



Raising and handling errors

Data Sciences Institute
University of Toronto

Simeon Wong

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Course overview

- ✓ 1. Version control with Git
- ✓ 2. Configuration
- 3. Raising errors**
- 4. Testing software
- 5. Modules and packages
- 6. Application programming interfaces

Python Exceptions

Handling errors

Errors in Python

- Two main types of errors in Python:
 - Syntax Error: code that is not understandable (e.g. not valid Python statements)
 - Exceptions: code that is syntactically correct, but cannot be executed
- When Python executes code that results in an error, it **raises** an **Exception**
- If not **handled**, exceptions will cause Python to stop executing your code and quit
 - In Python notebooks like colab, it will stop executing code, but it won't close your notebook

Handling errors

Why errors occur

- Syntax Errors: usually easy-to-fix issue with the code
- Exceptions:
 - Can be a coding error or an error external to the program (e.g. network issues)
 - Usually caused by a **combination of / an interaction** between the code and some external factors
 - e.g. The file specified by the user doesn't exist, but your program wasn't coded to check first

Handling errors

The Exception object

- Contains information about the nature of the Exception
 - e.g. ValueError, NameError, TypeError, ZeroDivisionError
- Contains the line of code that caused the error
- Contains the context where the exception was raised
 - Also called the stack traceback
 - List of function calls that led to the current error
- **This info is useful to a programmer, but maybe not to a user**
 - **Recall:** user more about how to use it rather than how it works inside

Handling errors

try/except

- Allows your code to **handle** an error without stopping code execution
- Some possibilities:
 - Raise a modified Exception with more helpful error messages
 - *Recall*: Error info is useful to a programmer, but maybe not to a user
 - Diagnose and try to fix the error
 - Cannot reach primary API server, let's try the secondary one
 - Ignore/log the error and move on
 - Processing a batch of files in a loop: move onto the next file and alert the user at the end which files didn't work
 - Be careful of **failing silently**: undetected errors are problematic

Handling errors

\$> Interactive live coding

- Handle a `ValueError` when converting `str` to `int`
 - Add a note using `e.add_note()`

```
value = 'not a number'
```

```
try:
```

```
    value = int(value)
```

```
except ValueError as e:
```

```
    e.add_note(f'Input value {value} is not an integer')
```

Questions?

Throwing errors

Defensive programming

- Mistakes will happen. Guard against mistakes.
- Your code can raise errors when it detects a problem.
- We can add user-defined error messages to indicate the error.

Throwing errors

Writing your own exceptions

- Use the `raise` keyword along with an Exception object

```
def calc_circle_area(radius:float) -> float:
    ''' Calculates the area of a circle given a radius. '''
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2
```

Throwing errors

Writing your own exceptions

```
def calc_circle_area(radius:float) -> float:
    ''' Calculates the area of a circle given a radius. '''
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2
```

```
calc_circle_area(-1)
```

⊗ 0.3s

ValueError Traceback (most recent call last)

Cell **In[2]**, **line 1**

→ **1** calc_circle_area(-1)

c:\repos\UTDSI_202401_building_software\lessons\2 - Documentation\exceptions_raiseyourown.py in line 5, in calc_circle_area(radius)

3 ''' Calculates the area of a circle given a radius. '''

4 if radius < 0:

→ **5** raise ValueError("Radius cannot be negative")

6 return math.pi * radius ** 2

ValueError: Radius cannot be negative

Throwing errors

\$> Interactive live coding

- In a Python notebook, paste in function to calculate circle area

```
import math
```

```
def calc_circle_area(radius):  
    return math.pi * radius**2
```

```
radius = 2  
area = calc_circle_area(radius)
```

- Edit the code to raise `ValueError` if radius is negative

Throwing errors

Writing useful error messages

- Be specific, clear, concise, and actionable
 - **Not specific:** "Error"
 - **Better, but still not specific:** "ValueError"
 - **Great:** "ValueError: got 52.1 (float) but expected an integer"
- Write for your audience's level of understanding.
 - "Authorization error: credentials have expired" vs. "HTTPError_401"
- Use consistent vocabulary within projects and organizations
 - Is a "Loading Error" the same as a "File not found" error?

Throwing errors

Writing useful error messages

- Don't blame the user
 - "OSError: data.txt not found" is better than "UserError: Your path is wrong"
- Avoid catastrophe words
 - "fatal", "illegal", "danger", "aborted" may make users worry unnecessarily about device or data damage
- Avoid jokes and cutesy language
 - **Don't do this:** "Oopsie-daisy! Looks like the value you provided wasn't an integer! Why don't you try again pal!"


Assertions

Assertions

- Short-hand way of raising an error if a statement is false
- Useful for checking the consistency of your program state
 - guard against programming error
 - a form of documenting your code and thought process
- Use for conditions that should **never** be true (invariant)

```
def calc_circle_area(radius:float) -> float:
    ''' Calculates the area of a circle given a radius. '''
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2

radius = 2
area = calc_circle_area(radius)
assert area > 0, f"Area is {area} but should be positive"
```



If this is false, something is very wrong in our code!

Assertions

When to **raise** vs **assert**

raise

- Use for most things (e.g. input validation, failed operations)
- Exception types (ValueError, etc.) provide detailed information

assert

- Use to detect invariants – things that should *never happen*
- Always raises an AssertionError with an optional message
- Helps catch coding mistakes and annotate programmer's inner mindset

Assertions

\$> Interactive live coding

- Revisiting the `calc_circle_area` function
- Assert that the function output must be a float
- Assert that the function output must be positive
- Change output of the `calc_circle_area` function and test!

```
assert isinstance(area, float)
assert area > 0
```

Assertions

\$> Interactive live coding

- In a Python notebook, load the Iris dataset in a DataFrame
 - Assert that the variable is a DataFrame
 - Assert that expected columns exist
- In another cell:
 - Assert that the variable is a DataFrame
 - Overwrite the dataset with the sepal_length column
- Run this cell again. What happens?

Assertions

\$> Interactive live coding

Cell 1

```
import pandas as pd
iris_data = pd.read_csv(
    'https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv')
assert isinstance(iris_data, pd.DataFrame), "iris_data should be a DataFrame"
assert 'sepal_length' in iris_data.columns, "sepal_length should be a column in iris_data"
```

##

```
assert isinstance(iris_data, pd.DataFrame), "iris_data should be a DataFrame"
iris_data = iris_data['sepal_length']
```

Questions?

Logging

The Python logging library

- Not all detectable issues are full-blown errors
- Python's `logging` library allows for more nuanced messages of different severity levels
 - `DEBUG`: very detailed information used to diagnose outputs
 - `INFO`: confirmation messages
 - `WARNING`: possible unexpected result or scenario, but code can continue
 - `ERROR`: the program cannot continue, but no "permanent damage"
 - `CRITICAL`: potential loss of data, security issues

Logging

The Python logging library

- Allows the user to specify which level of messages they want to see at any given time
- Logging levels always include messages of higher severity

```
import logging

logging.basicConfig(level=logging.DEBUG, filename='logging.log')

logging.debug('This is for debugging.')
logging.info('This is just for information.')
logging.warning('This is a warning.')
logging.error('Something went wrong.')
logging.critical('Something went seriously wrong.')
```

```
DEBUG:root:This is for debugging.
INFO:root:This is just for information.
WARNING:root:This is a warning.
ERROR:root:Something went wrong.
CRITICAL:root:Something went seriously wrong.
```


Logging

\$> Interactive live coding

- Modify the UCI Iris dataset code.
- Setup logging to file, and show INFO messages
- Load dataset in a function.
 - Log the loaded dataset path
- Catch **HTTPError**
 - Add a note
 - Log as error

Logging

\$> Interactive live coding

```
import pandas as pd
import logging

logging.basicConfig(
    level=logging.INFO,
    handlers=[logging.StreamHandler(), logging.FileHandler('uci_iris.log')],
)

def data_loader(path:str) -> pd.DataFrame:
    logging.info('Loading data from {}'.format(path))
    try:
        data = pd.read_csv(path)
    except Exception as e:
        logging.error('Error loading data from {}'.format(path), exc_info=True)
        e.add_note('Error loading data from {}'.format(path))
        raise e

    return data

iris_data = data_loader('https://raw.githubusercontent.com/uiuc-cse/data-fa14/g-pages/data/iris.csv')

assert isinstance(iris_data, pd.DataFrame), "iris_data should be a DataFrame"
assert 'sepal_length' in iris_data.columns, "sepal_length should be a column in iris_data"
```

Questions?

Course overview

- ✓ 1. Version control with Git
- ✓ 2. Configuration
- 3. Raising errors**
- 4. Testing software
- 5. Modules and packages
- 6. Application programming interfaces

Course objective

How to write robust software in a team that we, our colleagues, and the public can trust and use with confidence.

Homework

- In your Homework 1 script:
 - Raise an error or make an assertion
 - Implement `try/except` where it could be useful to add context
 - Implement the `logging` module
- Ungraded. Submit before next class for feedback.

References

- Research Software Engineering with Python by Damien Irving, Kate Hertweck, Luke Johnston, Joel Ostblom, Charlotte Wickham, and Greg Wilson (<https://merely-useful.tech/py-rse/config.html>)
- [PEP 257 \(python.org\)](https://www.python.org/dev/peps/pep-0257/)
- [Style guide — numpydoc v1.7.0rc0.dev0 Manual](#)