CS1 34 Lecture 4: Functions, Booleans, and Conditionals

Announcements & Logistics

- **Homework 2** is due Monday 10 pm
  - Ten multiple-choice questions on Glow
  - Try to answer them using pencil and paper first
  - Can verify answers using interactive Python if you wish
- **Lab 2** has been posted, 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
- Functions can return values, change the state of the world, or both
- 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
    - Such functions "secretly" (or ***implicitly***) return a **None** value
    - Well-written code will almost always return a value **with the same type** for all paths through the function

Today's Plan

- Wrap up discussion of functions
  - Discuss return statements and variable scope in more detail
  - Functions with multiple arguments
- Introduce conditionals and Boolean data type
  - Making decisions in Python using `if else` statements

Variable Scope

- **Local variables:** An assignment to a variable *within a function* definition creates/modifies a *local variable*
- Local variables exist and are valid *only* within a function's body, and cannot be referred to outside of it
- **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)
```

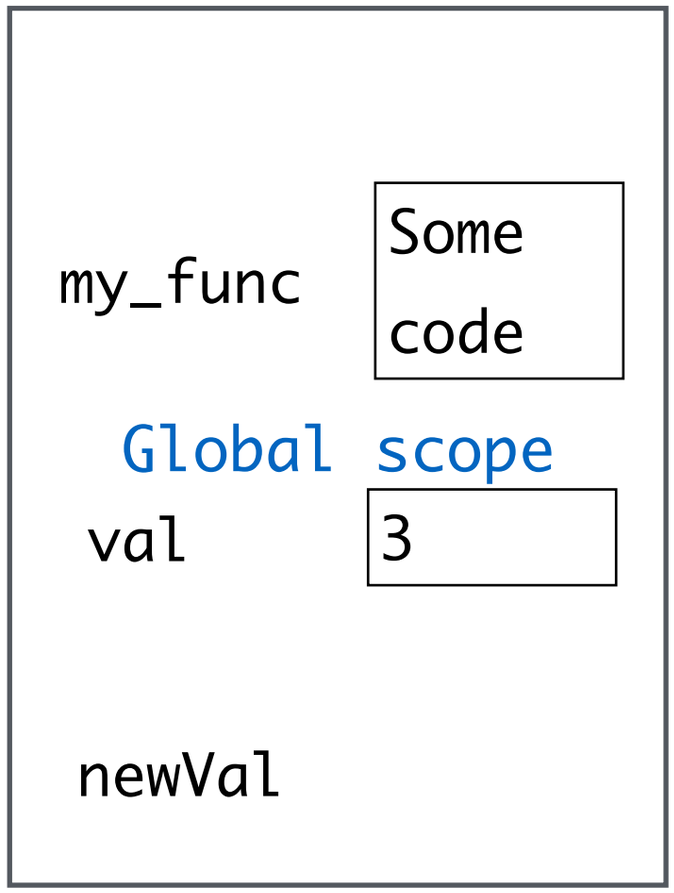
What is printed here?

What is returned?

What is printed here?

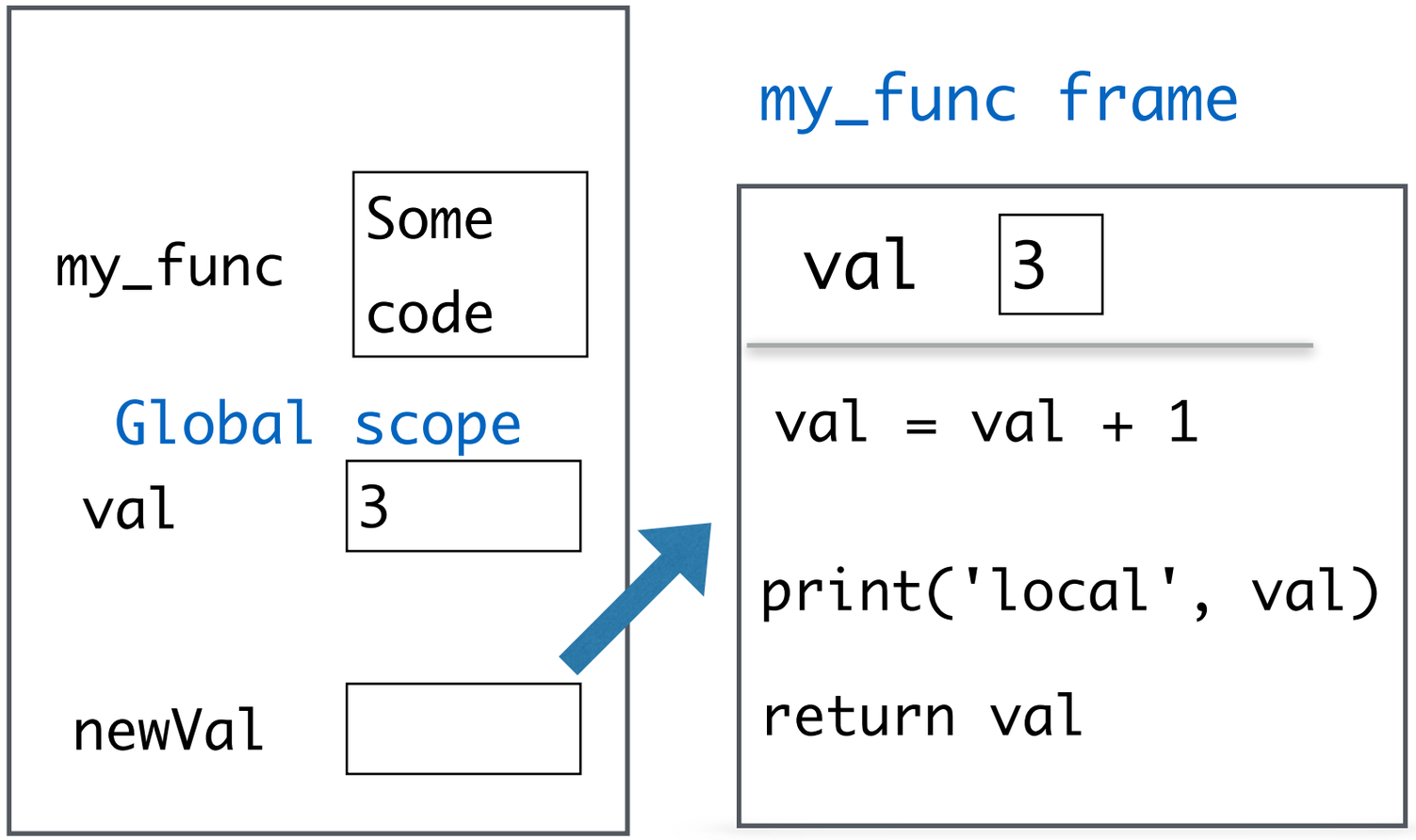
Variable Scope: A Tricky Example

```
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

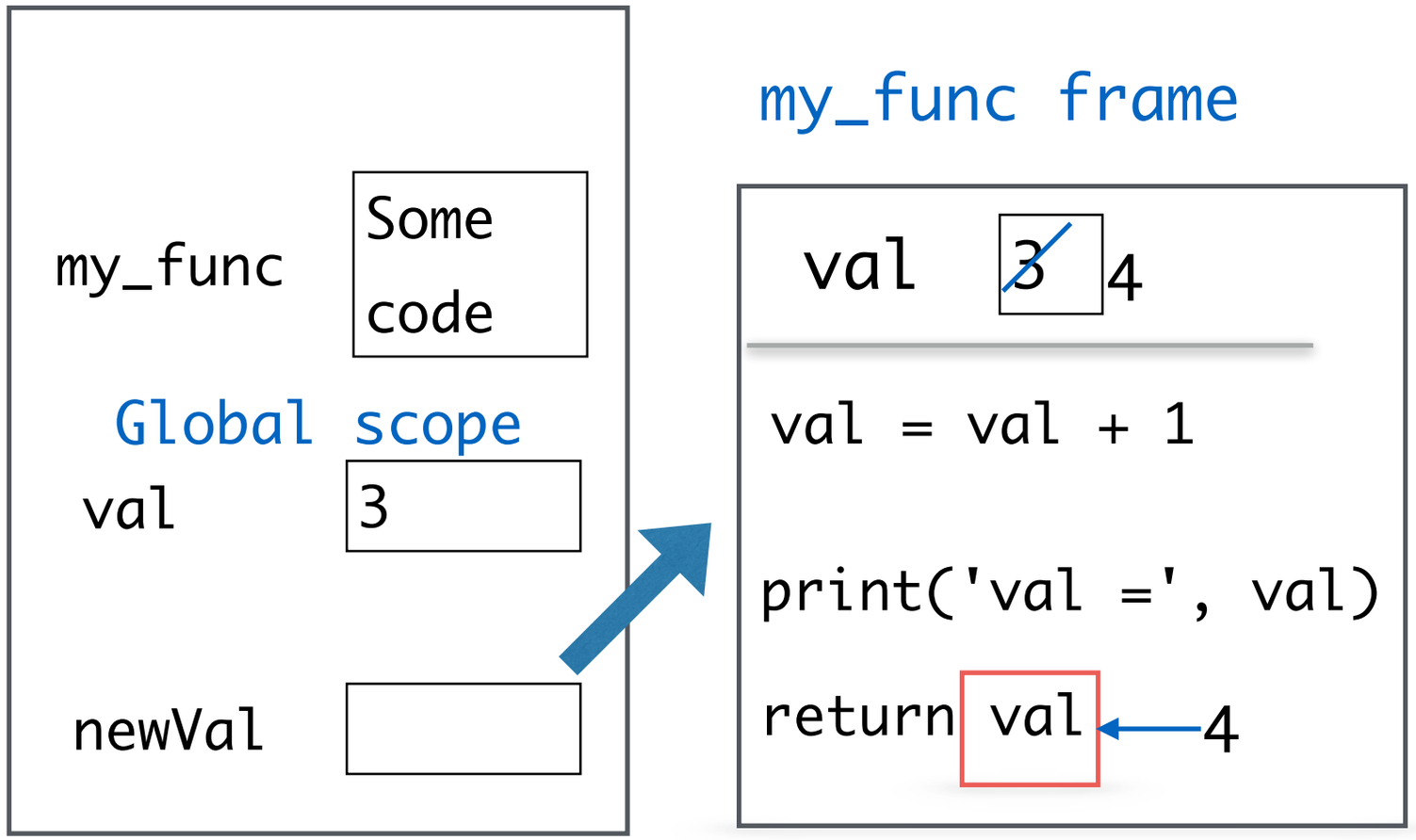
```
def my_func (val):
    val = val + 1
    print('local val', val)
    return val


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)
```

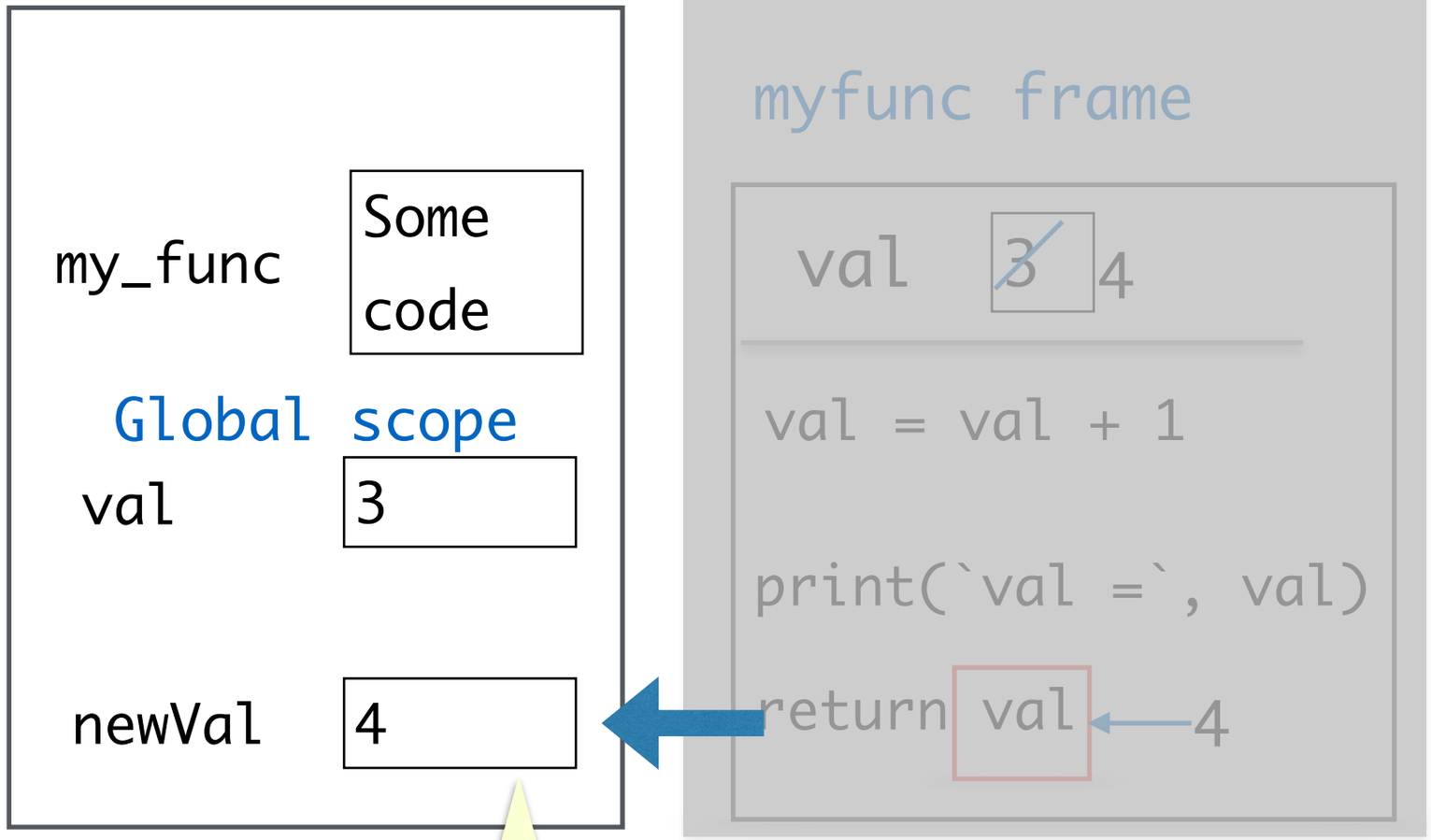
What is printed here?



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)
```

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



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

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

Moving On:

Making Decisions

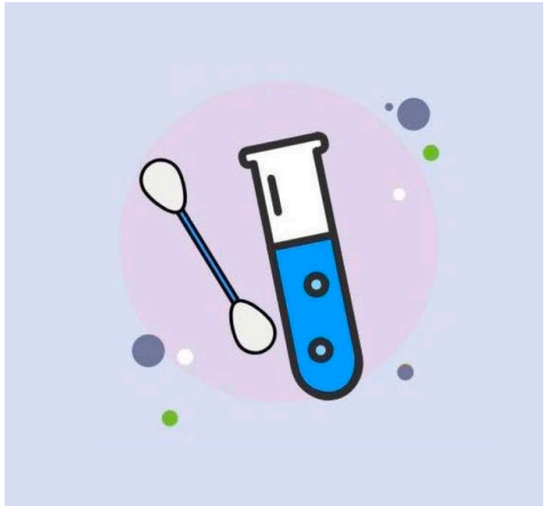
Making Decisions

If it is raining, then bring an umbrella.

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

If you are inside an academic building, wear a mask.

If your name starts with letters A-L, test on Tuesdays.



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 inside an academic building, wear a mask.

If your name starts with letters A-L, test on Tuesdays.

Decisions Based on Yes/No Questions

If it is raining, then bring an umbrella.

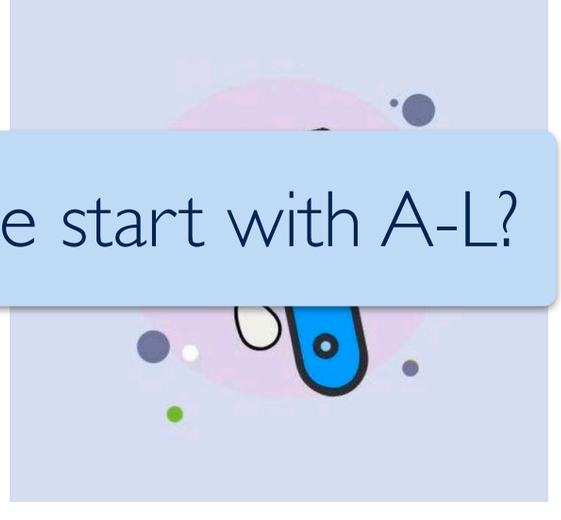
Is it raining?

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

Is it yellow? red? green?

If you are inside an academic building, wear a mask.

Are you inside ?



Does your name start with A-L?

If your name starts with letters A-L, test on Tuesdays.

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
```

Boolean Expressions and If Statement

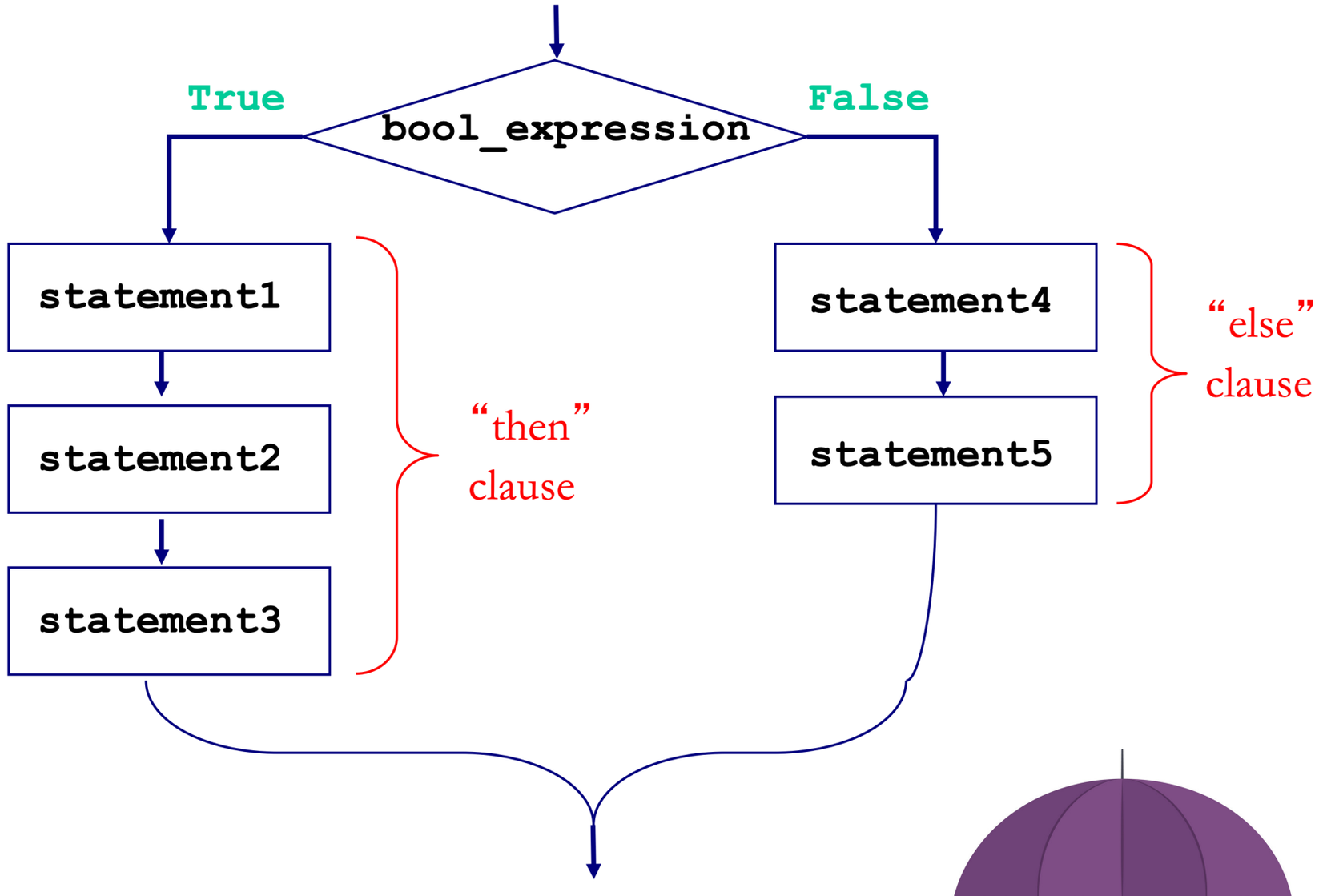
- Python expressions that result in a `True/False` output are called **boolean expressions**
  - For example, checking if a user's entered number, **`num`**, is even
- How do we do this? (What is a property of even numbers that we can use to test this condition?)

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
- 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

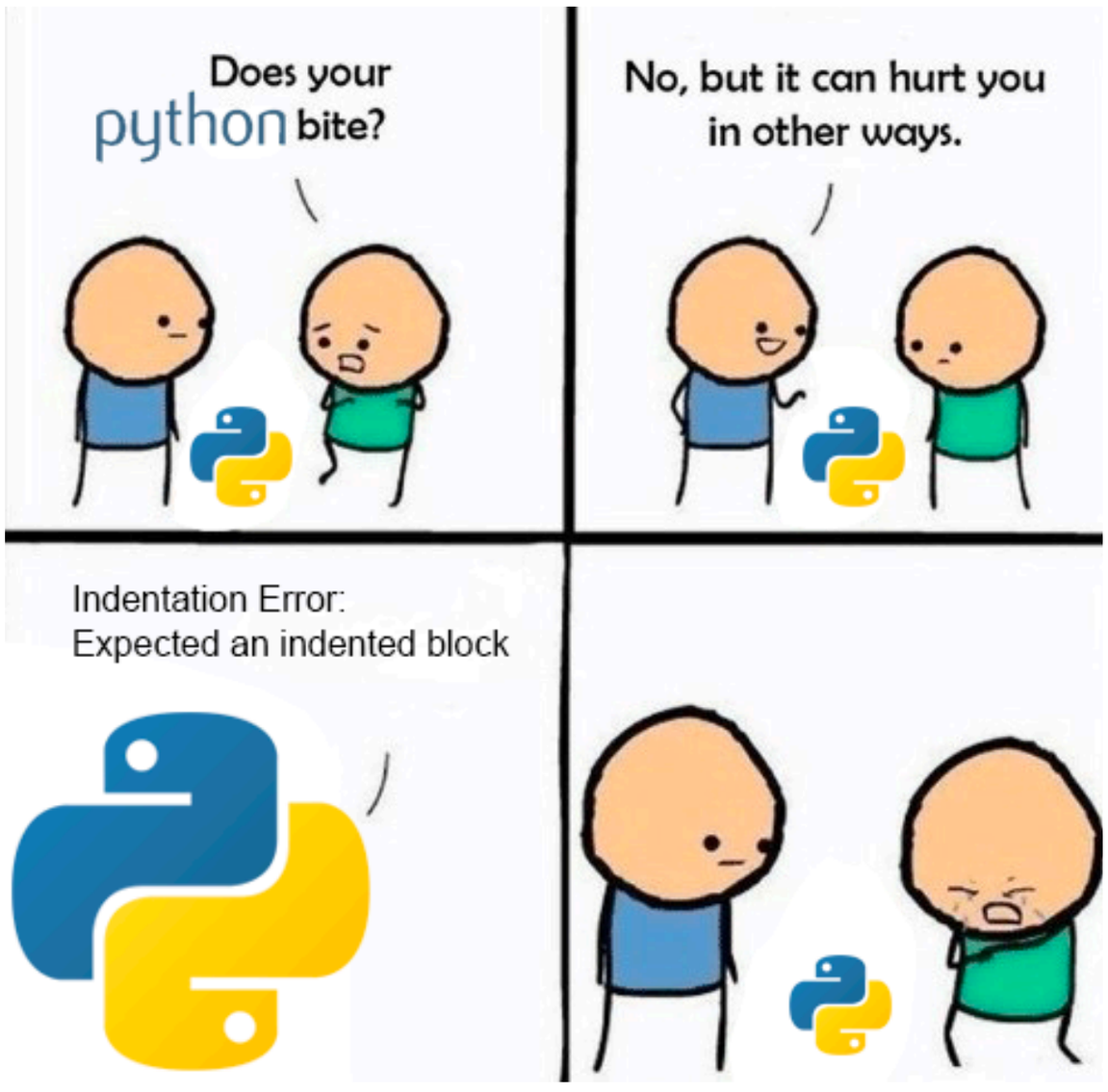
```
if num % 2 == 0:
    return True
else:
    return False
```

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

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

Python Conditionals (`if` Statements)

- Don't forget proper indentation!



(Credit to u/ufoludek_ on r/ProgrammerHumor)

Let's See Some Examples

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**

Nested Conditionals

- Sometimes, we need a more complicated conditional structure with more than 2 options
- Example: Write a function that takes a temp value in Fahrenheit
    - If temp is above 80, print "It is a hot one out there."
    - If temp is between 60 and 80, print "Nice day out, enjoy!"
    - If temp is below 60, print "Chilly day, don't forget a jacket."
- Notice that temp **can only be in one of those** ranges
    - If we find that temp is greater than 80, no need to check the rest!

Nested Conditionals

```
if booleanExpression1:
    statement 1
    ...
else:
    if booleanExpression2:
        statement 2
        ...
    else:
        statement 3
        ...
```

Attempt 1: Chained Conditionals

- We can **nest** if-else statements (using indentation to distinguish between matching if-else blocks)
- Works, but this can quickly become unnecessarily complex (and hard to read!) This is an example of what NOT to do!

```
def weather1(temp):
    if temp > 80:
        print("It is a hot one out there.")
    else:
        if temp >= 60:
            print("Nice day out, enjoy!")
        else:
            if temp >= 40:
                print("Chilly day, wear a sweater.")
            else:
                print("Its freezing out, bring a winter jacket!")
```

Attempt 2: Chained Ifs

- What if we use a bunch of if statements (w/o else) one after the other to solve this problem?
- What are the advantages/disadvantages of this approach?

```
def weather2(temp):
    if temp > 80:
        print("It is a hot one out there.")
    if temp >= 60 and temp <= 80:
        print("Nice day out, enjoy!")
    if temp <60 and temp >= 40:
        print("Chilly day, wear a sweater")
    if temp < 40:
        print("Its freezing out, bring a winter jacket!")
```

If Elif Else Statements

- Fortunately, there is a simpler way to specify several options by **chaining** conditionals

```
if booleanExpression1:
    statement 1
    ...
elif booleanExpression2:
    statement 2
    ...
else:
    statement 3
    ...
```

A better approach that avoids too many indented blocks and improves code readability

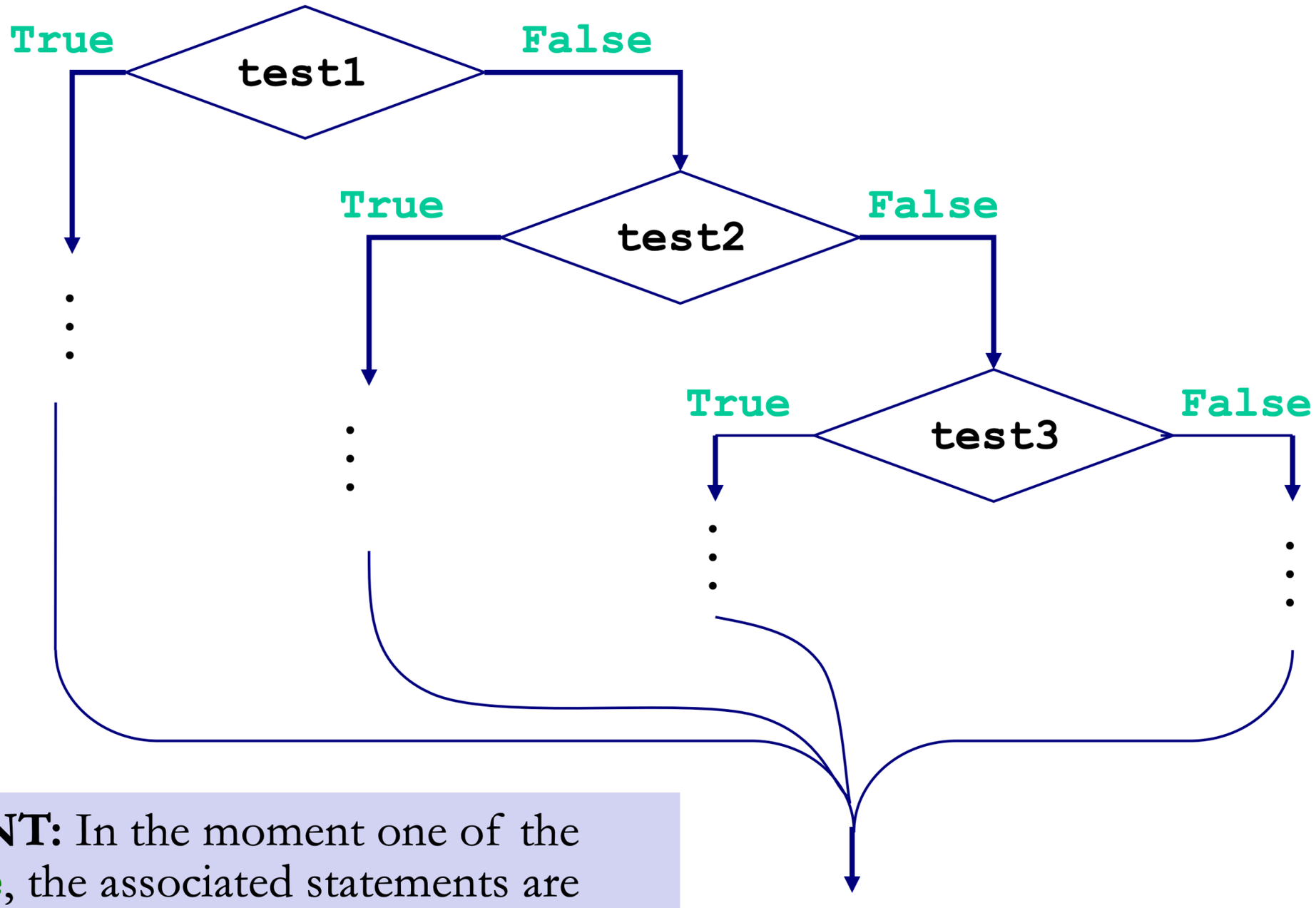
Can have any number of `elif` conditions, but only one (optional) `else` (at the end)

Attempt 3: Chained Conditionals

- We can chain together any number of elif blocks
- The else block is **optional** (not a required part of the syntax)

```
def weather3(temp):
    if temp > 80:
        print("It is a hot one out there.")
    elif temp >= 60:
        print("Nice day out, enjoy!")
    elif temp >= 40:
        print("Chilly day, wear a sweater.")
    else:
        print("Its freezing out, bring a winter jacket!")
```

Flow Diagram: Chained Conditionals



IMPORTANT: In the moment one of the tests is `True`, the associated statements are executed and the chained conditional is exited. Only in the case when tests are False, we continue checking to find a True test.

Takeaways

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

CS Colloquium Today

- Almost Every Friday
- Time: **2:35pm**
- Normal Location: **TCL 123** (Wege Auditorium)
- Today: Thesis Proposals (Part 2)

Next Time:
Leap Year Function

Exercise: `leapYear` Function

- Let's write a function `leapYear` that takes a `year` (int) as input, and returns `True` if `year` is a leap year, else returns `False`
- When is a given year a leap year?
  - *"Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years, if they are exactly divisible by 400."*

How do we structure this logic using booleans and conditionals?

Exercise: `leapYear` Function

- Let's write a function `leapYear` that takes a `year` (int) as input, and returns `True` if `year` is a leap year, else returns `False`
- When is a given year a leap year?
  - *"Every year that is exactly divisible by four is a leap year, except for years that are exactly divisible by 100, but these centurial years are leap years, if they are exactly divisible by 400."*

Decomposition!

  - If year is **not** divisible by 4: year is not a leap year
  - Else (divisible by 4) and if **not** divisible by 100: is a leap year
  - Else (divisible by 4 and by 100) and **not** divisible by 400: not a leap year
  - Else (if we make it to here must be divisible by 400): is a leap year