Environments

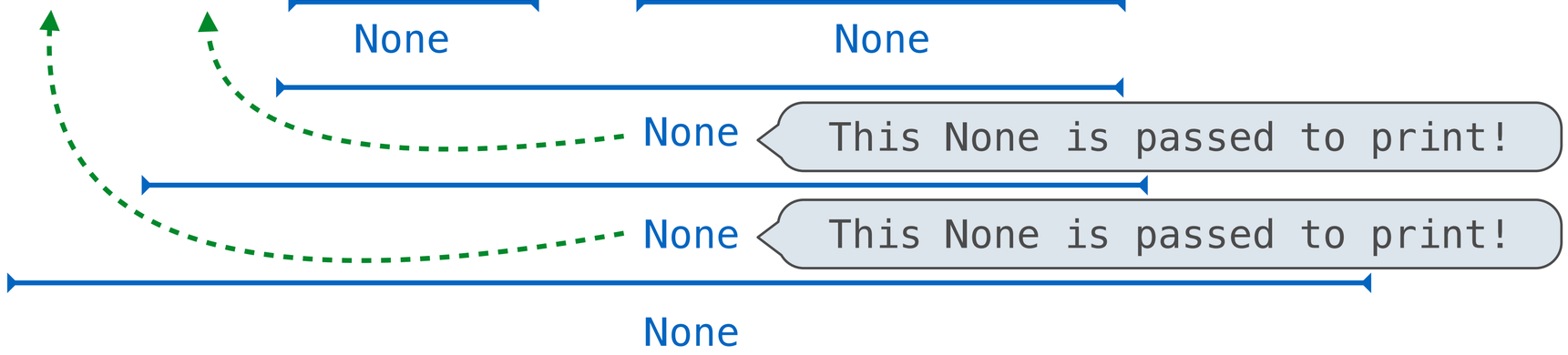
Announcements

Print and None Review

Fall 2022 CS 61A Midterm 1, Question 1 (c)

What does the long expression print?

```
s = "Knock"

print(print(print(s, s) or print("Who's There?")), "Who?")
```




```
Knock Knock
Who's There?
None
None Who?
```

False values in Python: False, 0, '', None *(more to come)*

To evaluate the expression **<left> or <right>**:

1. Evaluate the subexpression **<left>**.
2. If the result is a true value **v**, then the expression evaluates to **v**.
3. Otherwise, the expression evaluates to the value of the subexpression **<right>**.

Iteration Review

Fall 2022 CS 61A Midterm 1, Question 3 It's Perfect

A **perfect** number is a positive integer n whose **proper factors** (the factors of n below n) sum to exactly n. A number n is **abundant** if the sum of n's proper factors is greater than n and **deficient** if that sum is less than n.

Implement classify, a function that takes an integer n greater than 1. It returns the string 'deficient', 'perfect', or 'abundant' that correctly describes n.

```
>>> classify(6) # Proper factors 1, 2 and 3 sum to exactly 6.
'perfect'
>>> classify(24) # Proper factors 1, 2, 3, 4, 6, 8, and 12 sum to 36.
'abundant'
>>> classify(23) # Proper factor 1 sums to 1.
'deficient'
```

pollev.com/c88c

What do we need to compute?
(a) print the factors less than n
(b) count the factors less than n
(c) sum the factors less than n
(d) find the biggest factor less than n

What should we iterate over?
(a) the factors less than n
(b) every integer less than n
(c) every integer less than or equal to n

Fall 2022 CS 61A Midterm 1, Question 3 (a) It's Perfect

```
def classify(n):
    """Return whether n > 1 is 'deficient', 'perfect', or 'abundant'.
    >>> classify(6) # Proper factors 1, 2 and 3 sum to exactly 6.
    'perfect'
    >>> classify(24) # Proper factors 1, 2, 3, 4, 6, 8, and 12 sum to 36.
    'abundant'
    >>> classify(23) # Proper factor 1 sums to 1.
    'deficient'
    """
    total, k = 0, 1
    while k < n:
        if n % k == 0:
            total += k
        k = k + 1
    if total == n:
        return 'perfect'
    elif total < n:
        return 'deficient'
    else:
        return 'abundant'
```

pollev.com/c88c

What are we doing?

Sum the factors below n

Iterate over every integer below n, check if k is a factor

https://pythontutor.com/cp/composingprograms.html#code=def%20classify%28n%29%3A%0A%20%20%20%20%22%22%22Return%20whether%20n%20%3E%201%20is%20'deficient',%20'perfect',%20or%20%E2%80%98abundant'.%0A%20%20%20%20%3E%3E%3E%20classify%286%29%20%23%20Proper%20factors%201,%202%20and%203%20sum%20to%20exactly%206.%0A%20%20%20%20%E2%80%98perfect'%0A%20%20%20%20%3E%3E%3E%20classify%2824%29%20%23%20Proper%20factors%201,%202,%203,%204,%206,%208,%20and%2012%20sum%20to%2036.%0A%20%20%20%20%E2%80%98abundant'%0A%20%20%20%20%3E%3E%3E%20classify%282329%20%23%20Proper%20factor%201%20sums%20to%201.%0A%20%20%20%20%E2%80%98deficient'%0A%20%20%20%20%22%22%22%0A%20%20%20%20total,%20k%20%3D%20k%20%2B%201%0A%20%20%20%20while%20k%20%3C%20n%3A%0A%20%20%20%20%20%20%20%20if%20n%20%25%20k%20%3D%3D%200%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20total%20%2B%3D%20k%0A%20%20%20%20%20%20%20%20k%20%3D%20k%20%2B%201%0A%20%20%20%20if%20total%20%3D%3D%20n%3A%0A%20%20%20%20%20%20%20%20return%20'perfect'%0A%20%20%20%20elif%20total%20%3C%20n%3A%0A%20%20%20%20%20%20%20%20return%20'deficient'%0A%20%20%20%20else%3A%0A%20%20%20%20%20%20%20%20return%20'abundant'%0A%20%20%20%20%20%20%20%20%0Aclassify%286%29&cumulative=true&mode=edit&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

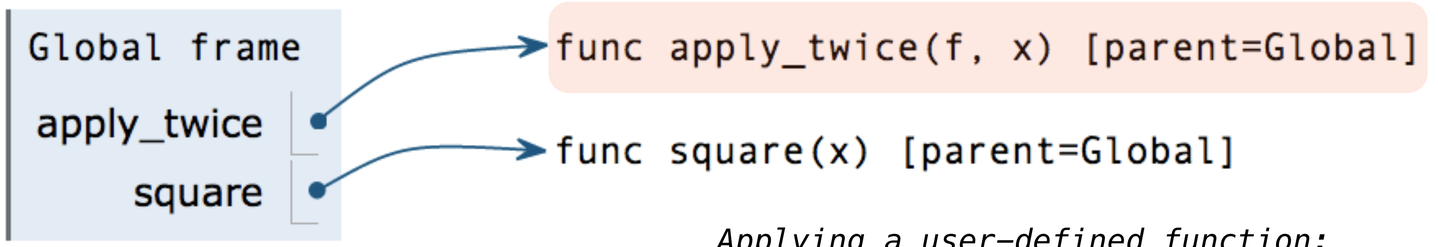
Environments for Higher-Order Functions

Student advice from final survey:

"ENVIRONMENT DIAGRAMS ARE EXTREMELY IMPORTANT! Taking this class with no prior Python experience and minimal overall programming experience, taking time to understand environment diagrams helped me fully understand step-by-step how my code is interpreted, and any areas where my code may be going wrong. This made coding more intuitive for me, as it helped me gain a understanding of the connections being made between my code and carried out functions."

Names can be Bound to Functional Arguments

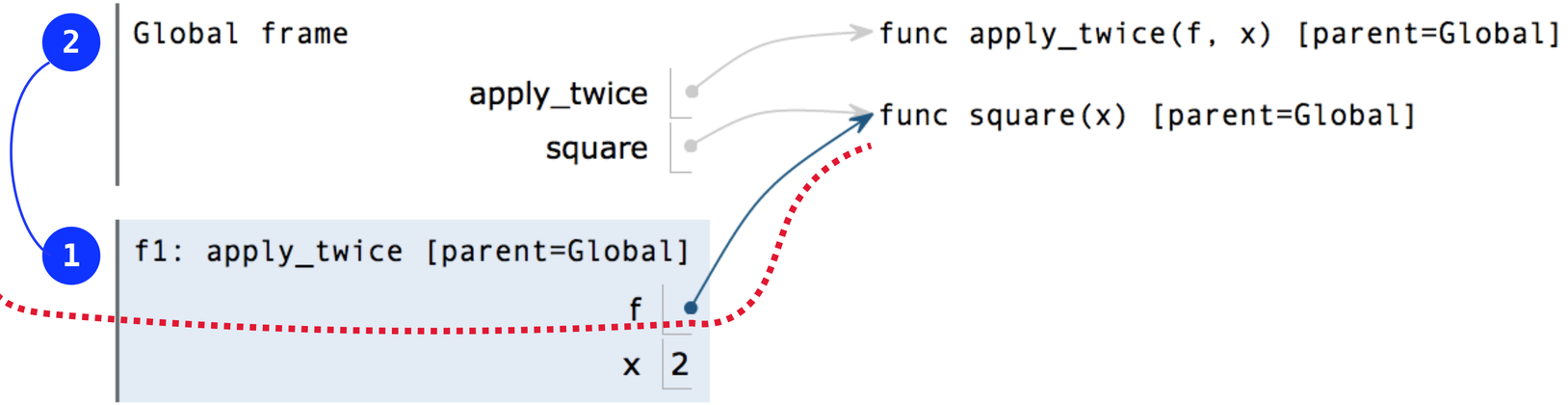
```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```



Applying a user-defined function:

- Create a new frame
- Bind formal parameters (f & x) to arguments
- Execute the body: return f(f(x))

```
1  def apply_twice(f, x):
2      return f(f(x))
3
4  def square(x):
5      return x * x
6
7  result = apply_twice(square, 2)
```

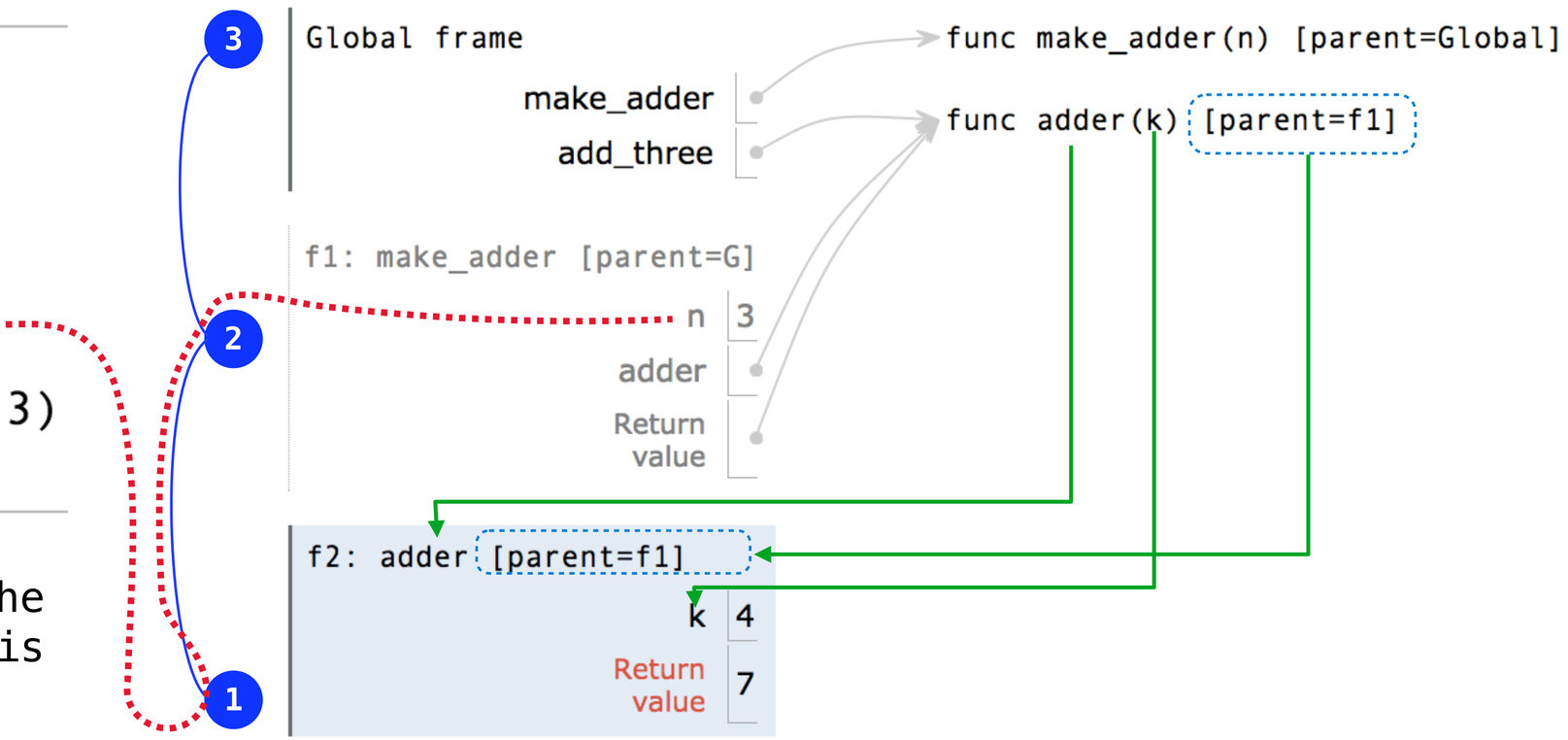


pythontutor.com/composingprograms.html#code=def%20apply_twice%28f,%20x%29%3A%0A%20%20%20%20return%20f%28f%28x%29%29%0A%20%20%20%20%0Adef%20square%28x%29%3A%0A%20%20%20%20return%20x%20*%20x%0A%20%20%20%20%0Aresult%20%3D%20apply_twice%28square,%202%29&mode=display&origin=composingprograms.js&cumulative=true&py=3&rawInputLstJSON=[]&curInstr=0

Environment Diagrams for Nested Def Statements

Nested def

```
1  def make_adder(n):
2      def adder(k):
3          return k + n
4      return adder
5
6  add_three = make_adder(3)
7  add_three(4)
```



- When a function is defined, the parent of the function value is the current frame
- Every local frame has a parent frame (often global)
- The parent of a frame is the parent of the function called

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20k%20%2B%20n%0A%20%20%20%20return%20adder%0A%20%20%20%20%0Aadd_three%20%3D%20make_adder%283%29%0Aresult%20%3D%20add_three%284%29&cumulative=false&mode=edit&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Wrong make_adder()?

```
def make_adder(n):
    k = 5
    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)

print(add_three(4))
```

(A)

```
def make_adder(n):
    n = 5
    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)

print(add_three(4))
```

(B)

```
def make_adder(n):

    def adder(k):
        k = 5
        return n + k

    return adder

add_three = make_adder(3)

print(add_three(4))
```

(C)

```
def make_adder(n):

    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)
add_five = make_adder(5)
print(add_three(4))
```

(D)

Which of these implementations **do not print 7**?

pollev.com/c88c

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20k%20%3D%205%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20return%20adder%0Aadd_three%20%3D%20make_adder%283%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=2&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20return%20adder%0Aadd_three%20%3D%20make_adder%283%29%0Aadd_five%20%3D%20make_adder%285%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5

How to Draw an Environment Diagram

When a function is defined:

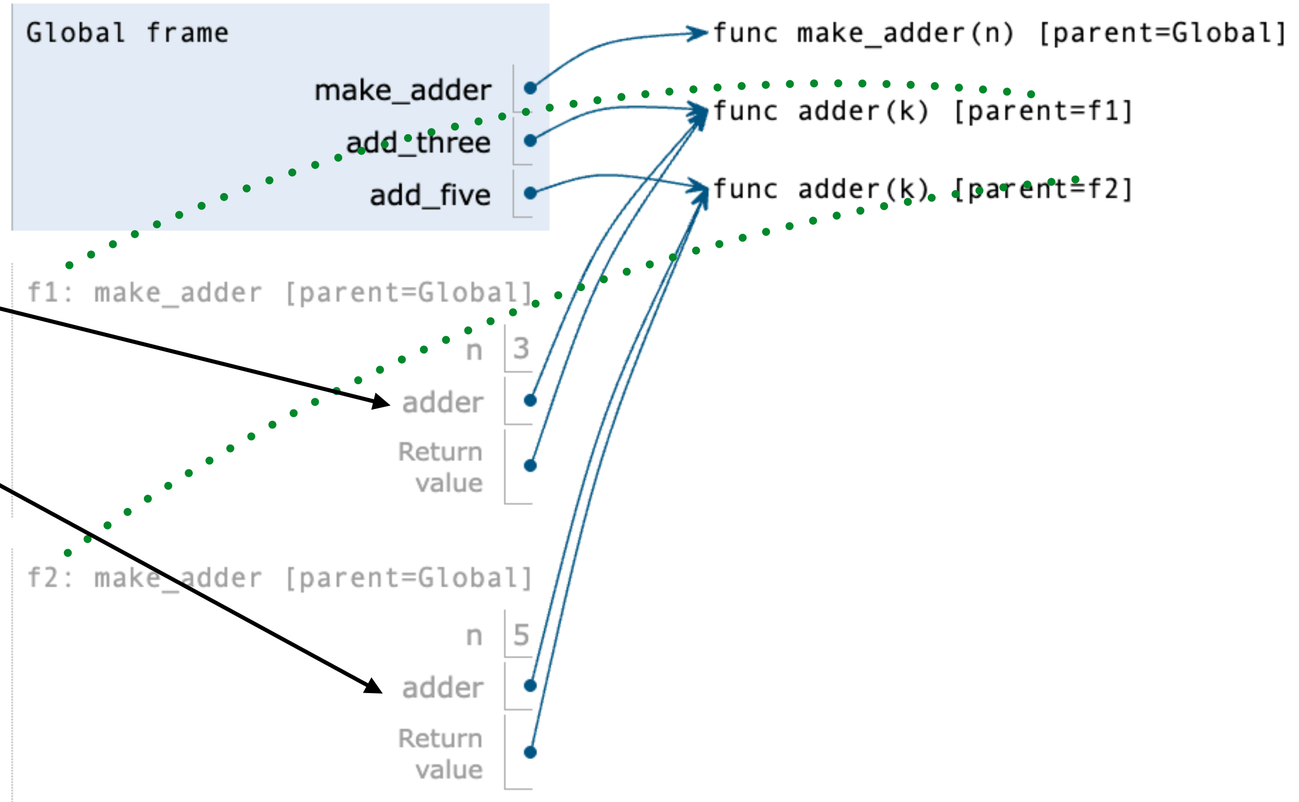
Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

Bind <name> to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.



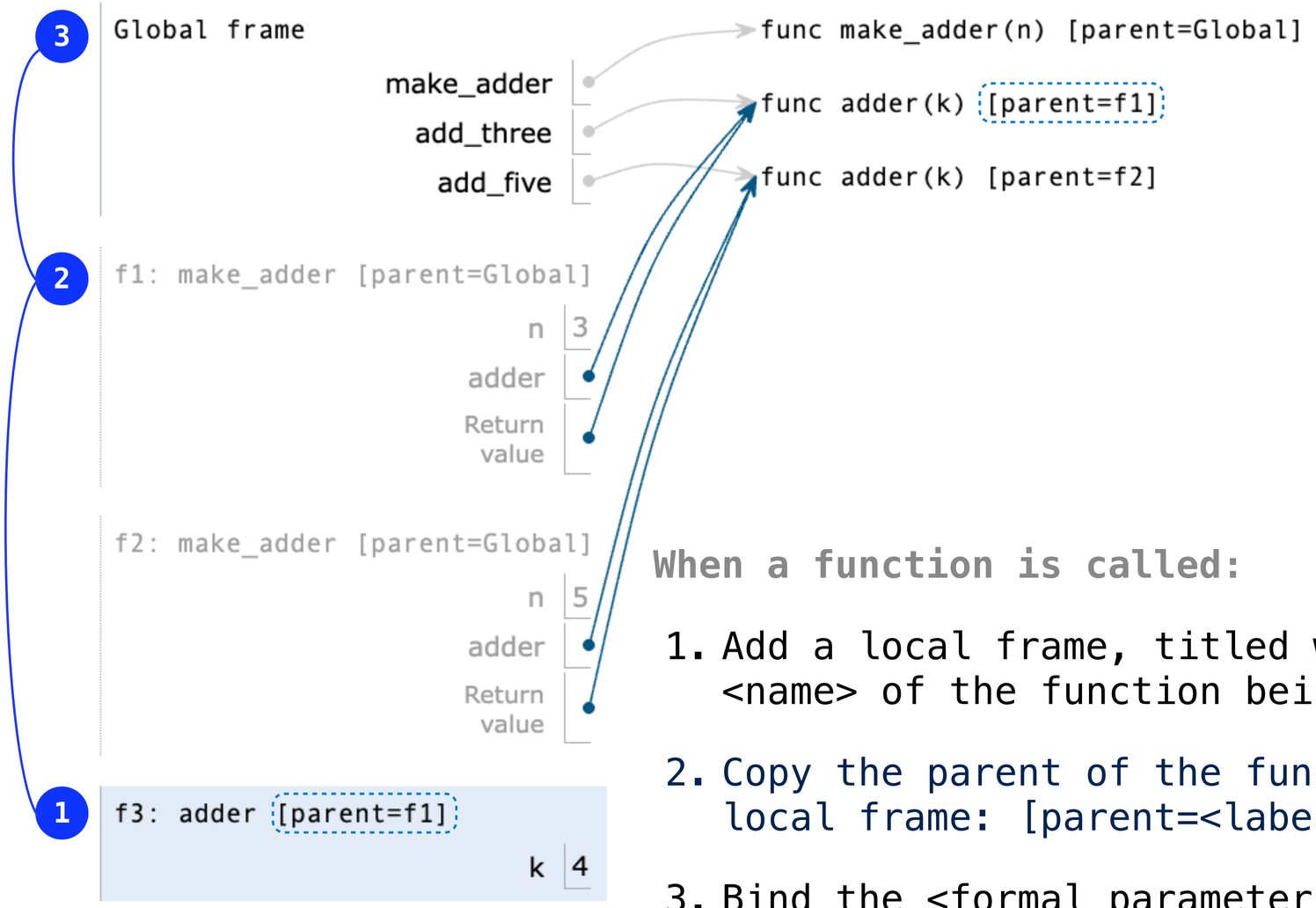
(Demo)

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20return%20adder%0Aadd_three%20%3D%20make_adder%283%29%0Aadd_five%20%3D%20make_adder%285%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5

How to Draw an Environment Diagram

```
def make_adder(n):

    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)
add_five = make_adder(5)
print(add_three(4))
```

(D)



When a function is called:

1. Add a local frame, titled with the <name> of the function being called.
2. Copy the parent of the function to the local frame: [parent=<label>]
3. Bind the <formal parameters> to the arguments in the local frame.

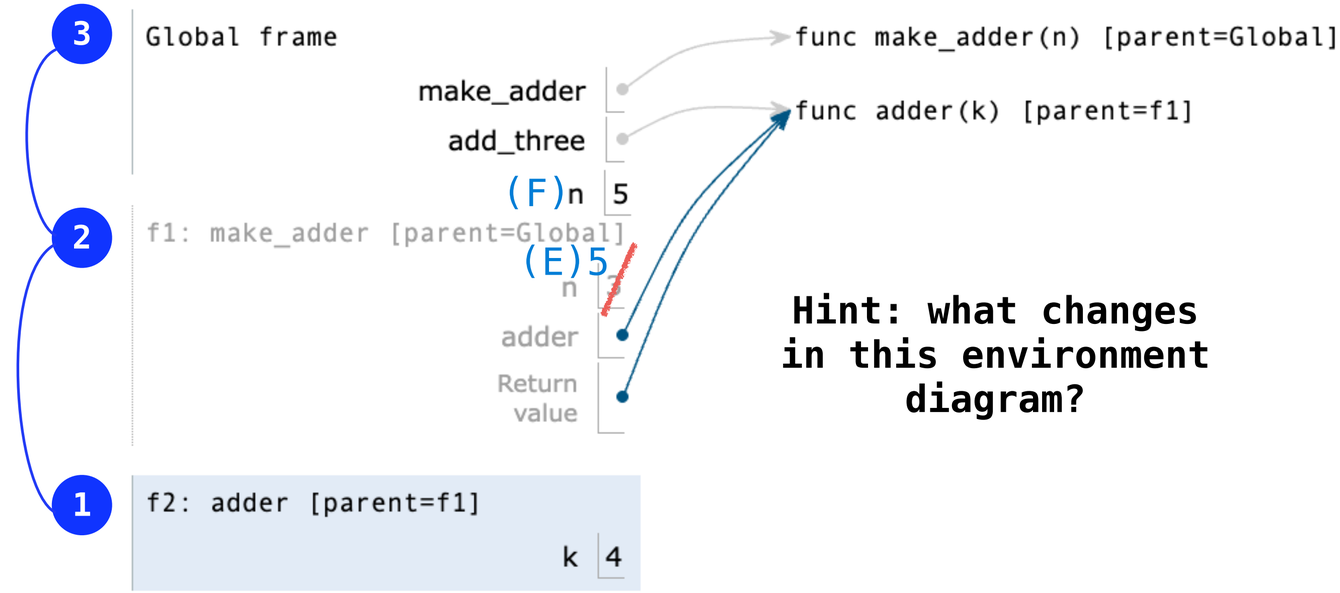
Wrong make_adder()?

```
def make_adder(n):

    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)

print(add_three(4))
```

(E)

```
def make_adder(n):

    def adder(k):

        return n + k

    return adder

add_three = make_adder(3)
n = 5
print(add_three(4))
```

(F)



Hint: what changes in this environment diagram?

Which of these implementations results in an incorrect add_three function, so prints a **different result**?

pollev.com/c88c

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20return%20adder%0Aadd_three%20%3D%20make_adder%283%29%0Aadd_five%20%3D%20make_adder%285%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5

Wrong make_adder()?

```
1  def make_adder(n):
2      return adder
3
4  def adder(k):
5      return n + k
6
7  add_three = make_adder(3)
8  print(add_three(4))
```

When a function is defined:

Create a function value: func <name>(<formal parameters>) [parent=<label>]

Its parent is the current frame.

Bind <name> to the function value in the current frame

https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20return%20adder%0A%20%20%20%20%0Adef%20adder%28k%29%3A%0A%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20%0Aadd_three%20%3D%20make_adder%283%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5D

Recap: How to Draw an Environment Diagram

When a function is defined:

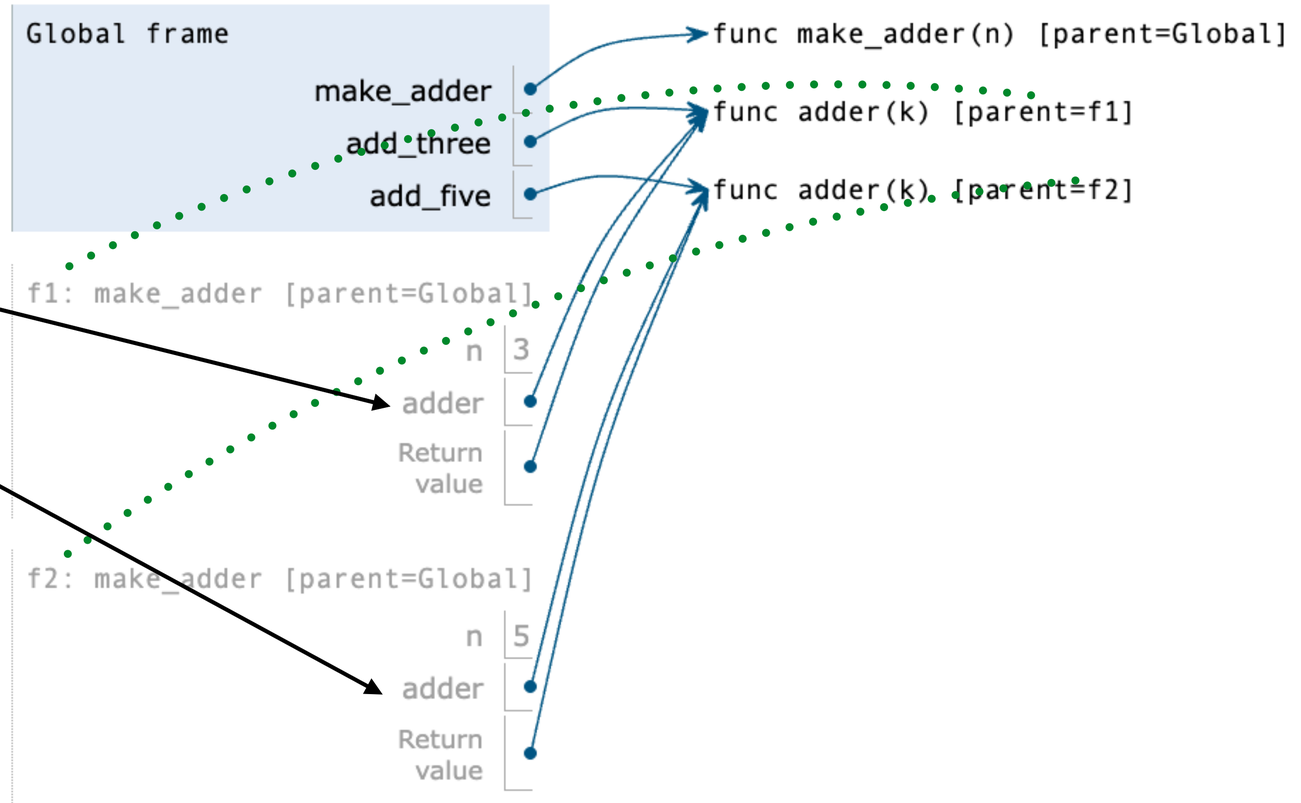
Create a function value: `func <name>(<formal parameters>) [parent=<label>]`

Its parent is the current frame.

Bind `<name>` to the function value in the current frame

When a function is called:

1. Add a local frame, titled with the `<name>` of the function being called.
2. **Copy the parent of the function to the local frame: `[parent=<label>]`**
3. Bind the `<formal parameters>` to the arguments in the local frame.



https://pythontutor.com/cp/composingprograms.html#code=def%20make_adder%28n%29%3A%0A%20%20%20%20def%20adder%28k%29%3A%0A%20%20%20%20%20%20%20%20return%20n%20%2B%20k%0A%20%20%20%20return%20adder%0Aadd_three%20%3D%20make_adder%283%29%0Aadd_five%20%3D%20make_adder%285%29%0Aprint%28add_three%284%29%29&cumulative=true&curInstr=0&mode=display&origin=composingprograms.js&py=3&rawInputLstJSON=%5B%5