# Debugging and Error Handling

## Our Learning Goals

- Identify common errors in Python
- Read error messages for guidance to fix errors
- Implement try/except error handling

## Making Errors Into Friends

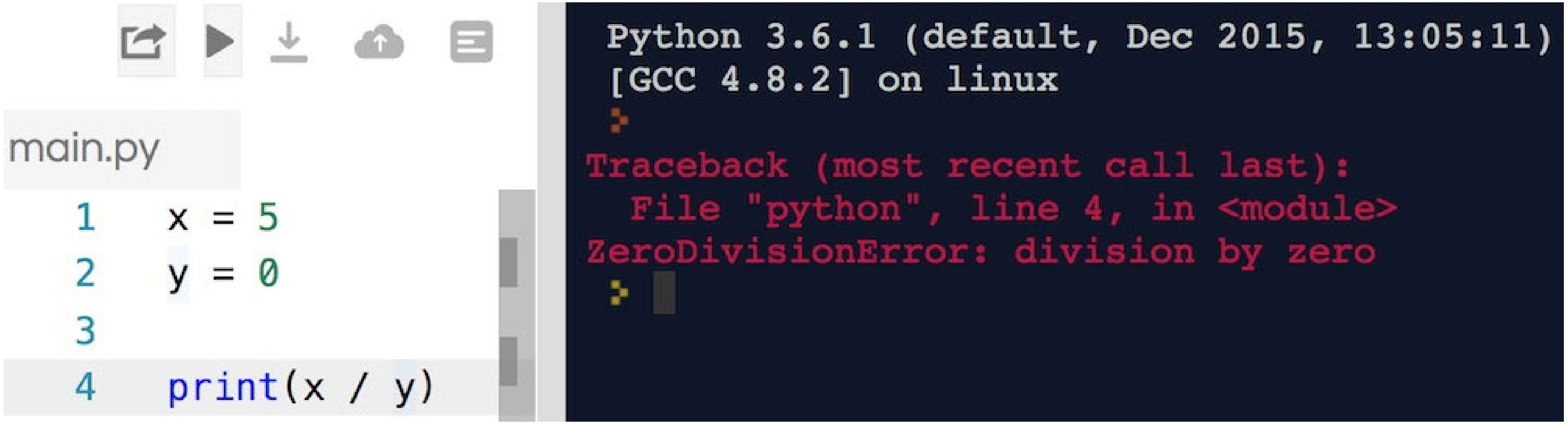
Python errors are very helpful and typically have clear messages.

- Errors sometimes say exactly what's wrong.
- Some errors have very common causes.
- Errors may say exactly how to fix the issue.

Once you've seen the same error a few times, you'll start to know exactly where to look to fix the problem.

## Error Messages

What's wrong with the code snippet below? What information is the error message giving us?



```
x = 5
y = 0

print(x / y)
```

```
Python 3.6.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
›
Traceback (most recent call last):
  File "python", line 4, in <module>
ZeroDivisionError: division by zero
›
```

Solo Exercise:

## Error Scavenger Hunt

30 minutes

We've curated a selection of the most common errors available in Python.

Run each of the code blocks in the notebook, read the error message, then describe in your own words what the error type means.

## Logic Errors

**Program Errors** are easy to spot: your code won't run, and it will complain about why.

**Logic Errors** are tougher to track down. The code snippet on the right, while running error-free, is not doing what the programmer expected.

```
def multiply(num_1, num_2):
    return num_1 + num_2


hundred = multiply(10, 10)

# hundred is actually 20!
```

## Print Your Way to Freedom!

When debugging logic errors, **print** statements at each step of the way can reveal where the program went wrong. You might add print statements to:

- Determine which blocks of code execute, and in what order
- Reveal the value of specific variables and parameters used
- Test out individual blocks of code to ensure they work as expected

Group Exercise:

## Debugging a Sort Function

30 minutes

We're trying to write a common sort function, bubble sort. However, our code has run into some errors, both logical and programmatic.

Use print statements and error codes to determine where the code is going wrong, taking things one step at a time.

Debugging and Error Handling

## Programmatic Error Handling

## User Input: Causing Errors Since Forever

Common offenders:

- Applications that require user input will, inevitably, get input from the user that isn't valid or doesn't make sense for the program.
- Applications that make requests to external data sources can sometimes get bad results, or no results, in response to their requests.

We need a way of creating programs that can anticipate when a block of code might run into an error.

## Try and Except Your Code

Fortunately, errors don't have to be the end of the world! Once you start to anticipate what could go wrong, you can use **try** and **except** statements to plan ahead.

```
try:
        starting_value = int(input("Give me a number one through ten"))
except ValueError:
        print("Your input was not a number. Try again.")
```

## Try-Except Options

. We can catch:

- One error
    - `except ValueError:`
- Multiple errors
    - `except (ValueError, KeyError):`
- Any/every error
    - `except err:`

Try to specify the error, if possible! You can even string several except statements to handle different errors differently.

## Accessing an Error

You can use the **as** statement, or aliasing, to turn your exception into a specific, accessible object:

```
try:
        risky_function()
except ValueError as value_err:
        print("Value error: ", value_err)
except Exception as err:
        print(err)
```

## Creating Your Own Errors (On Purpose This Time)

```
lucky_number = int(input("Type a number that's less than 100"))
if(lucky_number > 100):
        raise ValueError("The number needs to be less than 100!")
```

Why would we do this?

Discussion:

## Error Handling Possibilities

5 minutes

Proper error handling is all about thinking through every possibility. You should have a plan for every logical situation. What's the plan below? Add comments to the code block to explain what each line is doing.

```
try:
        lucky_number = int(input("Type a number that's less than 100"))
        if(lucky_number > 100):
                raise ValueError("The number needs to be less than 
100!")
        process_valid_number(lucky_number)
except ValueError as err:
        print("You need to provide a number")
```

Debugging and Error Handling

## Wrapping Up

## Recap

### In today's class, we…

- Identify common errors in Python
- Read error messages for guidance to fix errors
- Implement try/except error handling

## Looking Ahead

### Next Class:

Python Fundamentals Review Lab

## Don’t Forget: Exit Tickets!