

Lecture 12:

LIFE SKILLS 2 & 3: DEBUGGING & DOCUMENTATION

CSC111: Introduction to CS through Programming

R. Jordan Crouser

Assistant Professor of Computer Science

Smith College

Overview

- ✓ Monday: Loops
 - ✓ `for...in` (looping through items in a list)
 - ✓ the `range()` function (getting a list of numbers)
 - ✓ `while` (looping until something happens)
- ✓ Lab: Loops
- Wednesday: Life Skill #2/3: Debugging & Documentation
- Friday: Dictionaries and Sets

Assignment 3 (what's hard so far?)

In this assignment, you will write a python program that **simulates a Magic 8 Ball** (a toy first popularized in the 1950s, produced by Mattel).

The user interface of your Magic 8 Ball should first ask the user to input **a question**.
For example:

Your question: Will lunch at Chase/Duckett be good today?

Your program will then output a randomly selected answer, e.g.

Ask again later.



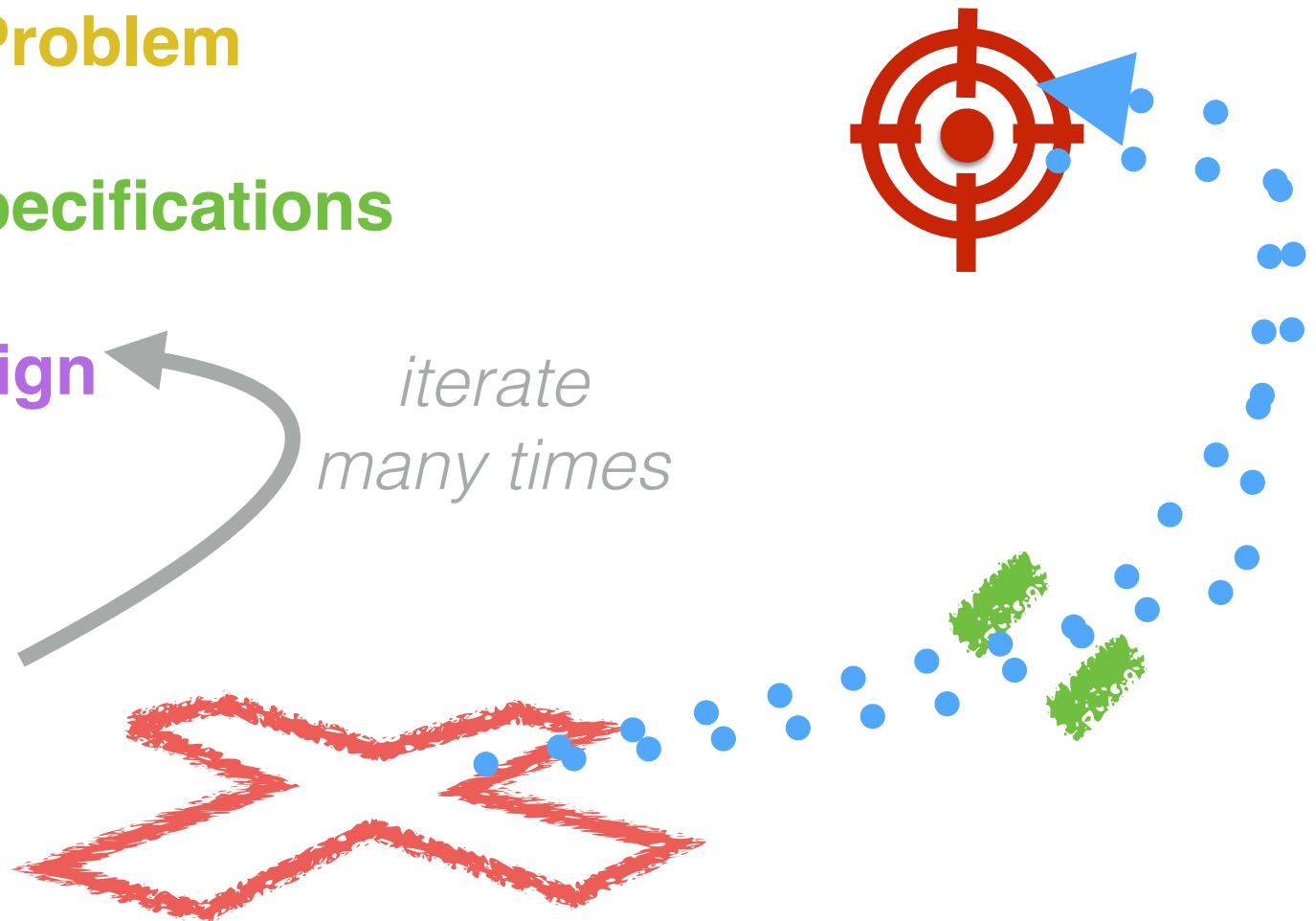
Discussion

What is your plan
for getting **unstuck**?



RECAP: the programming process

- Analyze the **Problem**
- Determine **Specifications**
Refine the
- ~~Create a~~ **Design**
- **Implement**
- **Test & Debug**



Fun history: the term “debug”

9/9

0800 arctan started
1000 " stopped - arctan ✓
13"cc (032) MP - MC
(033) PRO 2
cosine 2.130476415
cosine 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10.000 test .

1100 Started Cosine Tapc (Sine check)
1525 Started Mult + Adder Test.

1545



Relay #70 Panel F
(moth) in relay.

16160 arctangent started.
1700 closed down .



RDML Grace M. Hopper
b.1906 – d.1992

Some problems are obvious

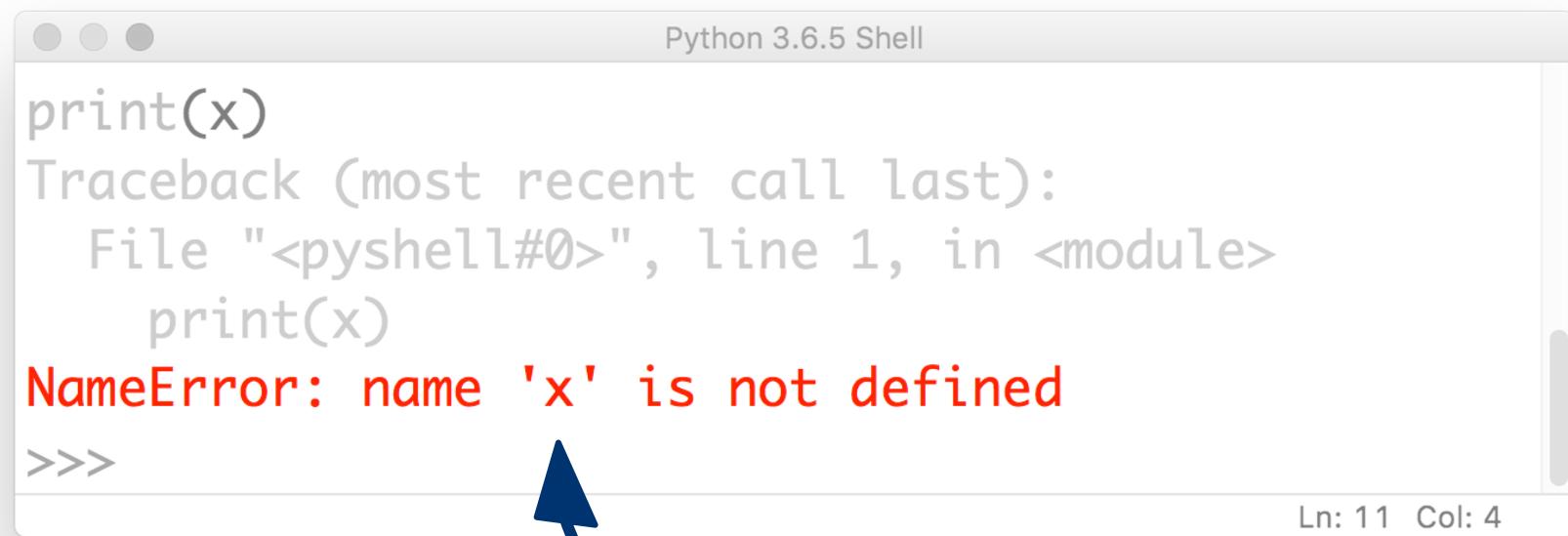
this is called
an **Exception**



```
Python 3.6.5 Shell
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Ln: 11 Col: 4

Some problems are obvious



A screenshot of a Python 3.6.5 Shell window. The title bar says "Python 3.6.5 Shell". The code input area shows "print(x)". A "Traceback (most recent call last):" message follows, detailing the error. The error itself is highlighted in red: "NameError: name 'x' is not defined". The bottom status bar shows "Ln: 11 Col: 4". A blue arrow points from the text "the kind of error gives you a **clue** about what the problem is" down to the red error message.

```
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

the kind of error gives you
a **clue** about what the problem is

Some problems are obvious

it also tells you **where** the problem is
(but be careful!)

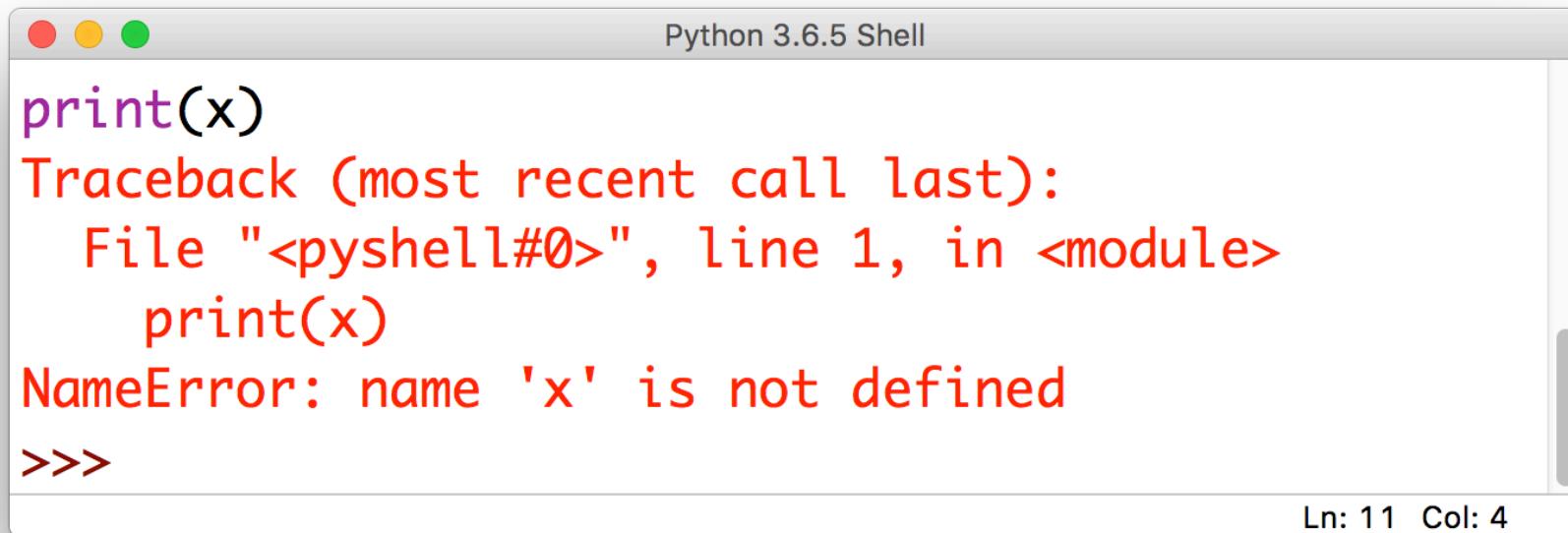


```
Python 3.6.5 Shell
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Ln: 11 Col: 4

Common Exceptions

- **NameError**: raised when Python can't find the thing you're referring to (a variable or a function)



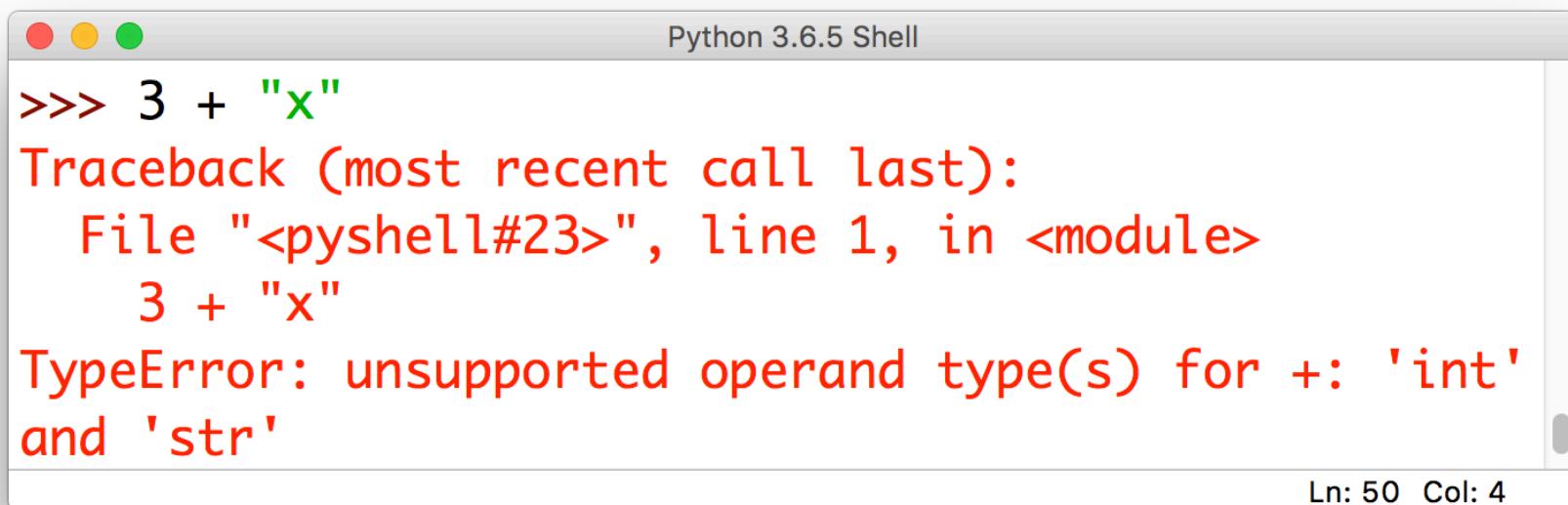
The screenshot shows a Mac OS X style window titled "Python 3.6.5 Shell". Inside the window, a command-line interface is displayed. The user has typed "print(x)" and then pressed Enter. A red error message follows:

```
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

In the bottom right corner of the window, there is a status bar with the text "Ln: 11 Col: 4".

Common Exceptions

- **TypeError:** raised when you try to perform an operation on an object that's not the right type (i.e. a string instead of a number)



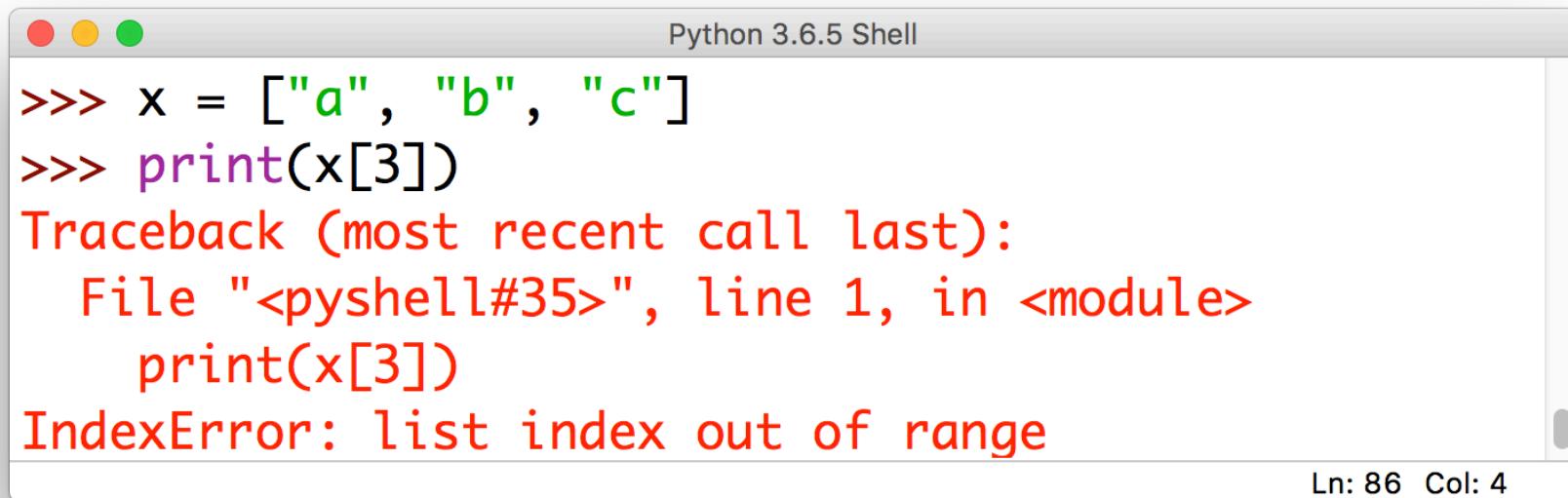
The screenshot shows a window titled "Python 3.6.5 Shell". In the terminal area, the following code and error message are displayed:

```
>>> 3 + "x"
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    3 + "x"
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In the bottom right corner of the window, the text "Ln: 50 Col: 4" is visible.

Common Exceptions

- **IndexError:** raised when you try to use an index that's out of bounds



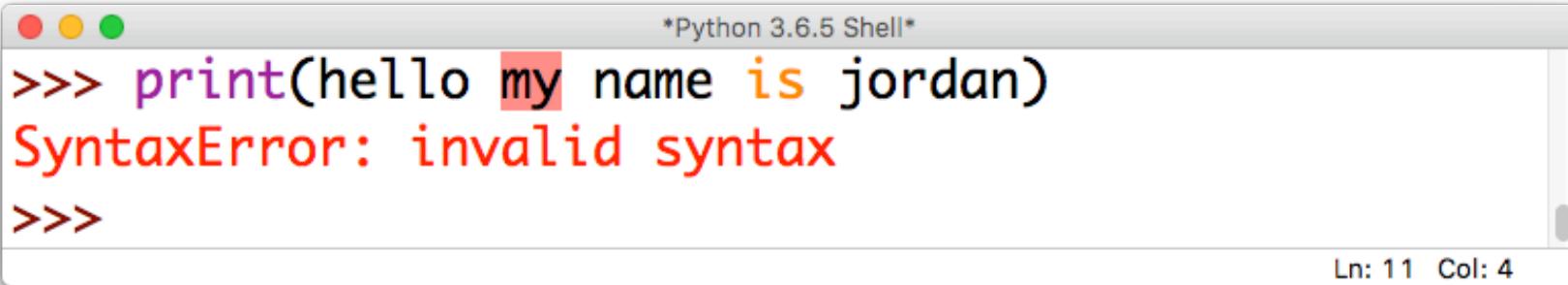
The screenshot shows a window titled "Python 3.6.5 Shell". Inside, a user has run the following code:

```
>>> x = ["a", "b", "c"]
>>> print(x[3])
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    print(x[3])
IndexError: list index out of range
```

The output shows a standard Python traceback for an `IndexError`, indicating that the list index is out of range.

Common Exceptions

- **SyntaxError:** raised when you try to run a command that isn't a valid Python statement



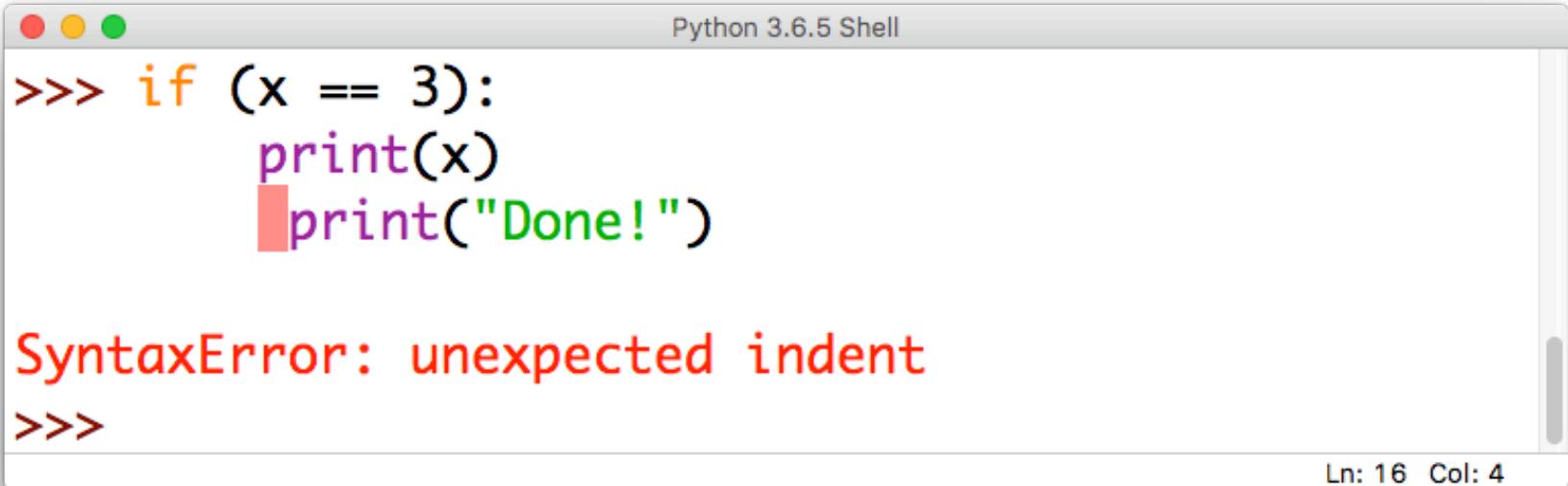
A screenshot of a Python 3.6.5 Shell window. The title bar says "*Python 3.6.5 Shell*". The console area shows the following text:

```
>>> print(hello my name is jordan)
SyntaxError: invalid syntax
>>>
```

The word "hello" is highlighted in red, and the word "is" is highlighted in orange. In the bottom right corner of the shell window, it says "Ln: 11 Col: 4".

Common Exceptions

- **SyntaxError:** also raised if your indentation is messed up (this is a special kind of SyntaxError called an IndentationError)



The screenshot shows a Python 3.6.5 Shell window. The code entered is:

```
>>> if (x == 3):
    print(x)
    print("Done!")
```

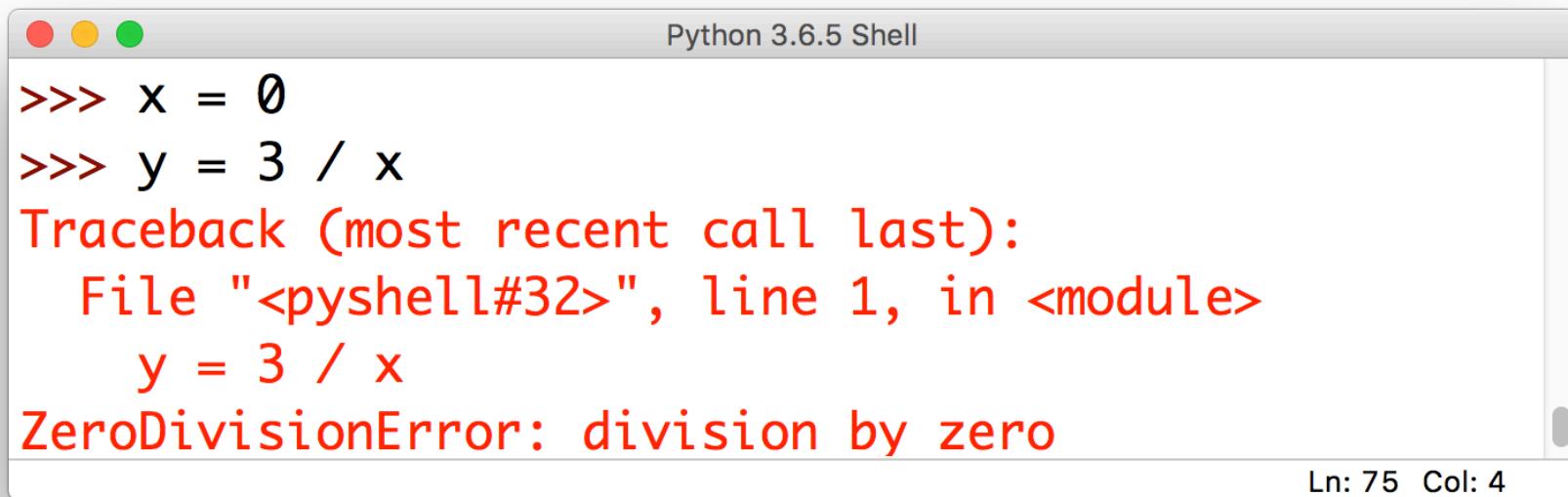
The line "print("Done!")" is highlighted with a red rectangle. The error message displayed is:

SyntaxError: unexpected indent

The status bar at the bottom right indicates "Ln: 16 Col: 4".

Common Exceptions

- **ZeroDivisionError**: raised when you try to divide by zero (or do modular arithmetic with zero)



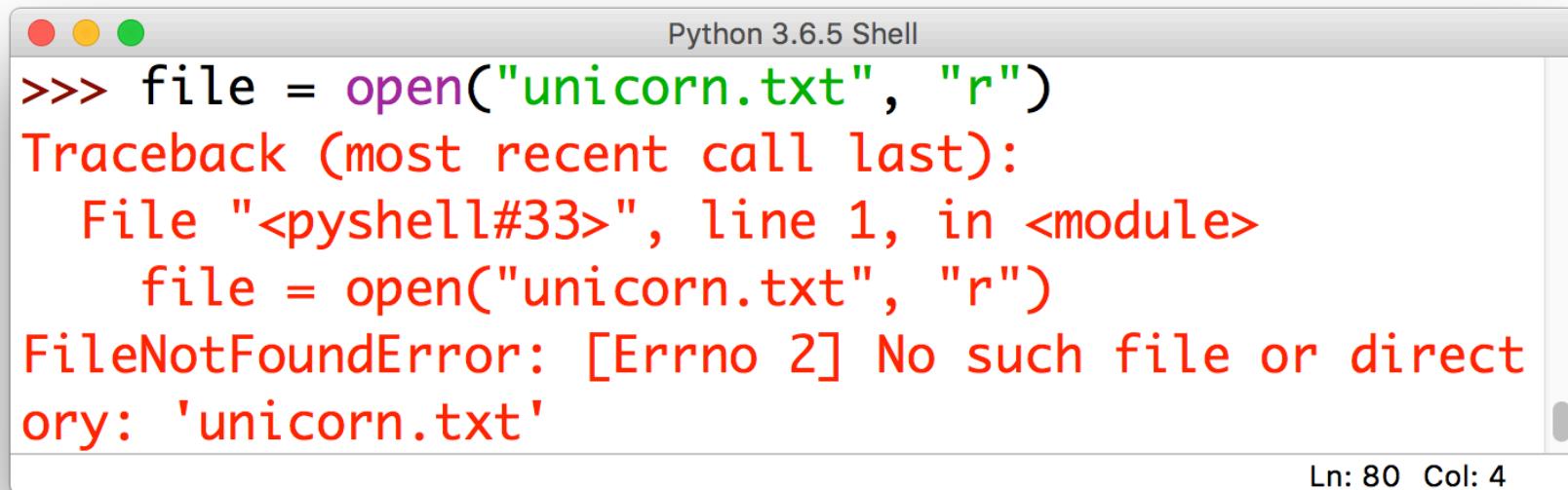
A screenshot of a Mac OS X window titled "Python 3.6.5 Shell". The window contains a command-line interface. The user has entered the following code:

```
>>> x = 0
>>> y = 3 / x
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    y = 3 / x
ZeroDivisionError: division by zero
```

The error message "ZeroDivisionError: division by zero" is displayed in red text. In the bottom right corner of the window, there is a status bar with the text "Ln: 75 Col: 4".

Common Exceptions

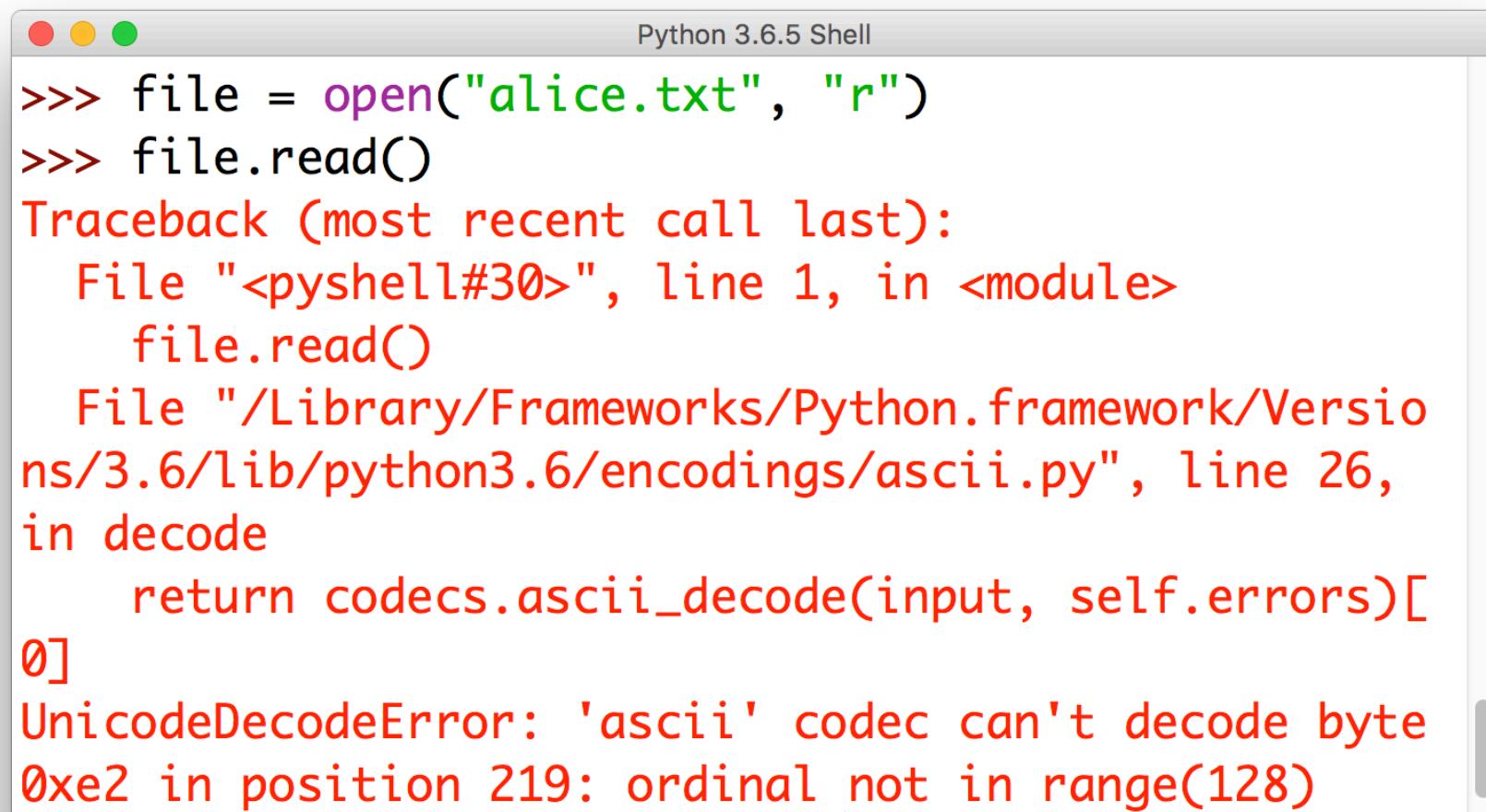
- **FileNotFoundException**: raised when Python can't find the thing you're referring to (a file)



The image shows a screenshot of a Python 3.6.5 Shell window. The title bar reads "Python 3.6.5 Shell". The code input is: `>>> file = open("unicorn.txt", "r")`. The output shows a traceback and an error message: `Traceback (most recent call last):
 File "<pyshell#33>", line 1, in <module>
 file = open("unicorn.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'unicorn.txt'`. In the bottom right corner of the shell window, there is a status bar with the text "Ln: 80 Col: 4".

Common Exceptions

- **UnicodeDecodeError**: raised when you try to read a file that has weird characters in it (most common culprit: apostrophe vs. the *single quote*)



A screenshot of a Mac OS X window titled "Python 3.6.5 Shell". The window contains a command-line interface. The user has run the following code:

```
>>> file = open("alice.txt", "r")
>>> file.read()
```

The output shows a traceback indicating the error occurred while reading from the file "alice.txt". The error message at the bottom is:

```
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    file.read()
      File "/Library/Frameworks/Python.framework/Versions/3.6/lib/python3.6/encodings/ascii.py", line 26,
    in decode
        return codecs.ascii_decode(input, self.errors)[
0]
UnicodeDecodeError: 'ascii' codec can't decode byte
0xe2 in position 219: ordinal not in range(128)
```

Less common **Exceptions**

Did your program throw an **Exception** not listed here?

Look it up at:

<https://docs.python.org/3/library/exceptions.html>

Exceptions = relatively easy to fix

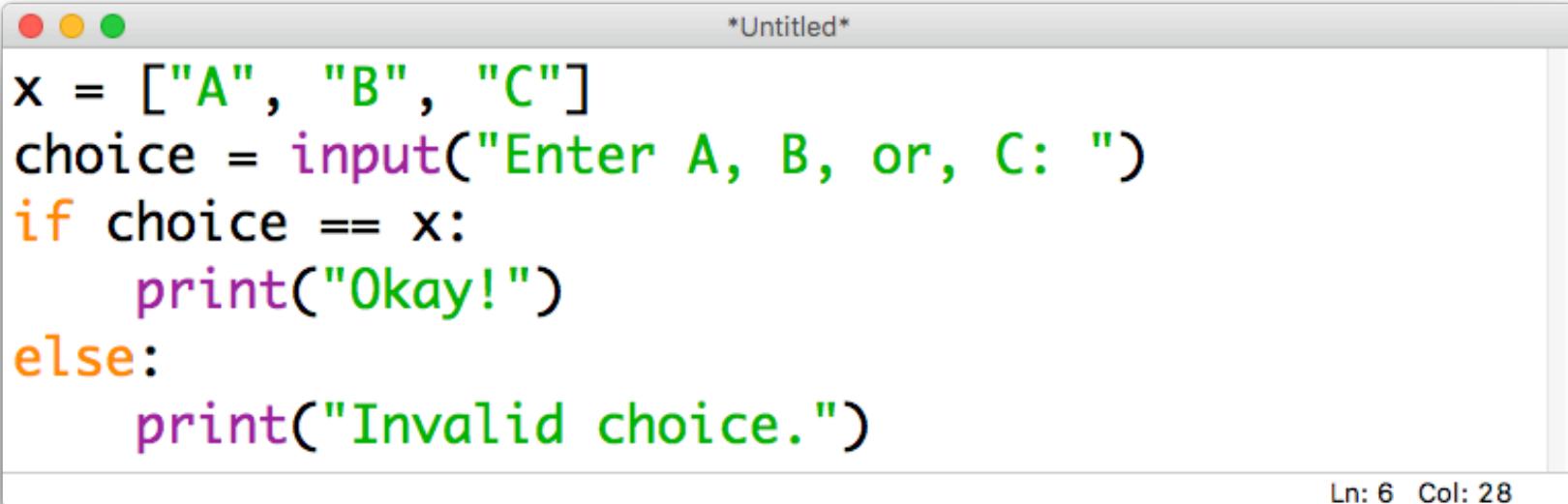
Why would I say that?

What's the alternative?



Logical errors

- Mistakes in the **reasoning** behind the code (though the statements are valid and there are no Exceptions), e.g.



The screenshot shows a window titled "Untitled*" containing Python code. The code defines a list `x` with elements "A", "B", and "C". It then prompts the user for a choice and compares it with the list `x`. If the choice matches an element in `x`, it prints "Okay!". Otherwise, it prints "Invalid choice.". The code is as follows:

```
x = ["A", "B", "C"]
choice = input("Enter A, B, or, C: ")
if choice == x:
    print("Okay!")
else:
    print("Invalid choice.")
```

In the status bar at the bottom right, it says "Ln: 6 Col: 28".



perfectly **valid**
(just not what we wanted)

Logical errors

- Mistakes in the **reasoning** behind the code (though the statements are valid and there are no Exceptions), e.g.



```
*Untitled*
```

```
x = ["A", "B", "C"]
choice = input("Enter A, B, or, C: ")
if choice in x:
    print("Okay!")
else:
    print("Invalid choice.")
```

Ln: 6 Col: 28

what we were
actually going for

An analogy

Syntactic Error

Their is no
reason to be
concerned.

Logical Error

If an animal is
green, it must
be a frog.

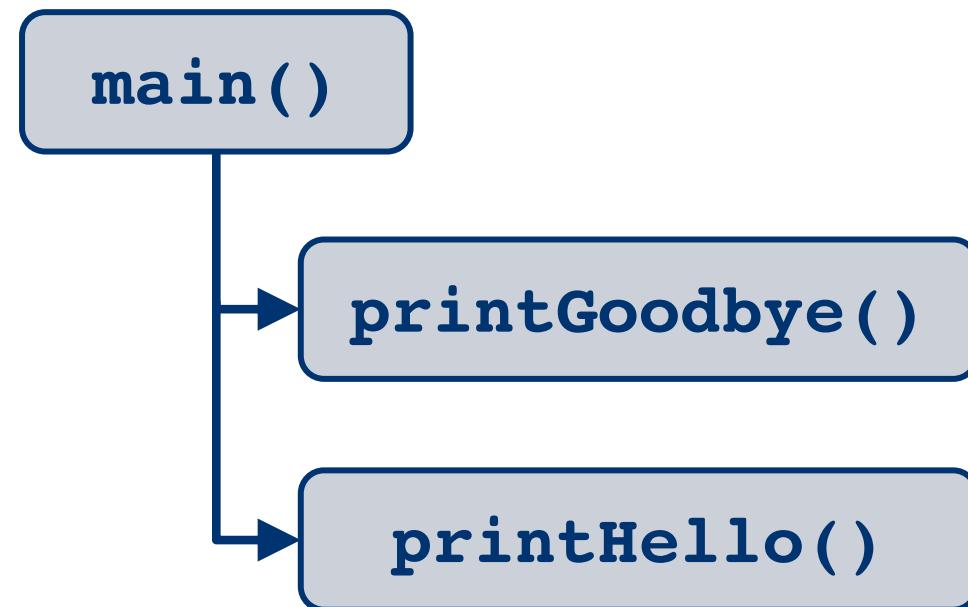
Discussion

How do you find and fix **logical** errors?



Step 1: map out the code

- It is impossible to debug code that you **don't understand** (and it's possible to not understand code even if **you wrote it!**)
- It's often helpful to map out how the code fits together:



Step 2: “rubber ducking”

- Still stuck? Try explaining it to someone else (or historically, to a rubber duckie)
- This is the debugging equivalent of **pair programming**

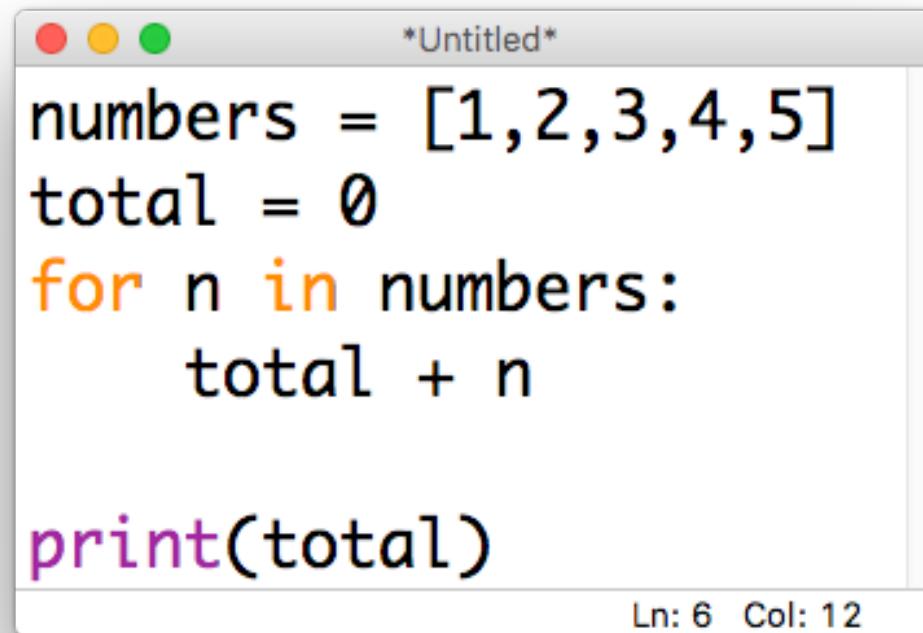
“Okay, so first we
are going to `round()` the
user’s input and then
...oh wait...

I think maybe the problem is
that I forgot to `eval()` the
input first, so it’s
still a string!



Step 3: add `print()` statements

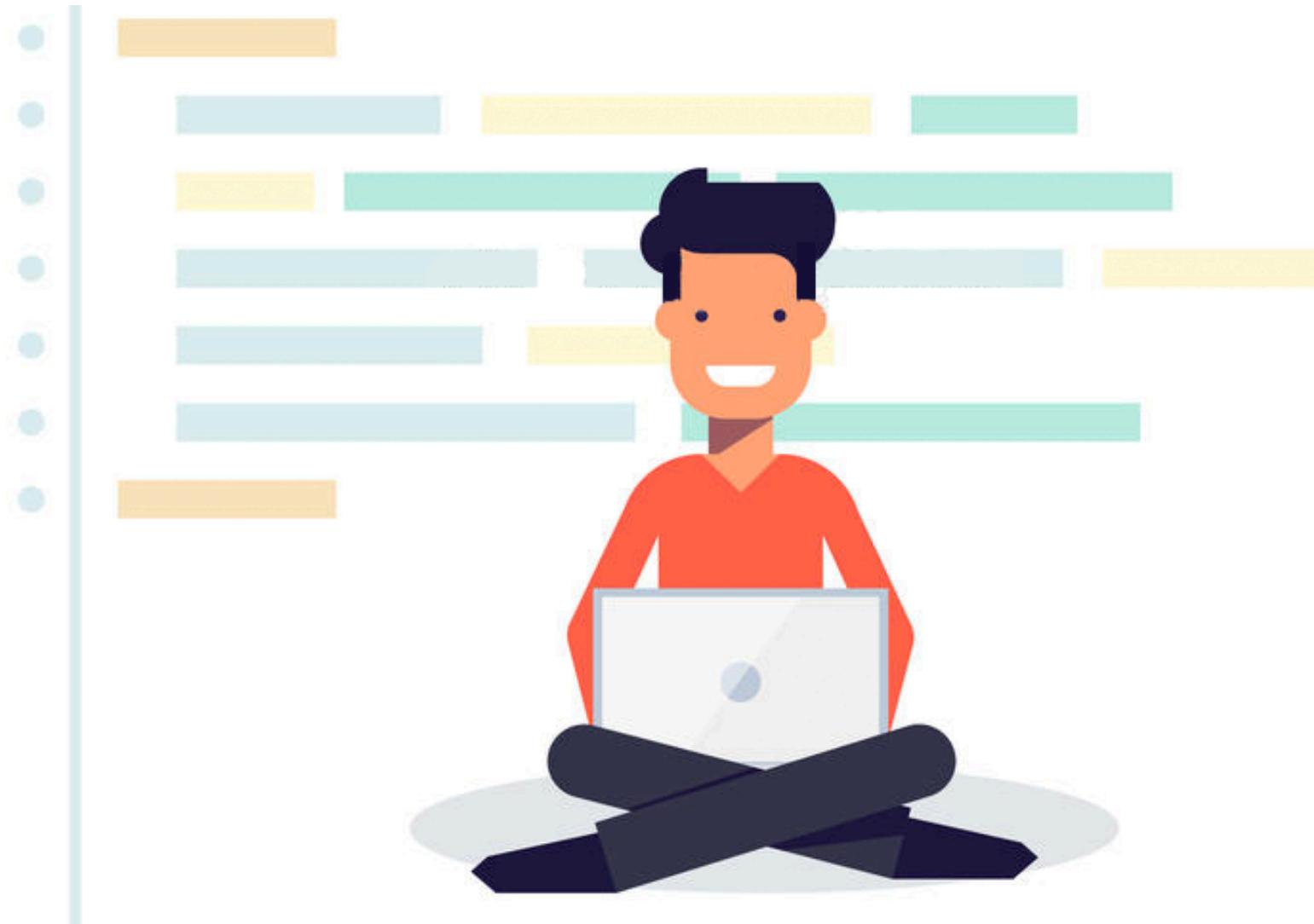
- Not sure exactly where things are going wrong (esp. inside a loop)?
- Add `print()` statements to leave a “trail” on the console



```
*Untitled*
numbers = [1,2,3,4,5]
total = 0
for n in numbers:
    total + n

print(total)
Ln: 6 Col: 12
```

In the beginning...



Now...



Eventually...

A screenshot of a GitHub repository page for 'github/gitignore'. The page shows a collection of useful .gitignore templates. Key statistics include 2,852 commits, 6 branches, 0 releases, 969 contributors, and CC0-1.0 licensing. The 'Fork' button in the top right corner is highlighted with a yellow box. The repository has 2,596 watchers, 69,757 stars, and 32,086 forks.

github / gitignore

2,852 commits | 6 branches | 0 releases | 969 contributors | CC0-1.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

File	Description	Time Ago
.github	Trim trailing whitespace	a month ago
Global	Merge pull request #2738 from jayvdb/patch-artifacts	7 days ago
.travis.yml	stub a placeholder Travis config	24 days ago
Actionscript.gitignore	Fix comments on same line causing ignore to break	11 months ago

The point

- **Other people** need to be able to understand your code
- **Future you** needs to be able to understand your code
- **Document it**

... but how?

Step 4: meaningful nouns for variables

```
*Untitled*
```

```
x = "Jordan Crouser"
y = "Professor"
z = "Smith College"
print(x, "-", y, "at", z)
```

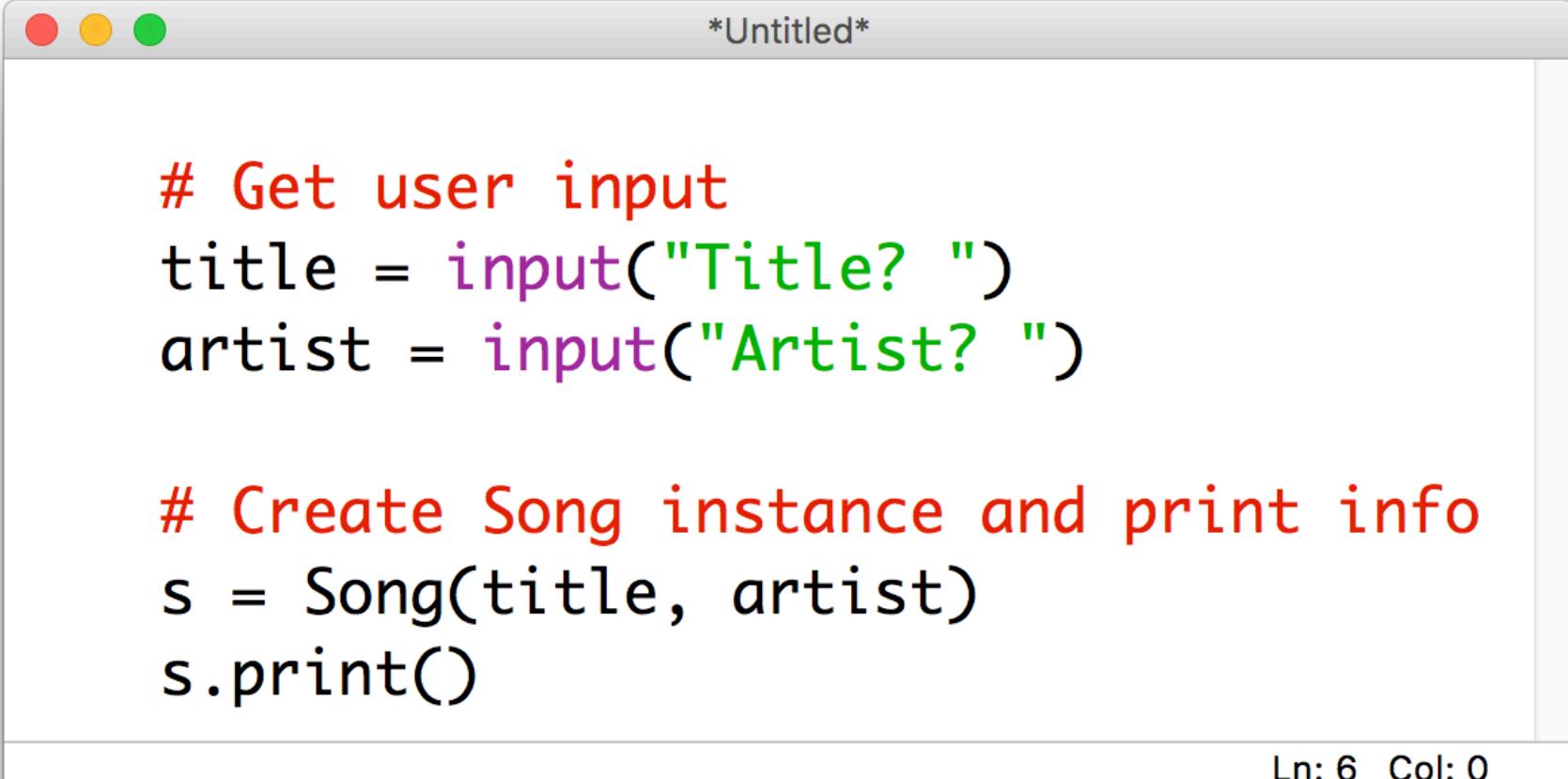
Ln: 1 Col: 1

```
*Untitled*
```

```
name = "Jordan Crouser"
title = "Professor"
institution = "Smith College"
print(name, "-", title, "at", institution)
```

Ln: 4 Col: 41

Step 5: lots of comments



```
# Get user input
title = input("Title? ")
artist = input("Artist? ")

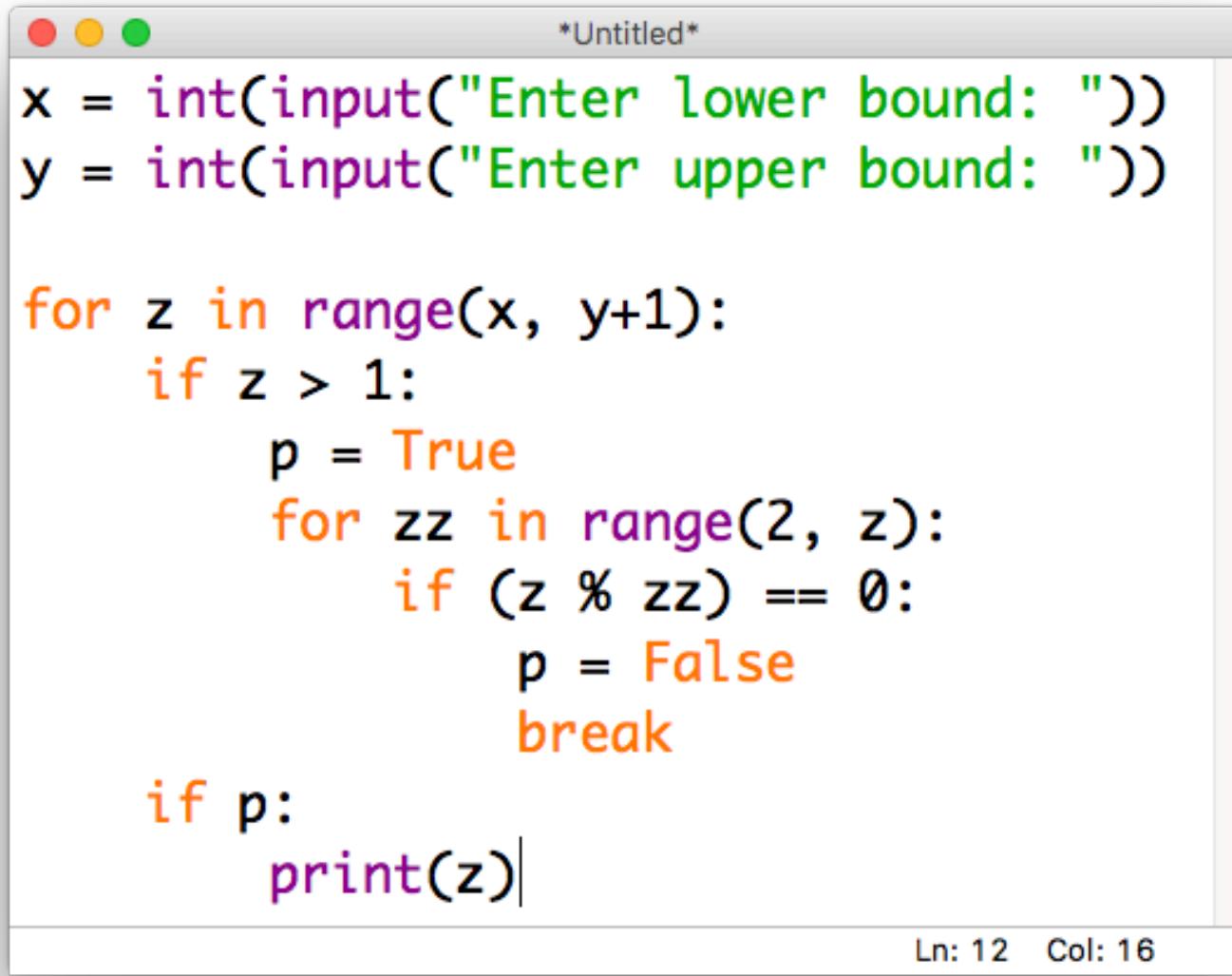
# Create Song instance and print info
s = Song(title, artist)
s.print()
```

Ln: 6 Col: 0

A useful technique: code tracing

- **Given:** a very short, poorly-documented program
- **Your goal:** try to figure out what it's doing
- **Recommendations:**
 - walk through the program step-by-step (“**trace its execution**”) using the **whiteboard or paper** instead of typing it into a repl
 - once you understand what’s happening, then rewrite it using **informative variable names** and **comments**

Example



```
*Untitled*
x = int(input("Enter lower bound: "))
y = int(input("Enter upper bound: "))

for z in range(x, y+1):
    if z > 1:
        p = True
        for zz in range(2, z):
            if (z % zz) == 0:
                p = False
                break
        if p:
            print(z)

Ln: 12 Col: 16
```

Takeaways

- This is a crash course in **basic** debugging / documentation
- There are **lots** of other techniques for both dealing with and **preventing** bugs, but for now this will suffice
- The most important part is to understand:
 - what the code is **trying** to do
 - what the code is **actually** doing
- Tips:
 - change **one thing** at a time
 - **keep track** of what you change!

Demo



Your task

```
1 # -----
2 #      Names: <YOUR NAMES HERE>
3 #      Filename: connectFour-broken.py
4 #      Date: <TODAY'S DATE HERE>
5 #
6 # Description: This file contains a broken version
7 #                  of Jordan's ConnectFour game.
8 #
9 # There are 5 SYNTACTIC ERRORS (mistakes
10 # that are not correct Python statements
11 # and so cause the program to throw
12 # Exceptions) as well as 5 LOGICAL ERRORS
13 # (mistakes that are technically correct
14 # Python statements, but which cause the
15 # program not to behave the way we want).
16 #
17 # Your job is to find (and correct!) each
18 # of these mistakes using your new
19 # DEBUGGING TECHNIQUES.
20 # -----
```

Discussion

What did you find?



Up next

- ✓ Monday: Loops
 - ✓ `for...in` (looping through items in a list)
 - ✓ the `range()` function (getting a list of numbers)
 - ✓ `while` (looping until something happens)
- ✓ Lab: Loops
- ✓ Wednesday: Life Skill #2/3: Debugging & Documentation
- A3: Magic 8 Ball due Thursday at 11:55pm EST
- Friday: Dictionaries and Sets