

Intro to Coding with Python– Documentation and Debugging

Dr. Ab Mosca (they/them)

Plan for Today

- Documenting code
- Tracing code
- Debugging

Big Picture

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

The point

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

... but how?

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 1:
meaningful
nouns for
variables



```
temp.py
UNREGISTERED

temp.py
1 x = "Ab Mosca"
2 y = "Professor"
3 z = "Westfield State"
4
5 print(x, "-", y, "at", z)
```

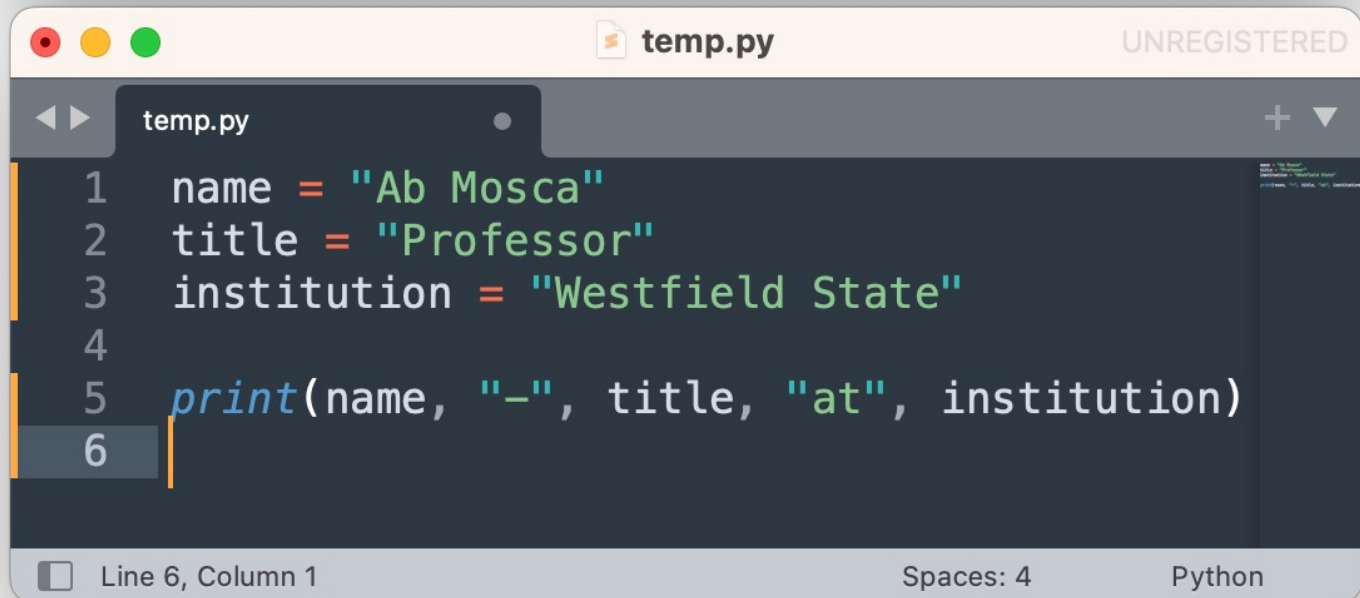
Line 5, Column 26 Spaces: 4 Python

Step 1:
meaningful
nouns for
variables



```
temp.py UNREGISTERED
temp.py x
1 x = "Ab Mosca"
2 y = "Professor"
3 z = "Westfield State"
4
5 print(x, "-", y, "at", z)
```

Line 5, Column 26 Spaces: 4 Python



```
temp.py UNREGISTERED
temp.py
1 name = "Ab Mosca"
2 title = "Professor"
3 institution = "Westfield State"
4
5 print(name, "-", title, "at", institution)
6
```

Line 6, Column 1 Spaces: 4 Python

Step 2: lots of
comments

```
*Untitled*  
def makeSong():  
    # 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 running lines
 - 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

Step 3*:
describe the
action for
functions



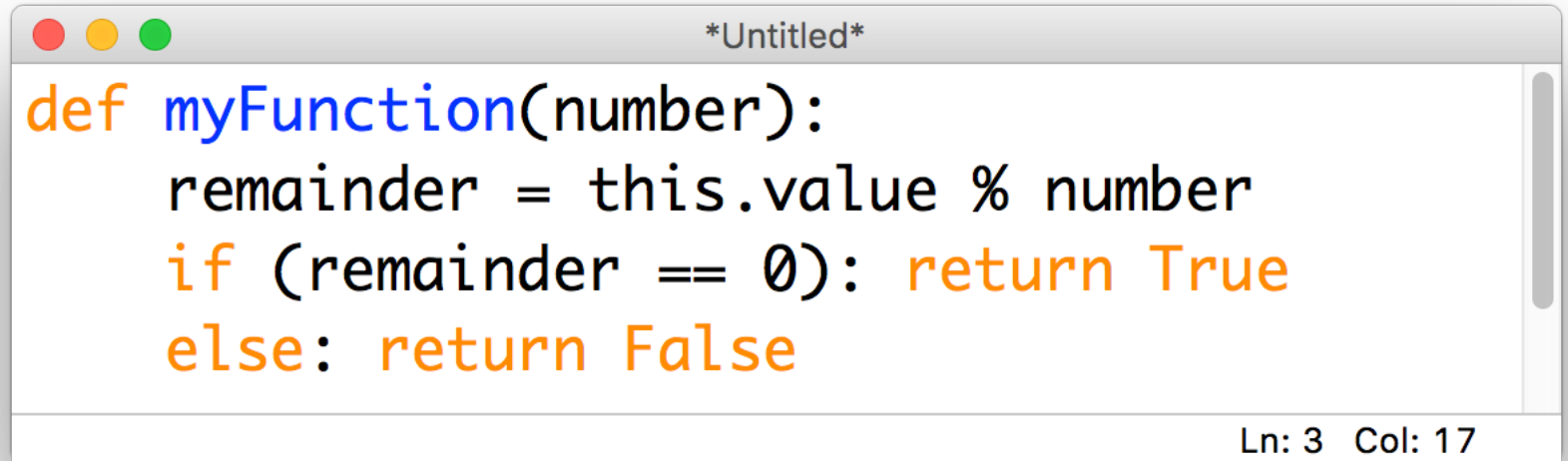
```
*Untitled*  
def bloop(x):  
    return x.title().replace("'S", "'s")  
Ln: 1 Col: 9
```

Step 3*:
describe the
action for
functions

```
*Untitled*  
def bloop(x):  
    return x.title().replace("'S", "'s")  
Ln: 1 Col: 9
```

```
*Untitled*  
def capitalizeWords(x):  
    return x.title().replace("'S", "'s")  
Ln: 2 Col: 40
```

Step 3*:
describe the
action for
functions



```
def myFunction(number):  
    remainder = this.value % number  
    if (remainder == 0): return True  
    else: return False
```

Ln: 3 Col: 17

Step 3*:
describe the
action for
functions

```
*Untitled*  
def myFunction(number):  
    remainder = this.value % number  
    if (remainder == 0): return True  
    else: return False  
Ln: 3 Col: 17
```

```
*Untitled*  
def isDivisibleBy(number):  
    remainder = this.value % number  
    if (remainder == 0): return True  
    else: return False  
Ln: 4 Col: 22
```

Step 4*: docstrings

```
*Untitled*
def makeSong():
    """ Creates and prints a Song instance
        from user input """
    # Get user input
    title = input("Title? ")
    artist = input("Artist? ")

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

Ln: 5 Col: 0

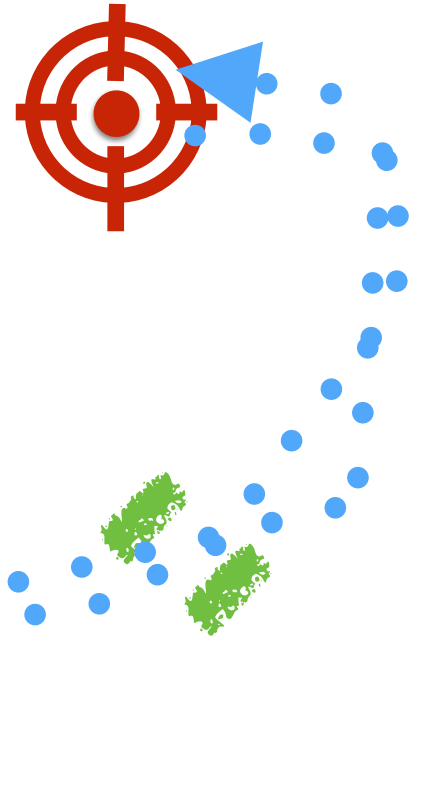


Debugging

RECAP: the programming process

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

iterate many times



Fun history:
the term
"debug"

9/9


0800 Antcom started
1000 " stopped - antcom ✓

1300 (032) MP - MC ~~1.982647000~~ 2.130476415 (3) 4.615925059 (-2)
(033) PRO 2 2.130476415
conck 2.130676415

Relays 6-2 in 033 failed special speed test
in relay " 11.000 test -

Relays changed

1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

1545  Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 Antcom started.
1700 closed down.

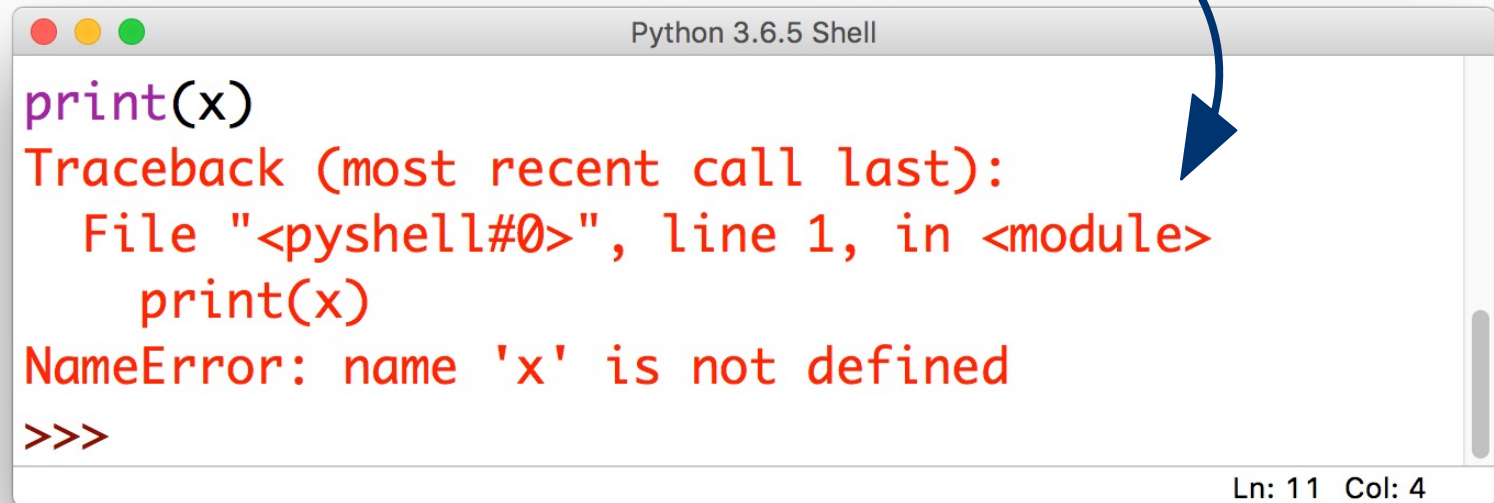


RDML Grace M. Hopper

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

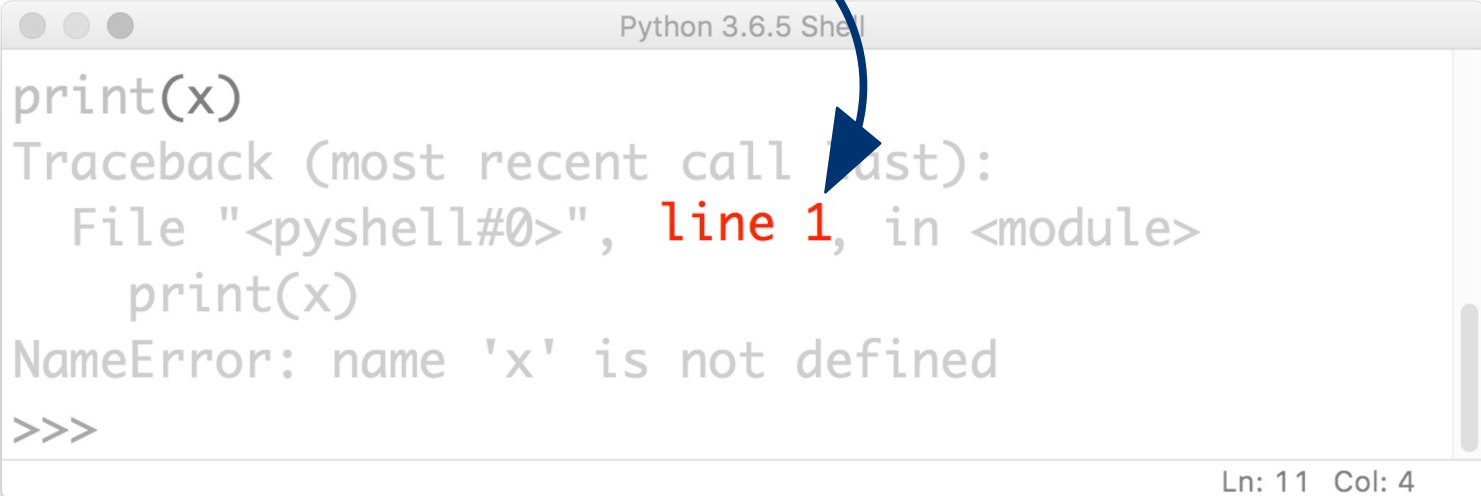
```
Python 3.6.5 Shell
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Ln: 11 Col: 4

this 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!)

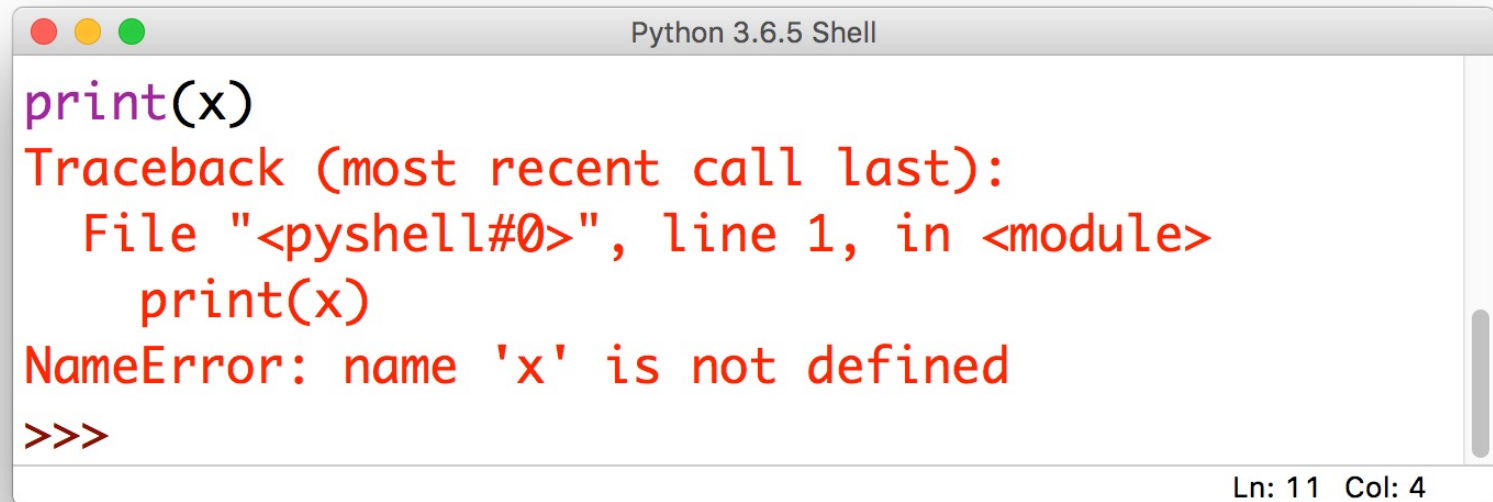


```
Python 3.6.5 Shell  
print(x)  
Traceback (most recent call last):  
  File "<pyshell#0>", line 1, in <module>  
    print(x)  
NameError: name 'x' is not defined  
>>>
```

Ln: 11 Col: 4

Common Exceptions

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

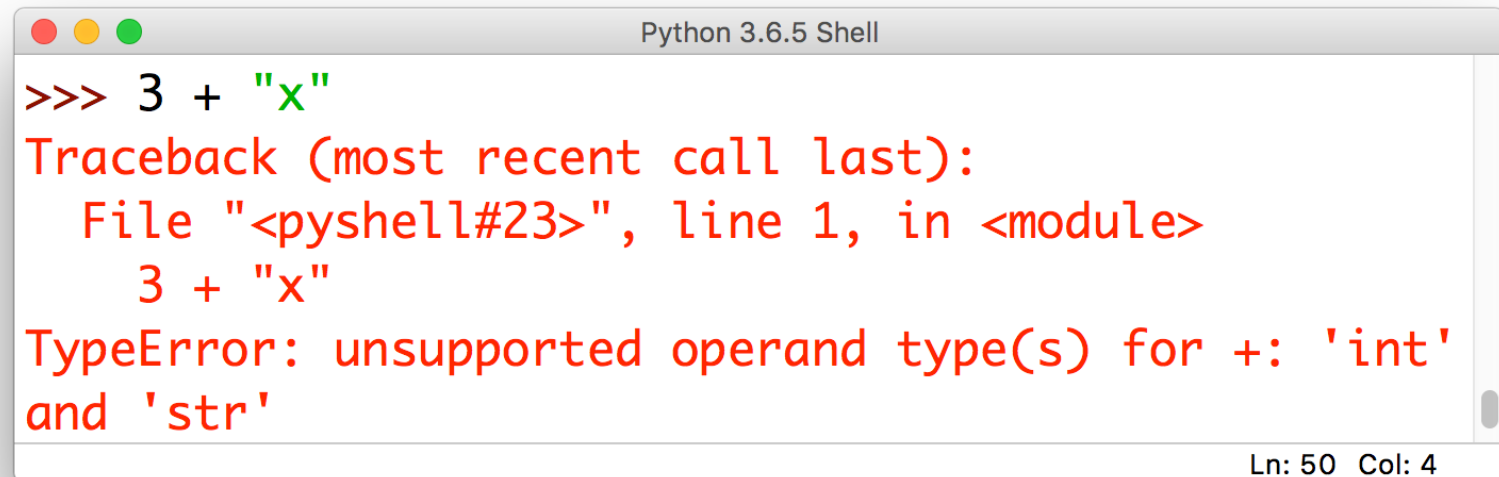


```
Python 3.6.5 Shell
print(x)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
>>>
```

Ln: 11 Col: 4

Common Exceptions

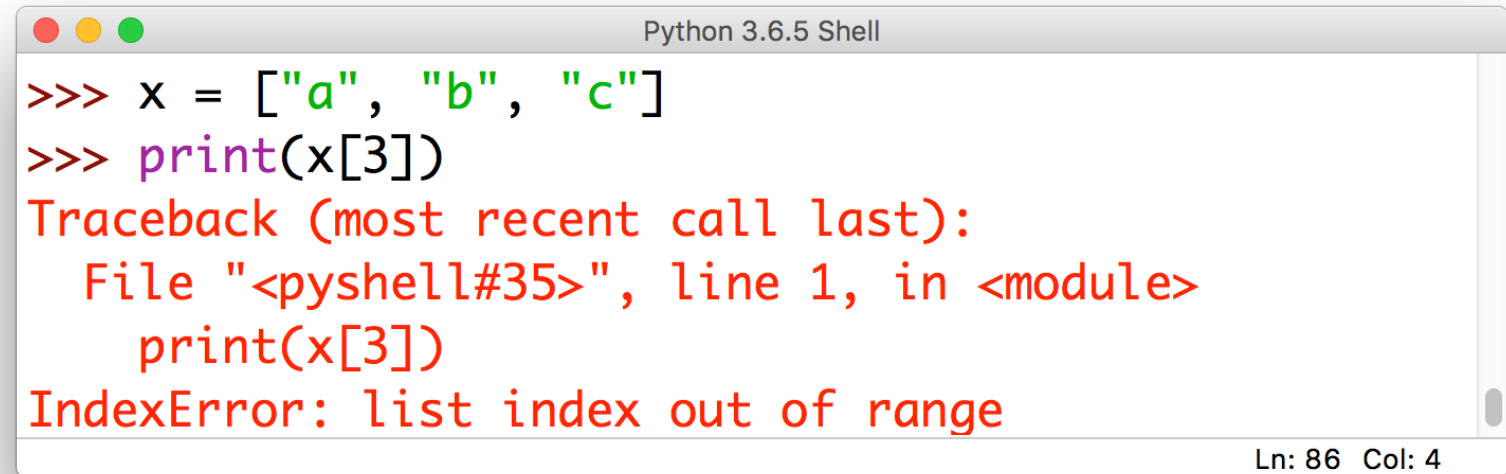
- **TypeError**: raised when you try to perform an operation on an object that's not the right type (i.e. a `string` instead of a number)



```
Python 3.6.5 Shell
>>> 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'
Ln: 50 Col: 4
```

Common Exceptions

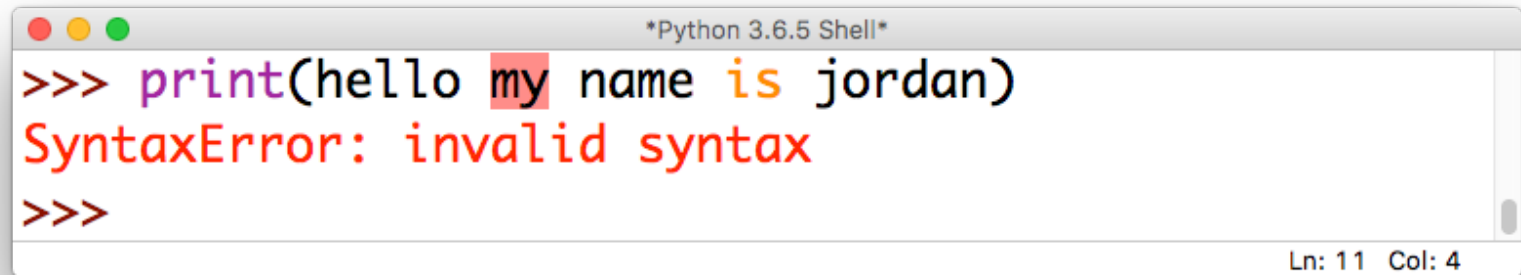
- **IndexError**: raised when you try to use an index that's out of bounds



```
Python 3.6.5 Shell
>>> 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
Ln: 86 Col: 4
```


Common Exceptions

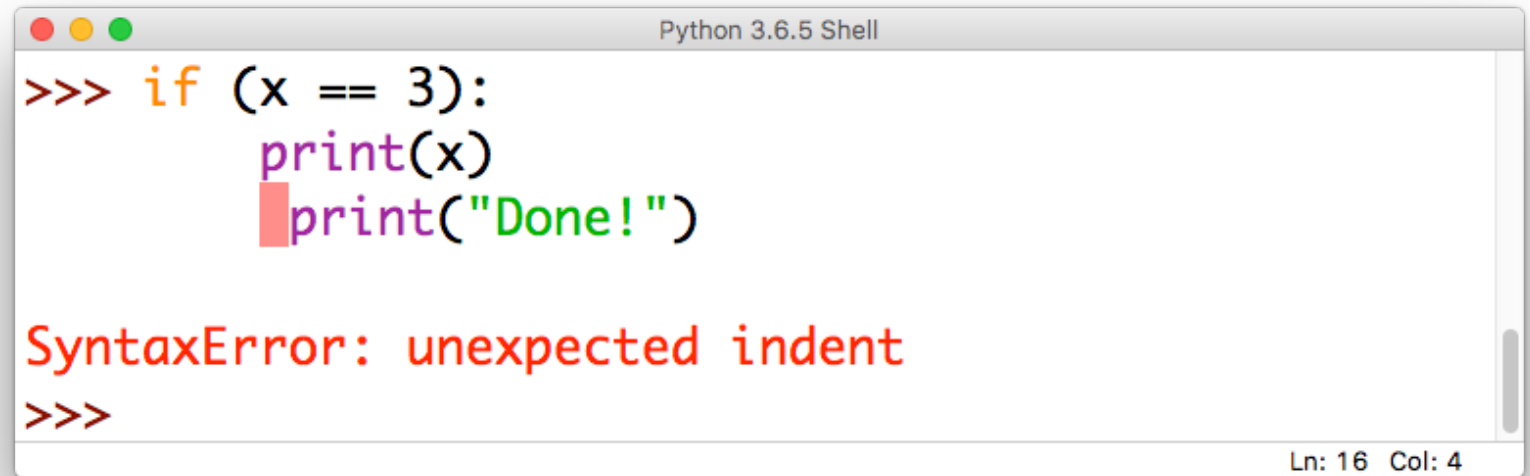
- **SyntaxError**: raised when you try to run a command that isn't a valid Python statement

A screenshot of a Python 3.6.5 Shell window. The window title is "*Python 3.6.5 Shell*". The prompt is ">>>". The user has entered the code "print(hello my name is jordan)", where "my" is highlighted in red. The shell has responded with "SyntaxError: invalid syntax" in red text. The prompt ">>>" is shown again on the next line. In the bottom right corner, the status bar shows "Ln: 11 Col: 4".

```
*Python 3.6.5 Shell*
>>> print(hello my name is jordan)
SyntaxError: invalid syntax
>>>
```

Common Exceptions

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



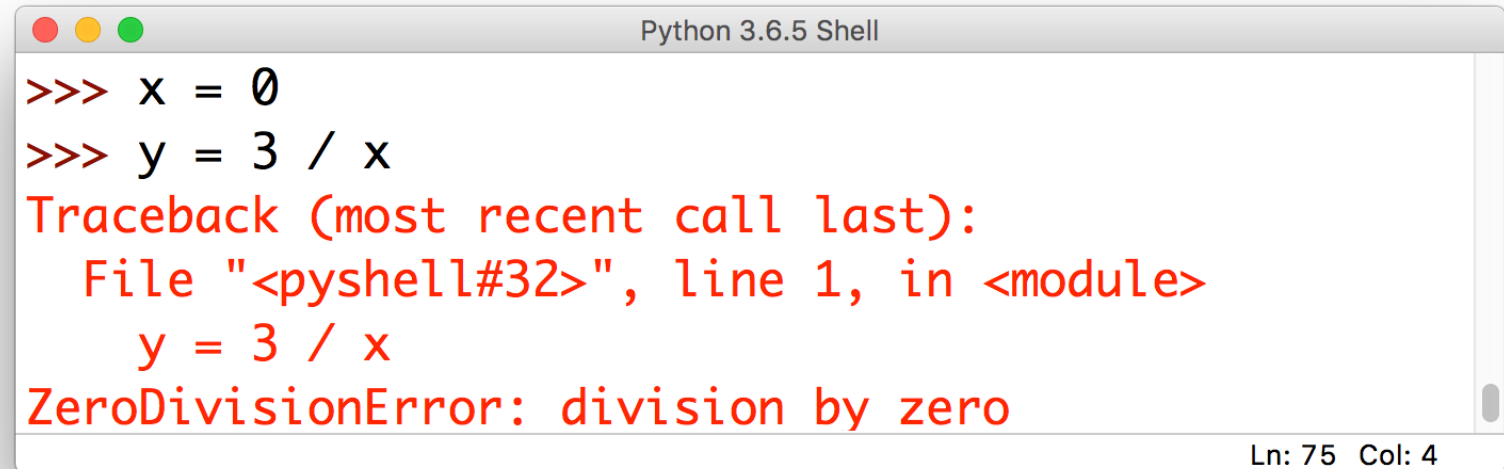
```
Python 3.6.5 Shell
>>> if (x == 3):
      print(x)
      print("Done!")

SyntaxError: unexpected indent
>>>
```

Ln: 16 Col: 4

Common Exceptions

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



```
Python 3.6.5 Shell
>>> x = 0
>>> y = 3 / x
Traceback (most recent call last):
  File "<pyshell#32>", line 1, in <module>
    y = 3 / x
ZeroDivisionError: division by zero
Ln: 75 Col: 4
```

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.

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

There is no
reason to be
concerned.

Logical Error

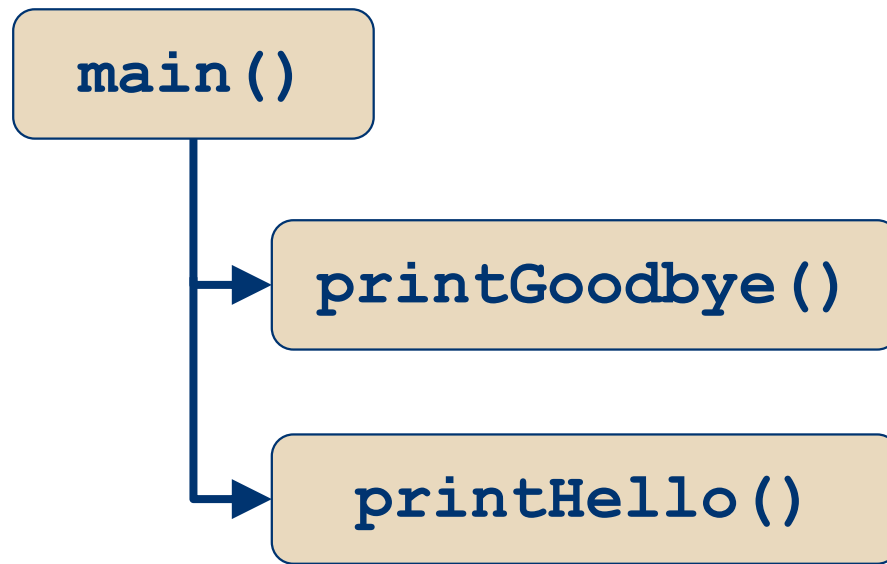
If an animal is
green, it must
be a frog.

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”

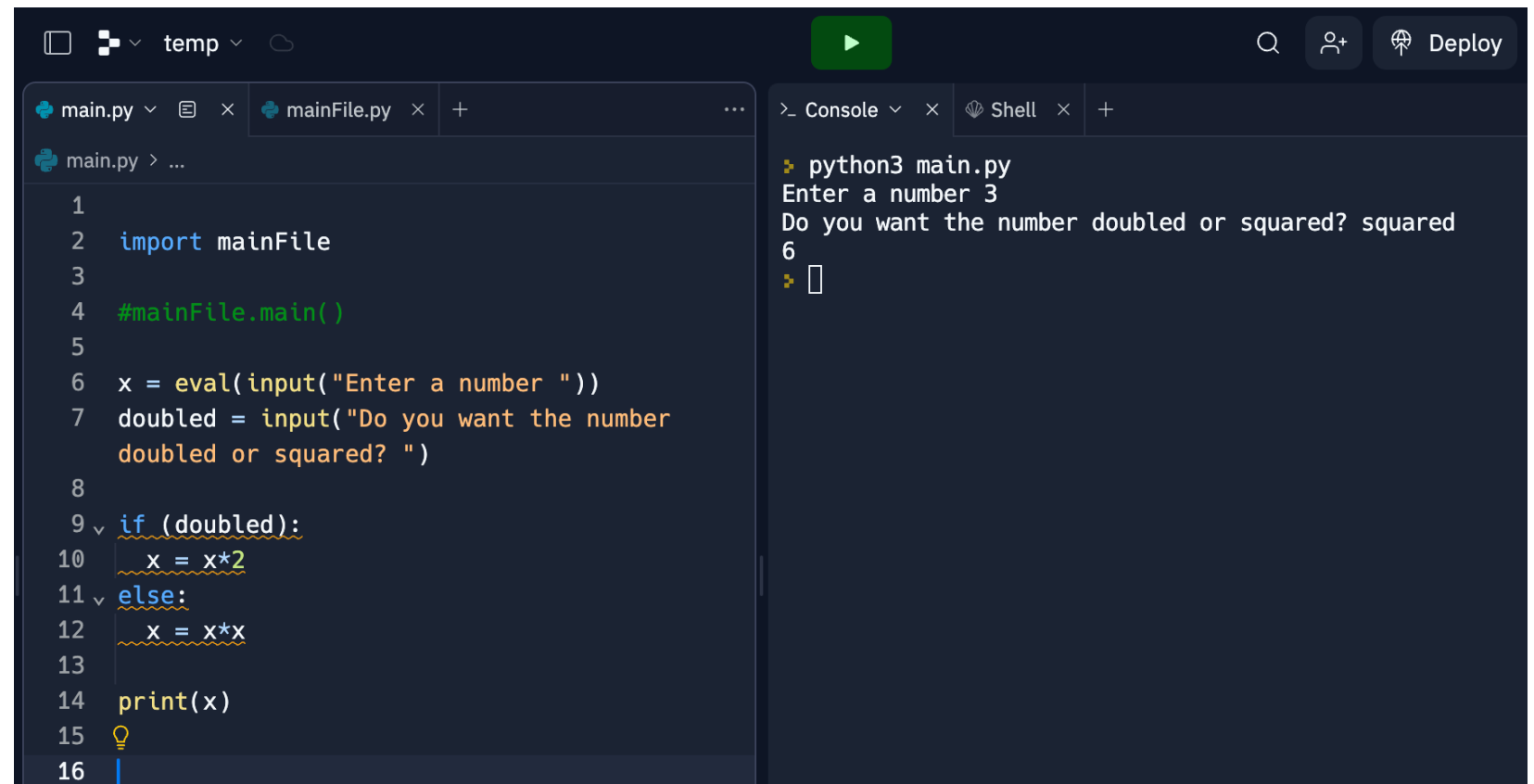
- Still stuck? Try explaining it to someone else (or historically, to a rubber duckie)
- This is the debugging equivalent of **pair programming**

“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?
- Add `print()` statements to leave a “trail” on the console



The screenshot shows a code editor with two tabs: `main.py` and `mainFile.py`. The `main.py` tab is active, displaying the following Python code:

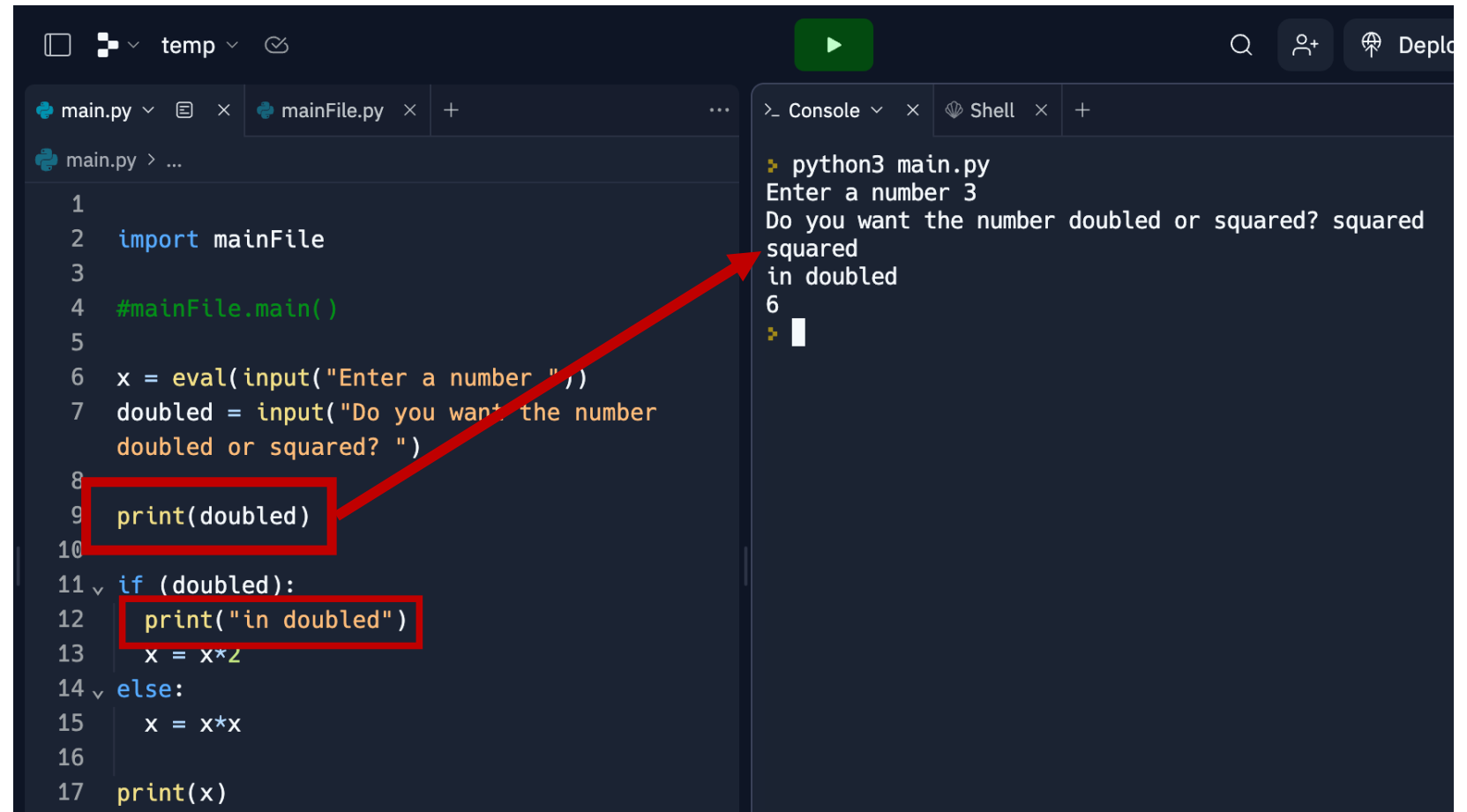
```
1
2 import mainFile
3
4 #mainFile.main()
5
6 x = eval(input("Enter a number "))
7 doubled = input("Do you want the number
8 doubled or squared? ")
9
10 if (doubled):
11     x = x*2
12 else:
13     x = x*x
14 print(x)
15
16
```

The right-hand side of the editor shows the console output for the command `python3 main.py`:

```
> python3 main.py
Enter a number 3
Do you want the number doubled or squared? squared
6
```

Step 3: add `print()` statements

- Not sure exactly where things are going wrong?
- Add `print()` statements to leave a "trail" on the console



The screenshot shows a code editor with two files: `main.py` and `mainFile.py`. The `main.py` file contains the following code:

```
1
2 import mainFile
3
4 #mainFile.main()
5
6 x = eval(input("Enter a number "))
7 doubled = input("Do you want the number
8 doubled or squared? ")
9 print(doubled)
10
11 if (doubled):
12     print("in doubled")
13     x = x*2
14 else:
15     x = x*x
16
17 print(x)
```

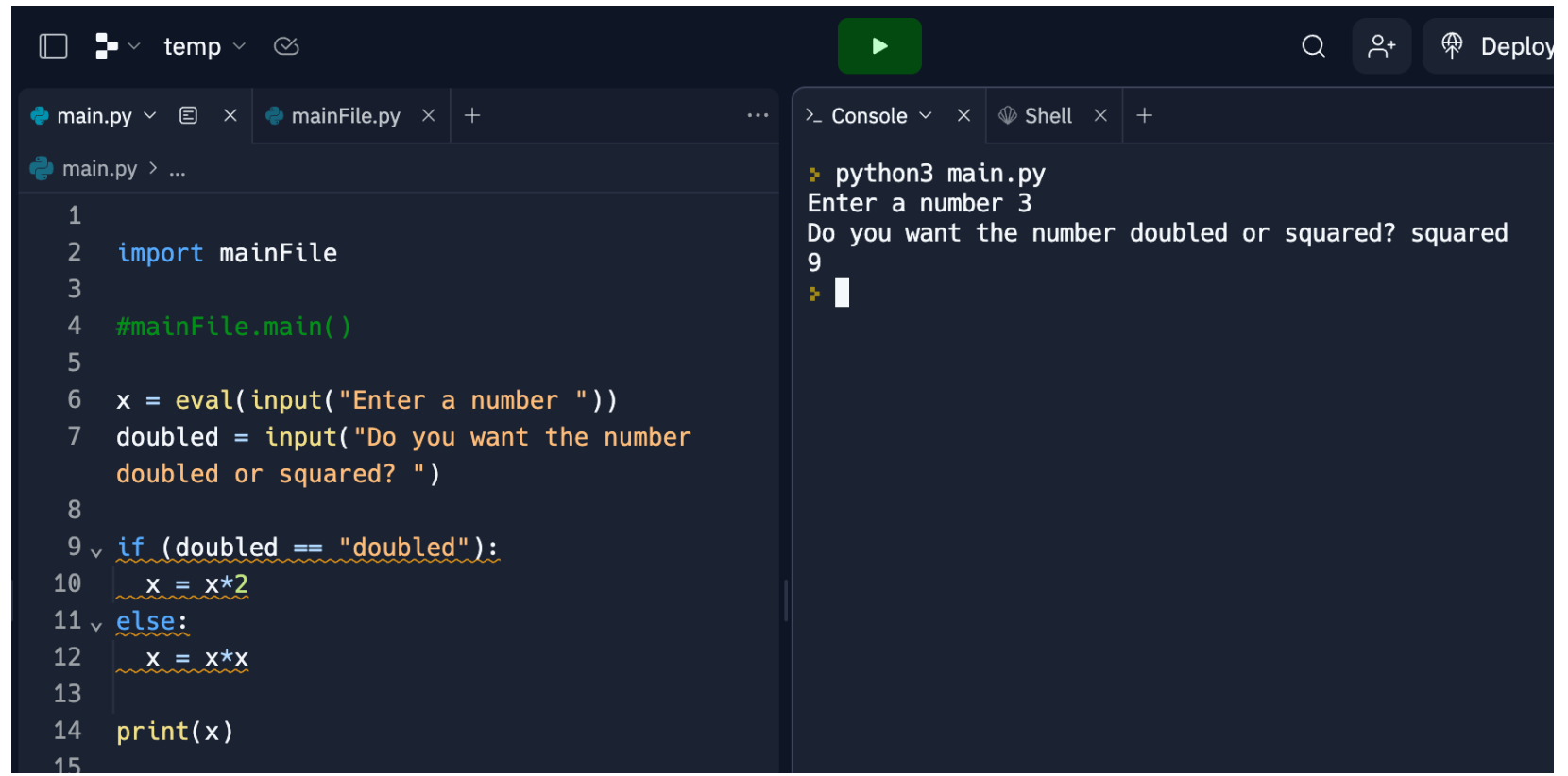
The console output shows the execution of `python3 main.py` with the following input and output:

```
> python3 main.py
Enter a number 3
Do you want the number doubled or squared? squared
in doubled
6
```

A red arrow points from the `print(doubled)` statement in the code to the `in doubled` output in the console. Another red arrow points from the `print("in doubled")` statement in the code to the `6` output in the console.

Step 3: add `print()` statements

- Not sure exactly where things are going wrong?
- Add `print()` statements to leave a "trail" on the console



```
temp
main.py mainFile.py
main.py > ...
1
2 import mainFile
3
4 #mainFile.main()
5
6 x = eval(input("Enter a number "))
7 doubled = input("Do you want the number
  doubled or squared? ")
8
9 if (doubled == "doubled"):
10     x = x*2
11 else:
12     x = x*x
13
14 print(x)
15

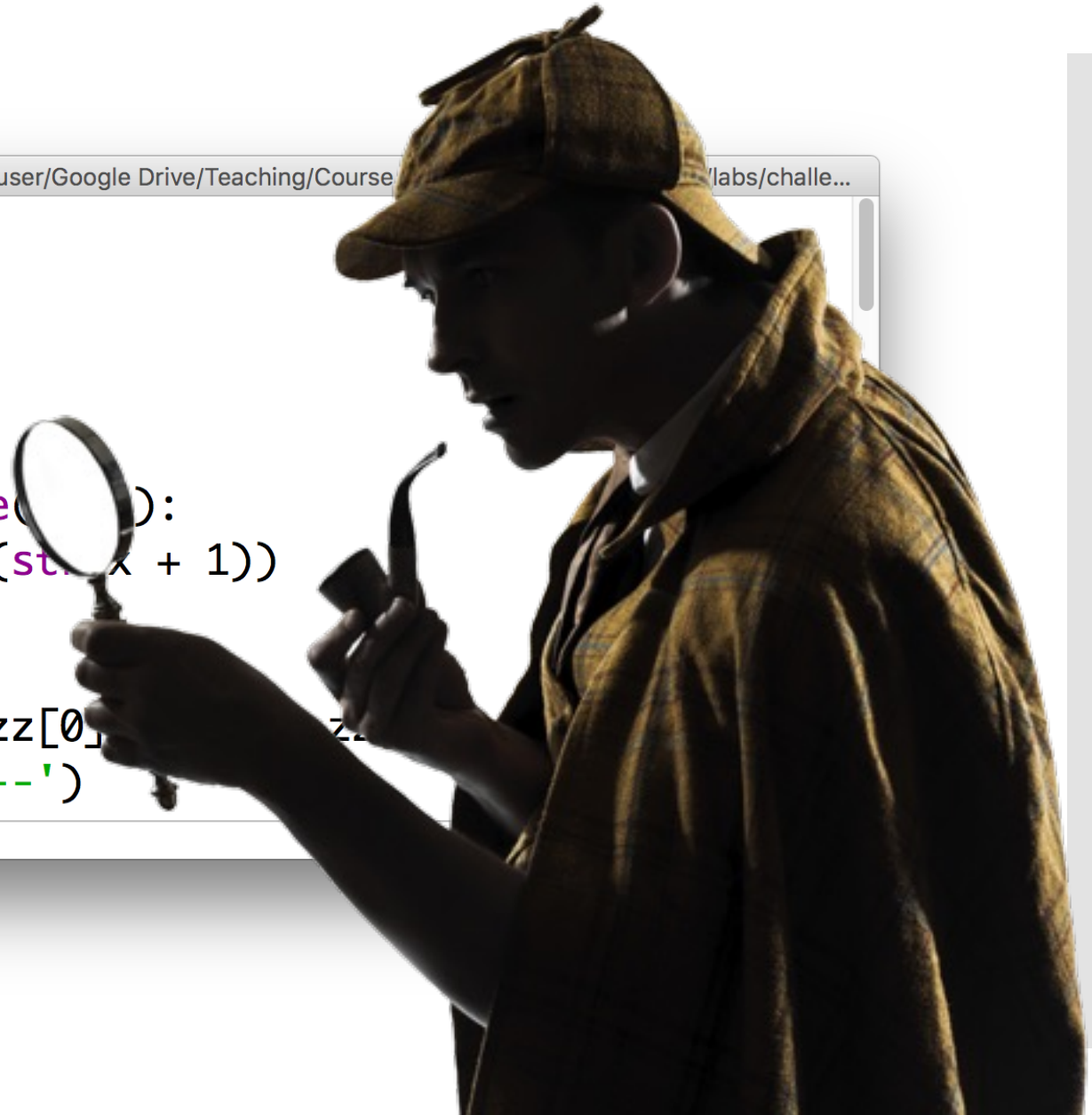
python3 main.py
Enter a number 3
Do you want the number doubled or squared? squared
9
```

Takeaways

- There are **lots** of other techniques for both dealing with and **preventing** bugs
- 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!

ic03

Activity: "code detective"



```
challenge2.py - /Users/jcrouser/Google Drive/Teaching/Course/labs/challe...
zz = []
p1go = True
w = False

def su():
    for x in range(10):
        zz.append(str(x + 1))

def pb():
    print( ' ' + zz[0] + ' ' )
    print( '---+---' )
```