Lecture 10: Environment Diagrams, Lambda Functions

Comp 102 (sec: A,B)

Fakhir Shaheen

Forman Christian University

Recap

You Try

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

```
1 def add(x,y):
2     return x+y
3 def mult(x,y):
4     print(x*y)
5
6 add(1,2)
7 print(add(2,3))
8 mult(3,4)
9 print(mult(4,5))
```

You Try

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

```
1 def add(x,y):
2     return x+y
3 def mult(x,y):
4     print(x*y)
5
6 add(1,2)
7 print(add(2,3))
8 mult(3,4)
9 print(mult(4,5))
```

Output: 5, 12, 20, None

You Try

Fix the code that tries to write this function:

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

You Try

Fix the code that tries to write this function:

```
1  def is_triangular(n):
2      """ n is an int > 0
3      Returns True if n is triangular, i.e. equals a
          continued summation of natural numbers
          (1+2+3+...+k), False otherwise """
4      total = 0
5      for i in range(n):
6          total += i
7          if total == n:
8              print(True)
9      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

```
1 x = 10
2 y = 20
3 z = x + y
```

Global Variables

Variables created outside functions live in the Global Frame

```
1 x = 10
2 y = 20
3 z = x + y
```



Global Variables

Variables created outside functions live in the Global Frame

```
1 x = 10
2 y = 20
3 z = x + y
```



Global Variables

Variables created outside functions live in the Global Frame

```
1 x = 10
2 y = 20
3 z = x + y
```



Global Variables

Variables created outside functions live in the Global Frame

```
1 x = 10
2 y = 20
3 z = x + y
```



Function Definitions

Function definitions create function objects in the Global Frame

```
1 def add(a, b):
2     return a + b
3
4 def mult(a, b):
5     return a * b
```

Function Definitions

Function definitions create function objects in the Global Frame

```
1 def add(a, b):
2     return a + b
3
4 def mult(a, b):
5     return a * b
```



Function Definitions

Function definitions create function objects in the Global Frame

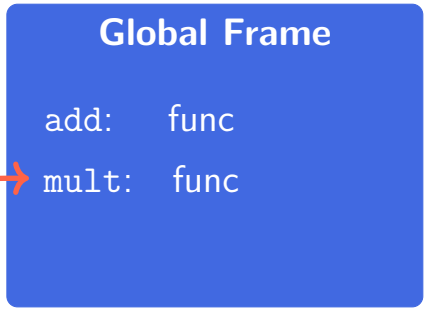
```
1 def add(a, b):
2     return a + b
3
4 def mult(a, b):
5     return a * b
```



Function Definitions

Function definitions create function objects in the Global Frame

```
1 def add(a, b):
2     return a + b
3
4 def mult(a, b):
5     return a * b
```



Function Definitions

Function definitions create function objects in the Global Frame

```
1 def add(a, b):
2     return a + b
3
4 def mult(a, b):
5     return a * b
```



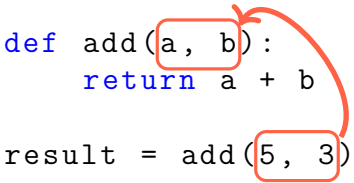
Function Call Creates Frame

When you call a function, Python creates a new frame

```
1 def add(a, b):
2     return a + b
3
4 result = add(5, 3)
```

Function Call Creates Frame

When you call a function, Python creates a new frame



Function Call Creates Frame

When you call a function, Python creates a new frame

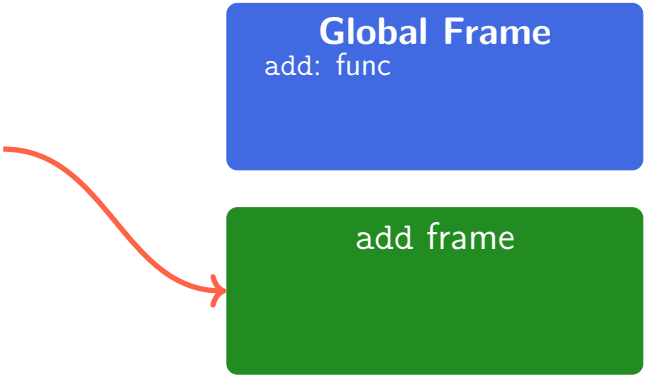
```
1 def add(a, b):
2     return a + b
3
4 result = add(5, 3)
```



Function Call Creates Frame

When you call a function, Python creates a new frame

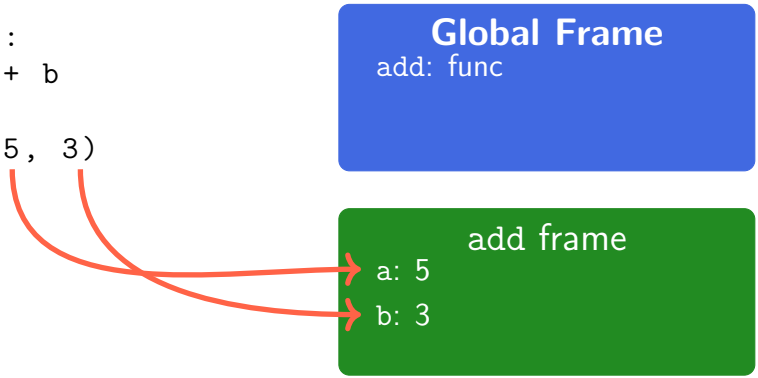
```
1 def add(a, b):
2     return a + b
3
4 result = add(5, 3)
```



Function Call Creates Frame

When you call a function, Python creates a new frame

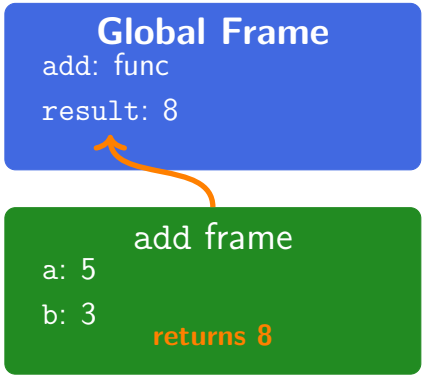
```
1 def add(a, b):
2     return a + b
3
4 result = add(5, 3)
```



Function Call Creates Frame

When you call a function, Python creates a new frame

```
1 def add(a, b):
2     return a + b
3
4 result = add(5, 3)
```



Local vs Global Variables

Local variables exist only in their function's frame

```
1 # global
2 x = 100
3
4 def compute(y):
5     z = x + y
6     return z
7
8 result = compute(50)
```

Local vs Global Variables

Local variables exist only in their function's frame

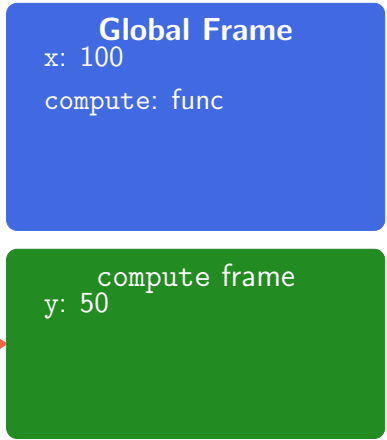
```
1  # global
2  x = 100
3
4  def compute(y):
5      z = x + y
6      return z
7
8  result = compute(50)
```



Local vs Global Variables

Local variables exist only in their function's frame

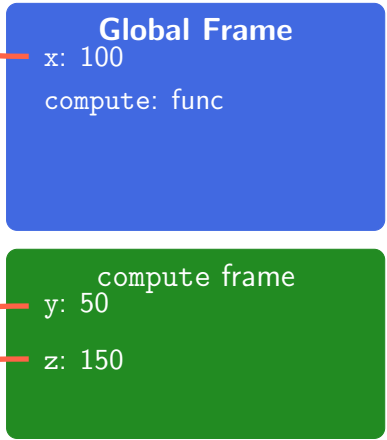
```
1  # global
2  x = 100
3
4  def compute(y):
5      z = x + y
6      return z
7
8  result = compute(50)
```



Local vs Global Variables

Local variables exist only in their function's frame

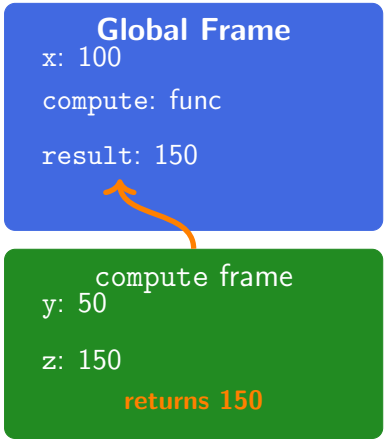
```
1  # global
2  x = 100
3
4  def compute(y):
5      z = x + y
6      return z
7
8  result = compute(50)
```



Local vs Global Variables

Local variables exist only in their function's frame

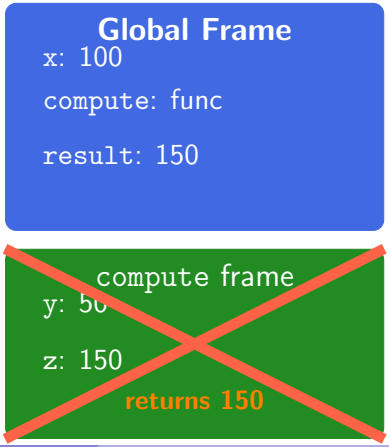
```
1  # global
2  x = 100
3
4  def compute(y):
5      z = x + y
6      return z
7
8  result = compute(50)
```



Local vs Global Variables

Local variables exist only in their function's frame

```
1  # global
2  x = 100
3
4  def compute(y):
5      z = x + y
6      return z
7
8  result = compute(50)
```



You Try

Draw the environment diagram:

```
1 def square(x):
2     return x * x
3
4 y = square(4)
```

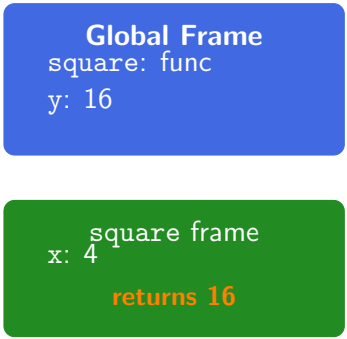
How many frames are created? What happens to them?

You Try

Draw the environment diagram:

```
1 def square(x):
2     return x * x
3
4 y = square(4)
```

How many frames are created? What happens to them?

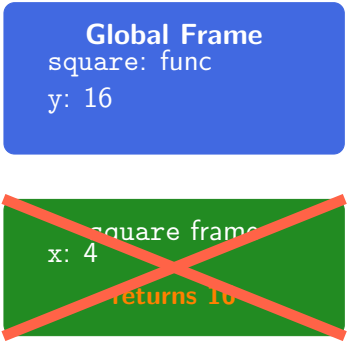


You Try

Draw the environment diagram:

```
1  def square(x):
2      return x * x
3
4  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

```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```

Multiple Function Calls

Each function call gets its own frame

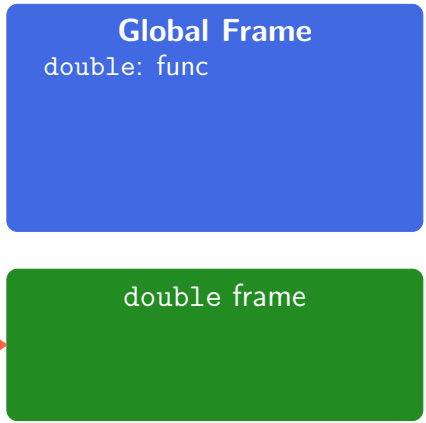
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

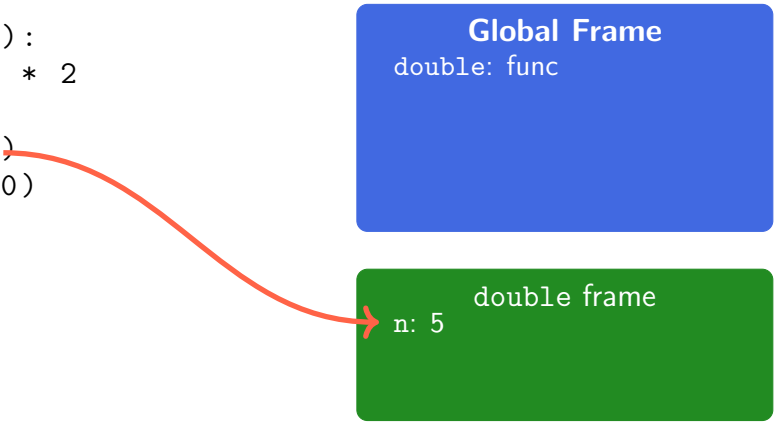
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

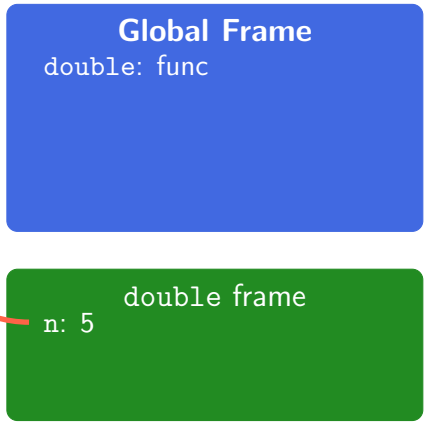
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

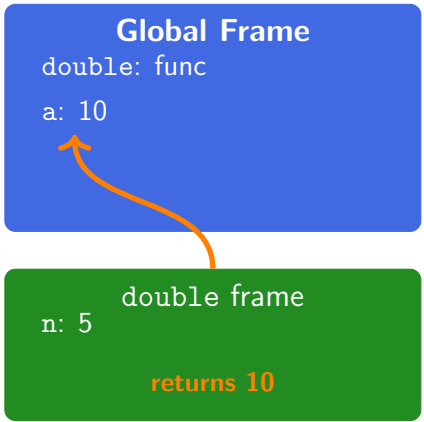
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

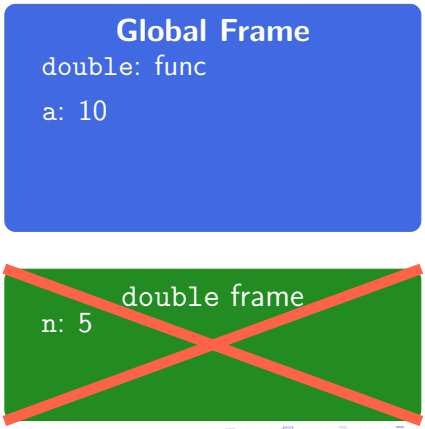
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

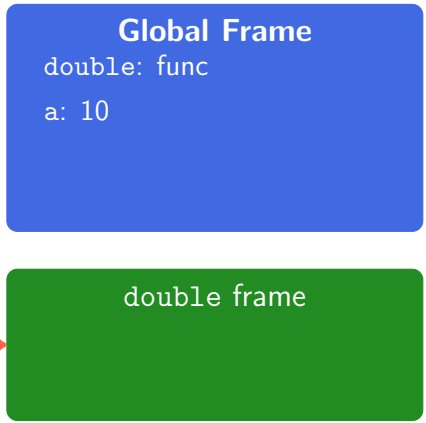
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

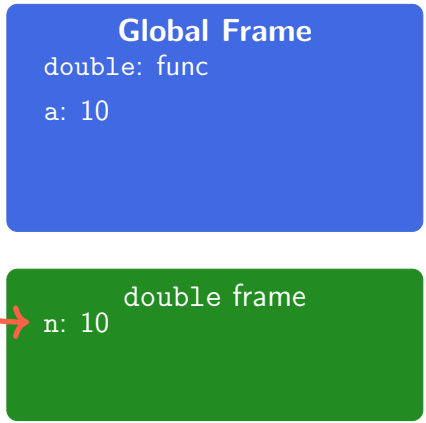
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

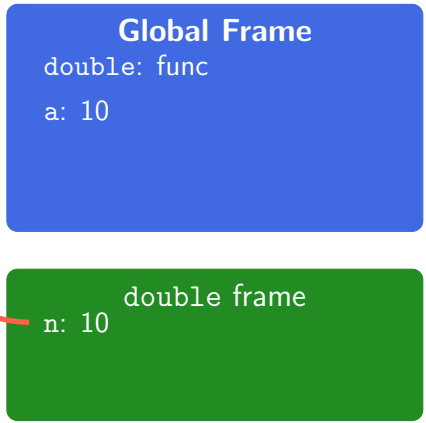
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

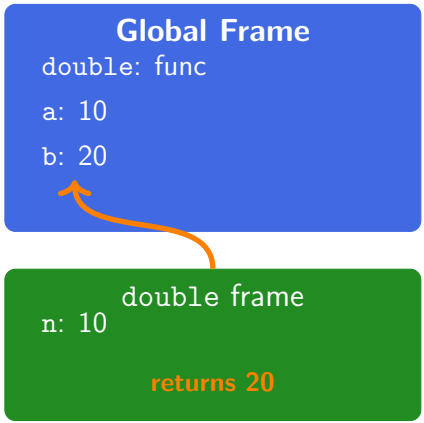
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

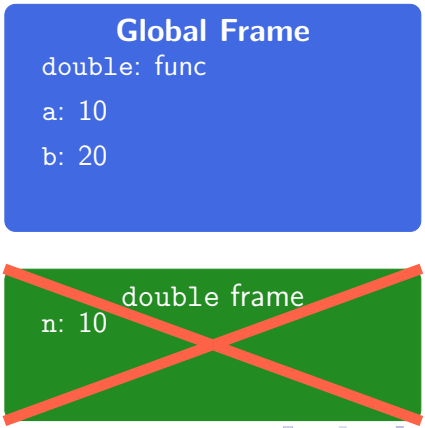
```
1 def double(n):
2     return n * 2
3
4 a = double(5)
5 b = double(10)
```



Multiple Function Calls

Each function call gets its own frame

```
1 def double(n):
2     return n * 2
3
4 a = double(5)
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

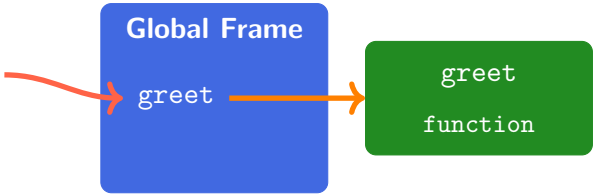
Functions can be stored in variables and called by different names

```
1 def greet(name):
2     return "Hi, " + name
3
4 say_hi = greet
5
6 print(say_hi("Alice"))
7 print(greet("Bob"))
```

Functions as Variables

Functions can be stored in variables and called by different names

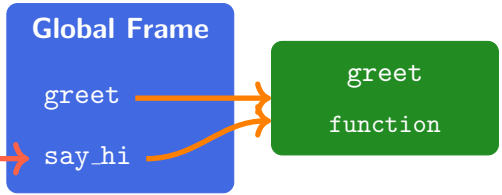
```
1 def greet(name):
2     return "Hi, " + name
3
4 say_hi = greet
5
6 print(say_hi("Alice"))
7 print(greet("Bob"))
```



Functions as Variables

Functions can be stored in variables and called by different names

```
1 def greet(name):
2     return "Hi, " + name
3
4 say_hi = greet
5
6 print(say_hi("Alice"))
7 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

```
1 def double(x):
2     return x * 2
3
4 def apply_twice(func, n):
5     return func(func(n))
6
7 result = apply_twice(double, 3)
8 print(result)  # 12
```

Functions as Arguments

Functions can be passed as arguments to other functions

```
1 def double(x):
2     return x * 2
3
4 def apply_twice(func, n):
5     return func(func(n))
6
7 result = apply_twice(double, 3)
8 print(result)  # 12
```

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

You Try

Write a function that meets these specs.

```
1 def apply(criteria,n):
2     """
3     inputs:
4         - criteria is a func that takes in a number and
             returns a bool
5         - n is an int
6     output:
7         - returns how many ints from 0 to n (inclusive)
             match the criteria (i.e. return True when run
             with criteria)
8     """
```

Solution

```
1  def apply(criteria,n):
2      """
3      * criteria: function that takes a number and returns a bool
4      * n: an int
5      Returns how many ints from 0 to n (inclusive) match the
6      criteria (i.e. return True when run with criteria) """
7      count = 0
8      for i in range(n+1):
9          if criteria(i):
10             count += 1
11     return count
12
13 def is_even(x):
14     return x%2==0
15
16 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.

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

Anonymous Functions

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

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

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

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

Anonymous Functions

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

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

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

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

Body of lambda
No return keyword

Anonymous Functions

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

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

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

```
1 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:

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

Anonymous Functions

- Function call with a named function:

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

- Function call with an anonymous function as parameter:

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

Anonymous Functions

- Function call with a named function:

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

- Function call with an anonymous function as parameter:

```
1 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?

```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```

You Try!

What does this print?

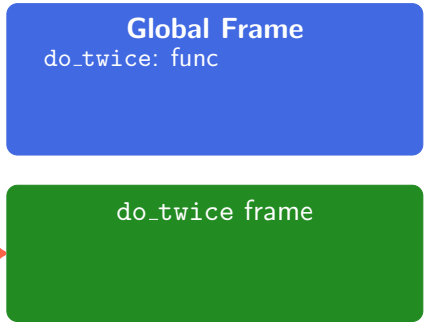
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

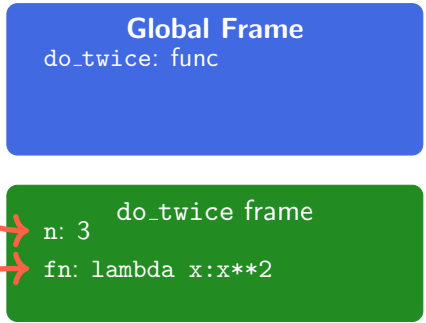
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

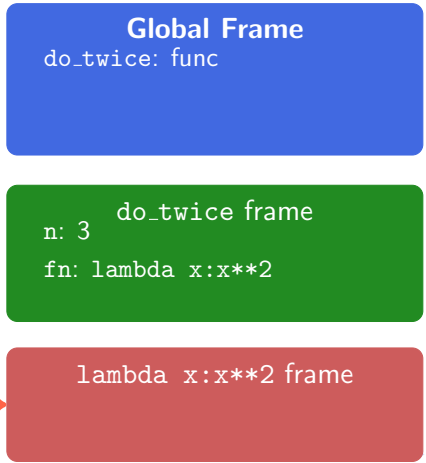
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

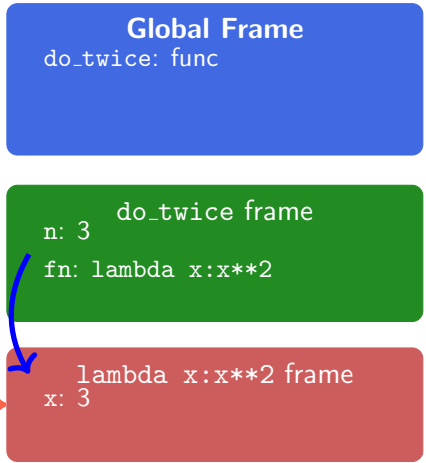
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

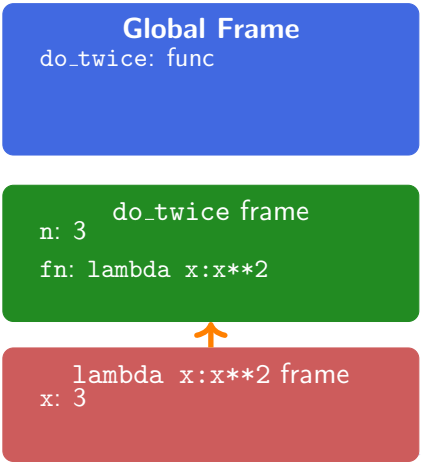
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

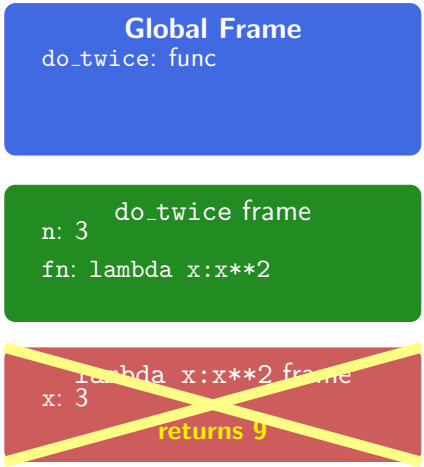
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

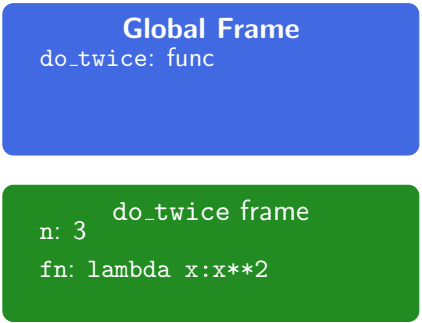
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

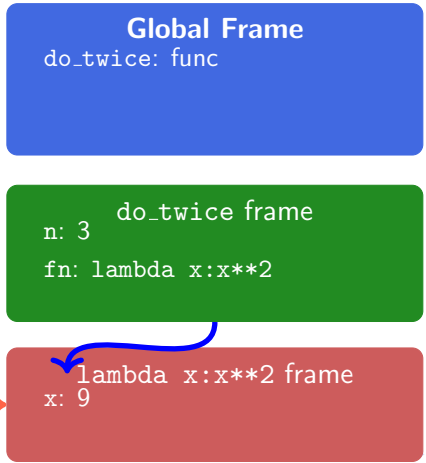
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

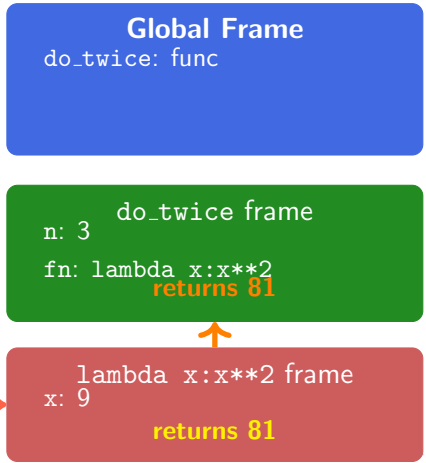
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

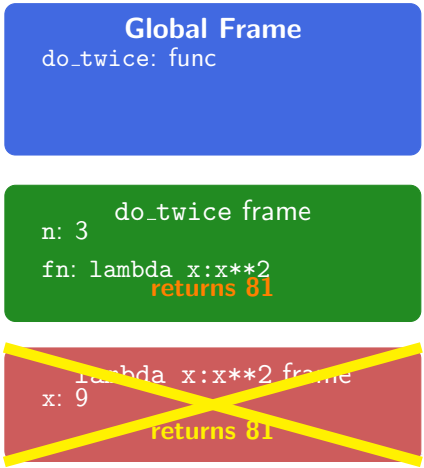
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

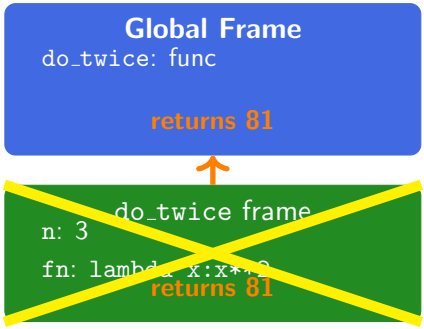
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

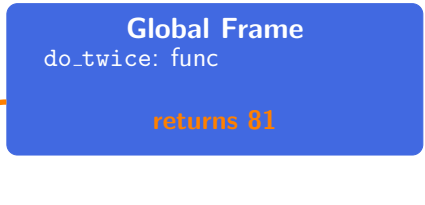
```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



You Try!

What does this print?

```
1 def do_twice(n, fn):
2     return fn(fn(n))
3
4 print(do_twice(3,
5         lambda x: x**2))
```



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

A function can **create** a new function and **return** it

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```

Advanced: Closures

A function can **create** a new function and **return** it

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```

1. **A function inside a function:** adder is created each time make_adder runs

Advanced: Closures

A function can **create** a new function and **return** it

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```

1. **A function inside a function:** `adder` is created each time `make_adder` runs
2. **It is returned:** `return adder` hands back the function itself, so `add5` is a function

Advanced: Closures

A function can **create** a new function and **return** it

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```

1. **A function inside a function:** `adder` is created each time `make_adder` runs
2. **It is returned:** `return adder` hands back the function itself, so `add5` is a function
3. **It uses `n`:** but `n` lives in `make_adder`'s frame, and that frame is **destroyed** after the return. So how can `add5(10)` be 15?

Advanced: Closures

A function can **create** a new function and **return** it

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))   # 15
```

1. **A function inside a function:** `adder` is created each time `make_adder` runs
2. **It is returned:** `return adder` hands back the function itself, so `add5` is a function
3. **It uses `n`:** but `n` lives in `make_adder`'s frame, and that frame is **destroyed** after the return. So how can `add5(10)` be 15?

Closure

A function that **remembers the variables** of the function it was created in, even after that function has **finished**. Here, `adder` remembers `n`.

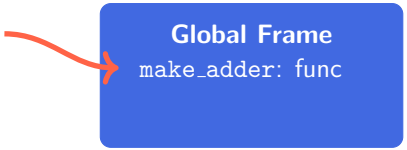
How a Closure Works

```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```

Draw the environment diagram line by line and **watch** `n`: where does it live?

How a Closure Works

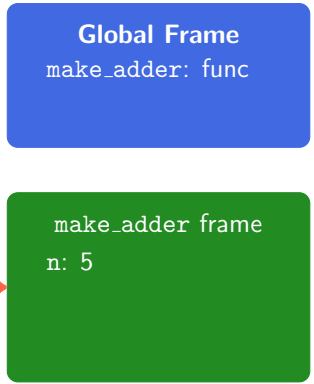
```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))  # 15
```



Step 1: `def` creates the function `make_adder`. Its body has **not run yet**, so there is no `adder` yet.

How a Closure Works

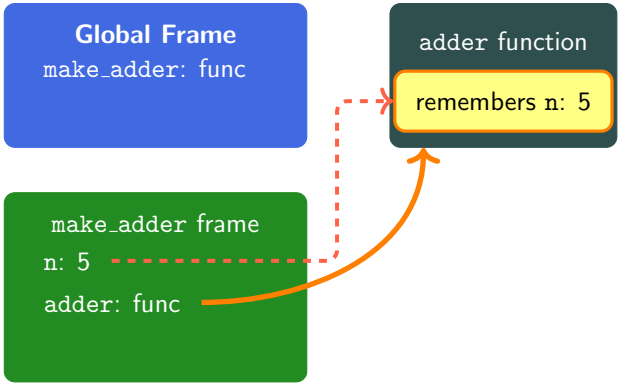
```
1 def make_adder(n):
2     def adder(x):
3         return n + x
4     return adder
5
6 add5 = make_adder(5)
7 print(add5(10))   # 15
```



Step 2: The call `make_adder(5)` creates a frame. The parameter `n` gets the value 5.

How a Closure Works

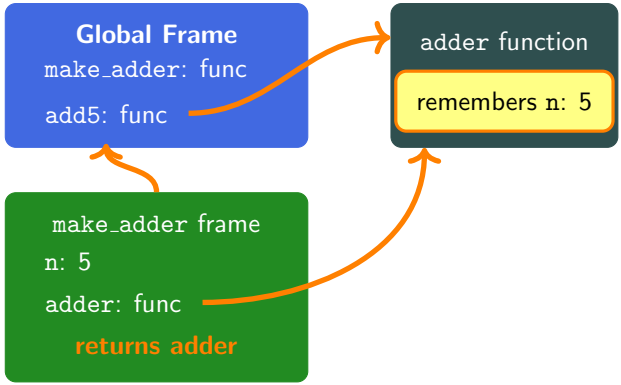
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 3: Now the inner `def` runs: a **new function** is created, named `adder` in this frame. Its body uses `n`, so it **remembers** this frame's `n`.

How a Closure Works

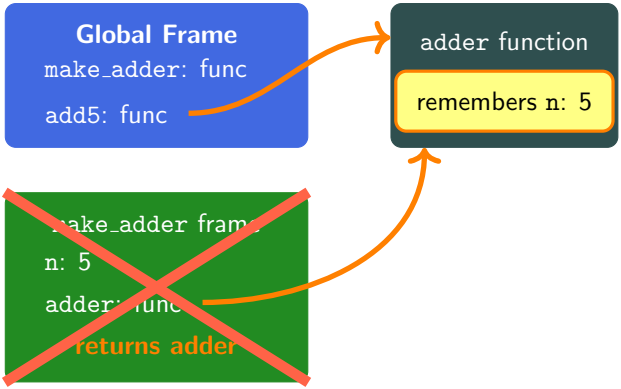
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 4: `return adder` hands the function back *(no brackets, so it is not called)*. In the Global Frame, `add5` is now a name for it.

How a Closure Works

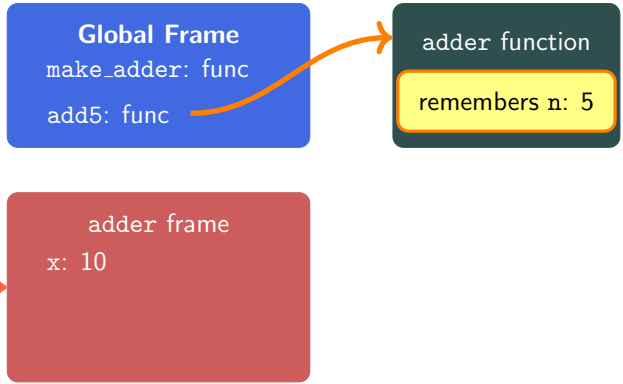
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 5: `make_adder` has returned: its frame is **destroyed**. The function lives on as add5, and it still remembers `n`: 5.

How a Closure Works

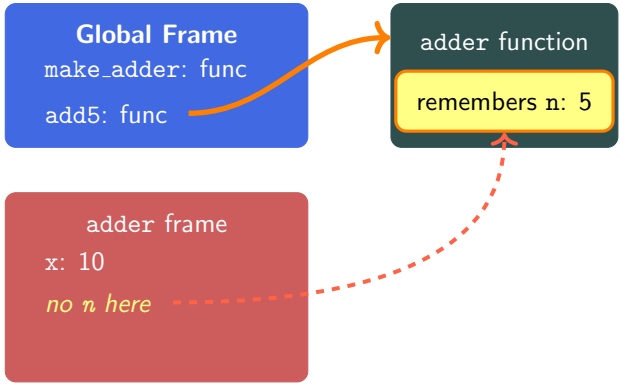
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 6: The call `add5(10)` runs that function: a new adder frame, with `x: 10`.

How a Closure Works

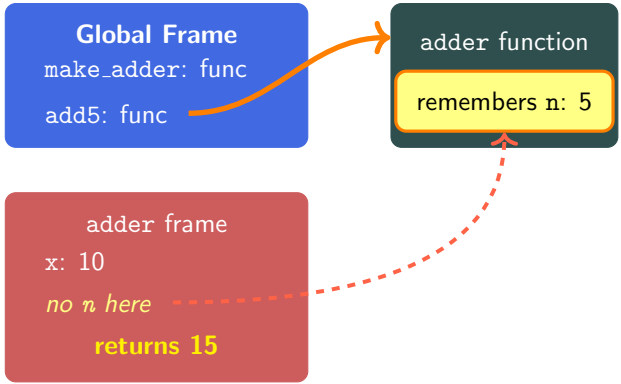
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 7: `return n + x`: Python finds `x` in the `adder` frame. There is no `n` there, so Python looks in **what the function remembers** before it tries the Global Frame.

How a Closure Works

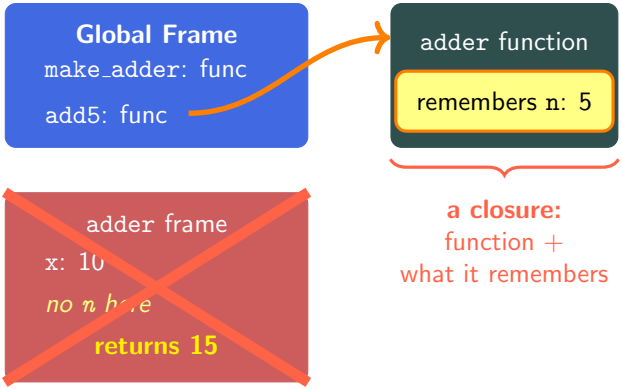
```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 8: 5 + 10 is 15. The function returns it, and `print` shows 15.

How a Closure Works

```
1  def make_adder(n):
2      def adder(x):
3          return n + x
4      return adder
5
6  add5 = make_adder(5)
7  print(add5(10))   # 15
```



Step 9: This frame is destroyed too. The function and the `n` it remembers stay: call add5 again and it still adds 5.

You Try

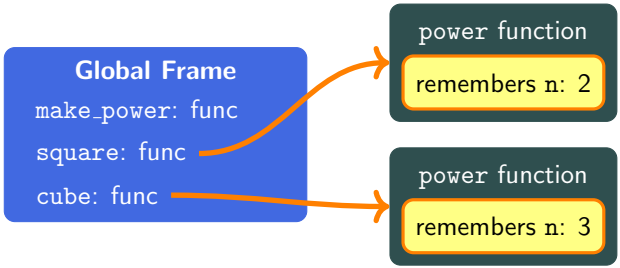
What does this print? What does each function remember?

```
1 def make_power(n):
2     def power(x):
3         return x ** n
4     return power
5
6 square = make_power(2)
7 cube = make_power(3)
8 print(square(5))
9 print(cube(2))
```

You Try

What does this print? What does each function remember?

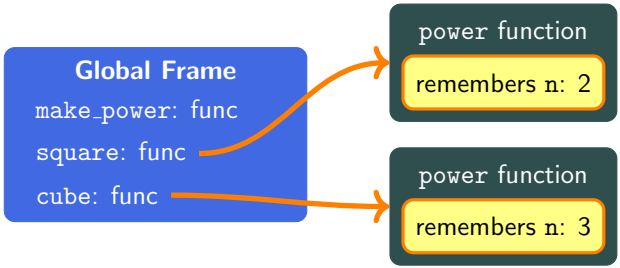
```
1  def make_power(n):
2      def power(x):
3          return x ** n
4      return power
5
6  square = make_power(2)
7  cube = make_power(3)
8  print(square(5))
9  print(cube(2))
```



You Try

What does this print? What does each function remember?

```
1  def make_power(n):
2      def power(x):
3          return x ** n
4      return power
5
6  square = make_power(2)
7  cube = make_power(3)
8  print(square(5))
9  print(cube(2))
```



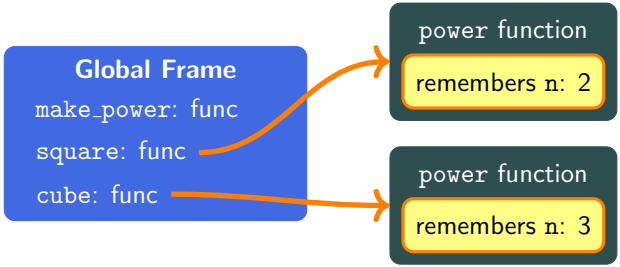
square(5): x is 5, n is 2 → 5 ** 2 → 25

cube(2): x is 2, n is 3 → 2 ** 3 → 8

You Try

What does this print? What does each function remember?

```
1  def make_power(n):
2      def power(x):
3          return x ** n
4      return power
5
6  square = make_power(2)
7  cube = make_power(3)
8  print(square(5))
9  print(cube(2))
```



square(5): x is 5, n is 2 → 5 ** 2 → 25

cube(2): x is 2, n is 3 → 2 ** 3 → 8

Key: each call to `make_power` creates a **new** function with its **own** `n`.
One maker, many functions!

Exit Ticket

Today's question *(2 minutes)*

Let `f = lambda x:  x * 3`

(a) What value is `f(4)`?

(b) When a function is called, Python creates a new _____ to hold its local variables. (one word)

COMP 102 — Exit Ticket #10

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.