fail ***loudly*** and ***early***.

- Assume your data is messy!
- Make your code complain clearly.
- Test with cases you expect to break.

https://swcarpentry.github.io/python-novice-inflammation/10-defensive.html

First let's revisit the Data Analysis notebook, and then get into Error Handling...