# Lecture 10: Environment Diagrams, Lambda Functions

**Comp 102**

Forman Christian University

## Recap

## You Try

What is printed if you run this code as a file?

```python
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)

add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

## You Try

What is printed if you run this code as a file?

```
def add(x,y):
    return x+y
def mult(x,y):
    print(x*y)

add(1,2)
print(add(2,3))
mult(3,4)
print(mult(4,5))
```

## You Try

Fix the code that tries to write this function:

```
def is_triangular(n):
    """ n is an int > 0
    Returns True if n is triangular, i.e. equals a
       continued summation of natural numbers
       (1+2+3+...+k), False otherwise """
    total = 0
    for i in range(n):
        total += i
        if total == n:
            print(True)
    print(False)
```

# You Try

Fix the code that tries to write this function:

```
def is_triangular(n):
    """ n is an int > 0
    Returns True if n is triangular, i.e. equals a
       continued summation of natural numbers
       (1+2+3+...+k), False otherwise """
    total = 0
    for i in range(n):
        total += i
        if total == n:
            print(True)
    print(False)
```

**Bugs:** (1) `print` → `return`   (2) `range(n)` → `range(1, n+1)`

## Environment Diagrams

## Environment Diagrams

- A visual tool to track **program execution**

## Environment Diagrams

- A visual tool to track **program execution**
- Shows how Python manages:
  - **Variables** and their values
  - **Function calls** and frames
  - **Scope** (where variables are accessible)

## Environment Diagrams

- A visual tool to track **program execution**
- Shows how Python manages:
  - **Variables** and their values
  - **Function calls** and frames
  - **Scope** (where variables are accessible)
- Every program starts with a **Global Frame**

## Environment Diagrams

- A visual tool to track **program execution**
- Shows how Python manages:
  - **Variables** and their values
  - **Function calls** and frames
  - **Scope** (where variables are accessible)
- Every program starts with a **Global Frame**
- Each function call creates a **new frame**

## Global Variables

Variables created outside functions live in the Global Frame

```
x = 10
y = 20
z = x + y
```

# Global Variables

Variables created outside functions live in the Global Frame

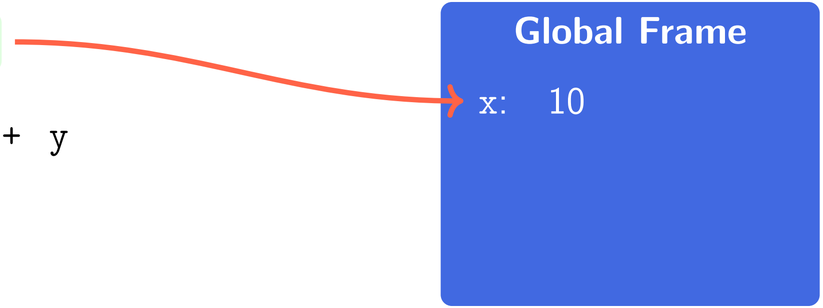
```
x = 10
y = 20
z = x + y
```

Global Frame

## Global Variables

Variables created outside functions live in the Global Frame

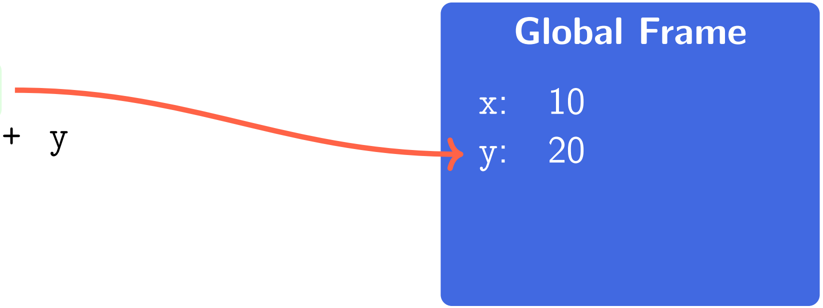
```
x = 10
y = 20
z = x + y
```



## Global Variables

Variables created outside functions live in the Global Frame

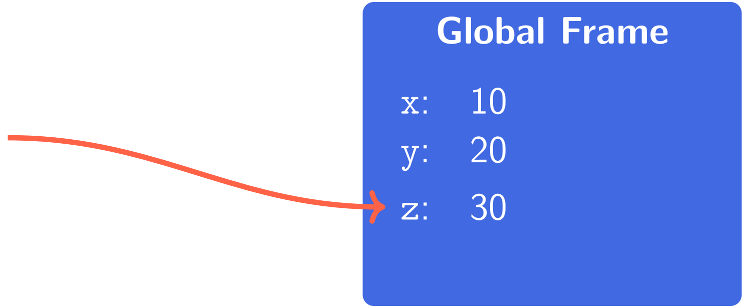
```
x = 10
y = 20
z = x + y
```



# Global Variables

Variables created outside functions live in the Global Frame

```
x = 10
y = 20
z = x + y
```



## Function Definitions

Function definitions create function objects in the Global Frame

```
def add(a, b):
    return a + b

def mult(a, b):
    return a * b
```

# Function Definitions

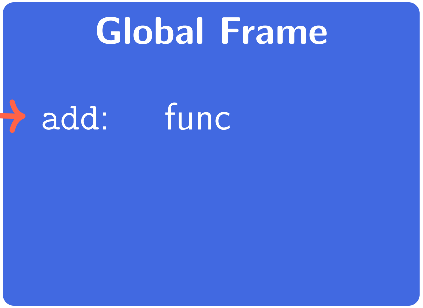
Function definitions create function objects in the Global Frame

```
def add(a, b):
    return a + b

def mult(a, b):
    return a * b
```

## Function Definitions

Function definitions create function objects in the Global Frame

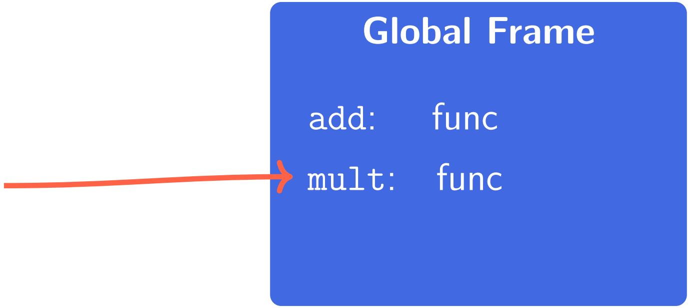
```
def add(a, b):
    return a + b

def mult(a, b):
    return a * b
```



# Function Definitions

Function definitions create function objects in the Global Frame

```python
def add(a, b):
    return a + b

def mult(a, b):
    return a * b
```



## Function Definitions

Function definitions create function objects in the Global Frame

```
def add(a, b):
    return a + b

def mult(a, b):
    return a * b
```

**Global Frame**

add: func

mult: func

*No execution yet!*

## Function Call Creates Frame

When you call a function, Python creates a new frame

```python
def add(a, b):
    return a + b

result = add(5, 3)
```

## Function Call Creates Frame

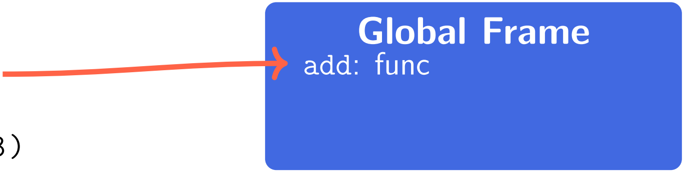
When you call a function, Python creates a new frame

```
def add(a, b):
    return a + b

result = add(5, 3)
```

# Function Call Creates Frame

When you call a function, Python creates a new frame

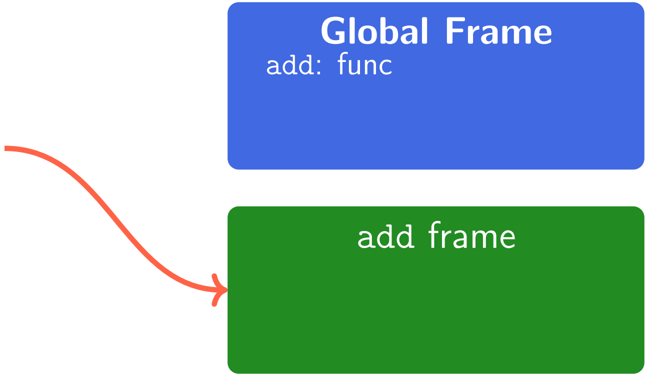
```
def add(a, b):
    return a + b

result = add(5, 3)
```



# Function Call Creates Frame

When you call a function, Python creates a new frame

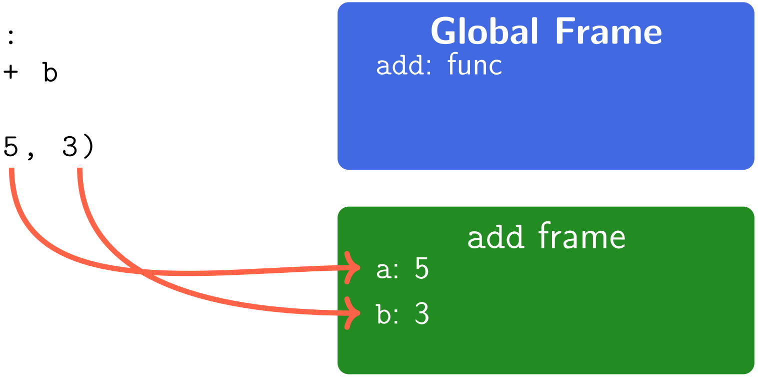
```
def add(a, b):
    return a + b

result = add(5, 3)
```



# Function Call Creates Frame

When you call a function, Python creates a new frame

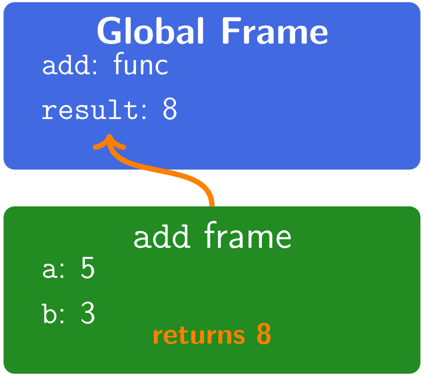
```
def add(a, b):
    return a + b

result = add(5, 3)
```



# Function Call Creates Frame

When you call a function, Python creates a new frame

```
def add(a, b):
    return a + b

result = add(5, 3)
```



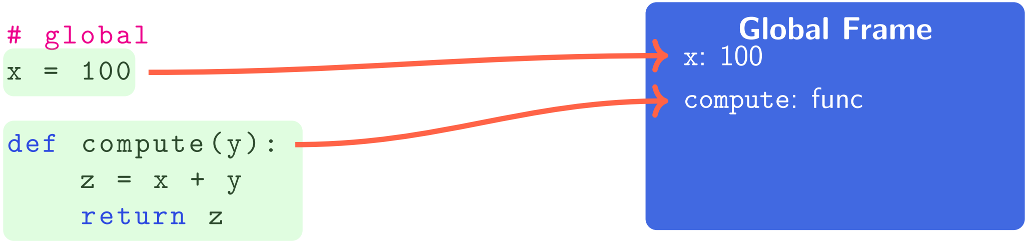
## Local vs Global Variables

Local variables exist only in their function's frame

```
# global
x = 100

def compute(y):
    z = x + y
    return z

result = compute(50)
```

# Local vs Global Variables

Local variables exist only in their function's frame



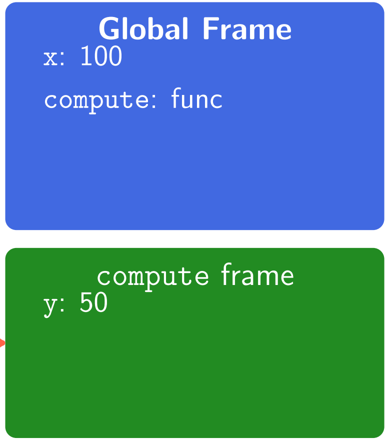
```
# global
x = 100

def compute(y):
    z = x + y
    return z
```

```
result = compute(50)
```

# Local vs Global Variables

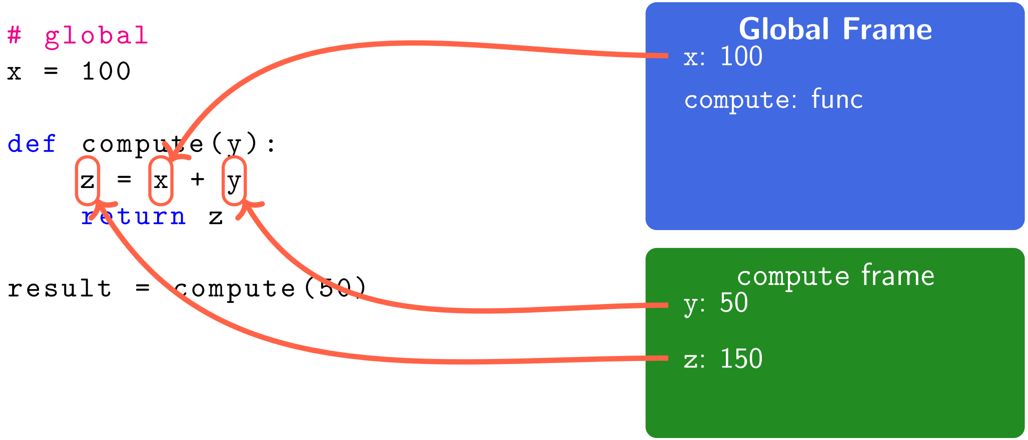
Local variables exist only in their function's frame

```
# global
x = 100

def compute(y):
    z = x + y
    return z

result = compute(50)
```



# Local vs Global Variables

Local variables exist only in their function's frame

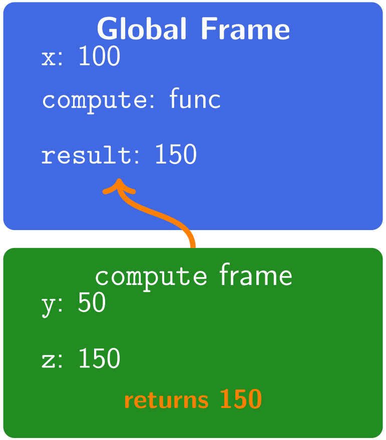


```
# global
x = 100

def compute(y):
    z = x + y
    return z

result = compute(50)
```

## Local vs Global Variables

Local variables exist only in their function's frame

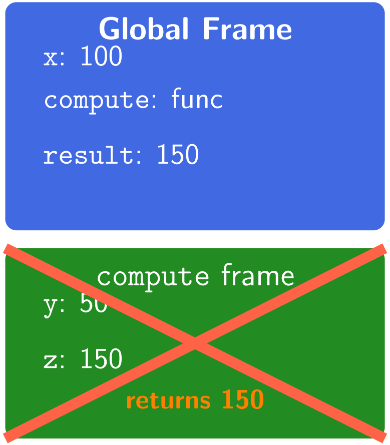
```
# global
x = 100

def compute(y):
    z = x + y
    return z

result = compute(50)
```



## Local vs Global Variables

Local variables exist only in their function's frame

```
# global
x = 100

def compute(y):
    z = x + y
    return z

result = compute(50)
```



## You Try

Draw the environment diagram:

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

y = square(4)
```

How many frames are created? What happens to them?

## You Try

Draw the environment diagram:

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

y = square(4)
```

How many frames are created? What happens to them?

**Global Frame**

square: func

y: 16

square frame

x: 4

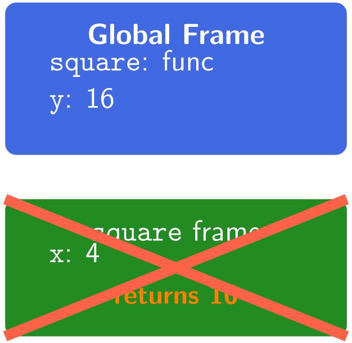
**returns 16**

# You Try

Draw the environment diagram:

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

y = square(4)
```

How many frames are created? What happens to them?



**2 frames** created:
Global + square

square frame is **destroyed** after return

# Multiple Function Calls

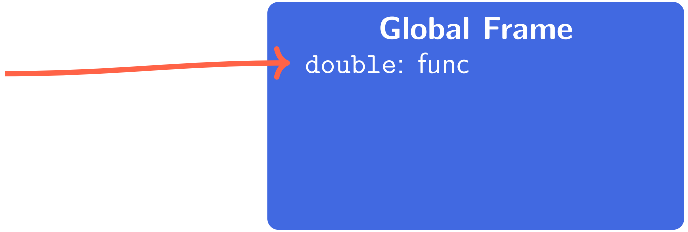
Each function call gets its own frame

```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```

# Multiple Function Calls

Each function call gets its own frame

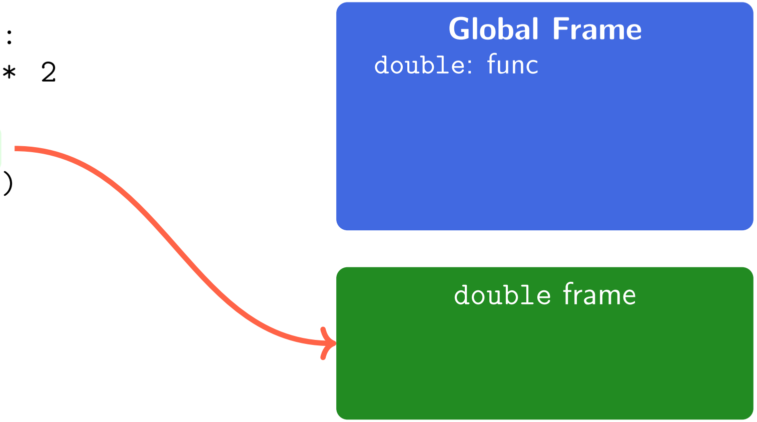
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

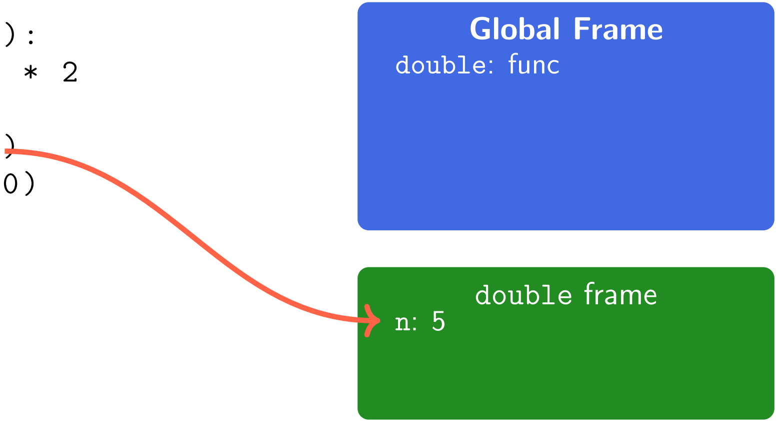
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

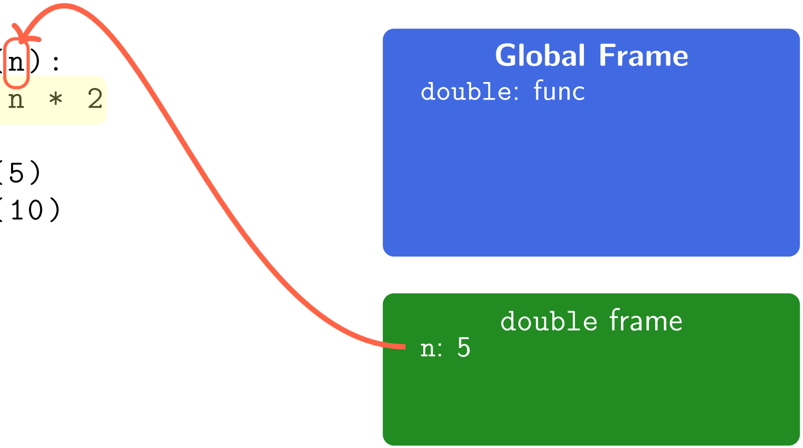
```python
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

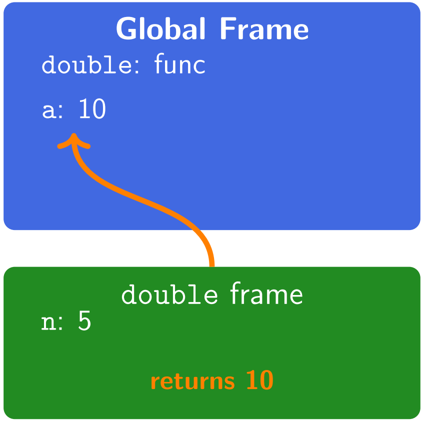
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



## Multiple Function Calls

Each function call gets its own frame

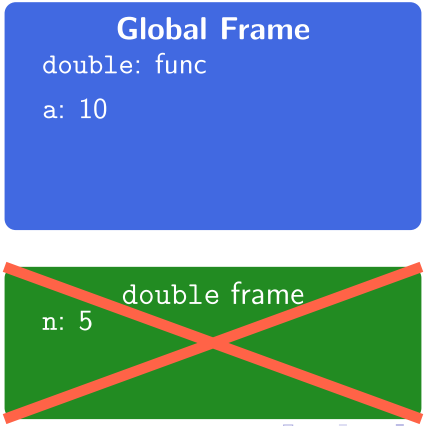
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

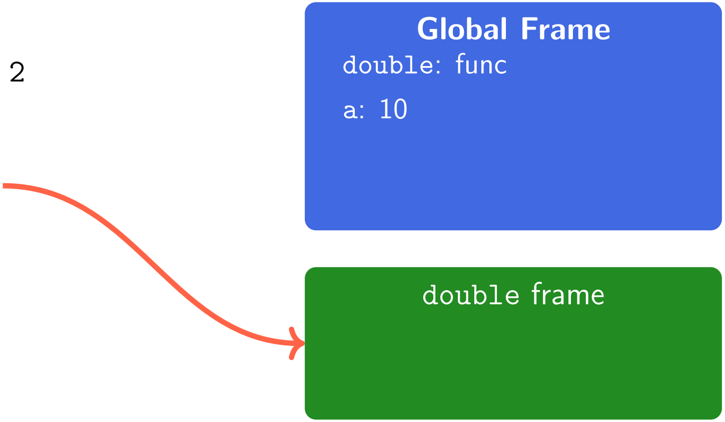
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

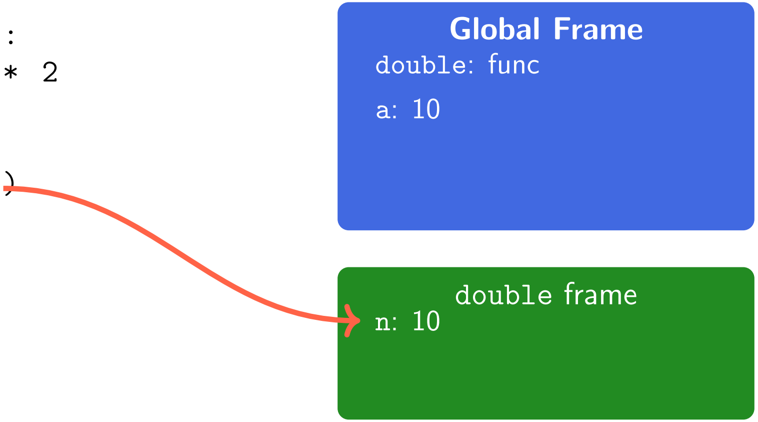
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

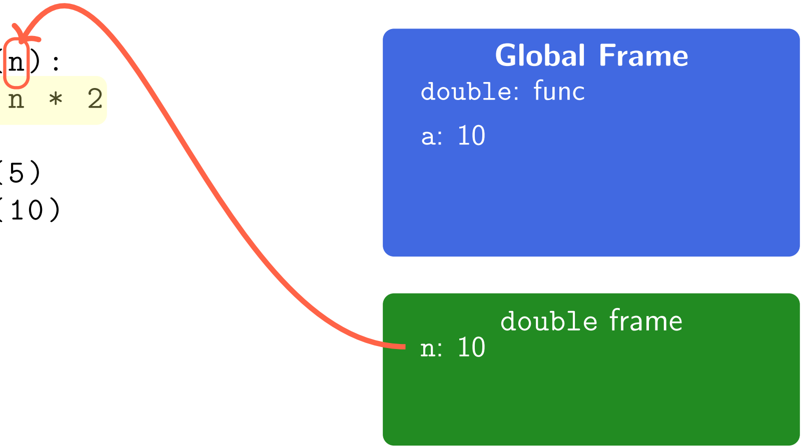
```python
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

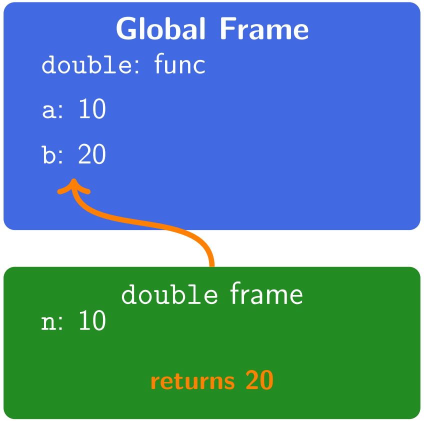
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

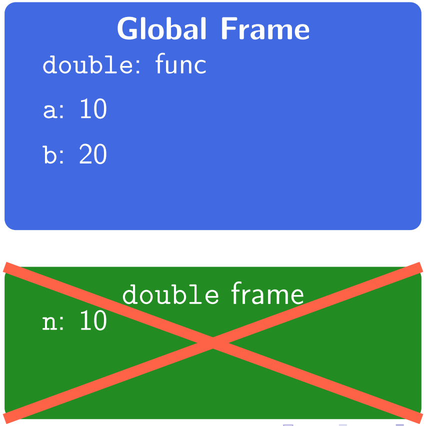
```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



# Multiple Function Calls

Each function call gets its own frame

```
def double(n):
    return n * 2

a = double(5)
b = double(10)
```



We've seen how Python creates and destroys frames for each function call.

We've seen how Python creates and destroys frames for each function call.

**What if the thing we pass into a function. . . is itself a function?**

## Higher Order Functions

## Higher Order Functions

- Functions are **first-class objects** in Python
  - Can be assigned to variables
  - Can be passed as arguments to other functions
  - Can be returned from functions

## Higher Order Functions

- Functions are **first-class objects** in Python
  - Can be assigned to variables
  - Can be passed as arguments to other functions
  - Can be returned from functions
- A **higher-order function** is a function that:
  - Takes one or more functions as arguments, **OR**
  - Returns a function as its result

## Functions as Variables

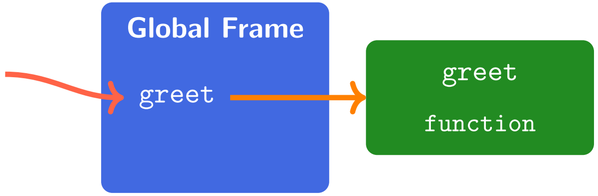
Functions can be stored in variables and called by different names

```
def greet(name):
    return "Hi, " + name

say_hi = greet

print(say_hi("Alice"))
print(greet("Bob"))
```

# Functions as Variables

Functions can be stored in variables and called by different names

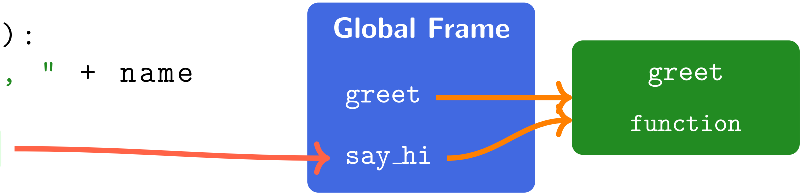
```python
def greet(name):
    return "Hi, " + name

say_hi = greet

print(say_hi("Alice"))
print(greet("Bob"))
```



## Functions as Variables

Functions can be stored in variables and called by different names

```
def greet(name):
    return "Hi, " + name

say_hi = greet

print(say_hi("Alice"))
print(greet("Bob"))
```



**Both names refer to the same function object!**

## Big Idea

Functions can be treated like **any other data**

Pass them around, store them, return them!

# Functions as Arguments

Functions can be passed as arguments to other functions

```
def double(x):
    return x * 2

def apply_twice(func, n):
    return func(func(n))

result = apply_twice(double, 3)
print(result)  # 12
```

## Functions as Arguments

Functions can be passed as arguments to other functions

```
def double(x):
    return x * 2

def apply_twice(func, n):
    return func(func(n))

result = apply_twice(double, 3)
print(result)  # 12
```

`apply_twice` takes a **function** as its first parameter!

## You Try

Write a function that meets these specs.

```
def apply(criteria,n):
    """
    inputs:
        - criteria is a func that takes in a number and
          returns a bool
        - n is an int
    output:
        - returns how many ints from 0 to n (inclusive)
          match the criteria (i.e. return True when run
          with criteria)
    """
```

## Solution

```
def apply(criteria,n):
    """
    * criteria: function that takes a number and returns a bool
    * n: an int
    Returns how many ints from 0 to n (inclusive) match the
    criteria (i.e. return True when run with criteria) """
    count = 0
    for i in range(n+1):
        if criteria(i):
            count += 1
    return count

def is_even(x):
    return x%2==0

print(apply(is_even,10))
```

## What are Anonymous Functions?

- Sometimes we need a **simple function** for one-time use

## What are Anonymous Functions?

- Sometimes we need a **simple function** for one-time use
- Creating a full `def` statement seems **wasteful**

## What are Anonymous Functions?

- Sometimes we need a **simple function** for one-time use
- Creating a full `def` statement seems **wasteful**
- **Anonymous functions** = functions without names

## What are Anonymous Functions?

- Sometimes we need a **simple function** for one-time use
- Creating a full `def` statement seems **wasteful**
- **Anonymous functions** = functions without names
- In Python, we create them using the `lambda` keyword

## What are Anonymous Functions?

- Sometimes we need a **simple function** for one-time use
- Creating a full `def` statement seems **wasteful**
- **Anonymous functions** = functions without names
- In Python, we create them using the `lambda` keyword
- Perfect for **short, simple operations** passed as arguments

## Anonymous Functions

- Sometimes don't want to name functions, especially simple ones.

```
def is_even(x):
    return x%2==0
```

## Anonymous Functions

- Sometimes don't want to name functions, especially simple ones.

```
def is_even(x):
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

# Anonymous Functions

- Sometimes don't want to name functions, especially simple ones.

```
def is_even(x):
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

**Body of lambda**

No return keyword

## Anonymous Functions

- Sometimes don't want to name functions, especially simple ones.

```
def is_even(x):
    return x%2==0
```

- Can use an **anonymous** procedure by using `lambda`

```
lambda x: x%2 == 0
```

**Body of lambda**

No return keyword

- `lambda` creates a function object, but simply does not bind name to it

# Anonymous Functions

- Function call with a named function:

```
apply(is_even, 10)
```

## Anonymous Functions

- Function call with a named function:

```
apply(is_even, 10)
```

- Function call with an anonymous function as parameter:

```
apply(lambda x: x%2 == 0, 10)
```

## Anonymous Functions

- Function call with a named function:

```
apply(is_even, 10)
```

- Function call with an anonymous function as parameter:

```
apply(lambda x: x%2 == 0, 10)
```

- `lambda` function is **one-time use**. It can't be reused because it has no name!

## You Try!

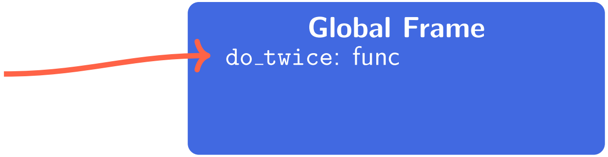
What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```

# You Try!

What does this print?

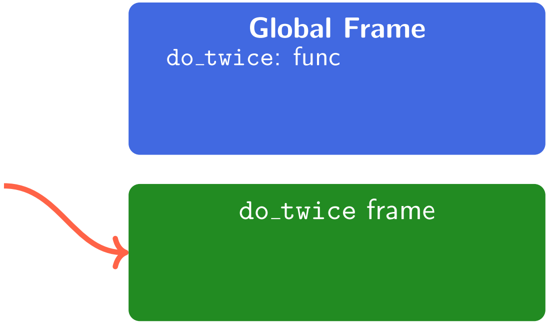
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



# You Try!

What does this print?

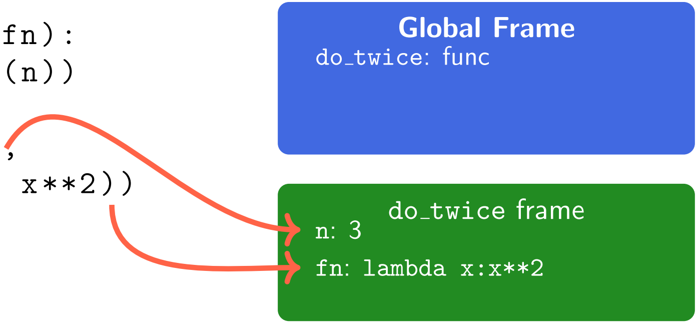
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
       lambda x: x**2))
```



## You Try!

What does this print?

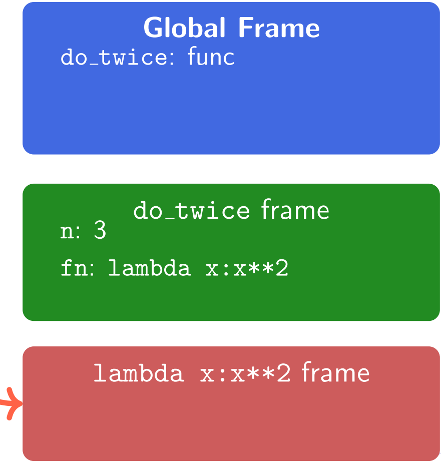
```python
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
          lambda x: x**2))
```



# You Try!

What does this print?

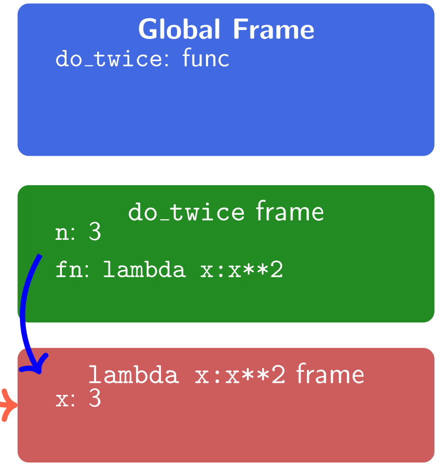
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



# You Try!

What does this print?

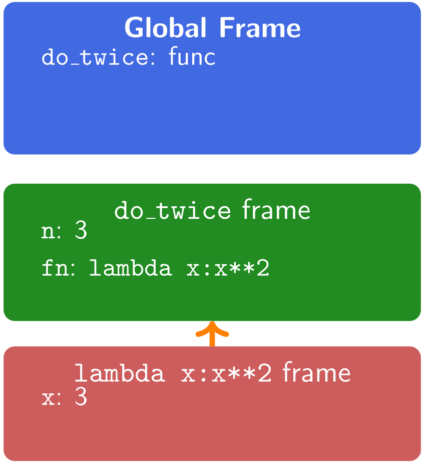
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



# You Try!

What does this print?

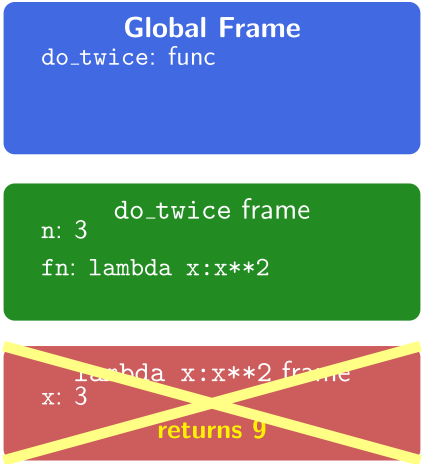
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



## You Try!

What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



## You Try!

What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```

**Global Frame**

do_twice: func

do_twice frame

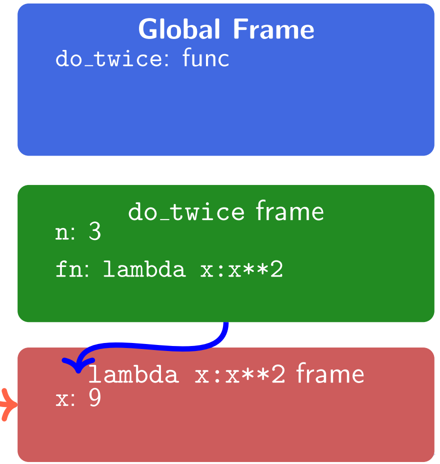
n: 3

fn: lambda x:x**2

## You Try!

What does this print?

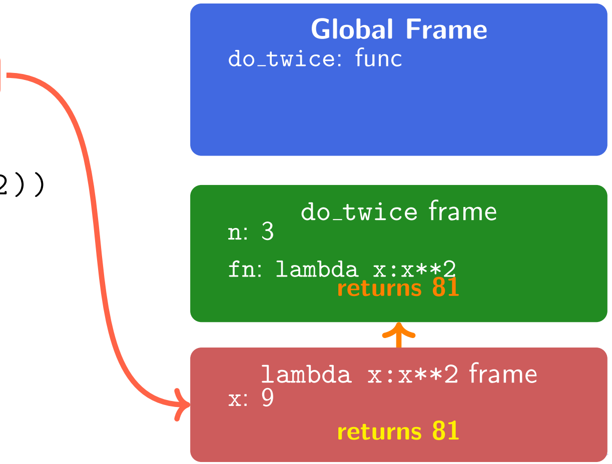
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
         lambda x: x**2))
```



# You Try!

What does this print?

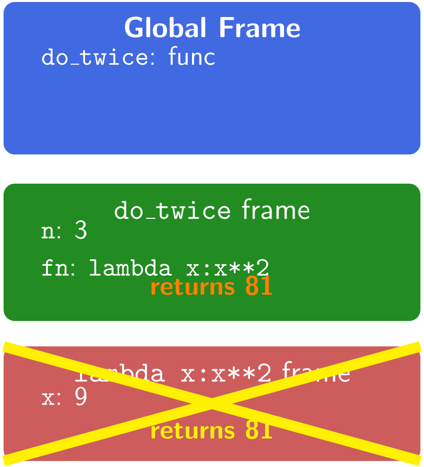
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



# You Try!

What does this print?

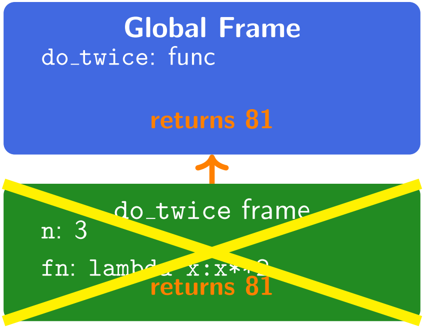
```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
         lambda x: x**2))
```



# You Try!

What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```



## You Try!

What does this print?

```
def do_twice(n, fn):
    return fn(fn(n))

print(do_twice(3,
        lambda x: x**2))
```

**Global Frame**

do_twice: func

**returns 81**

**print: 81**

## Common Lambda Pitfalls

- **When NOT to use lambdas:**
  - Complex logic requiring multiple statements
  - Functions that need documentation
  - Reusable functions used multiple times

## Common Lambda Pitfalls

- **When NOT to use lambdas:**
  - Complex logic requiring multiple statements
  - Functions that need documentation
  - Reusable functions used multiple times
- **Lambda limitations:**
  - Only **single expression** allowed (no statements!)
  - Cannot contain `return`, `assert`, `pass`, etc.

# Common Lambda Pitfalls

- **When NOT to use lambdas:**
  - Complex logic requiring multiple statements
  - Functions that need documentation
  - Reusable functions used multiple times
- **Lambda limitations:**
  - Only **single expression** allowed (no statements!)
  - Cannot contain `return`, `assert`, `pass`, etc.
- **Common syntax errors:**
  - Forgetting colon: `lambda x? x**2` ✗
  - Using return: `lambda x: return x**2` ✗

## Summary & Key Takeaways

**Environment Diagrams:** Track execution visually with frames

- Global Frame + Local Frames per function call
- Frames disappear when function returns

**Higher-Order Functions:** Functions as data

- Can be passed as arguments or returned

**Lambda Functions:** Anonymous functions

- Syntax: `lambda params: expression`
- For simple ops; use `def` for complex logic
- Useful with `map()`, `filter()`, `sorted()`

`lambda x, y: x + y` ≡ `def add(x, y): return x + y`

## Advanced: Closures

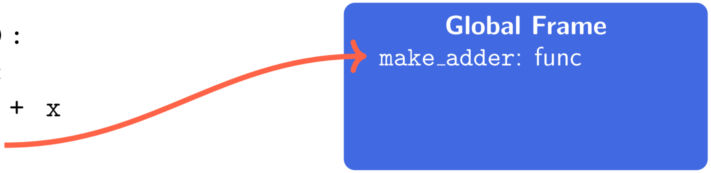
Functions can return other functions and "remember" their environment

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder

add5 = make_adder(5)
print(add5(10))  # 15
```

# Advanced: Closures

Functions can return other functions and "remember" their environment

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder
```

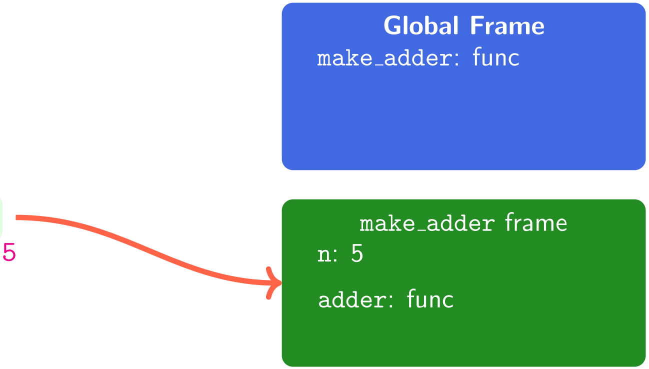


```
add5 = make_adder(5)
print(add5(10))  # 15
```

# Advanced: Closures

Functions can return other functions and "remember" their environment

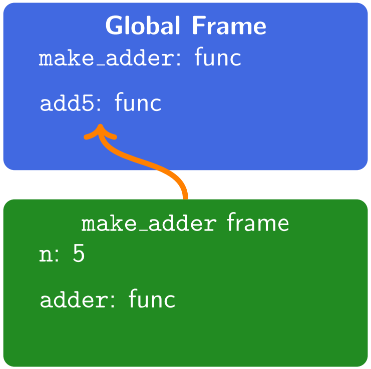
```
def make_adder(n):
    def adder(x):
        return n + x
    return adder

add5 = make_adder(5)
print(add5(10))  # 15
```



## Advanced: Closures

Functions can return other functions and "remember" their environment

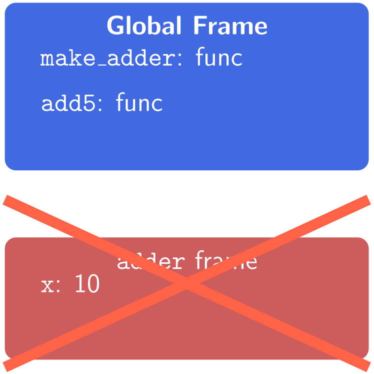
```
def make_adder(n):
    def adder(x):
        return n + x
    return adder

add5 = make_adder(5)
print(add5(10))  # 15
```



## Advanced: Closures

Functions can return other functions and "remember" their environment

```
def make_adder(n):
    def adder(x):
        return n + x
    return adder

add5 = make_adder(5)
print(add5(10))  # 15
```



## Advanced: Closures

Functions can return other functions and "remember" their environment

```python
def make_adder(n):
    def adder(x):
        return n + x
    return adder

add5 = make_adder(5)
print(add5(10))  # 15
```

**Global Frame**

make_adder: func

add5: func

adder frame

x: 10

**Key:** adder remembers n from parent frame!