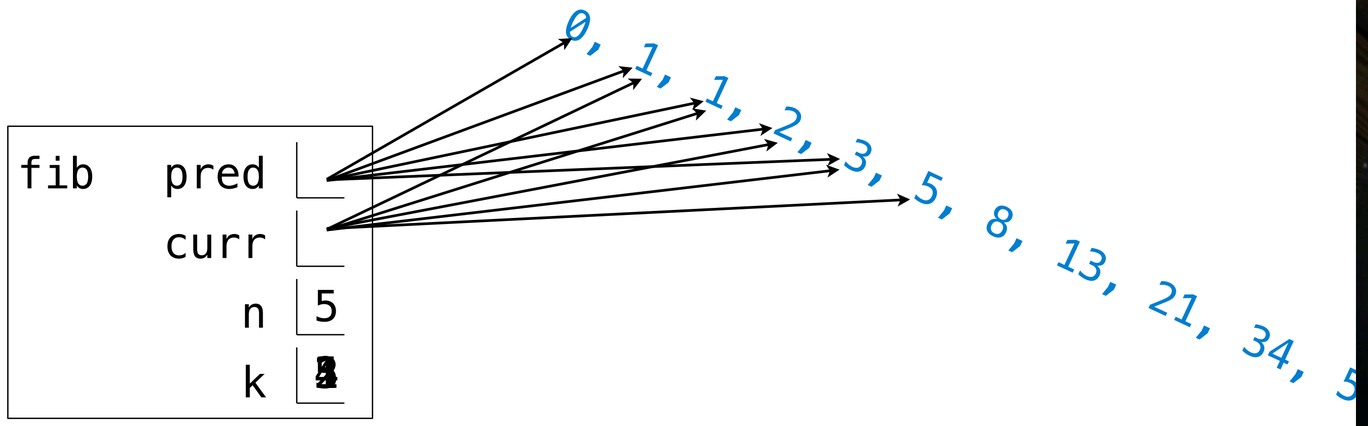
Higher-Order Functions

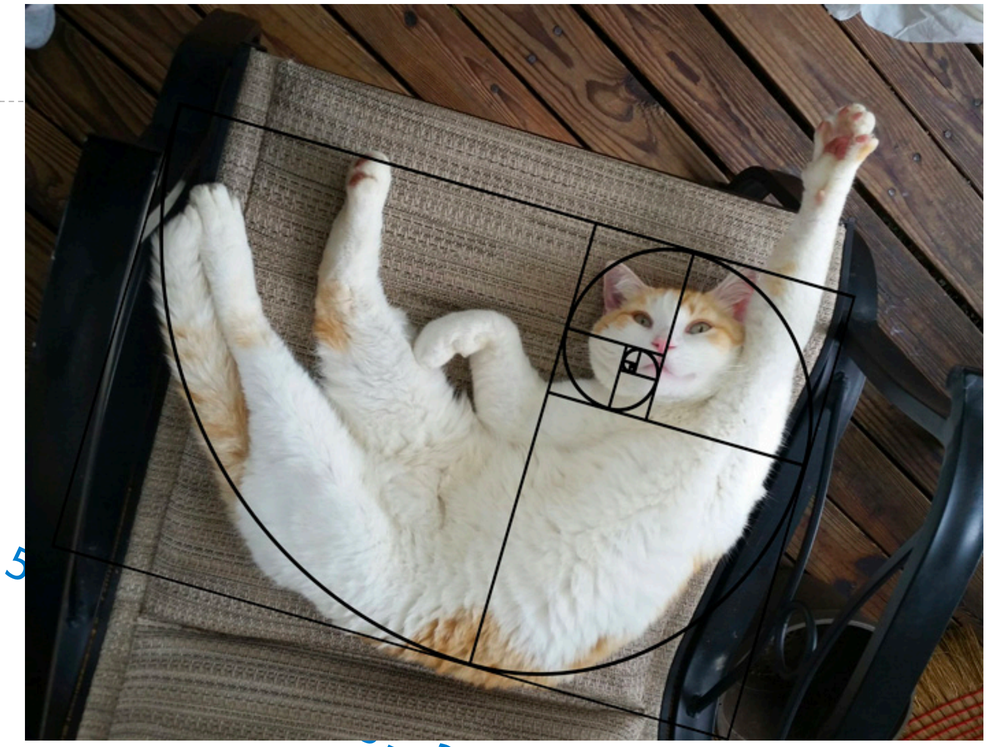
Announcements

- Hog is released!
  - Checkpoint 1: Due Thursday, July 1
  - Due: Wednesday, July 7
    - Submit a day early to get +1 EC point
  - You can work with a partner
- No lecture Monday – Juneteenth
  - Additional lecture content will be released on Friday
  - Lab 2 due date will be extended to next **Wednesday**
- Sign up for tutoring if you want to but haven't yet links.cs61a.org/tutorial-signup
- Feedback on Lab party would be appreciated – post on the Lab party Ed thread!
- Summer 2021 will have the same L&S CS declaration policy as Fall 2020 and Spring 2021
  - Details: https://piazza.com/class/hyq0br1u3kx7dg?cid=15211
  - Details: https://edstem.org/us/courses/5163/discussion/496330

Iteration Example

The Fibonacci Sequence



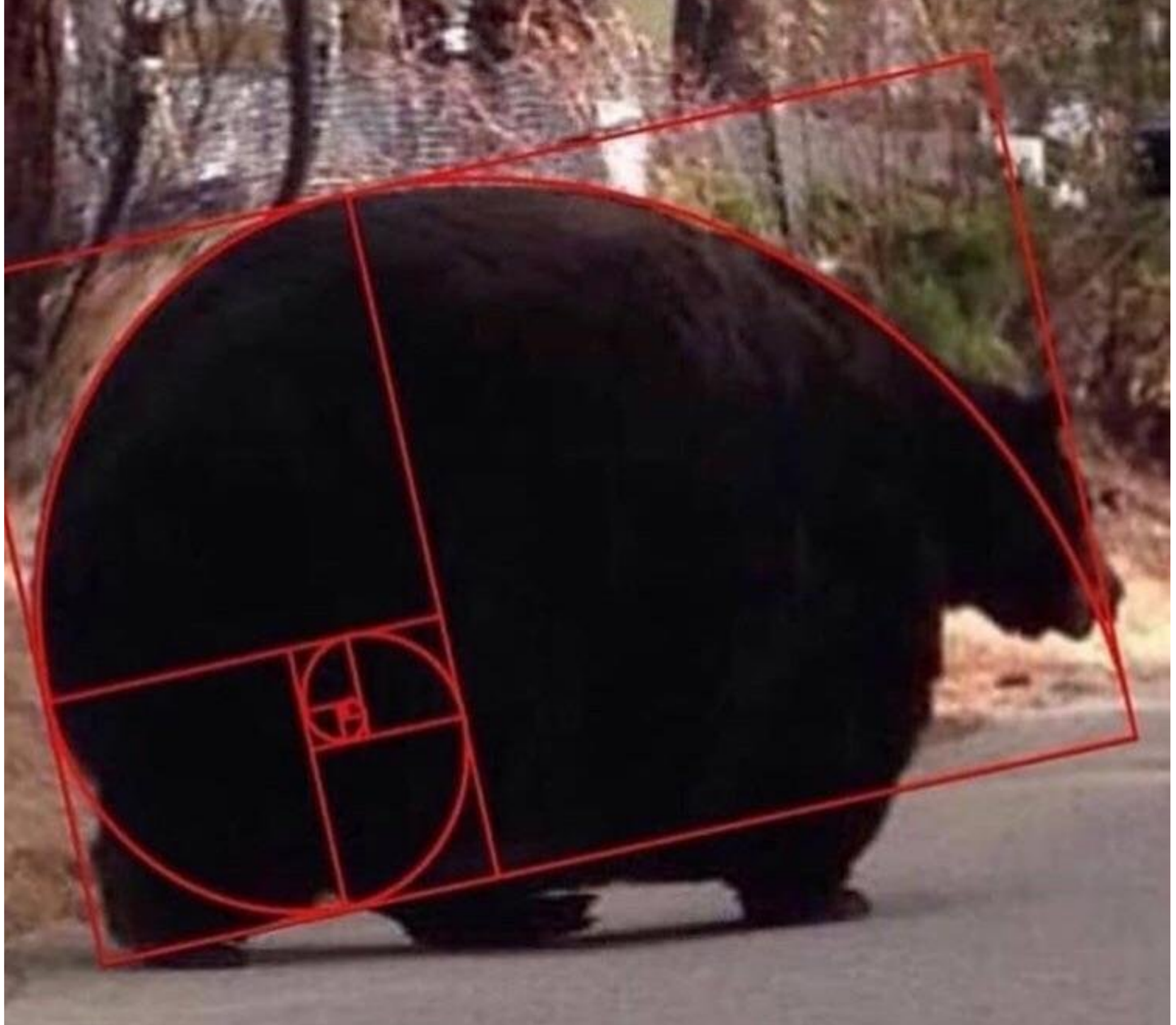


, 377, 610, 987

```
def fib(n):
    """Compute the nth Fibonacci number, for N >= 1."""
    pred, curr = 0, 1   # 0th and 1st Fibonacci numbers
    k = 1               # curr is the kth Fibonacci number
    while k < n:
        pred, curr = curr, pred + curr
        k = k + 1
    return curr
```

The next Fibonacci number is the sum of the current one and its predecessor

Go Bears!



Designing Functions

Describing Functions

A function's *domain* is the set of all inputs it might possibly take as arguments.

A function's *range* is the set of output values it might possibly return.

A pure function's *behavior* is the relationship it creates between input and output.

```
def square(x):
    """Return X * X."""
```

x is a number

square returns a non-negative real number

square returns the square of x

A Guide to Designing Function

Give each function exactly one job, but make it apply to many related situations

```
>>> round(1.23)
1
```

```
>>> round(1.23, 1)
1.2
```

```
>>> round(1.23, 0)
1
```

```
>>> round(1.23, 5)
1.23
```

Don’t repeat yourself (DRY): Implement a process just once, but execute it many times

Higher-Order Functions

Higher-Order Functions

A higher-order function is:

1. A function that takes in a function as an argument

   and / or

2. A function that returns another function

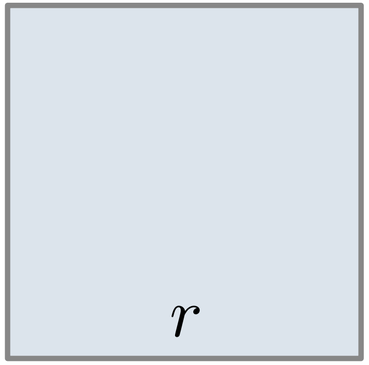
Higher-order functions allow us to design functions by expressing general patterns of computation.

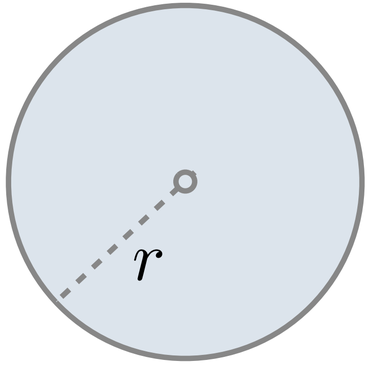
Generalization

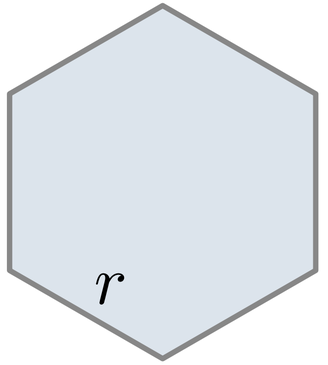
Generalizing Patterns with Arguments

Regular geometric shapes relate length and area.

Shape:







Area:

$1 \cdot r^2$ $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Finding common structure allows for shared implementation

(Demo)

Generalizing Over Computational Processes

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^{5} \frac{8}{(4k-3)\cdot(4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

(Demo)

Summation Example

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (*not called "term"*)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

0 + 1 + 8 + 27 + 64 + 125

The function bound to term gets called here

Calculating Pi

(Demo)

Functions as Return Values

(Demo)

Locally Defined Functions

Functions defined within other function bodies are bound to names in a local frame

A function that returns a function

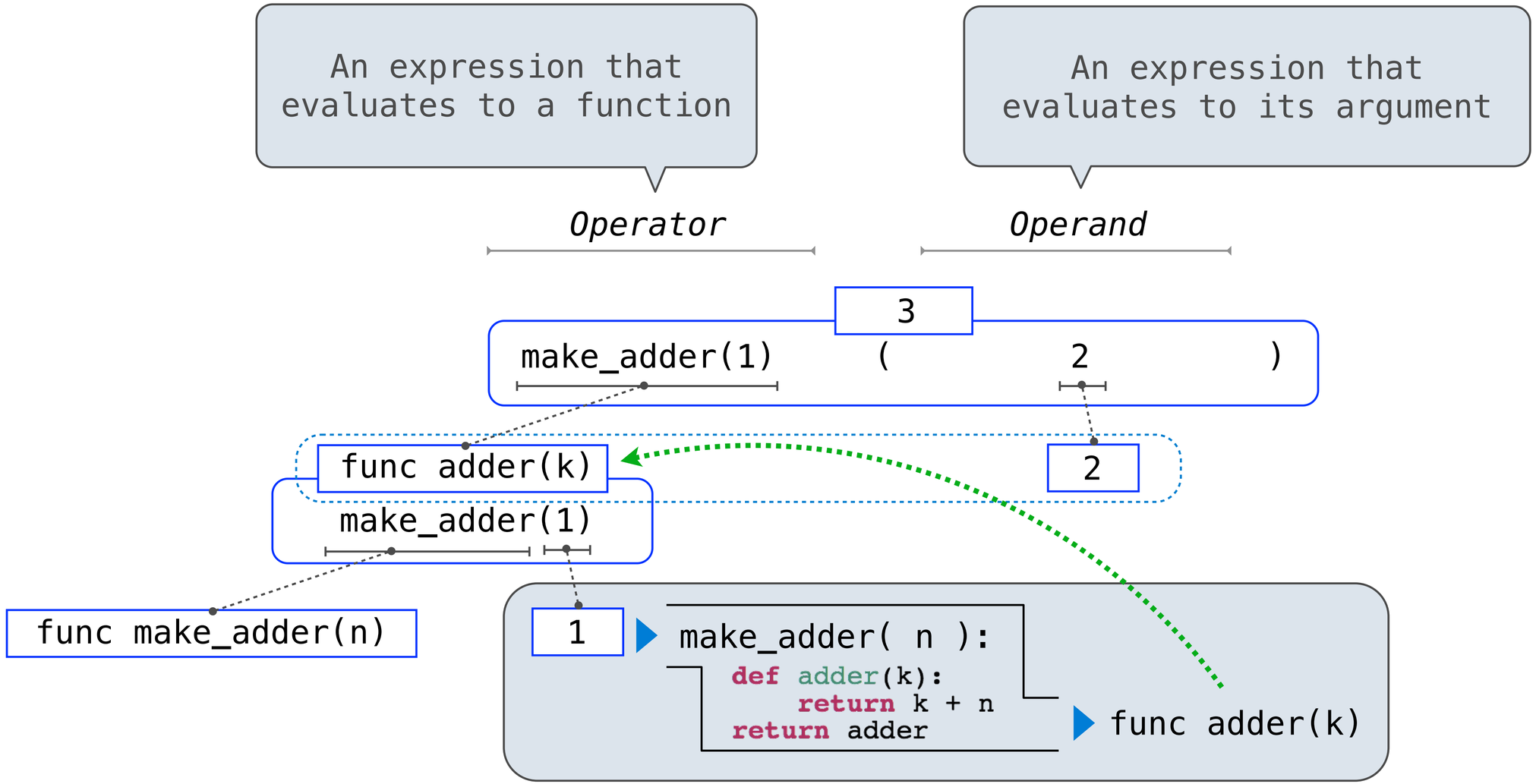
```
def make_adder(n):
    """Return a function that takes one argument k and returns k + n.

    >>> add_three = make_adder(3)
    >>> add_three(4)
    7
    """
    def adder(k):
        return k + n
    return adder
```

The name add_three is bound to a function

A def statement within another def statement

Can refer to names in the enclosing function

Call Expressions as Operator Expressions



Why do we use Higher-Order Functions?

A higher-order function is a function that takes in a function as an argument or function that returns another function.

Key concept: Functions can be manipulated as values in Python.

Higher-Order Functions:

- Express general methods of computation
- Remove repetition from programs

Lambda Expressions

(Demo)

Lambda Expressions

```
>>> x = 10
>>> square = x * x
>>> square = lambda x: x * x
>>> square(4)
16
```

An expression: this one evaluates to a number

Also an expression: evaluates to a function

Important: No "return" keyword!

A function
with formal parameter x
that returns the value of "x * x"

Must be a single expression

Lambda expressions are not common in Python, but important in general

Lambda expressions in Python cannot contain statements at all!

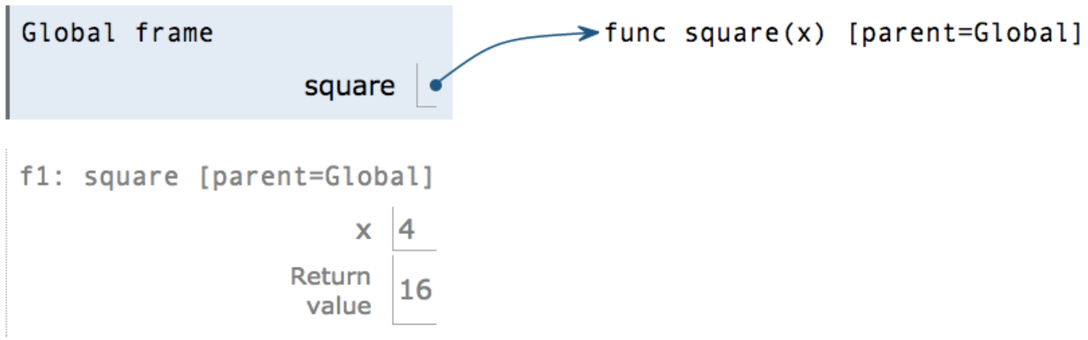
Lambda Expressions Versus Def Statements

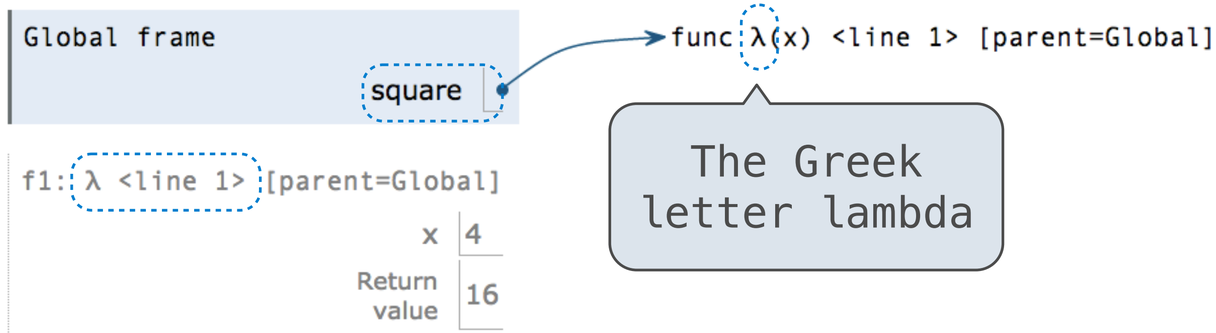
```
def square(x):
    return x * x
```

VS

```
square = lambda x: x * x
```

- Both create a function with the same domain, range, and behavior.
- Both bind that function to the name square.
- Only the def statement gives the function an intrinsic name, which shows up in environment diagrams but doesn't affect execution (unless the function is printed).





Return

Return Statements

A return statement completes the evaluation of a call expression and provides its value:

f(x) for user-defined function f: switch to a new environment; execute f's body

return statement within f: switch back to the previous environment; f(x) now has a value

Only one return statement is ever executed while executing the body of a function

```
def end(n, d):
    """Print the final digits of N in reverse order until D is found.

    >>> end(3467, 4)
    7
    6
    4
    """
    while n > 0:
        last, n = n % 10, n // 10
        print(last)
        if d == last:
            return None
```

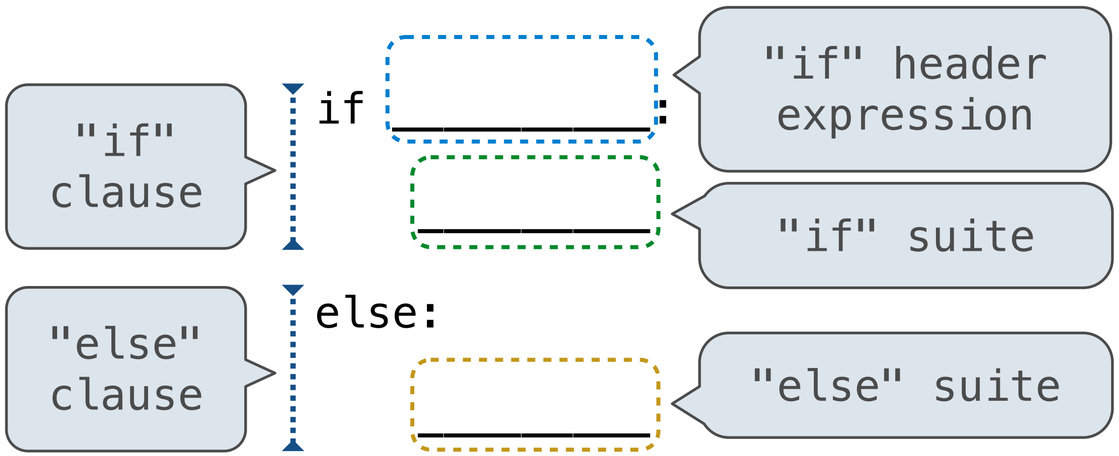
(Demo)

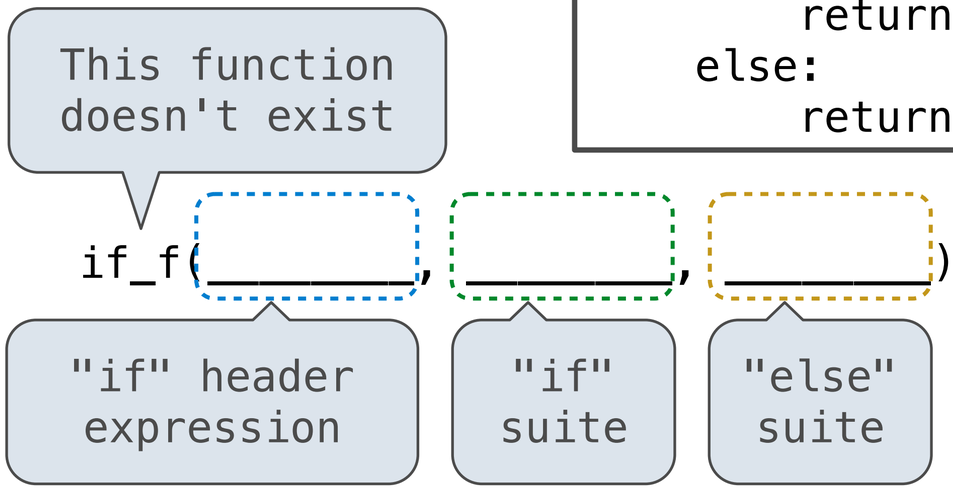
Control

If Statements and Call Expressions

Let's try to write a function that does the same thing as an if statement.

```
def if_f(c, t, f):
    if c:
        return t
    else:
        return f
```





Execution Rule for Conditional Statements:

Each clause is considered in order.

1. Evaluate the header's expression (if present).
2. If it is a true value (or an else header), execute the suite & skip the remaining clauses.

(Demo)

Evaluation Rule for Call Expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator to the arguments that are the values of the operands