Documentation and Handling Errors

Data Sciences Institute
University of Toronto

Simeon Wong

Previously at the DSI…

- Configuring programs
  - Configuration files make your use of programs documented and repeatable
  - Writing configuration files in YAML
  - Loading YAML configuration files into Python
- Application Programming Interfaces
  - Programs using programs
  - REST is the quasi-standard for Web APIs
  - Using Python requests to retrieve data from GitHub programmatically

Course overview

1. Configuration files & Environment variables
2. **Using and writing Application Programming Interfaces (APIs)**
3. **Handling errors**
4. Testing software
5. Building Python packages
6. Working in software teams using GitHub features

Daily scrum

Write in the chat:

1. How you’ll use one thing you’ve learned so far
2. One thing you’re struggling with / wish the course addressed
3. What you’re looking forward to in this course

Asking questions

- Zoom chat during class
  - Feel free to post and answer questions at any time
  - I will pause for questions occasionally, and review questions from the chat
- Pre- / Post-class office hours with Tong
- Email
  - simeonm.wong@mail.utoronto.ca
  - tong.su@mail.utoronto.ca

Assessments

- Outcomes-based learning
  - > 75% of course outcomes
- Assessed with both in-class exercises and summative project
- Summative project:
  - The next 4 classes build on each other
  - We will work on the summative project step-by-step through each class

Today’s learning outcomes

- Given a function, I can write API documentation
- Given documentation, I can write Python class and function headers described by that documentation
- I can interpret a Python error message
- I can catch and handle errors using try/except
- I can write helpful error messages
- I can use the Python logging library to control output from my code

Documentation

Why documentation?

> *"Code is more often read than written."*
> — Guido van Rossum

- Code is read by yourself (in the future) and by other developers who use it or contribute to it
- Code that cannot be properly understood …
  - e.g. its required inputs, its methods, and its outputs
- … can be used in situations or in ways that result in errors or mistakes

Documentation

Commenting vs Documenting code

- **Comments** describe your code for other developers
- It makes it easier to understand how (algorithmically) and why
- Helps with understanding intention, purpose, and design rationale

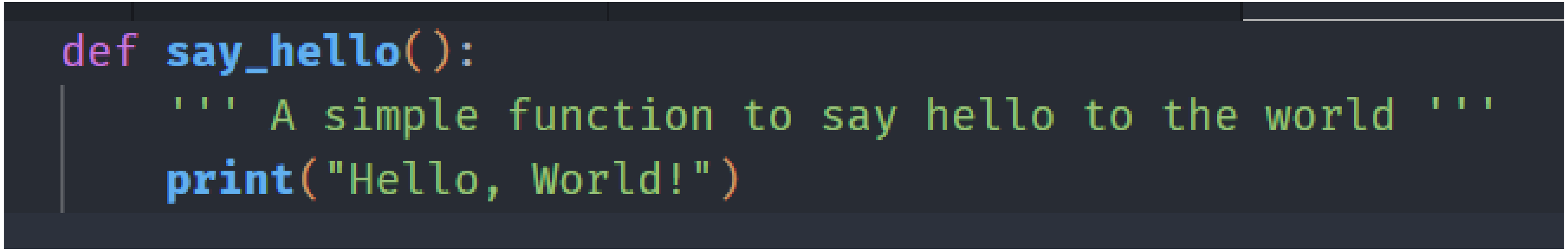
- **Documentation** describes your code for your users
- What your code does (for a user) and how to use it

Both are important!

Documentation

Python docstrings

- Official Python standard to describe your code’s functionality
- Wrapped in “’triple apostrophes or quotation marks’” at the very beginning of a function

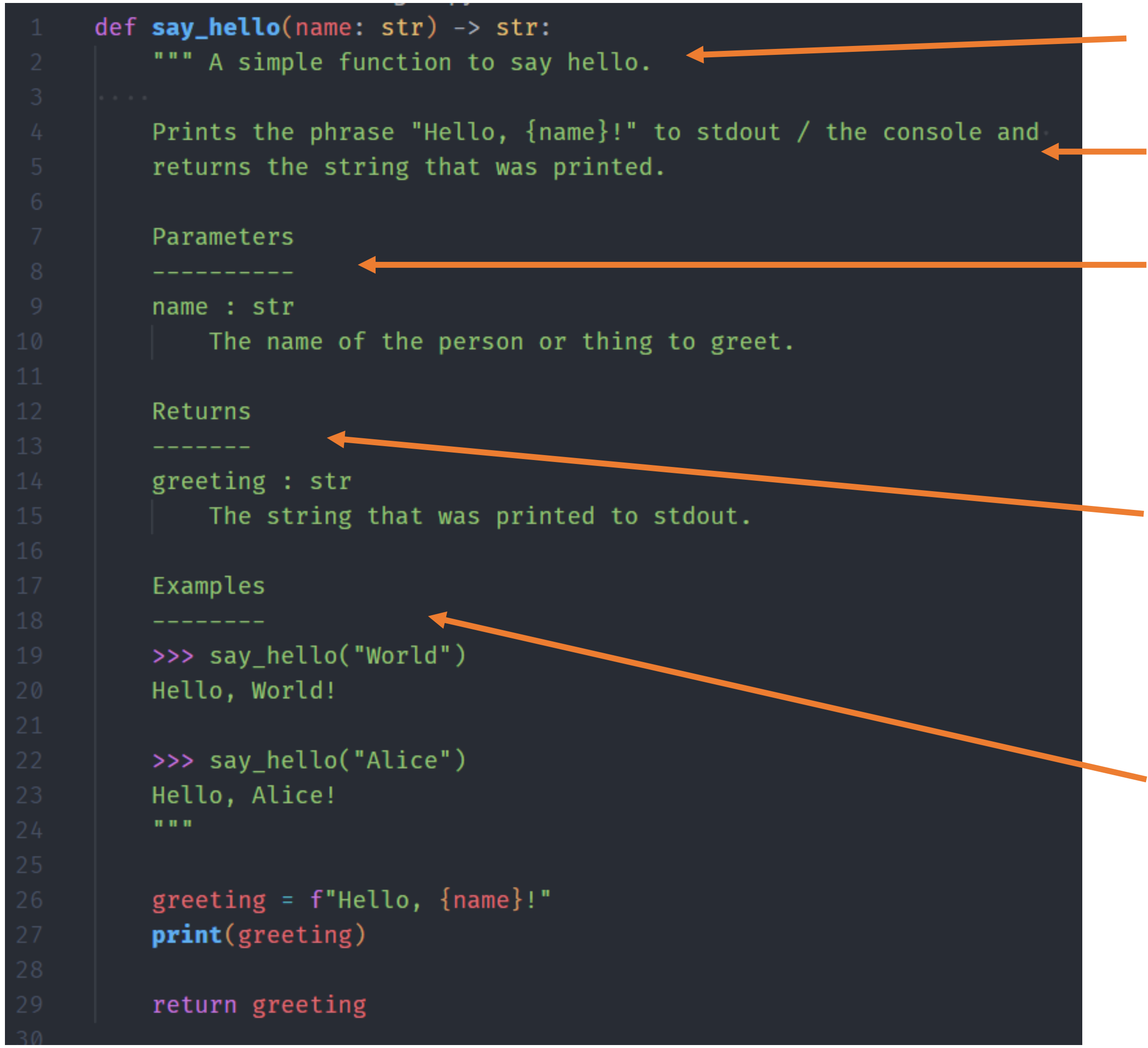


```
def say_hello():
    ''' A simple function to say hello to the world '''
    print("Hello, World!")
```

Reference: https://peps.python.org/pep-0257/

Documentation

The numpy docstring style



```python
def say_hello(name: str) -> str:
    """ A simple function to say hello.

    Prints the phrase "Hello, {name}!" to stdout / the console and
    returns the string that was printed.

    Parameters
    ----------
    name : str
        The name of the person or thing to greet.

    Returns
    -------
    greeting : str
        The string that was printed to stdout.

    Examples
    --------
    >>> say_hello("World")
    Hello, World!

    >>> say_hello("Alice")
    Hello, Alice!
    """

    greeting = f"Hello, {name}!"
    print(greeting)

    return greeting
```

Short summary (max one-line)

Extended summary / description of function

List of parameters formatted as:

```
param_name : param_type
      Description of parameter
```

List of returned values formatted as:

```
value_type
  Description of this value
```

Optional examples

Reference: https://numpydoc.readthedocs.io/en/latest/format.html

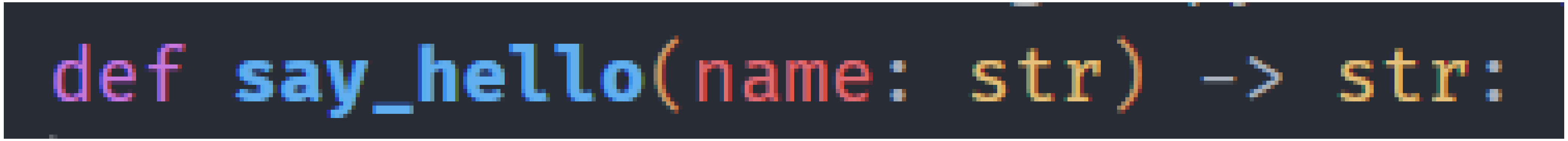
Documentation

Python type hints

- The Python standard for documenting the types of values expected by your function
- Concise way of representing a subset of the information from the numpy docstring

`name` should be a `str`

Function returns a `str`



```python
def say_hello(name: str) -> str:
```

References: https://peps.python.org/pep-0484/

Documentation

$> Interactive live coding

- Refactor the Hello World program as a function
- Rewrite the command-line arguments as function parameters
  - Use Python type hints
- Write a docstring for the function
- Call the Hello World function based on the parsed arguments

Exercise: Refactor your GitHub API code

- Which function parameters are needed?
- Write a docstring for the function in the numpy style

Documentation

Sphinx documentation generator

- The most common web-based / online documentation generator for Python projects
- Parses your docstrings and other indicated text files
- Compiles into HTML files
  - Easier to read and browse
  - Easier to share online (or on an internal site) with your users
- Requires a moderate amount of setup to get started

Documentation

$> Interactive live coding

- Install sphinx, numpydoc, and myst-parser (markdown parser)
- Copy and paste sphinx configuration and index files
- Generate documentation for your GitHub file

What questions do we have?

Course overview

1. Configuration files & Environment variables
2. **Using and writing Application Programming Interfaces (APIs)**
3. **Handling errors**
4. Testing software
5. Building Python packages
6. Working in software teams using GitHub features

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()`
- Handle a `ConnectionError` from the `requests` library
  - Refer to requests documentation about types of Exceptions raised
  - Try to connect to a backup API

What questions do we have?

Handling errors

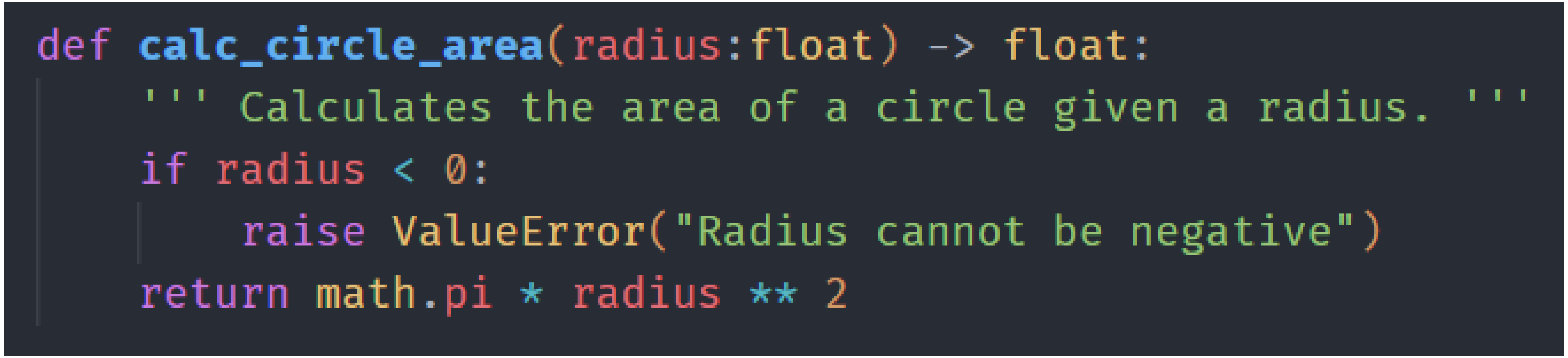
Defensive programming

- 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.

Handling errors

Writing your own exceptions

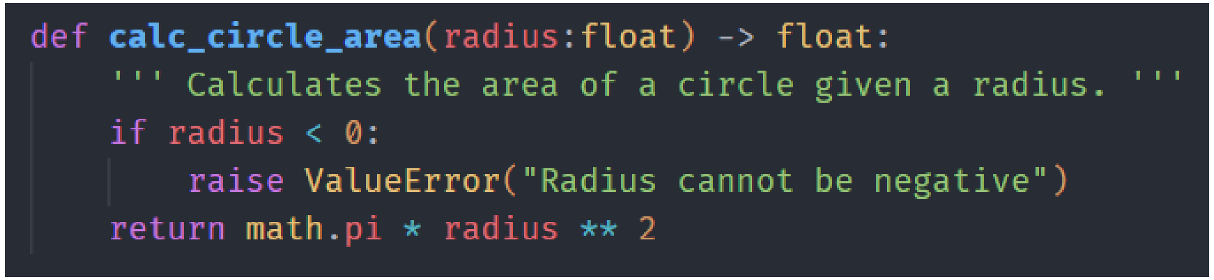
- Use the `raise` keyword along with an Exception object



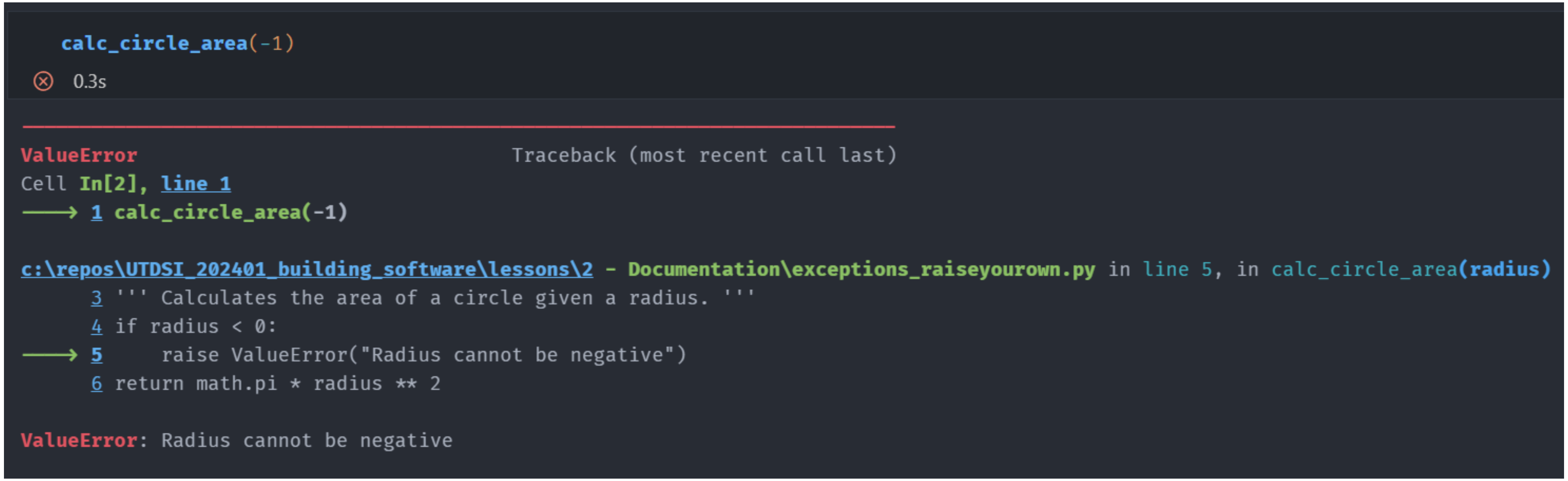
```python
def calc_circle_area(radius:float) -> float:
    ''' Calculates the area of a circle given a radius. '''
    if radius < 0:
        raise ValueError("Radius cannot be negative")
    return math.pi * radius ** 2
```

Handling errors

Writing your own exceptions



```python
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
```

Handling 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?

Handling errors

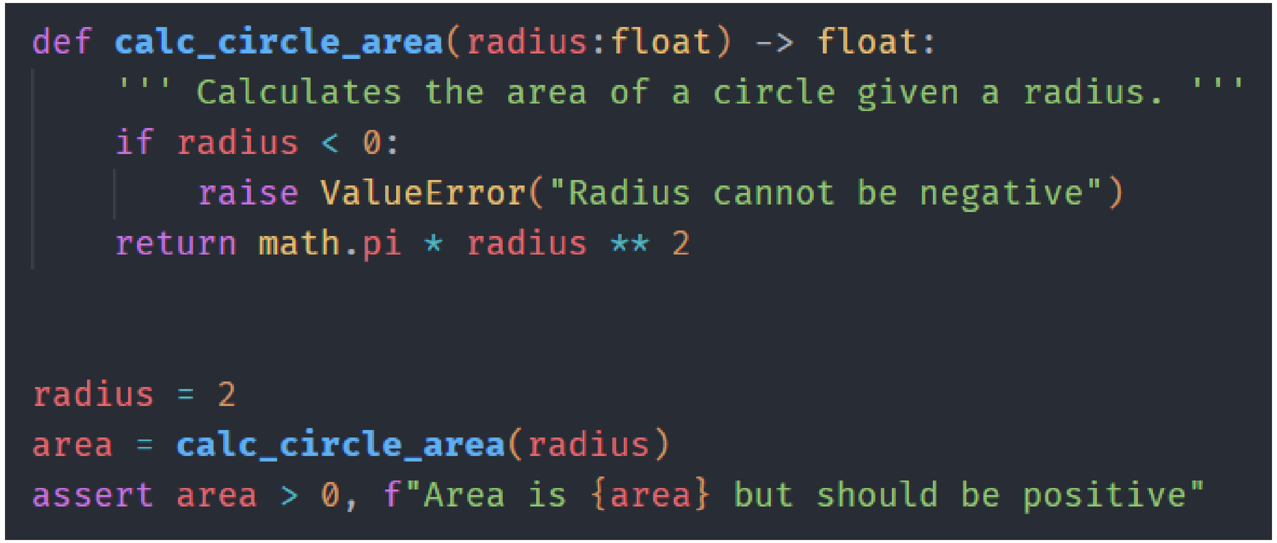
Writing useful error messages

- 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!”

Handling errors

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!

Handling errors

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

What questions do we have?

Handling errors

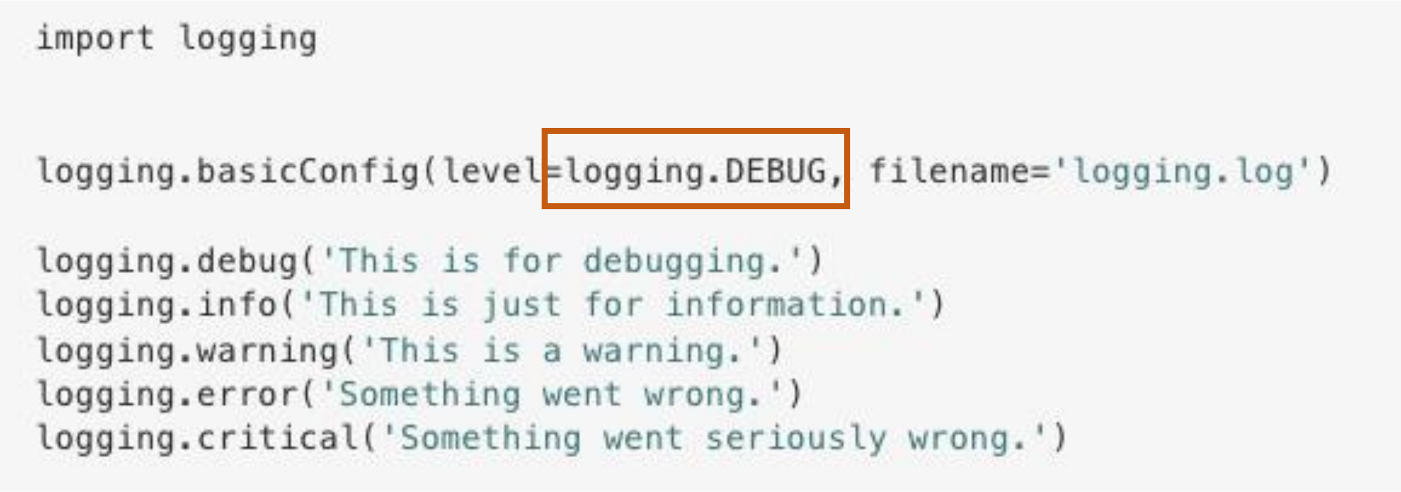
The Python logging library

- 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

Handling errors

The Python logging library

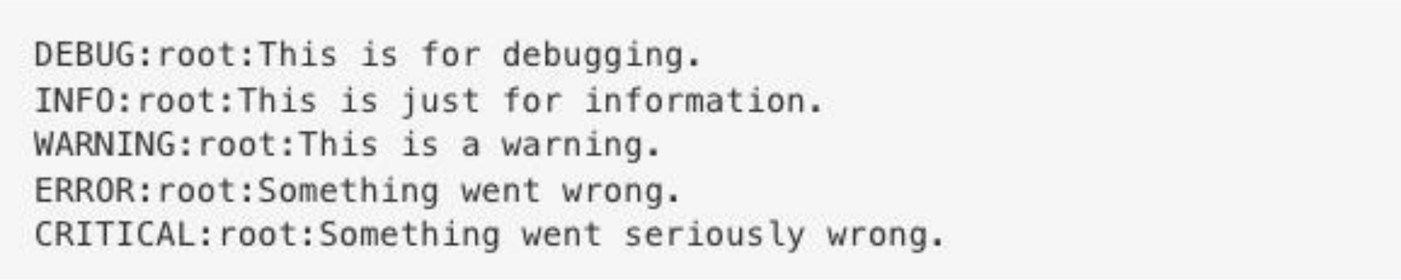
- Allows the user to specify which level of messages they want to see at any given time
- Logging levels always include messages of higher severity



```
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.
```

Handling errors

$> Interactive live coding

- Demonstrate the logging library

What questions do we have?

Today’s learning outcomes

- Given a function, I can write API documentation
- Given documentation, I can write Python class and function headers described by that documentation
- I can interpret a Python error message
- I can catch and handle errors using try/except
- I can write helpful error messages
- I can use the Python logging library to control output from my code

Handling errors

Exercise + Homework

- Add error handling and logging to your GitHub API code from lesson 1
    - Write useful error messages
    - Use try/catch
    - Validate inputs
    - Consider where DEBUG, INFO, and WARNING messages might be useful

- Commit and push your changes to the same repository from Lesson 1

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)
- Style guide — numpydoc v1.7.0rc0.dev0 Manual