Abstracting with Functions

Reminders and Announcements

- Make sure you pick up Homework 1 today
- Due Monday (Feb 17 in class)
- Monday labs: push your work (every week) by Wed 11 pm
- Tuesday labs: push your work (every week) by Thurs 1 pm

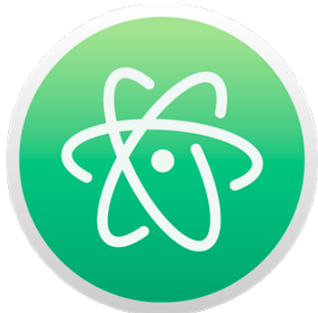
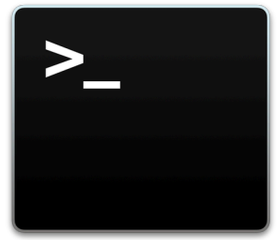
- **Late day policy for labs.** Each student has three late days, with at most two late day towards any particular lab.
- Late day: no-questions-asked 24-hour extension
- You must request a late day in advance on the Late day form located on the course webpage, under Course Policies

Check-in After First Lab!

- You have all had your first computer science lab
  - **Congratulations !**
- Computer science tools you used:
  - **Atom** as a text editor
  - **Terminal** as a text-based interface to the computer
  - **Git** for versioning, **Github/Gitlab** (cloud-based hosting service) for retrieving & submitting your work
  - **Python**, of course

Do You Have Any Questions?



Review and Reflect

- What is the difference between executing a **python** program as a **script** versus using **interactive python** on the terminal

- What's the difference between **Jupyter notebooks** we use in class versus an **interactive python** session?

- How can you test out and play with examples we do in a **Jupyter notebook** by yourself?

- What is the difference between `Out[]` when we run a command in Jupyter vs using the `print` command?

Structuring Code

- So far
    - We have written simple expressions
    - We can create small scripts to do certain tasks
- This is fine for small computations
    - Need more organization for larger problems
- Structuring code is good to
    - Keep track of which part of code is going what
    - What information needs to supplied where
    - **Reusability!** reusing blocks of code we write

Abstracting with Functions

- **Abstraction** to achieve code decomposition and reuse
- Real life example: a projector
    - We know how to switch it on and off **(public interface)**
    - How to connect it to our computer **(input/output)**
    - Don’t know how it works internally **(information hiding)**

- **Key idea:** We don’t need to know much about a projector to be able to use it

Decomposition Using Functions

- To write organized code, divide it tasks into functions
  - That are **self-contained**
  - Each function is a **small piece** of a **larger task**
  - Functions are **reusable**
  - Keep code **organized**
  - Keeps code **coherent**
- Today, we will learn how to decompose code and hide details using functions
- Later in the semester, we will learn a new abstraction which achieves decomposition and code hiding: classes

Anatomy of a Function

- Function **definition** characteristics:
    - Has a name #header
    - Has parameters (or more) #header
    - Has a docstring (optional but recommended) #header
    - Has a **body** (which may compute a value or produce a side-effect like printing)
    - Always **returns** something (even without an explicit `return` statement)
- Functions are not run in a program until they are “called” or “invoked” through a **function call**

Function Example

Function definition

```
def square(x):
    ```Takes a number and returns its square```
    return x*x
```

Function Calls/Invocations

```
In [1] square(5)
Out [1] 25
In [2] square(-2)
Out [2] 4
```

Important:

- Indent in function body (required)
- Colon after function name (required)
- Docstring (optional, good style)
- `x` in function definition is a parameter
- Single line body which returns the result of the expression `x * x`
- `return` always ends execution of function!
```

Parameters

- A **parameter names** are “holes” in the body of a function that will be filled in with **argument value** for each invocation
- A particular name for a parameter is irrelevant, as long as we use it consistently in the body

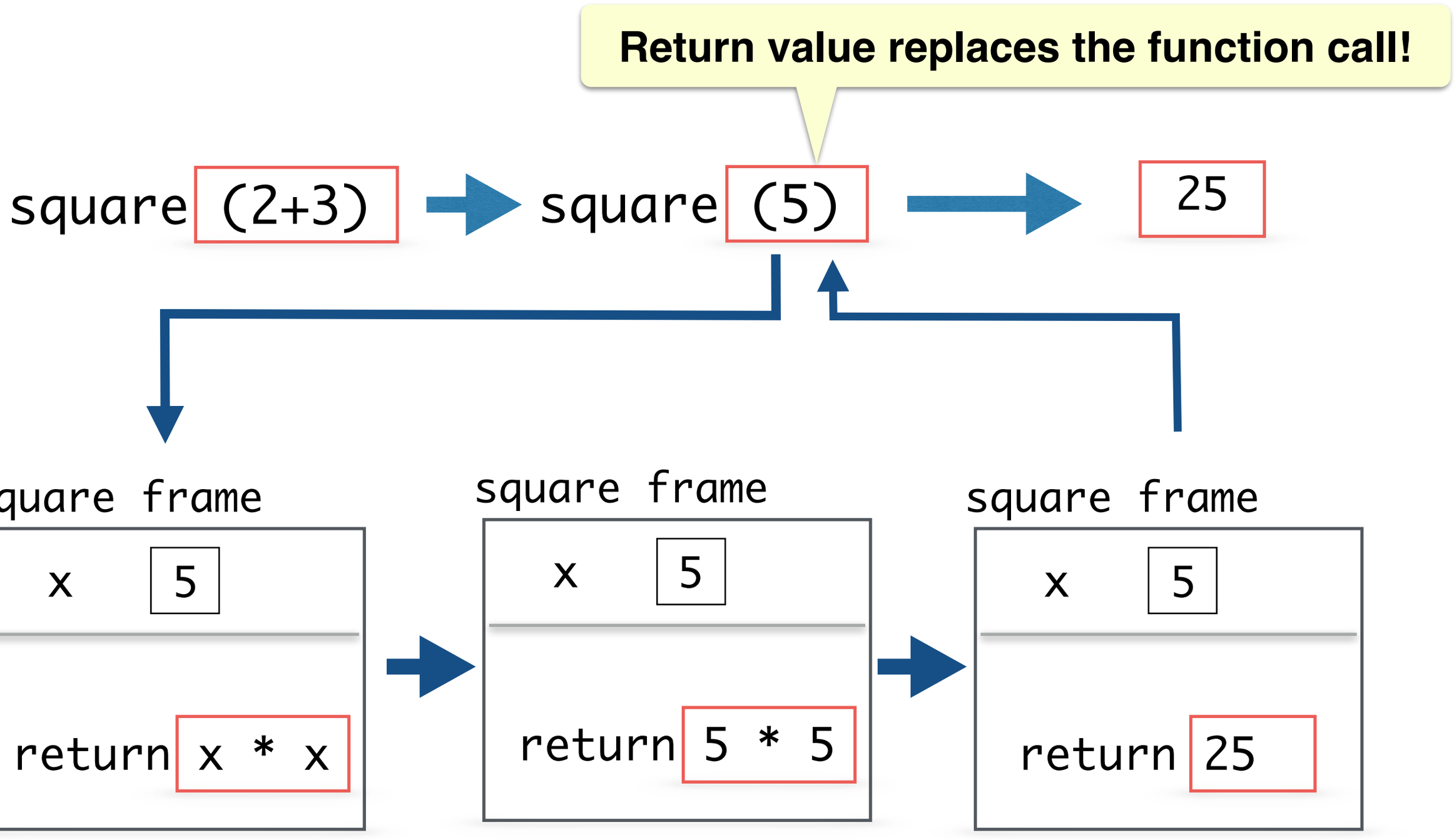
```
def square(x):
    return x*x
```

```
def square(apple):
    return apple*apple
```

```
def square(num):
    return num*num
```

Python Function Call Model

Function frame. Model to understanding how a function call works



Function Call Replaced by Return Value

```
17 + square (2+3)
```

```
17 + square (5)
```

```
17 + 25
```

```
42
```

Return Vs Print

Return

- `return` only has meaning inside a function definition
- A function definition may have multiple returns, but only the first one encountered is executed
- Any code after a return is reached will not be executed
- Has a value associated with it and can be used in expressions
- Function without an explicit return, return a **None**

Print

- print can be used inside or outside functions
- Has a side-effect (prints to console)
- Cannot be used in expressions expecting a value
- Is technically a function and always returns a **None** type
- (**None** is a special python type!)

Fruitful Vs None Functions

We call functions that return a None value **None-returning or None functions**. Such functions are invoked to perform an action (e.g., print something, change state), **and not to compute and return a result.**

We call functions that return a value other than **None fruitful functions** or **value-returning functions.**

Fruitful

```
def square(x):
    return x*x
```

None Function

```
def printHW():
    print(`Hello World')
```

What if I run `print(printHW)` or `print(print((printHW))`?

Exercise: Day of the Week

- Compute the day of the week for an arbitrary date, specified using a month, day, and year (1900—2099)
- Need a monthly adjustment, according to this table
- If it's a leap year and month is Jan or Feb, we must subtract one from the adjustment
- For now, we will just use our predefined function `monthAdjust` that does this part for us

Month	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Adjusment	1	4	4	0	2	5	0	3	6	1	4	6

Exercise: Day of the Week

- Given a month between 1 and 12, a `day` of the month between 1 and 31 and a year in the range 1900-2099
- **Step 1.** Compute the monthly adjustment `madj`
- **Step 2.** Compute the number of years `year` since 1900
- **Step 3.** Compute the `sum` of: `madj, day, year` and the the whole number of times 4 divides `year`
- **Step 4.** Compute the remainder of the `sum` computed above when divided by 7, this gives the day of the week as a num 0-6, where 0 is Saturday, 1 is Sunday etc.
- **Step 5.** Convert the day of the week number to its description

Test Your Steps

- Admiral Grace Hopper was born on **December 9, 1906**
- Monthly adjustment `madj`? **6**
- Year `year` since 1990? **6**
- Day of the week `day`? **9**
- Quotient when `year` is divided by 4? **1**
- `sum = 6 + 6 + 9 + 1 = 22`
- `22 % 7 = 1` ~ **Sunday!**

Month	1	2	3	4	5	6	7	8	9	10	11	12
Adjusment	1	4	4	0	2	5	0	3	6	1	4	6

Testing Functions Interactively

- Defined in a script, test interactively via terminal:
  - Suppose function definition is in a script `dow.py`
  - Can test functions in it interactively using interactive python
  - First compile `dow.py` and then go to interactive python and type `from dow import dayName` (for example)
  - Call `dayName(1)` to see return value and test!
- Function testing and testing on Jupyter notebook
  - Seamlessly combines definitions and testing in one place
  - But everything we do on Jupyter can be done in interactive python via the terminal!

Variable Scope

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

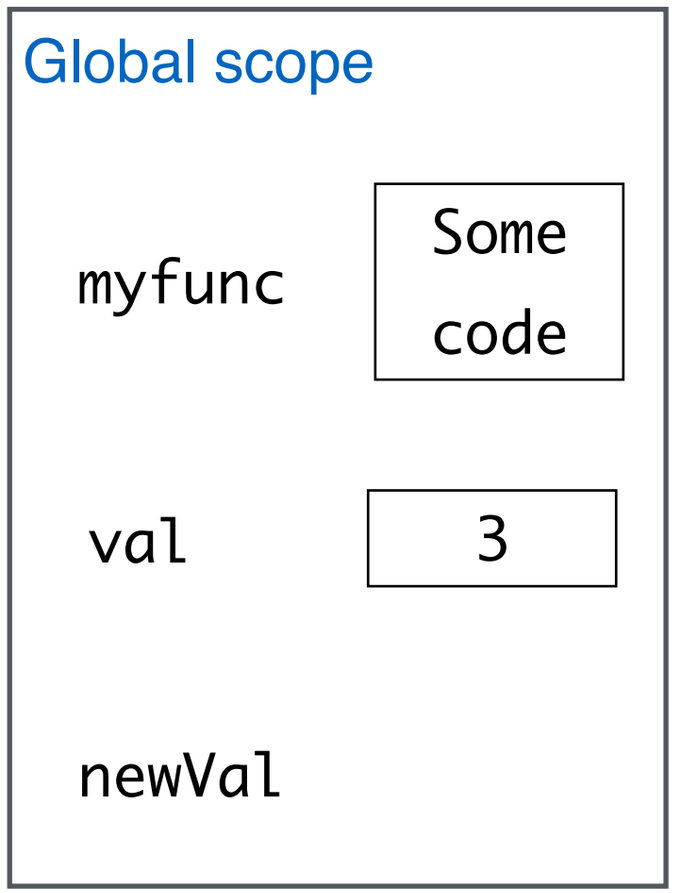
```
def square(num):
    return num*num
```

```
In [1] square (5)
Out [1] 25
In [2] num
NameError: name ‘num’ is not defined
```

Variable Scope

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

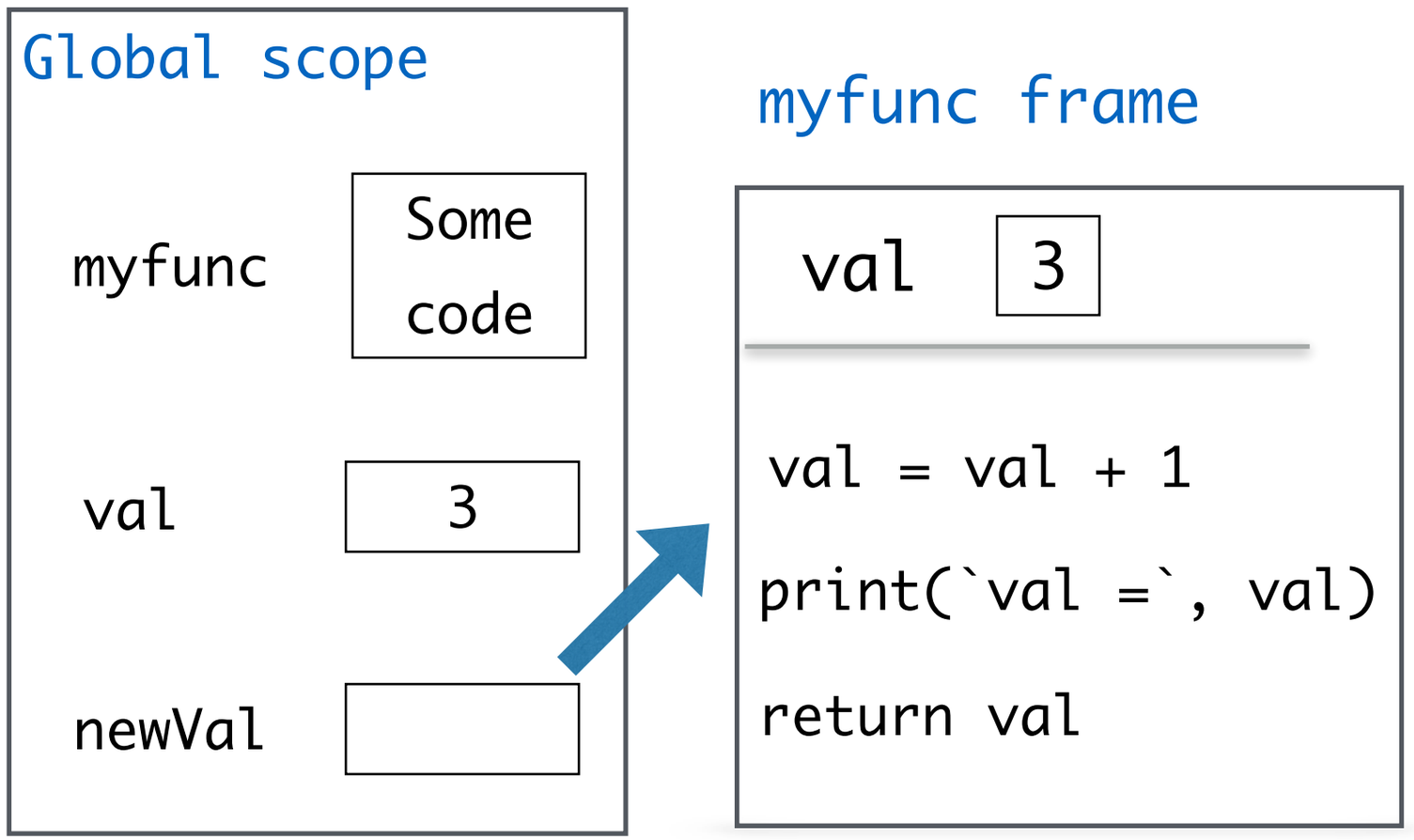
val = 3
newVal = myfunc(val)
```



Variable Scope

```
def myfunc (val):
    val = val + 1
    print(`val =`, val)
    return val

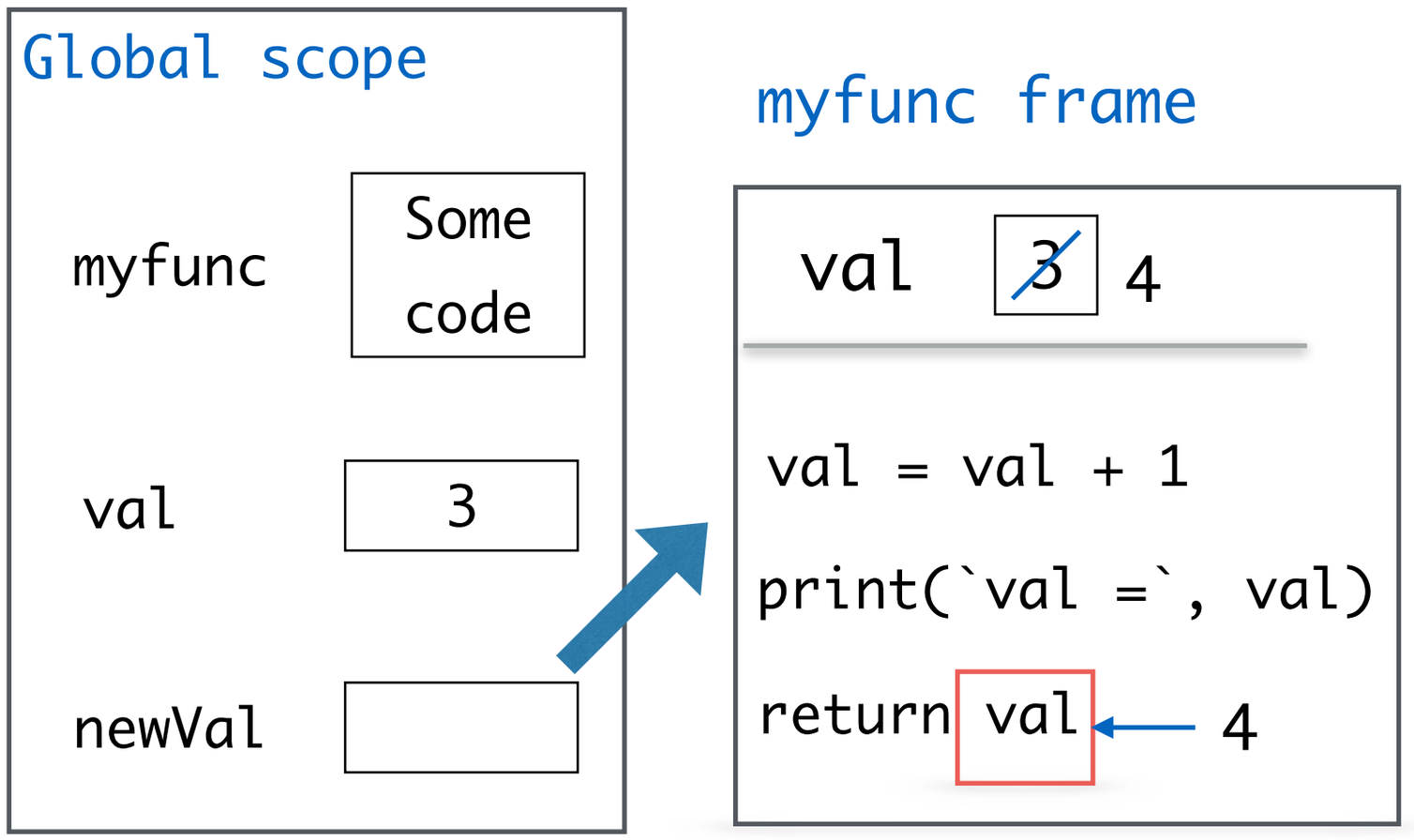
val = 3
newVal = myfunc(val)
```



Variable Scope

```
def myfunc (val):
    val = val + 1
    print(`val =`, val)
    return val

val = 3
newVal = myfunc(val)
```

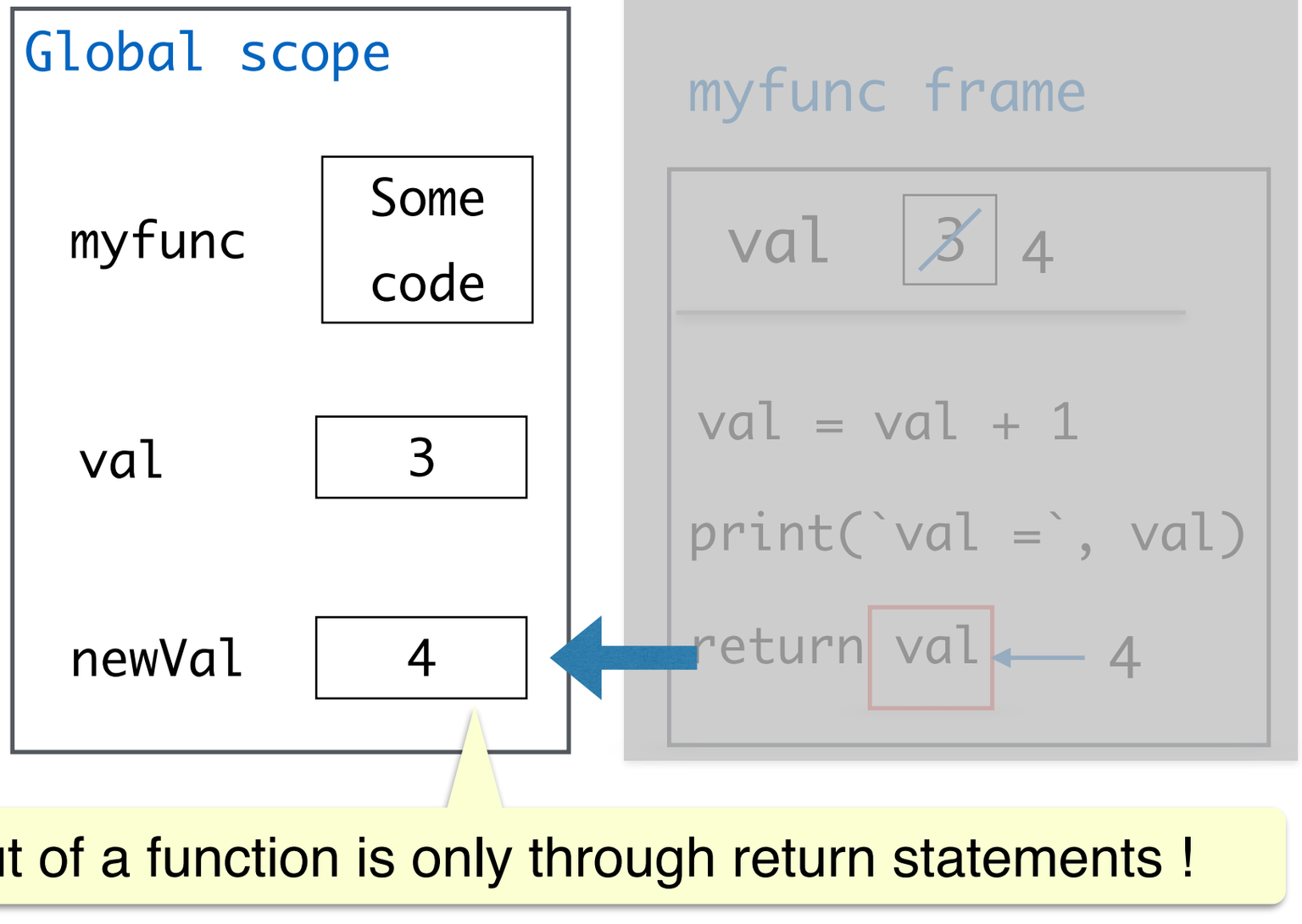


Variable Scope

```
def myfunc (val):
    val = val + 1
    print(`val =`, val)
    return val

val = 3
newVal = myfunc(val)
```

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



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

Acknowledgments

- These slides have been adapted from:
    - http://cs111.wellesley.edu/spring19 and
    - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/