Lecture 4

Higher - order functions
Asserts

Some slides were borrowed from John DeNero

Midterm Date

A: Week 5, Thursday
B: Week 6, Tuesday
C: Does not matter

Overall idea

- **Want**: is the ability to build **abstractions** by assigning names to common patterns and then to work in terms of the names directly.

- **Need**: to construct functions that can *accept* other functions as arguments or *return* functions as values.

- Functions that manipulate functions are called *higher-order* functions.

Nested Definitions (demo)

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


add3 = make_adder(3)
print(add3(9))
```

What will be printed?

A: 3

B: 9

C: 12

D: It does not make sense

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
```

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

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

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

```
make_adder(1)     (          2          )
```

Call Expressions as Operator Expressions

Operator

```
make_adder(1)     (          2          )
```

Call Expressions as Operator Expressions

Operator		*Operand*	
`make_adder(1)`	`(`	`2`	`)`

Call Expressions as Operator Expressions

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

An expression that evaluates to a function

Operator *Operand*

```
make_adder(1)    (    2    )
```

Call Expressions as Operator Expressions

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

An expression that evaluates to a function

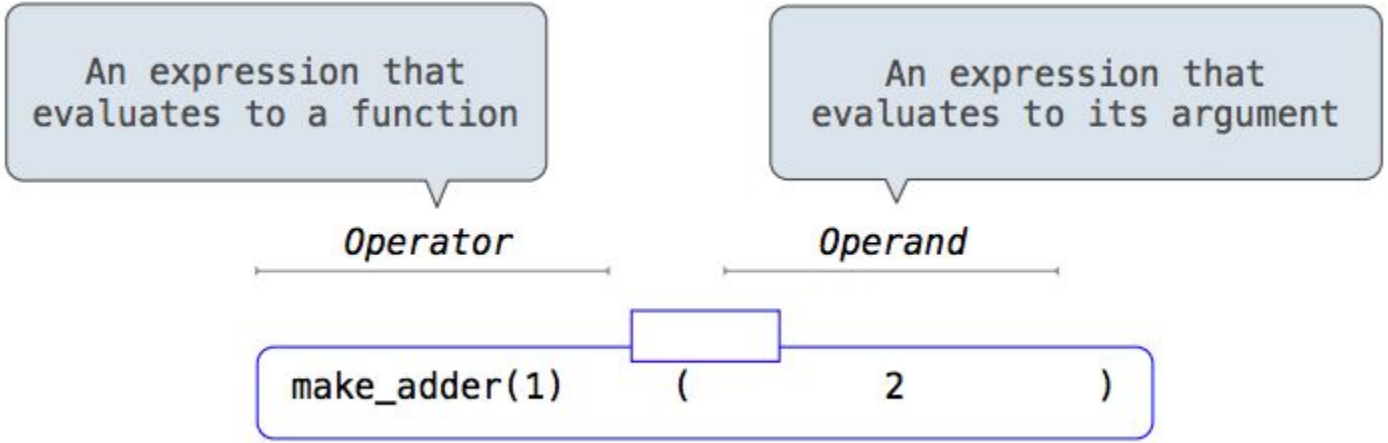
An expression that evaluates to its argument

Operator *Operand*

```
make_adder(1)    (    2    )
```

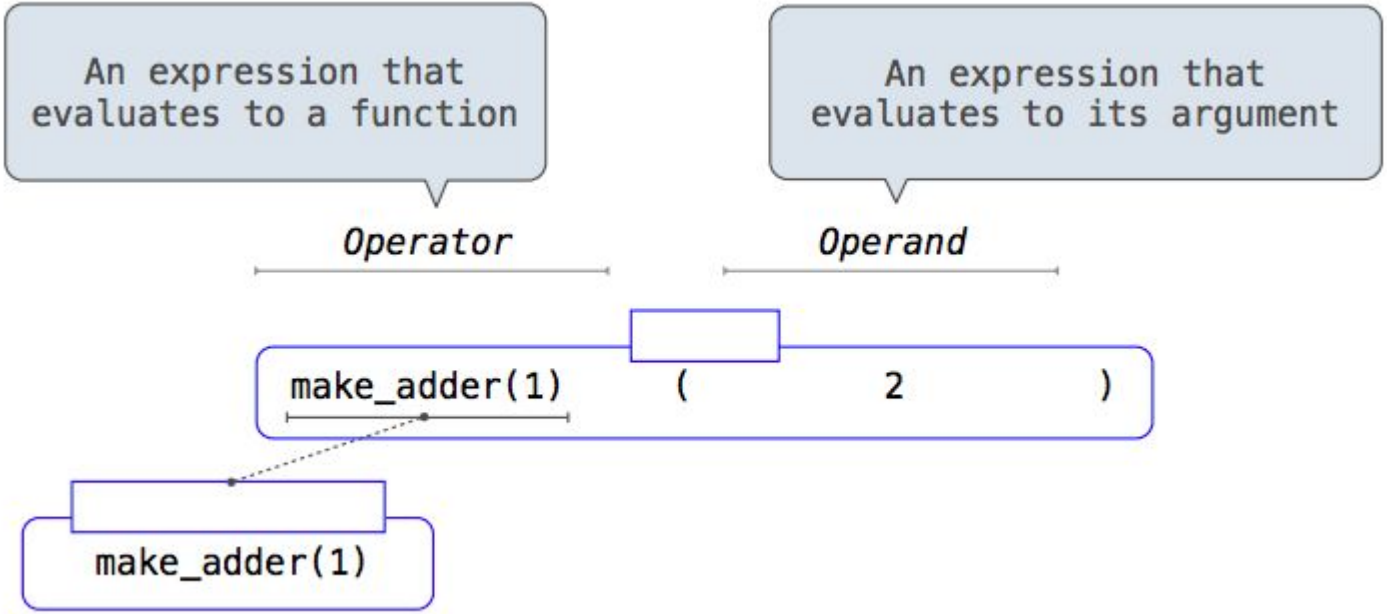
Call Expressions as Operator Expressions

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



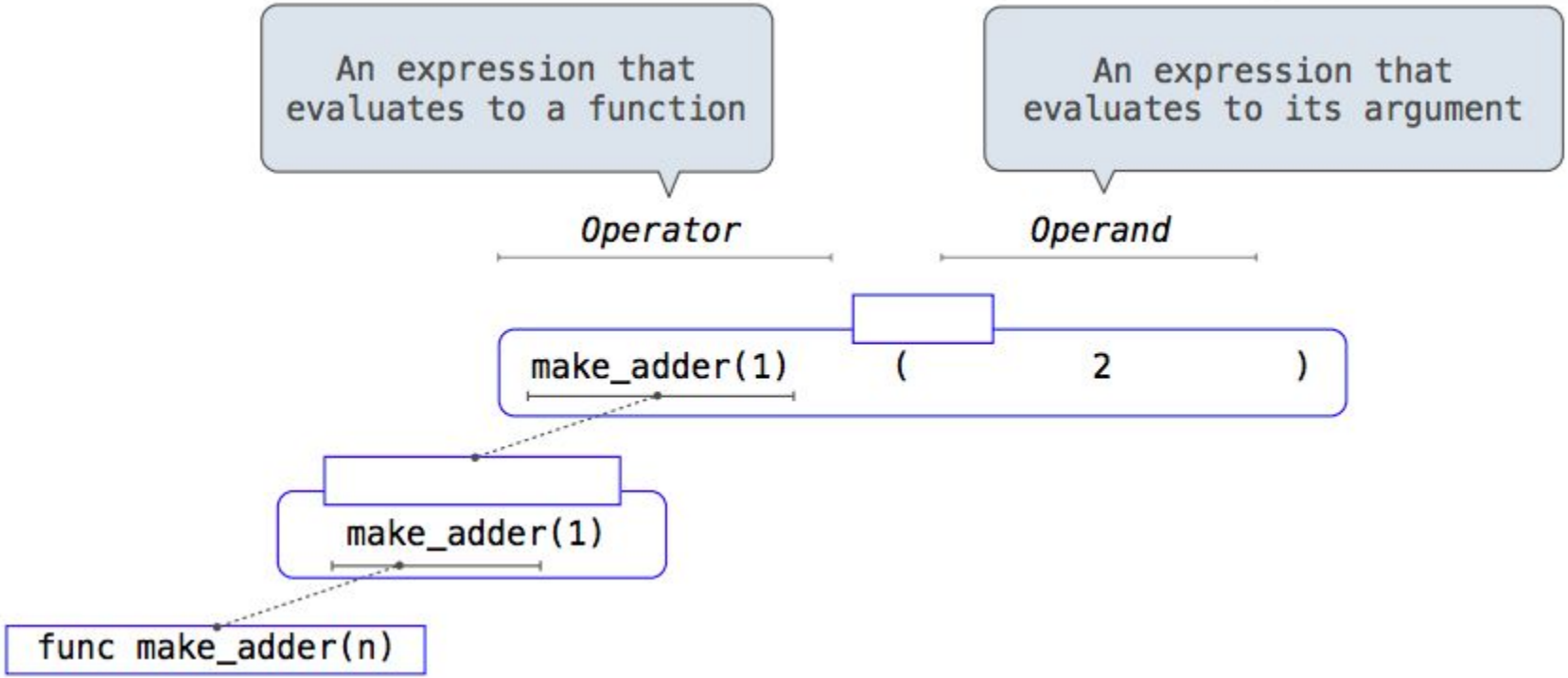
Call Expressions as Operator Expressions

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



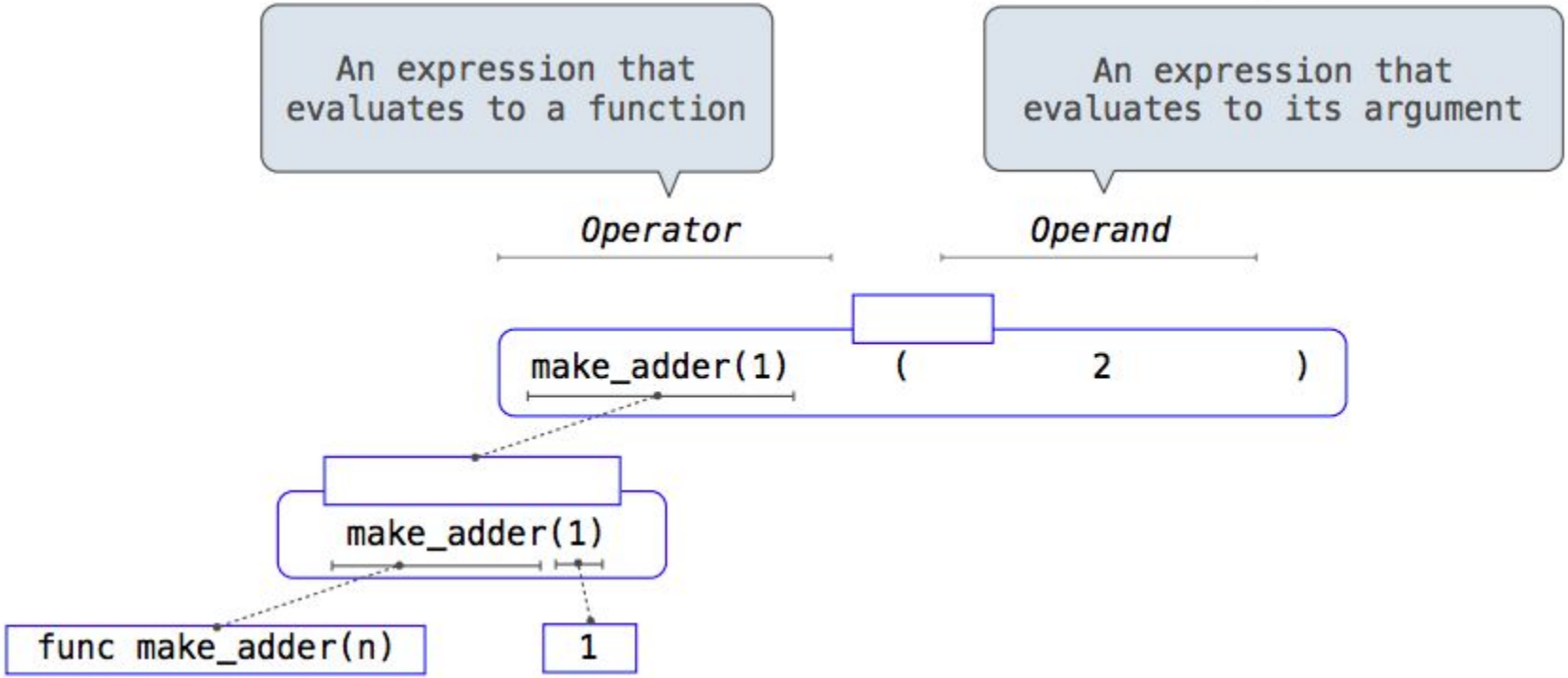
Call Expressions as Operator Expressions

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



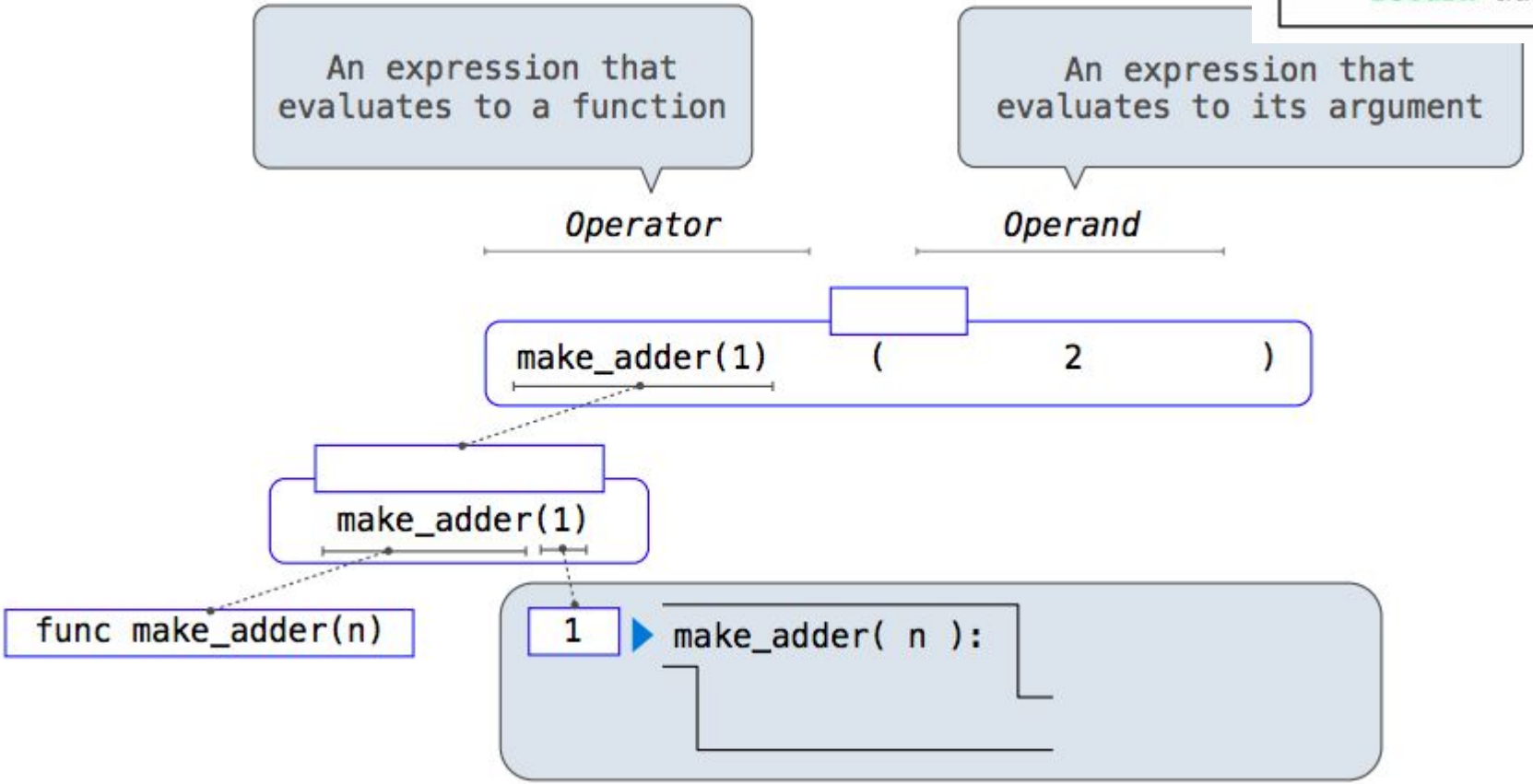
Call Expressions as Operator Expressions

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



Call Expressions as Operator Expressions

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



Call Expressions as Operator Expressions

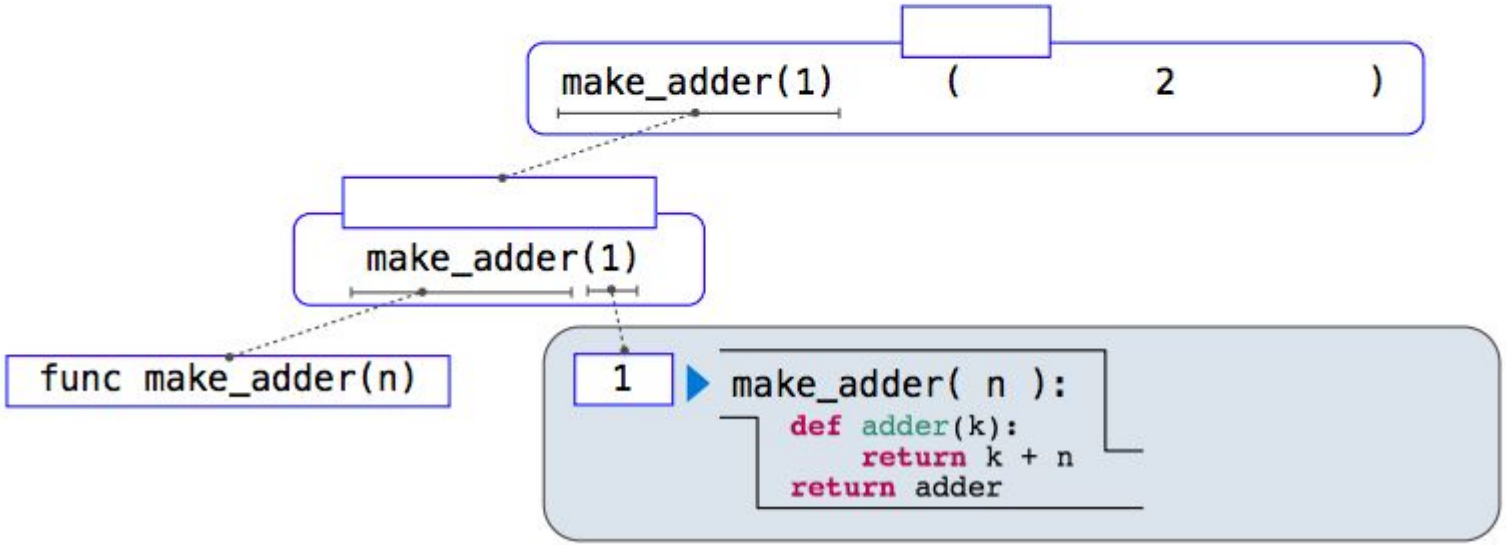
```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
```

An expression that evaluates to a function

Operator

An expression that evaluates to its argument

Operand



Call Expressions as Operator Expressions

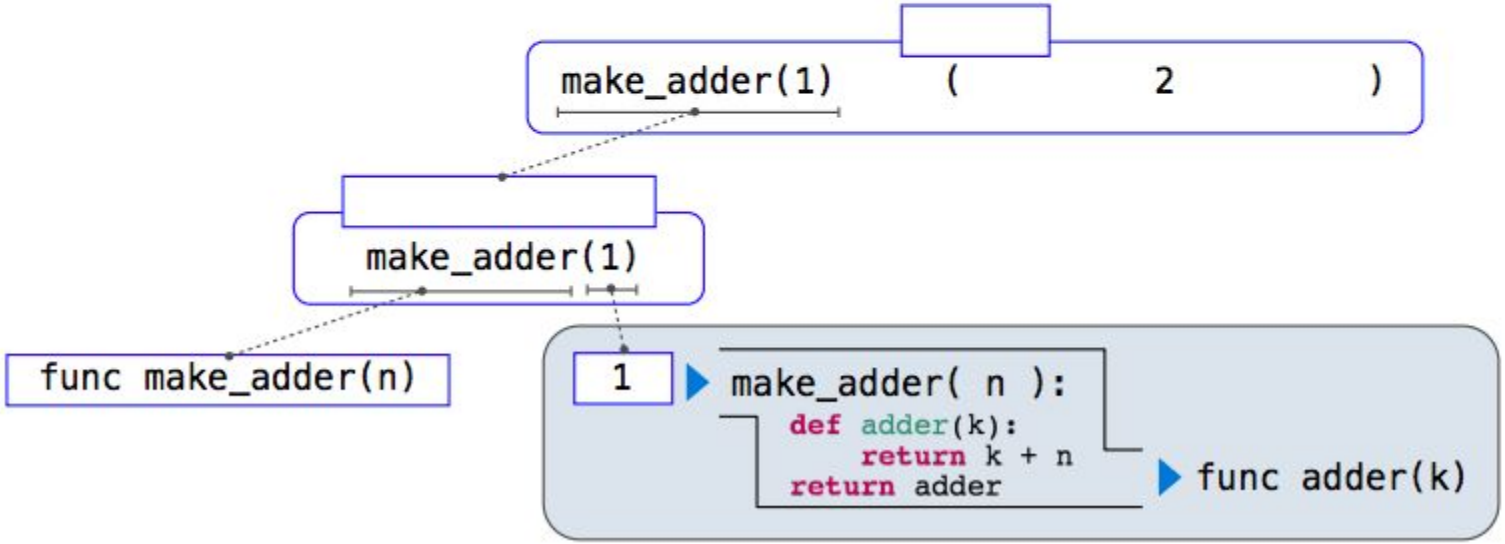
```
def make_adder(n):
    def adder(k):
        return k + n
    return adder
```

An expression that evaluates to a function

An expression that evaluates to its argument

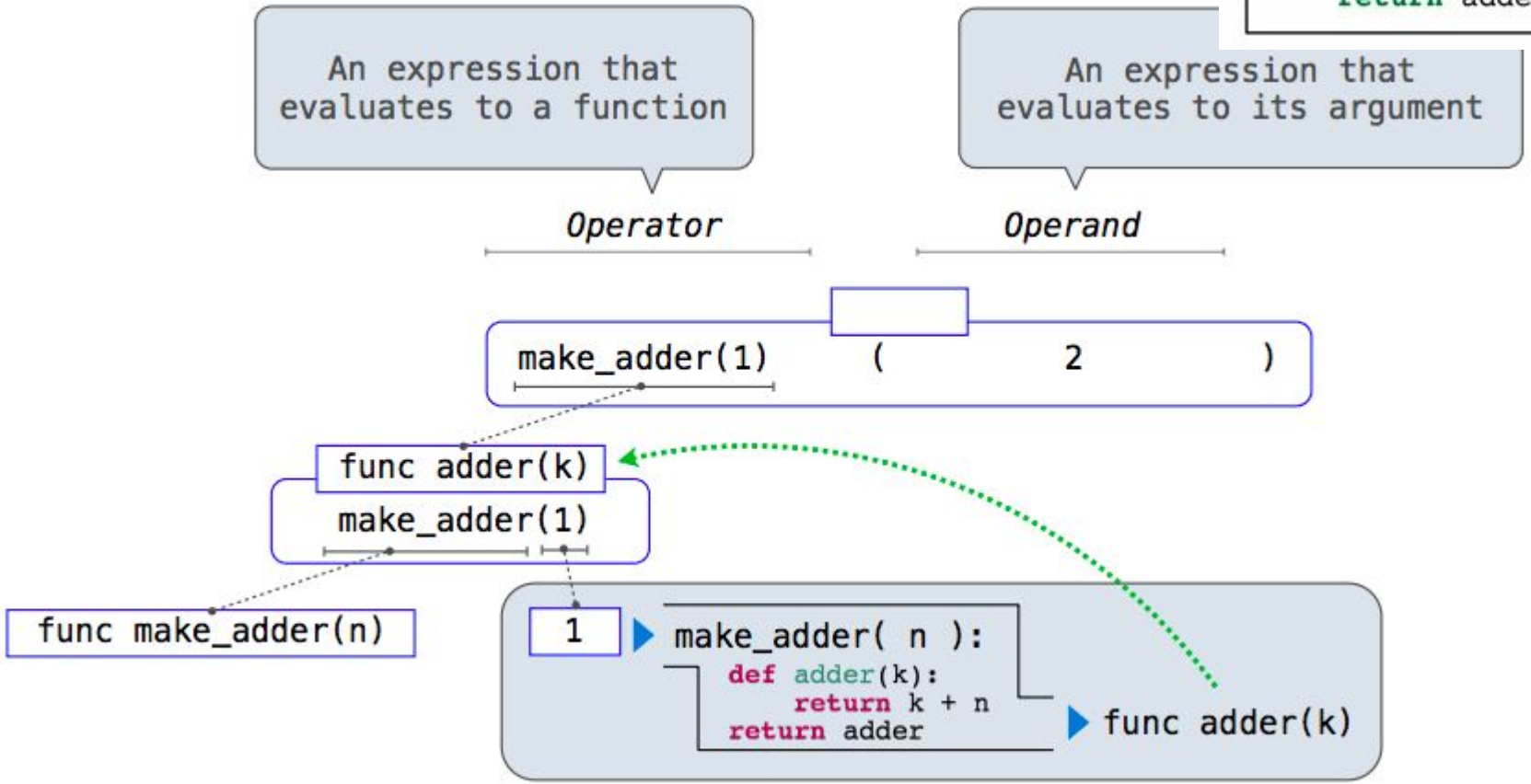
Operator

Operand



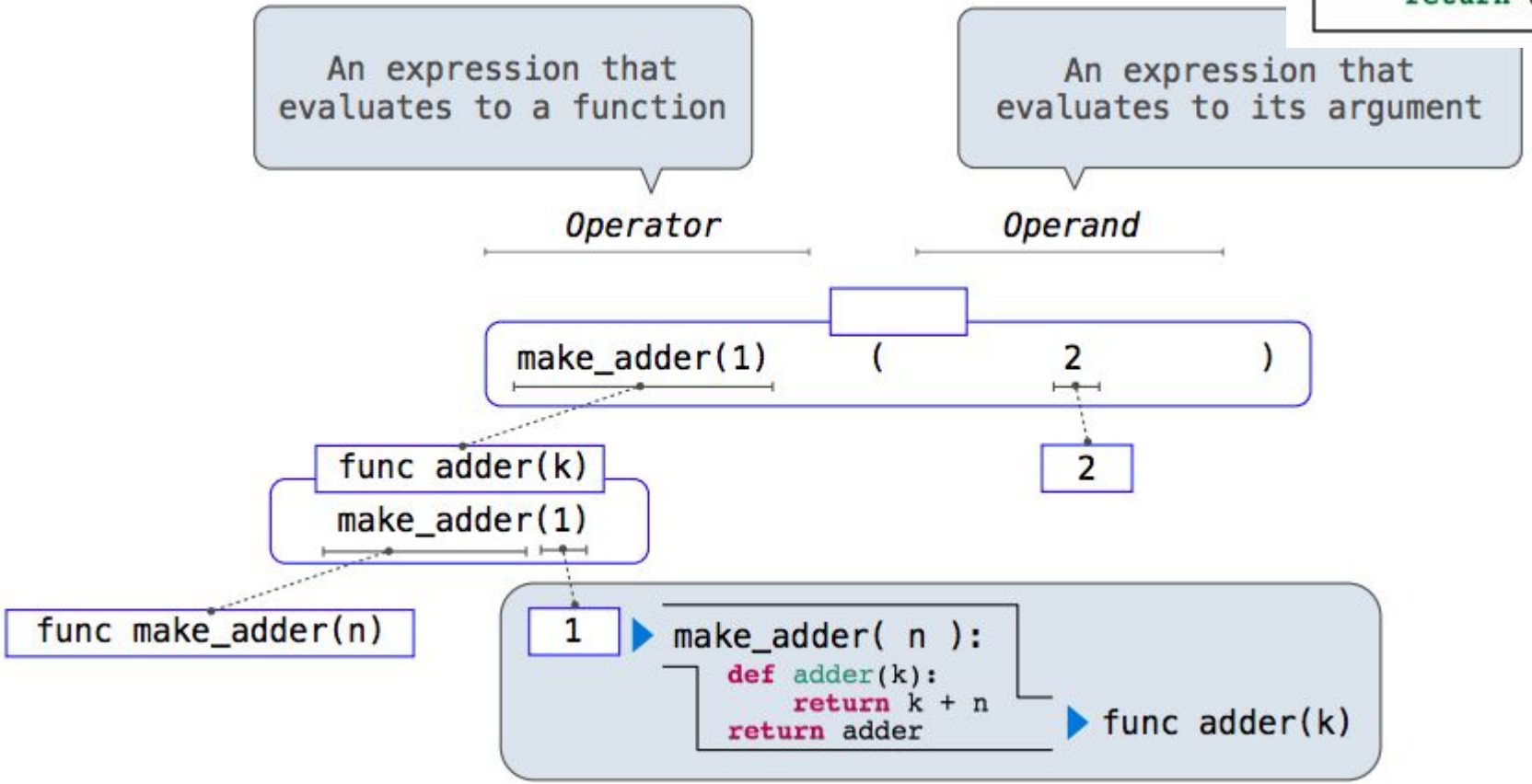
Call Expressions as Operator Expressions

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

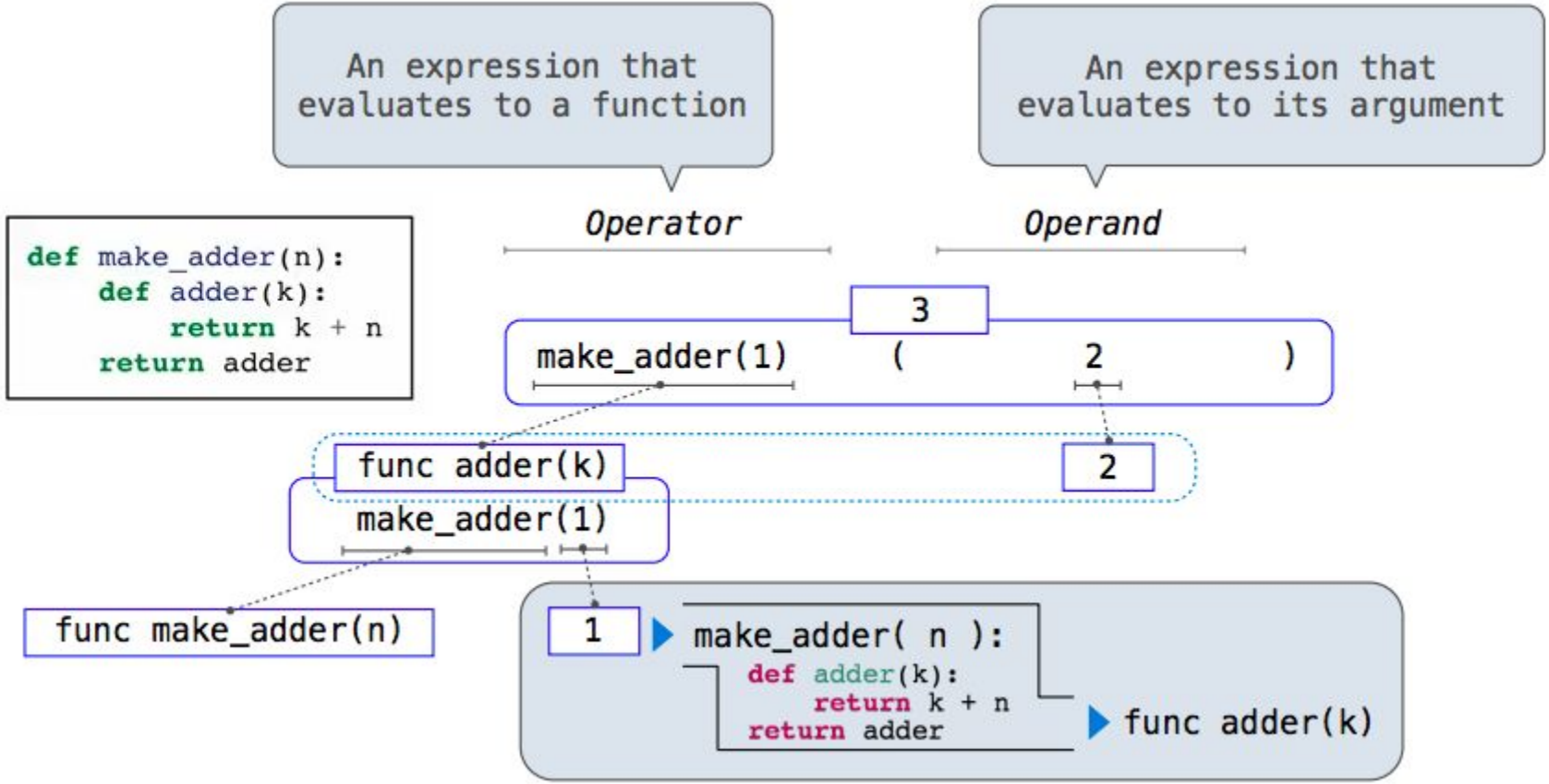


Call Expressions as Operator Expressions

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



Call Expressions as Operator Expressions (Demo)



The purpose of higher-order functions

Higher-order function: A function that takes a function as an argument value or returns a function as a return value

Useful:

- Express general methods of computation
- Remove repetition from programs
- Each function has exactly one job

Environment diagrams describe how higher-order functions work!

Nested Definitions. (Think before answering)

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


times3 = make_multiplier_of(3)
print(multiplier(9))
```

What will be printed?

A: 3

B: 9

C: 27

D: It does not make sense

E: Error

Log function

```
def logger (msg):

    def log_message():
        print('Log: ', msg)

    return log_message

log_hi = logger("Hi")
log_hi()
```

What will be printed?

A: Hi!

B: Log: Hi!

C: Log:

D: Still does not make sense

E: Error

Html_tags.

```
paragraph = html_tag('p')
paragraph('This is long paragraph')
```

```
<p>This is long paragraph</p>
```

```
print_h1 = html_tag('h1')
print_h1('Test Headline!')
print_h1('Another headline!')
```

```
<h1>Test Headline!</h1>
<h1>Another headline!</h1>
```

Html_tags. How would you write it?

```
paragraph = html_tag('p')
paragraph('This is long paragraph')
```

```
<p>This is long paragraph</p>
```

```
print_h1 = html_tag('h1')
print_h1('Test Headline!')
print_h1('Another headline!')
```

```
<h1>Test Headline!</h1>
<h1>Another headline!</h1>
```

```
>>> a = "Today"
>>> b = "June"
>>> print('{0} is not {1}, {1} is not {0}'.format(a,b))
Today is not June, June is not Today
```

```python
def html_tag(tag):

    def wrap_text(msg):
        print('<{0}>{1}</{0}>'.format(tag, msg))

    return wrap_text


print_h1 = html_tag('h1')
print_h1('Test Headline!')
print_h1('Another headline!')
```

```
<h1>Test Headline!</h1>
<h1>Another headline!</h1>
```

Question

```
def repeat(f, x):
    while f(x) != x:
        x = f(x)
    return x

def g(y):
    return (y+5)//3


result = repeat(g, 5)
```

What is the value of the *result*?

A. 0
B. Infinite loop
C. 2
D. 2.5
E. 3

Local Names
(Local parameters of the functions have *local* scope)

Question (demo)

```
def f(x,y):
    return g(x)

def g(a):
    return a + y

f(1, 2)
```

What will be the output?

A: 3

B: 1

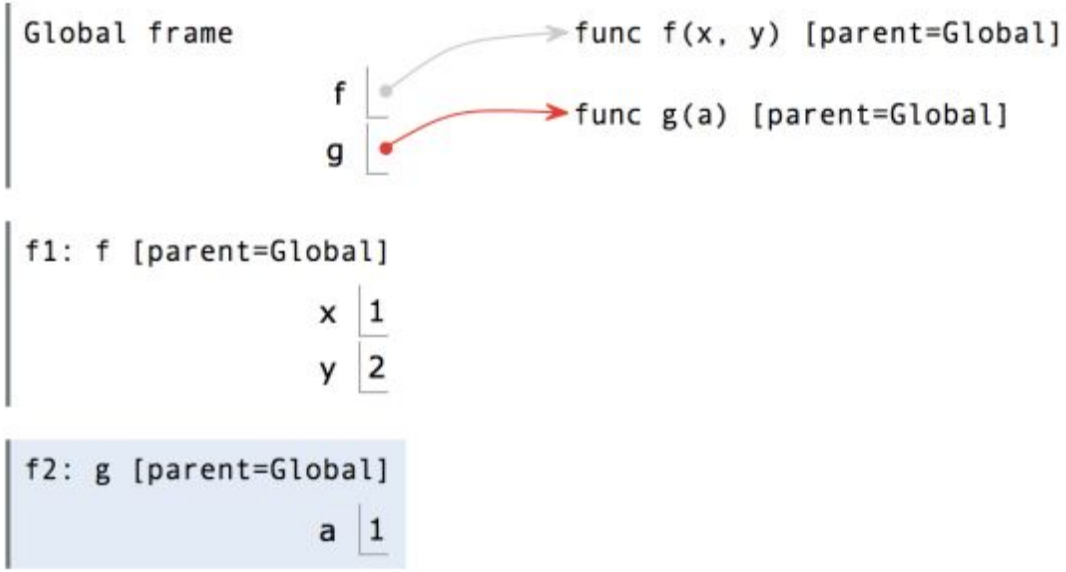
C: 2

D: Error

E: None of the above

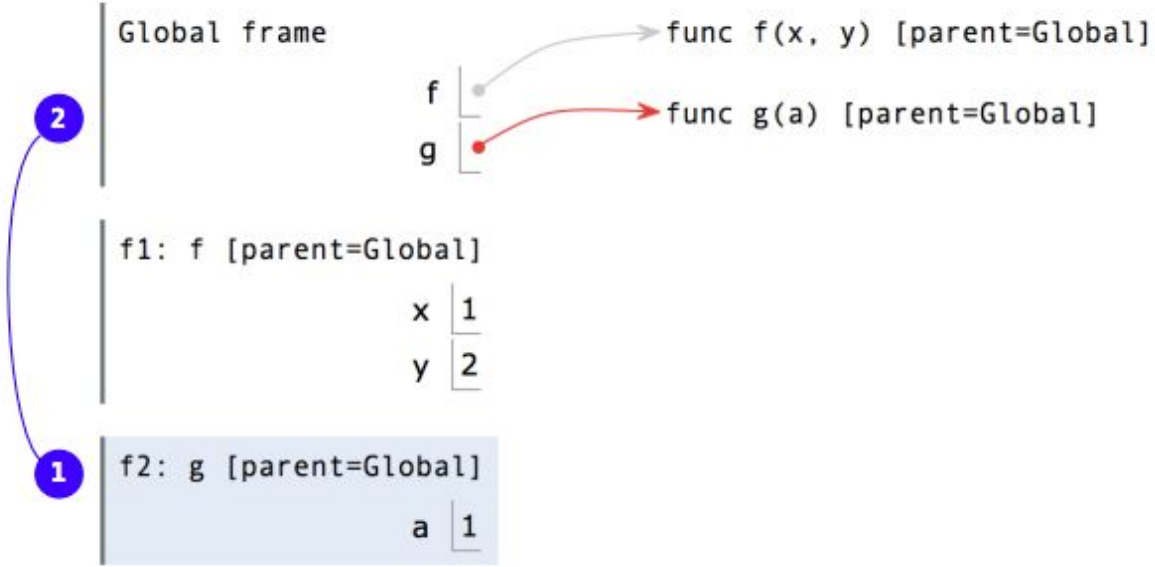
Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```



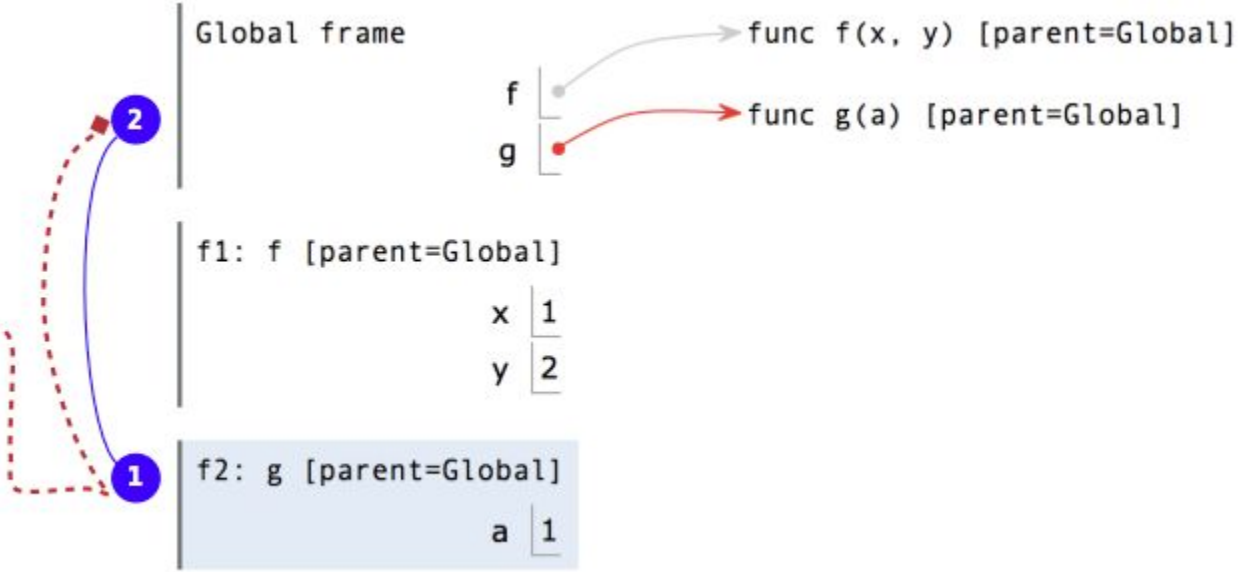
Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```



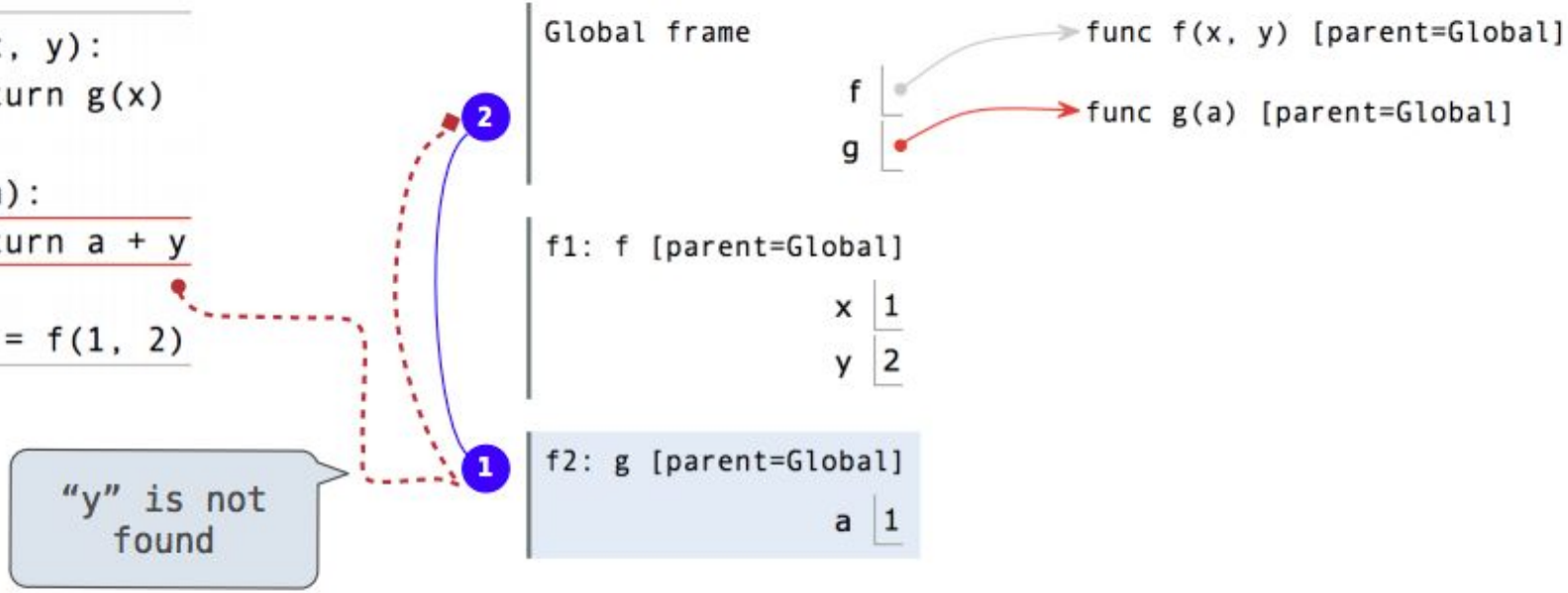
Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```



Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```



Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

"y" is not found, again

"y" is not found

Global frame

f → func f(x, y) [parent=Global]

g → func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

"y" is not found, again

Error

2

1

"y" is not found

Global frame

f → func f(x, y) [parent=Global]

g → func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

"y" is not found, again

Error

"y" is not found

Global frame

f → func f(x, y) [parent=Global]

g → func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

- An environment is a sequence of frames.

Local Names are not Visible to Other (Non-Nested) Functions

```
1  def f(x, y):
2      return g(x)
3
4  def g(a):
5      return a + y
6
7  result = f(1, 2)
```

"y" is not found, again

Error

"y" is not found

Global frame

f → func f(x, y) [parent=Global]

g → func g(a) [parent=Global]

f1: f [parent=Global]

x 1

y 2

f2: g [parent=Global]

a 1

- An environment is a sequence of frames.
- The environment created by calling a top-level function (no def within def) consists of one local frame, followed by the global frame.

Check Point

```
def func1(a, b):
    return 1 + func2(a + b)

def func2(c):
    a = 3
    return c + a


func1(1, 1)
```

Output?

A: Error

B: 5

C: 6

D: 7

E: Other

Practice with HOF. What is the output? (on your own)

```
def f(x, y):
    x, y = y, x
    def x(x):
        return y(x)
    return x

def y(x):
    return 6 + x

def z(y):
    return 4

g = f(y, z)
h = g(5)
```

Assert Statements In Python

Assertions and Exceptions

- Assertions should be used to check something *that should never happen*, while an exception should be used to check something *that might happen*.
  - Exceptions will be covered later.
- Assertions can be disabled at runtime using parameters, and are disabled by default, so don't count on them except for debugging purposes.
  - `Using -O`
- Assertions are used for **debugging** purposes only.

Assertions (demo)

- `assert`
  - Key word in Python

Syntax:

1. assert <condition>
2. assert <condition>, (optional) <error message>

If <condition> is false raise AssertError exception

Assertion: * not the best way to use it

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

```
def triple(func, num):
        assert isinstance(num, int), "Must be integer"

    return func(num) * 3
```

```
>>> triple(double, 10.1)

File "/Users/marinalanglois/Desktop/test.py", line 26, in triple
    assert isinstance(num, int), "Must be integer"
AssertionError: Must be integer
```

Assertion: * not the best way to use it

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

```
def triple(func, num):
        assert isinstance(num, int), "Must be integer"
        assert callable(func), "Must be function"
    return func(num) * 3
```

```
>>> doubl2 = 1000
>>> triple(double2, 10)

File "/Users/marinalanglois/Desktop/test.py", line 27, in triple
    assert callable(func), "Must be function"
AssertionError: Must be function
```

Lambda Expressions (Functions)

An expression that evaluates to a function

(Demo)

Lambda Expressions

- *An expression that evaluates to a function*
- Known as:
  - Anonymous functions
- We use lambda functions when we require a (nameless) function for a short period of time.
- We generally use it as an argument to a higher-order function
- Lambda functions are used along with built-in functions like *filter()*, *map()* and *reduce()*
- *Used with apply when working with Pandas*

Lambda Expressions

An expression that evaluates to a function

```
>>> double  = lambda x: 2 * x
```

A function

with formal parameter x

that returns the value of "2 * x"

Traditional way

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

Lambda Expressions

```
# add x and y

add = lambda x, y: x + y
```

```
def add (x, y):

     return x + y
```

Lambda Expressions

```
# max of x and y

max = lambda x, y: x if x > y else y
```

Return x, if x is greater than y, otherwise return y

```
def max(x, y):
    if x > y:
        return x
    else:
        return y
```

Lambda Expressions

- No “return” keyword
- **return**: Must be a single expression:
  - The **lambda**'s body is similar to what we'd put in a **def** body's **return** statement.
- Can not contain complex statements (like while or for loops)

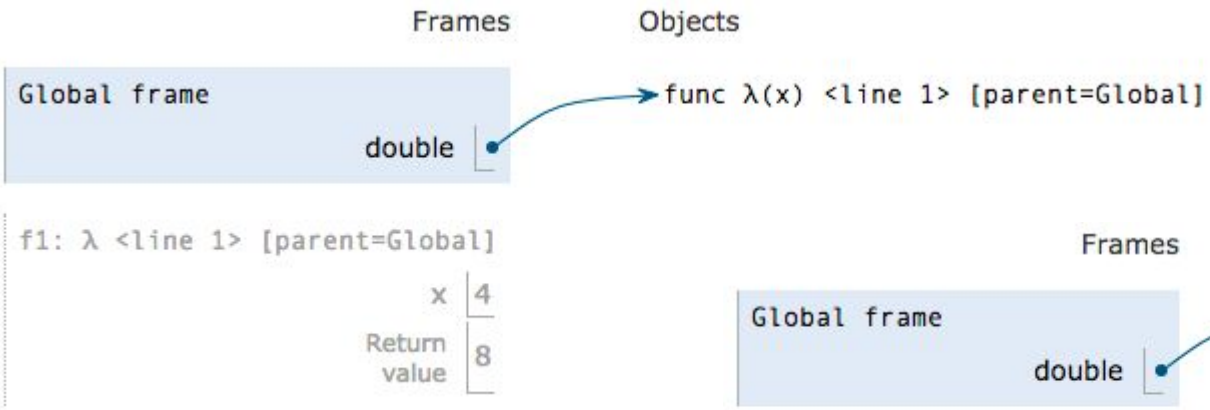
```
>>> def f(x, y, z): return x + y + z

>>> f(2, 30, 400)
```

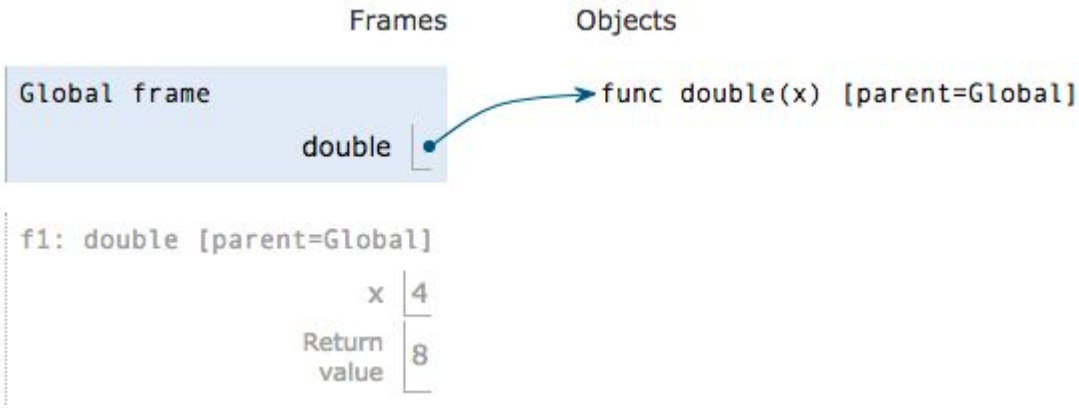
```
>>> f = lambda x, y, z: x + y + z

>>> f(2, 30, 400)
```

Intrinsic name

```
double  = lambda x: 2*x
```

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





Practice with lambdas. What is the output?

```
def writer():
    title = 'Sir'
    name = (lambda x:title + ' ' + x)
    return name

who = writer()
print(who('Arthur Ignatius Conan Doyle'))
```

Examples were taken from here: https://www.bogotobogo.com/python/python_functions_lambda.php

What is the output? (2 questions)

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]
for f in L:
  print(f(3))
```

_____

```
print(L[0](11))
```

Examples were taken from here: https://www.bogotobogo.com/python/python_functions_lambda.php

Re-write it using def statements

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]
for f in L:
  print(f(3))


print(L[0](11))
```

Examples were taken from here: https://www.bogotobogo.com/python/python_functions_lambda.php

Rewrite it using def statements

```
L = [lambda x: x ** 2,
     lambda x: x ** 3,
     lambda x: x ** 4]
for f in L:
  print(f(3))


print(L[0](11))
```

```
def f1(x): return x ** 2
def f2(x): return x ** 3
def f3(x): return x ** 4

L = [f1, f2, f3]
for f in L:
    print(f(3))
```

Examples takes from here: https://www.bogotobogo.com/python/python_functions_lambda.php


TIME FOR A
BREAK