CS134: Functions, Booleans, and Conditionals

CS Colloquium Today

- Almost Every Friday
- Time: **2:35pm**
- Normal Location: **TCL 123** (Wege Auditorium)
- Today: Bryan Perozzi (Google)
    - *Giving your Graph a Voice: Graph Representations and Large Language Models*

Gradescope

- Thanks for bearing with us as we figure out Gradescope
- It's the first time we're using it, so there's bound to be some issues!
  - Plus, the "Homeworks" are one of their new features, still in testing
- Telling cs134staff@williams.edu promptly is the best path to get things addressed quickly!

gradescope®
by Turnitin

Announcements & Logistics

- **Homework 2** is due Monday 10 pm
    - Ten multiple-choice questions (posted to course website)
    - Try to answer them using pencil and paper first
    - Can verify answers using interactive Python if you wish
- **Lab 2** posted today, due Wed 10pm / Thur 10pm
    - **Prelab**: warm-up pencil-and-paper exercise due at the start of lab
    - Read/think/work on the assignment ahead of your scheduled lab section
- Personal machine setup: reminder that you can (optionally) setup your machine
    - Setup instructions under Resources on Course Webpage

Do You Have Any Questions?

Last Time

Discussed **functions** in greater detail

- Reviewed the built-in functions:
    - `input()`, `print()`, `int()`, `float()`, `str()`
- Note: Some functions return an ***explicit*** value
    - `int()`, `input()`, our definition of `square()`
- Other functions "do something" but don't explicitly return
    - `print()`, user-defined functions *without* explicit return statement like `printMessage()`
    - Such functions "secretly" (or ***implicitly***) return a **None** value (more on this today!)

Today's Plan

- Write a non-trivial function together in VS Code
- Wrap up discussion of functions
  - Discuss return statements and variable scope in more detail
- Start learning about conditionals (Lab 2!)
  - Boolean data type
  - Making decisions in Python using `if else` statements

Variable Scope

Variable Scope

- **Local variables:** An assignment to a variable *within a function* definition creates/modifies a *local variable*
    - Local variables *only* exist within a function's body, and cannot be referred to outside of the function's body
- **Parameters** are also local variables that are assigned a value when the function is invoked

```
def square(num):

  return num*num
```

```
>>> square (5)
25
>>> num
NameError: name 'num' is not defined
```

Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val

val = 3
new_val = my_func(val)
print('global val', val)
```

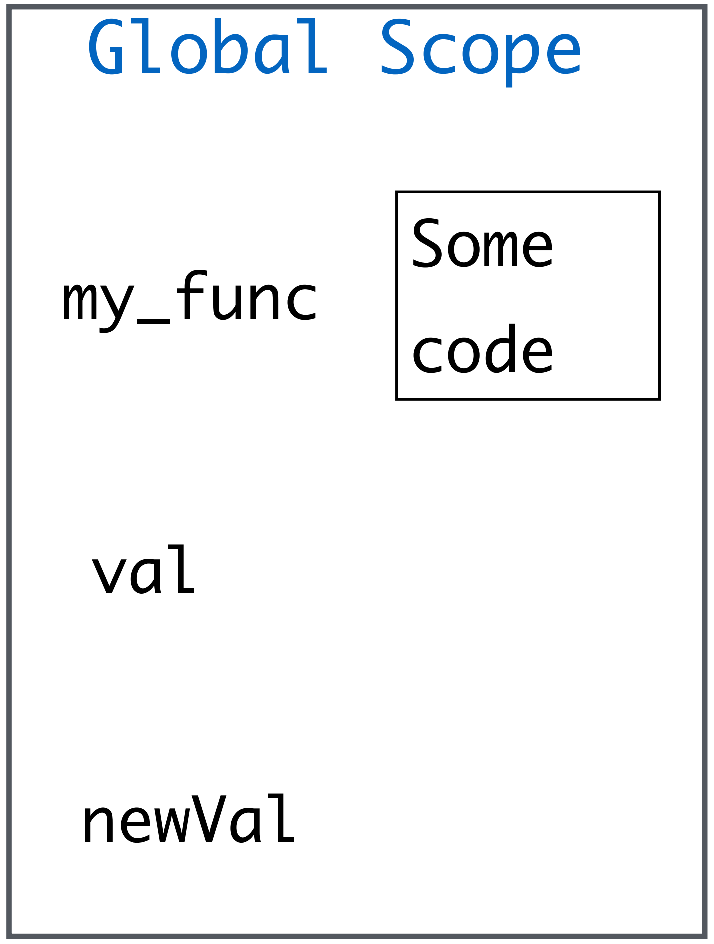
What is printed here?

What is returned?

What is printed here?

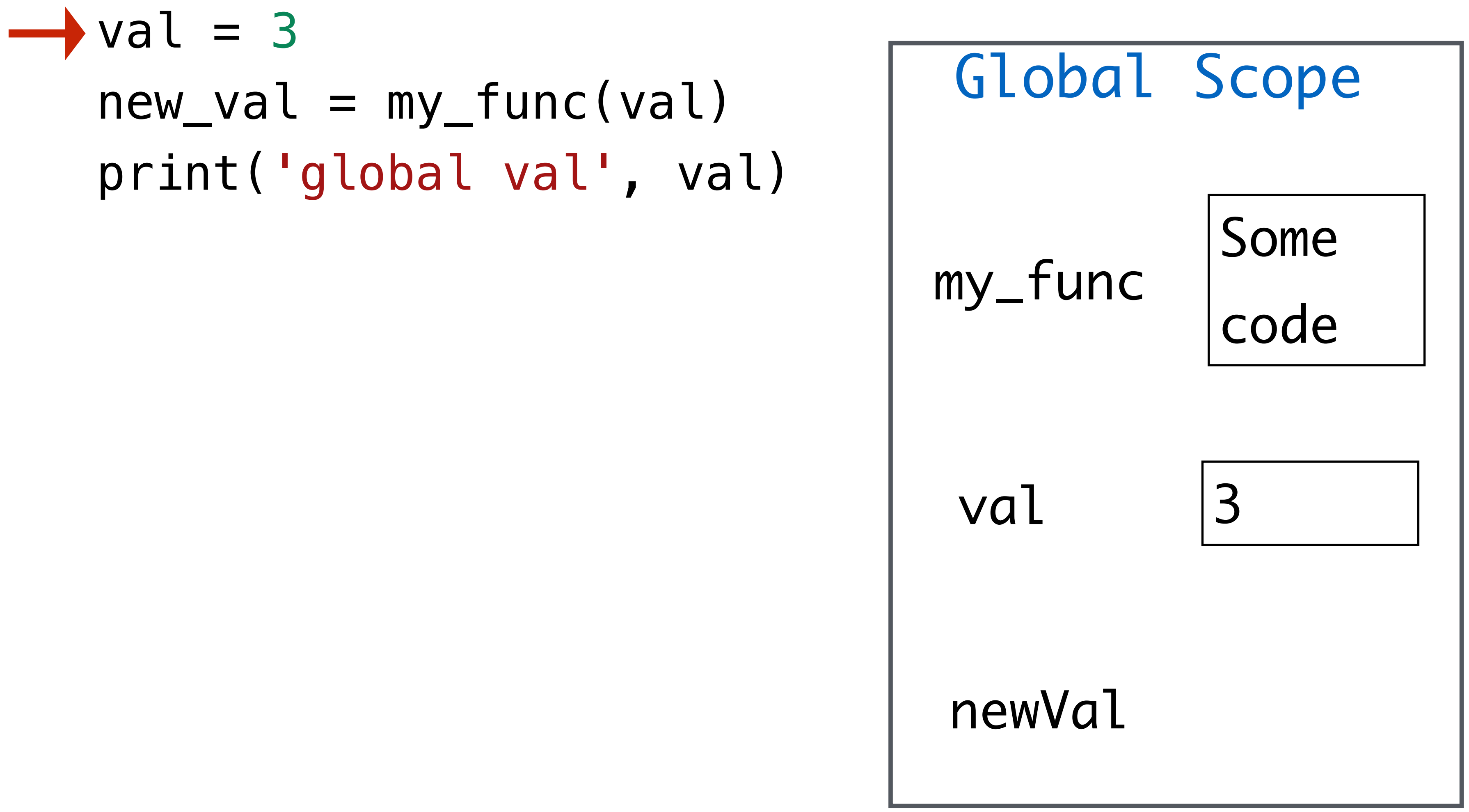
Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```



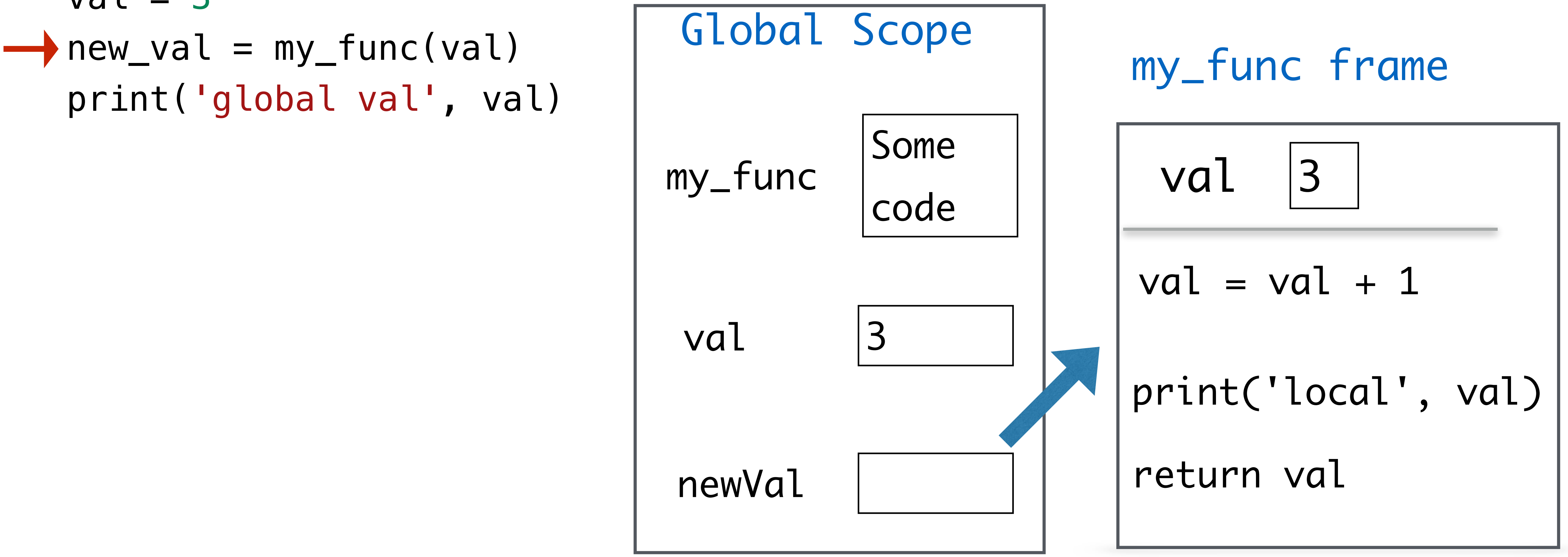
Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```



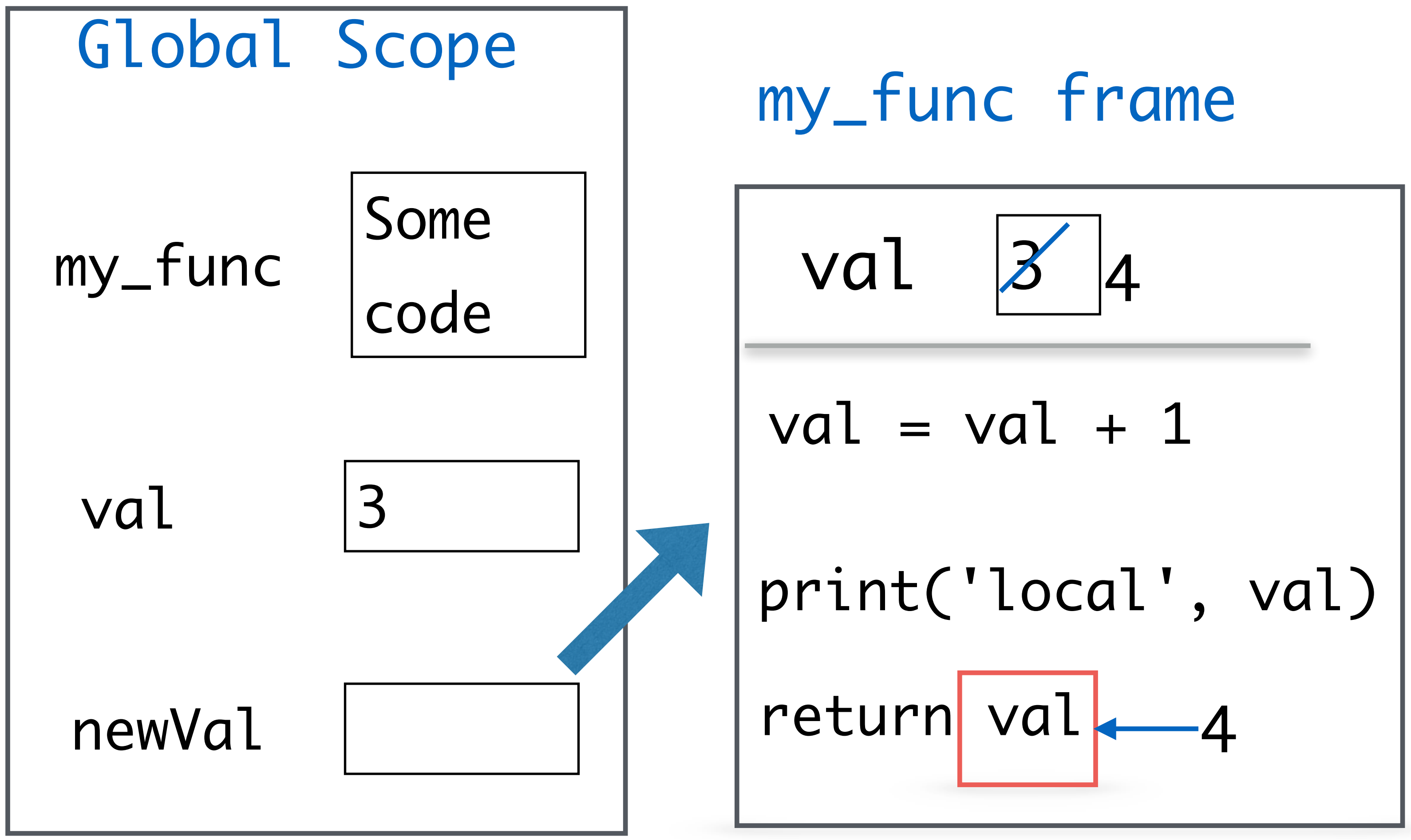
Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```



Variable Scope: A Tricky Example

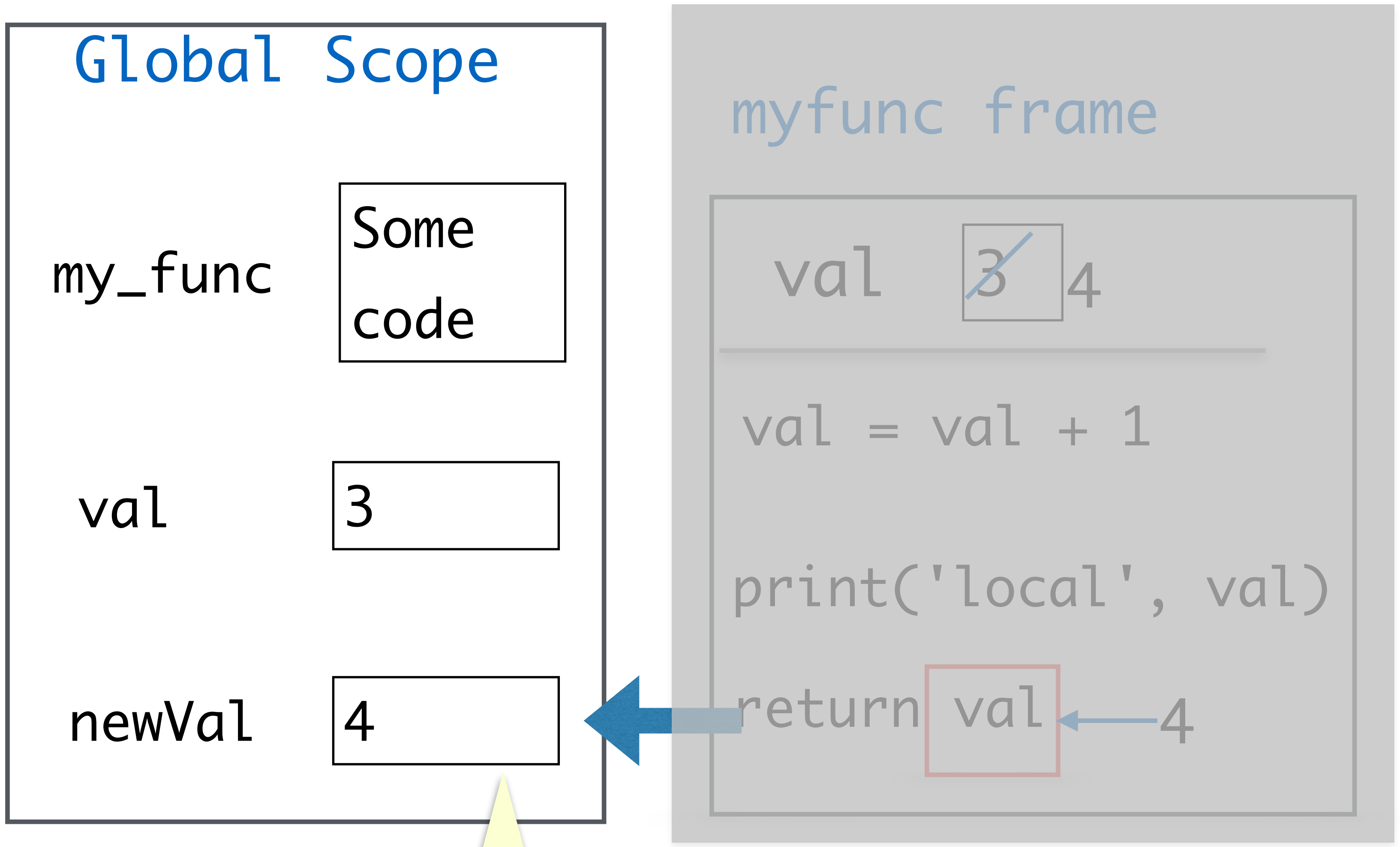
```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```



Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```

Function frame destroyed (and all local variables lost) after return from call

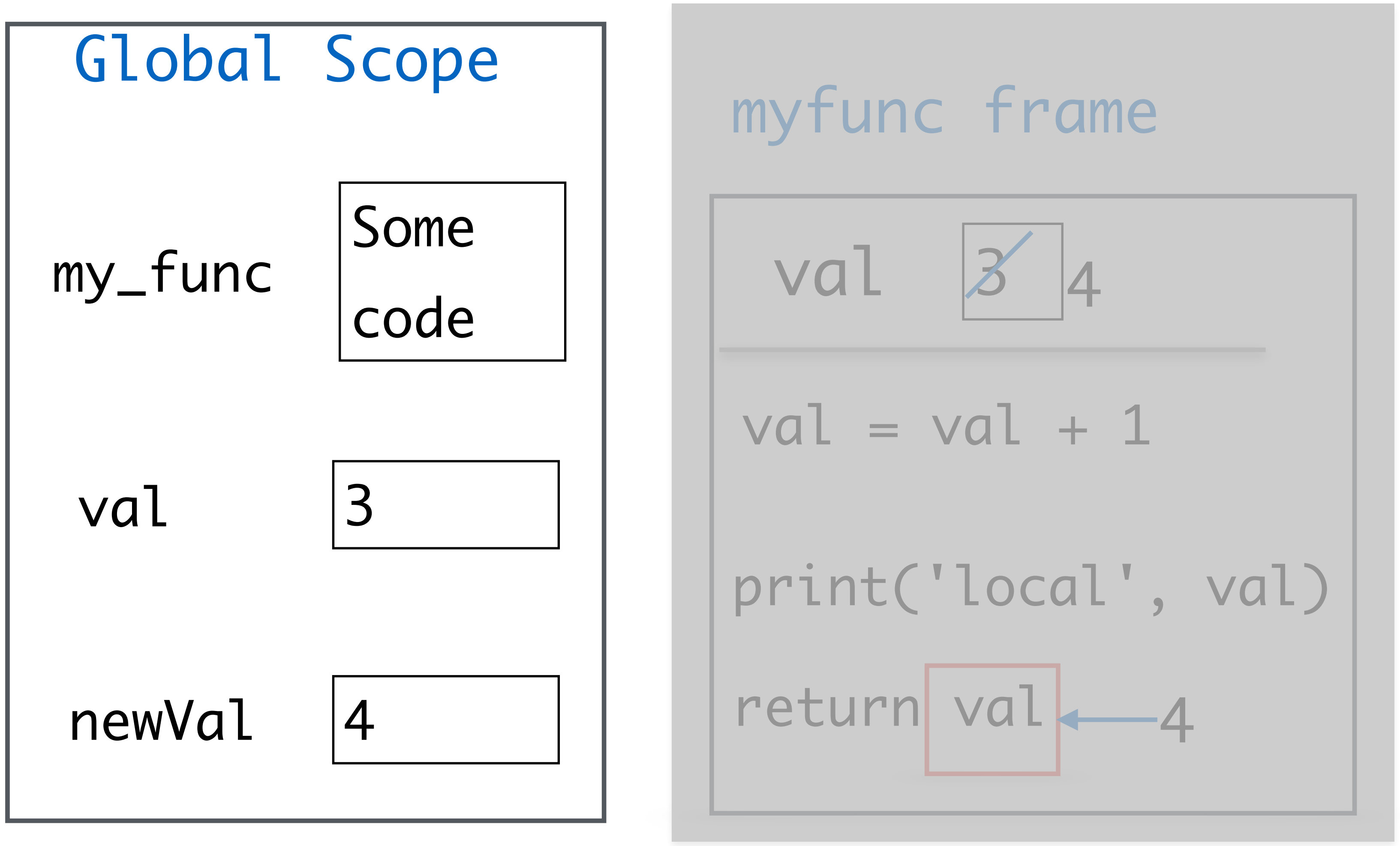


Information flow out of a function is only through return statements!

Variable Scope: A Tricky Example

```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


val = 3
new_val = my_func(val)
print('global val', val)
```

What is printed here?



scope.py

Example:
Making Change

Exercise: Making Change

- Suppose you are a cashier and you need to make change for a given number of cents using only quarters, dimes, nickels, and pennies
- Most cashiers use the following greedy strategy to make change using the fewest number of coins:
    - Use as many quarters as possible first, then as many dimes as possible next, and so on, using pennies last
    - Assume you have an unlimited supply of each coin

Exercise: Making Change

- **Problem**. Let us write a function `make_change`(`cents`) that takes as a parameter an integer `cents` and returns the fewest *number of coins* needed to make change for `cents` cents
- **Approach**: Decompose the problem into smaller pieces
  - What is the maximum number of quarters we can use?
    - `q = cents // 25`
  - How much money is left after we use `q` quarters?
    - `cents = cents % 25`
  - For the remaining cents, what is the maximum number of dimes can we use?

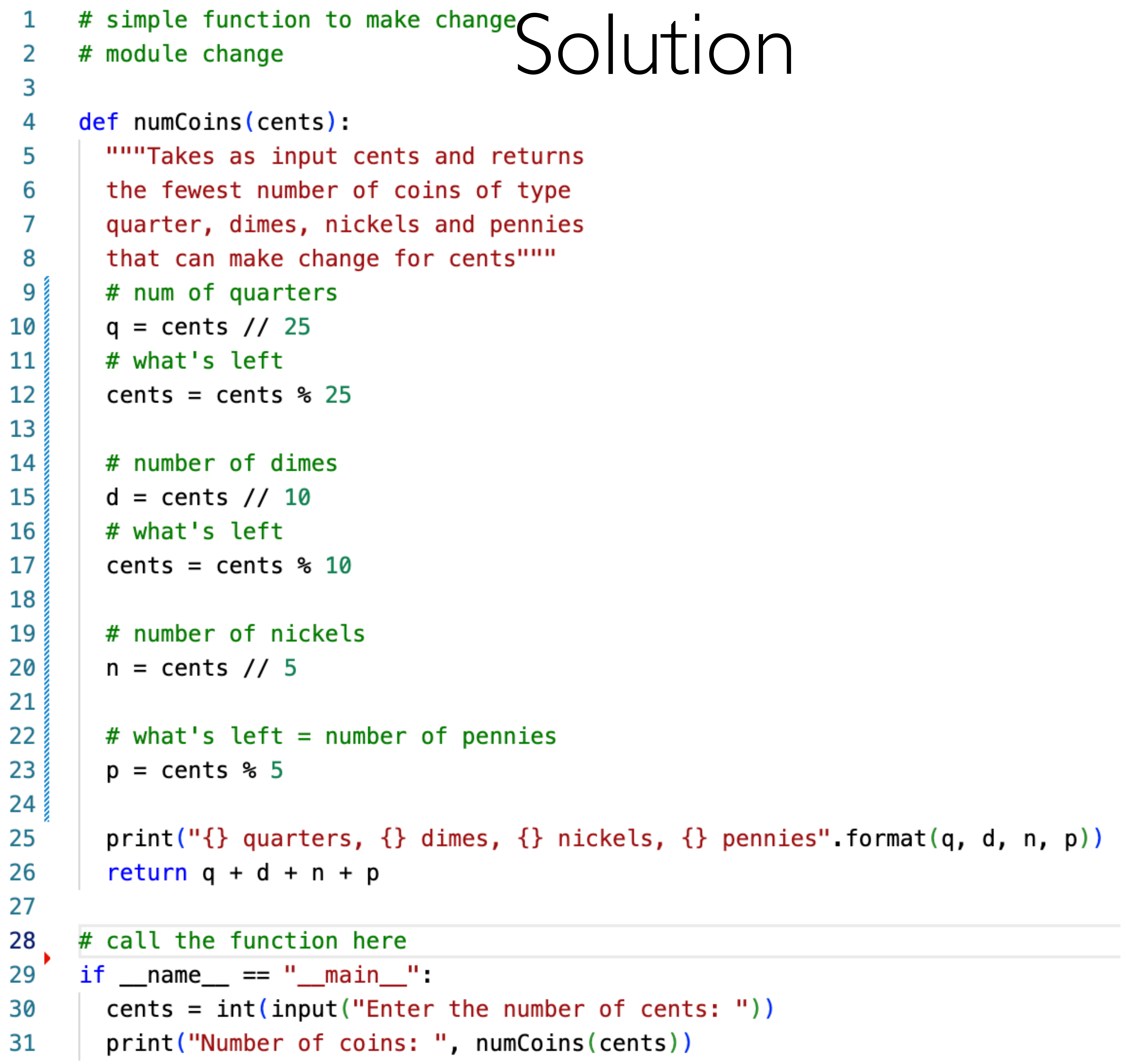
Example Code

```python
# simple function to make change
# module change

def numCoins(cents):
  """Takes as input cents (int) and returns the fewest
  number (int) of coins of type quarter, dimes, nickels
  and pennies that can make change for cents"""

# print("{} quarters, {} dimes, {} nickels, {} pennies".format(q, d, n, p))
pass

# call the function here
# ignore the next line for now
if __name__ == "__main__":
  cents = int(input("Enter the number of cents: "))
  print("Number of coins: ", numCoins(cents))
```

Ignore this for now... We will come back to this soon.

Let's implement this together!

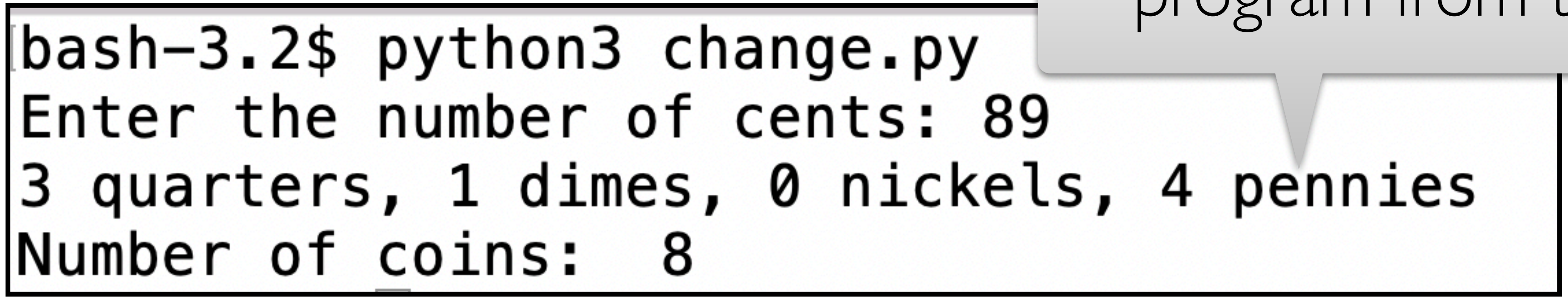
Solution



```python
# simple function to make change
# module change

def numCoins(cents):
  """Takes as input cents and returns
  the fewest number of coins of type
  quarter, dimes, nickels and pennies
  that can make change for cents"""
  # num of quarters
  q = cents // 25
  # what's left
  cents = cents % 25

  # number of dimes
  d = cents // 10
  # what's left
  cents = cents % 10

  # number of nickels
  n = cents // 5

  # what's left = number of pennies
  p = cents % 5

  print("{} quarters, {} dimes, {} nickels, {} pennies".format(q, d, n, p))
  return q + d + n + p

# call the function here
if __name__ == "__main__":
  cents = int(input("Enter the number of cents: "))
  print("Number of coins: ", numCoins(cents))
```

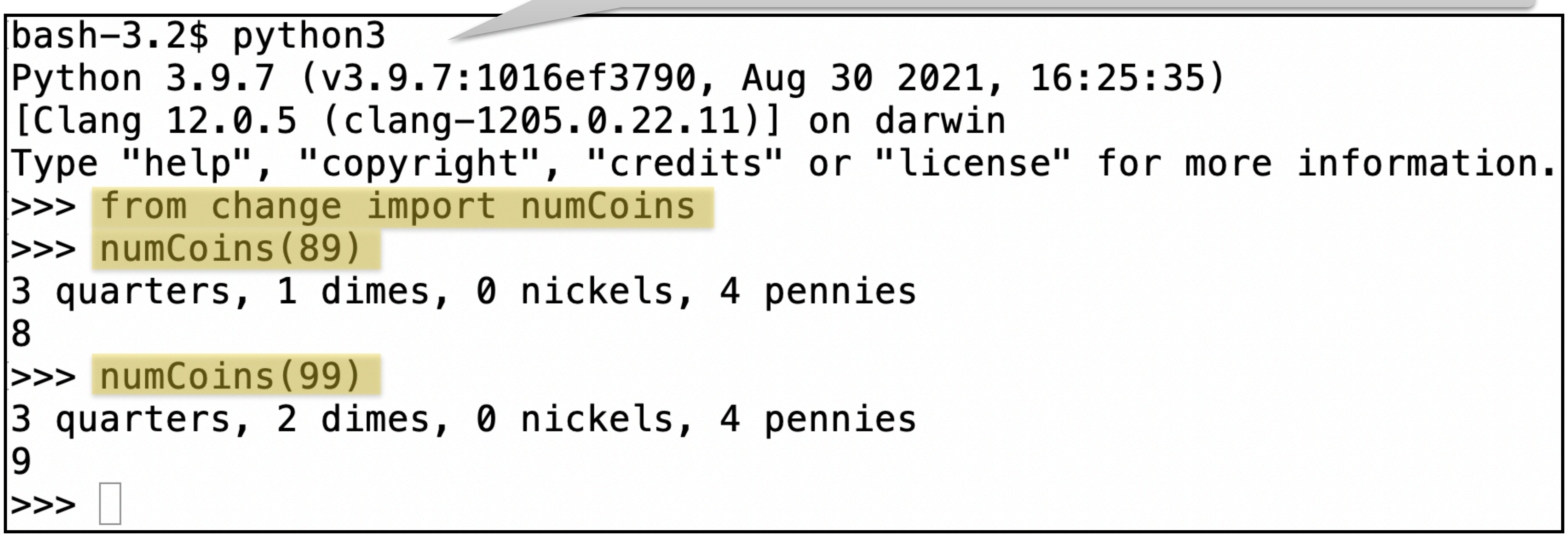
Two Ways To Test Our Code

1) Write code in a file change.py. Execute the program from the Terminal using python3.



```
bash-3.2$ python3 change.py
Enter the number of cents: 89
3 quarters, 1 dimes, 0 nickels, 4 pennies
Number of coins:  8
```

2) Test interactively by importing the function in ***interactive Python***. We'll see this again in Lab 2.



```
bash-3.2$ python3
Python 3.9.7 (v3.9.7:1016ef3790, Aug 30 2021, 16:25:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from change import numCoins
>>> numCoins(89)
3 quarters, 1 dimes, 0 nickels, 4 pennies
8
>>> numCoins(99)
3 quarters, 2 dimes, 0 nickels, 4 pennies
9
>>> 
```

Functions with Multiple Parameters

Function Parameters

- Functions can take any number of parameters:
  - Listed one by one in the definition, separated by commas
  - **Order matters!** Order of parameters in definition maps to order of arguments at function call

```
def exp(num, k):
    """Return the kth power of given number num"""
    return num ** k
```

- How to call this function to compute the 10th power of 2?

Review: Return Statements

- `return` only has meaning inside of a function body
- A function definition may have multiple `return` statements, **but only the first one encountered is executed!** (Why?)
    - We will see functions with multiple returns very soon
- Code that exists *after* a `return` statement is **unreachable** and will not be executed (Why?)
- Functions without an ***explicit*** return statement ***implicitly*** return **None**
    - Be careful when `None` returning functions are used in expressions or within other function calls

Function Calls are Expressions

- Return value of a function "replaces" the function call

```python
def three():
    return 3

x = three()
print(x)
print(three())

two_x = three() + three()
print(two_x)
print(three() + three())

y = print(three())
print(y)
print(print(three()))
```

```
>>> x = three()
>>> print(x)
3
>>> print(three())
3


>>> two_x = three()+three()
>>> print(two_x)
6
>>> print(three() + three())
6

>>> y = print(three())
3
>>> print(y)
None
>>> print(print(three()))
3
None
```

Moving On:
Making Decisions

Making Decisions

If it is raining, then bring an umbrella.

If the light is yellow, slow down. If it is red, stop.

If you are testing positive for COVID, wear a mask.

Making Decisions

If it is raining, then bring an umbrella.

If the light is yellow, slow down. If it is red, stop.

If you are testing positive for COVID, wear a mask.

Making Decisions

If it is raining, then bring an umbrella.

Is it raining?

If the light is yellow, slow down. If it is red, stop.

Is it yellow? red? green?

If you are testing positive for COVID, wear a mask.

Is your test positive? Has it been less than 10 days?

Boolean Types

- Python has two values of **bool** type, written **`True`** and **`False`**
- These are called logical values or Boolean values, named after 19th century mathematician George Boole
- **`True`** and **`False`** must be capitalized!
- Boolean values naturally result when answering a yes or no question
    - Is 10 greater than 5? Yes/True
    - Is 23 an even number? No/False
    - Does 'Williams' begin with a vowel? No/False
- Boolean values result naturally when using **relational** and **logical** operators

Relational Operators

< (less than), > (greater than)

<= (less than or equal to), > = (greater than or equal to)

== (equal to), ! = (not equal to)

Reminder that the single = is an assignment, double == is equality

```
>>> 3 > 5
False
>>> 5 != 6
True
>>> 5 == 5
True
```

Relational Operators

< (less than),   > (greater than)

<= (less than or equal to),   > = (greater than or equal to)

== (equal to),   ! = (not equal to)

Reminder that the single = is an assignment, double == is equality

```
>>> 0 == True
False
>>> True == True
True
>>> int(False)
0
>>> int(True)
1
```

Logical Operators

- Logical operators **`and`**, **`or`**, **`not`** are used to combine Boolean values
- For two Boolean expressions `exp1` and `exp2`
  - **`not`** `exp1` (! in other languages) returns the opposite of `exp1`
  - `exp1` **`and`** `exp2` (&& in other languages) is **`True`** iff `exp1` **and** `exp2` are **`True`**
  - `exp1` **`or`** `exp2` (|| in other languages) is **`True`** iff either `exp1` **or** `exp2` are **`True`**

Truth Table for or

exp1	**exp2**	**exp1 or exp2**
True	True	**`True`**
True	False	**`True`**
False	True	**`True`**
False	False	**`False`**

Truth Table for and

exp1	**exp2**	**exp1 and exp2**
True	True	**True**
True	False	**False**
False	True	**False**
False	False	**False**

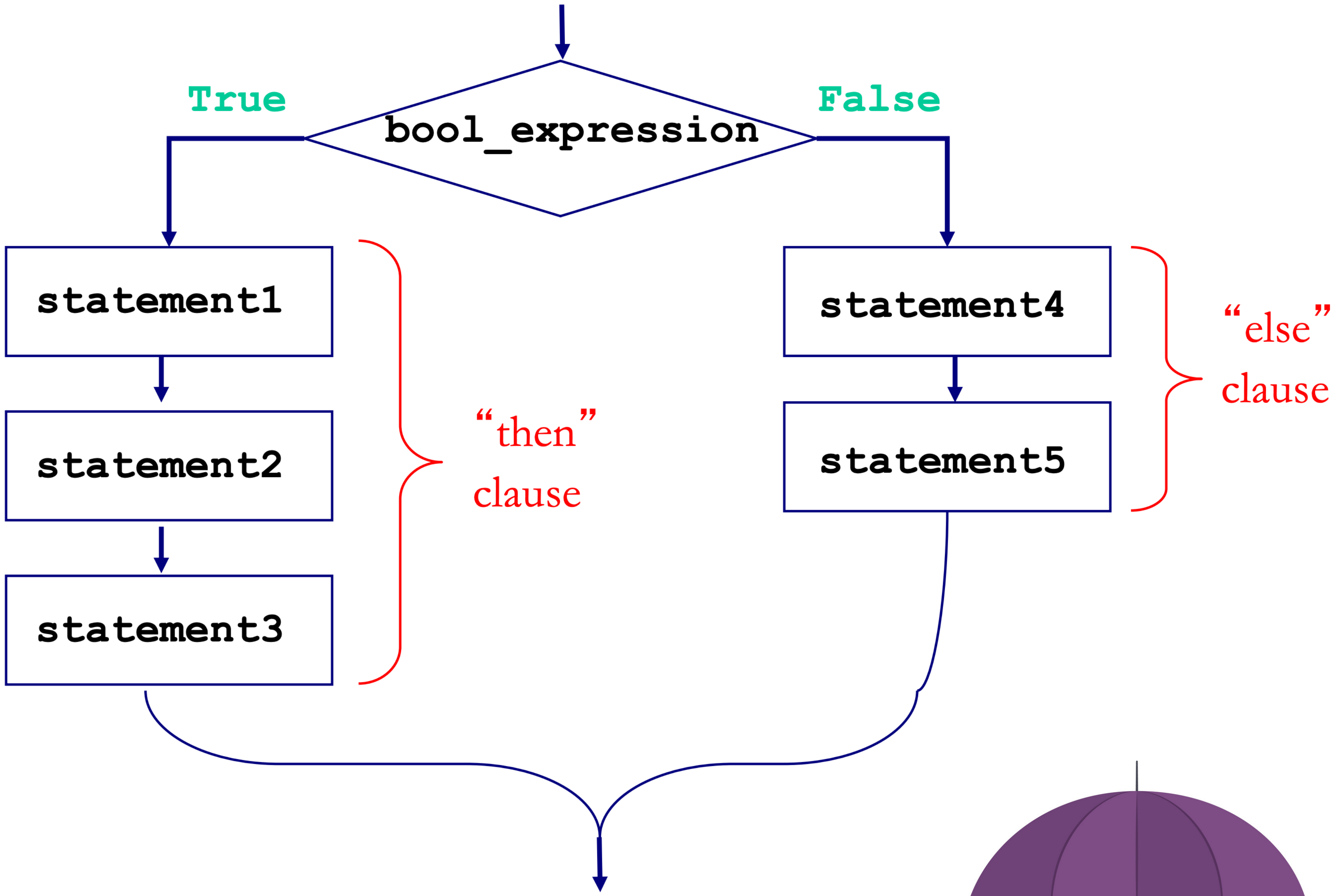
Boolean Expressions and If Statement

- Python expressions that result in a `True/False` output are called **boolean expressions**
  - For example, checking if a user's entered number, **`num`**, is even
- How do we do this? (What is a property of even numbers that we can use to test this condition?)
  - Even numbers are evenly divisible by 2 (remainder of zero)
  - Thus, **`num % 2`** should be zero if and only if **`num`** is even
- Now we have a Boolean expression we can test for: **`num % 2 == 0`**
- We can implement "conditional statements" in Python using Boolean expressions and an **if-else statement**

Python Conditionals (`if` Statements)

```
if <boolean expression>:
        statement1
        statement2
        statement3
else:
        statement4
        statement5
```

Note: (syntax) Indentation and colon after if and else



If it is raining, then bring an umbrella. Else, bring your sunglasses.

Optional Else & Simplifying Conditionals

- The else block is **optional**: not a requirement (not always needed!)
- Sometimes we can simplify conditionals
    - For example, all three below are equivalent inside the body of a function that returns `True` if num is even, and `False` otherwise

```
if num % 2 == 0:
    return True
else:
    return False
```

→

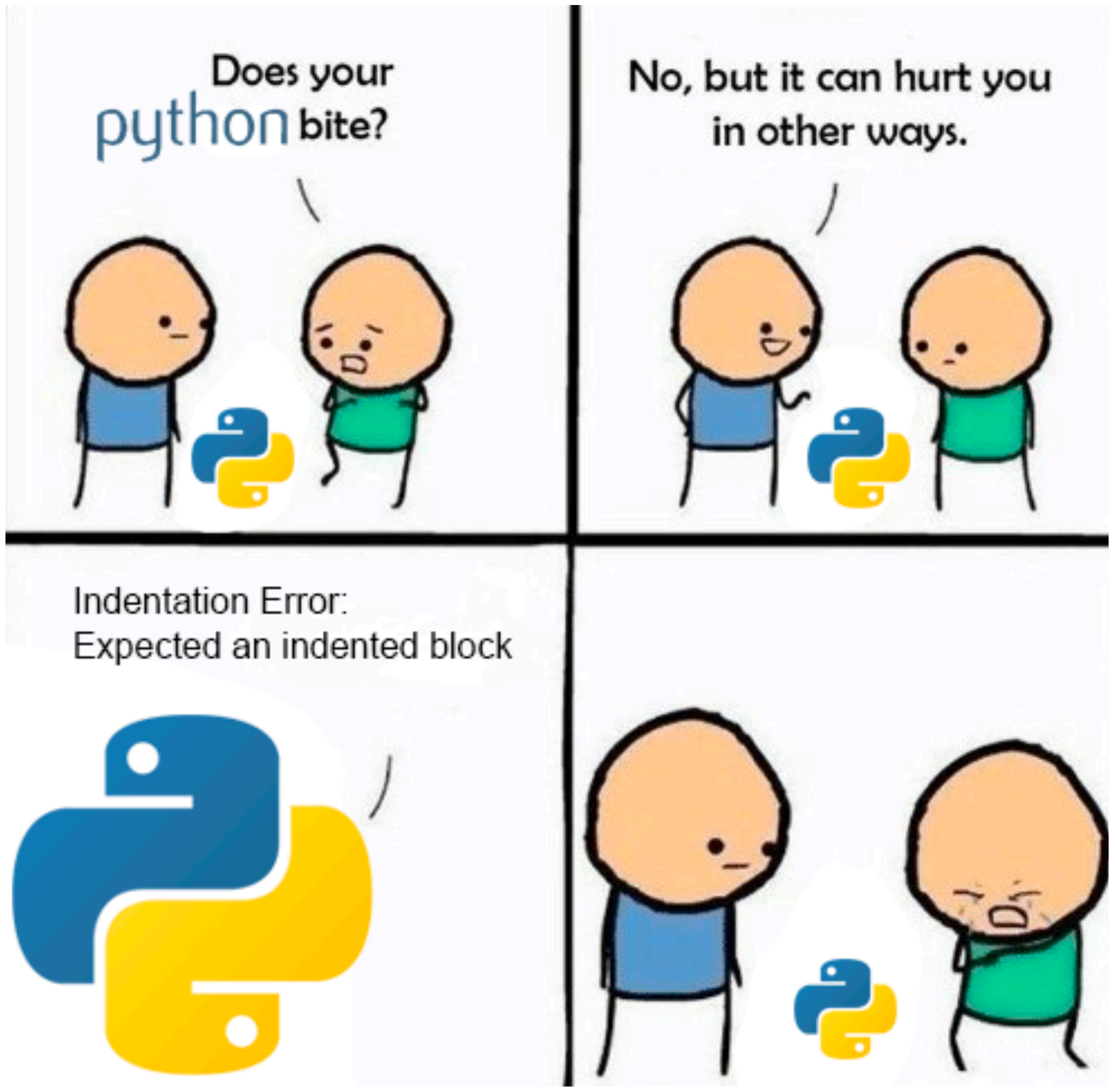
```
if num % 2 == 0:
    return True
return False
```

→

```
return num % 2 == 0
```

Python Conditionals (`if` Statements)

- Don't forget proper indentation!


Does your python bite?
No, but it can hurt you in other ways.
Indentation Error:
Expected an indented block

(Credit to u/ufoludek_ on r/ProgrammerHu

Example

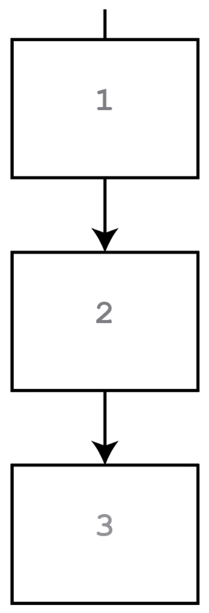
Conditional Statements: If Else

- Consider the following functions that check if a number is even or odd

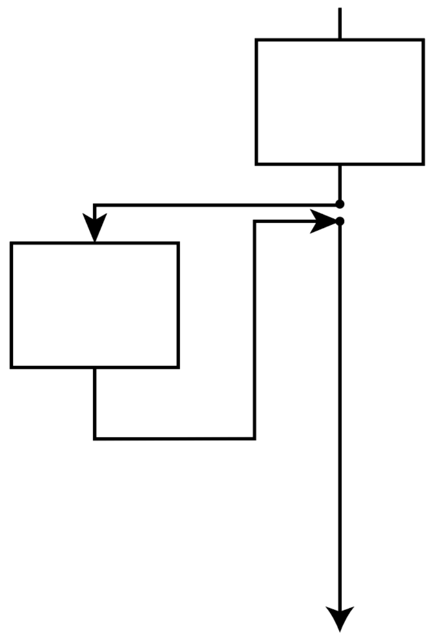
```
def print_even(num):
    '''Takes a number as parameter, prints Even if
it's even,
    else prints odd '''
    if num % 2 == 0: # if even
        print("Even")
    else:
        print("Odd")


def is_even(num):
    ''' Takes a number as parameter, returns True if
    it's even, else returns False'''
    return num %2 == 0

# MAIN PROGRAM
print("3? " + print_even(3))
print("22? " + print_even(22))

print("is_even(3)? " + str(is_even(3)))
print("is_even(22)? " + str(is_even(22)))
```

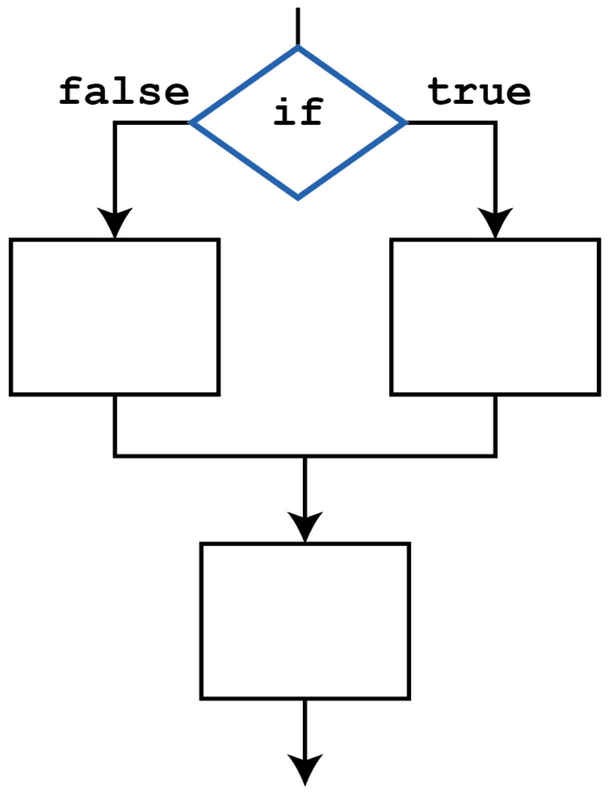
Takeaways

- Chained conditionals avoid messy nested conditionals
- Chaining reduces complexity and improves readability
- Since only one branches in a chained `if-else` block evaluates to `True`, using them avoids unnecessary checks incurred by chaining if statements one after the other



sequence


functions



conditionals

The end!