Discussion 1

Walkthrough

Announcements

- Start HOG early!
- Find partners on Piazza (https://piazza.com/class/irwl7o7shzu70z?cid=5)
- My office hours: T/W 4-5 @ 109 Morgan Hall
    - email me at katya.stukalova@berkeley.edu

Environment Diagram Rules

1. Assignment

ex: `a = 3`

1. evaluate the RHS
2. assign the value from step 1 to the name on the LHS

G:

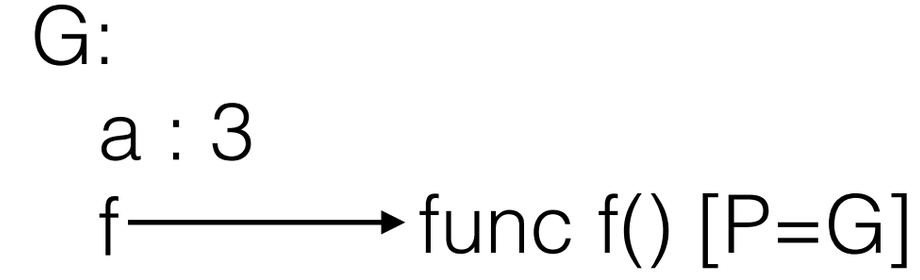
a : 3

Environment Diagram Rules

1. Assignment

ex: `a = 3`

1. evaluate the RHS
2. assign the value from step 1 to the name on the LHS



2. Defining a function

ex:
```
def f():
    return 1
```

1. write function signature
2. write the function name
3. point the name to the signature

Environment Diagram Rules

1. Assignment

ex: `a = 3`

1. evaluate the RHS
2. assign the value from step 1 to the name on the LHS

2. Defining a function

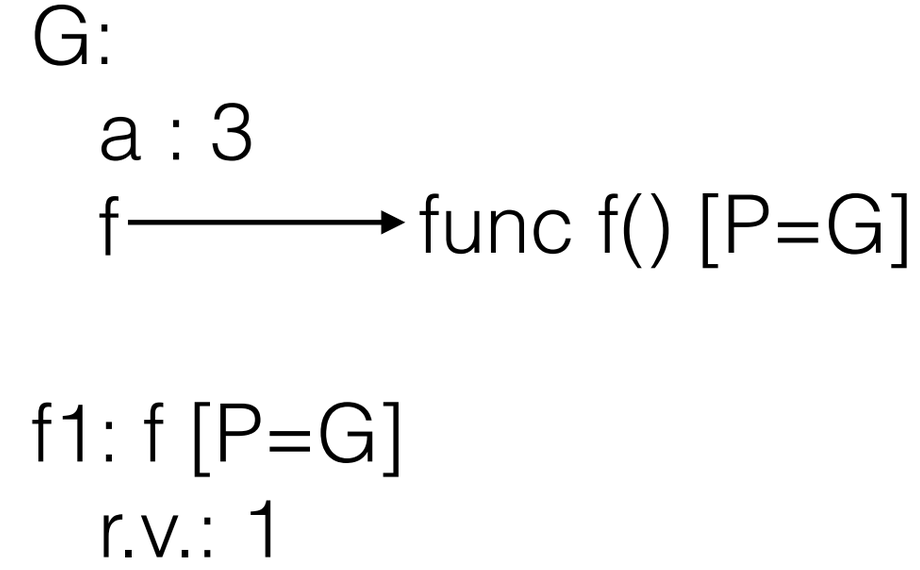
ex:
```
def f():
    return 1
```

1. write function signature
2. write the function name
3. point the name to the signature

3. Function call

ex: `a = f()`

1. evaluate the operator
2. evaluate the operands
3. open a new frame
   - label the frame with: f#, the intrinsic function name, [P=G]
4. copy the parameters into the new frame
   - remember to use the names from the function signature
5. execute the body of the function



2.1 #1

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
      return a + b
a = b(a)
a = b(a)
```

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
      return a + b
a = b(a)
a = b(a)
```

Reasoning

Solution

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
      return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

Solution

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
      return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

Solution

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)
- now we have a def statement (remember to skip the definition of the function)

Solution

G

a : 1

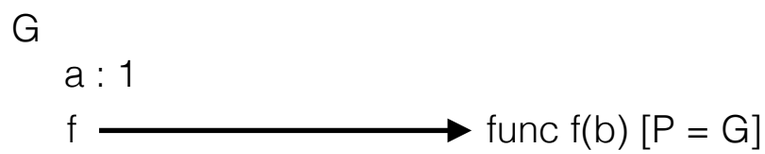
Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)
- now we have a def statement (remember to skip the definition of the function)

Solution



Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

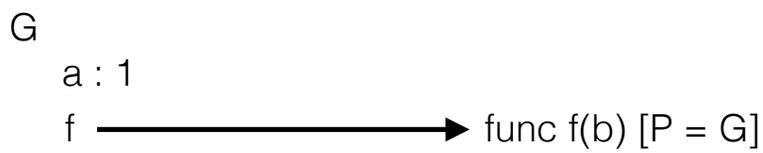
- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

Solution



Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

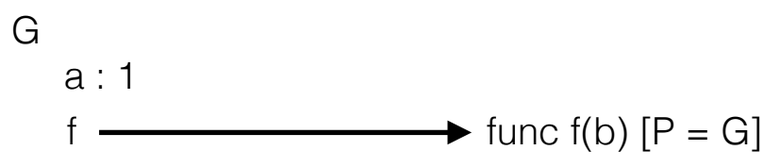
- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

Solution



Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

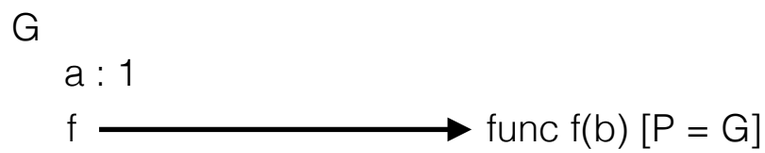
- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

Solution



Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

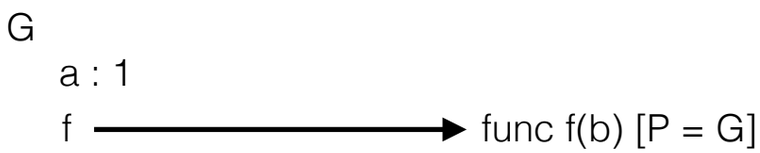
- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

Solution



f1: f [P=G]

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

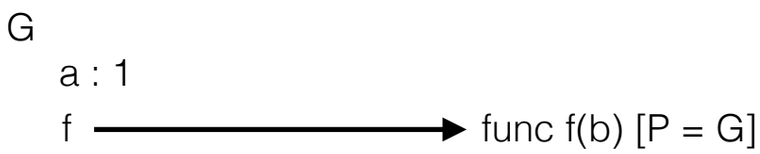
- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

Solution



f1: f [P=G]

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

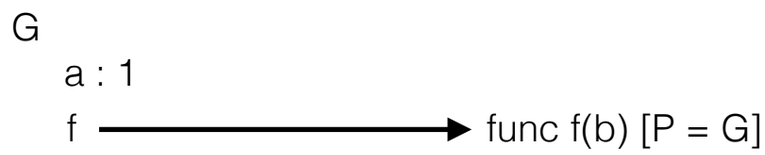
- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

Solution



f1: f [P=G]

b: 1

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

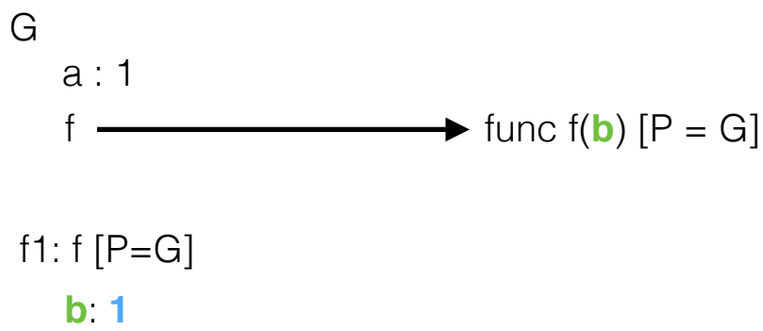
- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

Solution



Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

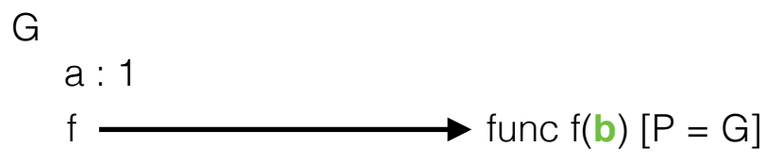
- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

Solution



f1: f [P=G]

b: **1**

r.v.: 2

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?

Solution

G

a : 1

f → func f(**b**) [P = G]

f1: f [P=G]

b: 1

r.v.: 2

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?

Solution

G

a : ~~1~~, 2

f → func f(**b**) [P = G]

f1: f [P=G]

b: **1**

r.v.: 2

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?
- run through what happens in the last line of the code, using the hints from above

Solution

G

a : ~~1~~, 2

f → func f(**b**) [P = G]

f1: f [P=G]

b: **1**

r.v.: 2

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?
- run through what happens in the last line of the code, using the hints from above

Solution

G

a : ~~1~~, 2

f → func f(**b**) [P = G]

f1: f [P=G]

b: 1

r.v.: 2

f2: f [P=G]

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?
- run through what happens in the last line of the code, using the hints from above

Solution

G

a : ~~1~~, 2

f → func f(**b**) [P = G]

f1: f [P=G]

b: **1**

r.v.: 2

f2: f [P=G]

b: 2

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?
- run through what happens in the last line of the code, using the hints from above

Solution

G

a : ~~1~~, 2

f → func f(**b**) [P = G]

f1: f [P=G]

b: **1**

r.v.: 2

f2: f [P=G]

b: 2

r.v.: 4

Draw the environment diagram that results from running the following code.

```
a = 1
def b(b):
    return a + b
a = b(a)
a = b(a)
```

Reasoning

- the first line is an assignment statement (recall the procedure for assignment statements)

- now we have a def statement (remember to skip the definition of the function

- there are 2 things in the line following the def statement — an assignment and a function call

- do the function call first: whats the **operator**? the **operand**?

- recall what we mean when we say “evaluate”. “evaluate” is synonymous to “do i know the value of this name?”
- do we know the value bound to **b**? **a**?

- now we’re ready to open up a new frame (how do we label the frame?)

- what parameters do we pass in? why is the name b? why is the value 1?

- now we’re ready to execute the body of the function. what do we return in this case?

- remember where we were when we started doing the function call! we were in the middle of an assignment statement. what do we need to change in G?
- run through what happens in the last line of the code, using the hints from above

Solution

G

a : ~~1~~, ~~2~~, 4

f → func f(**b**) [P = G]

f1: f [P=G]

b: **1**

r.v.: 2

f2: f [P=G]

b: 2

r.v.: 4

2.1 #2

Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

Solution

func min(…) [P=G]

Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)

Solution

func min(…) [P=G]

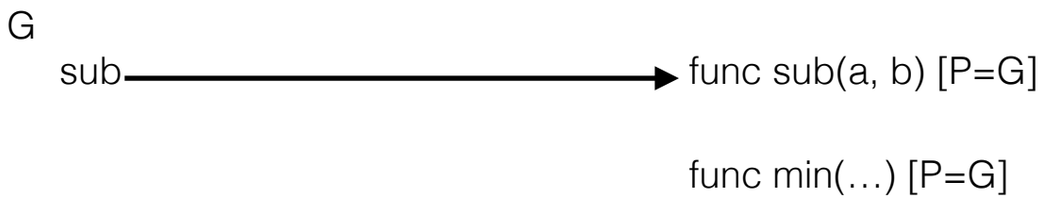
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)

Solution



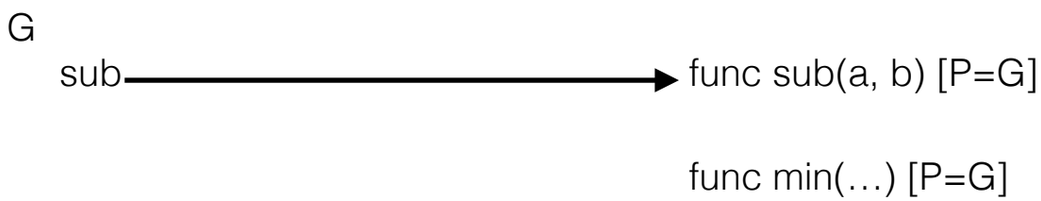
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?

Solution



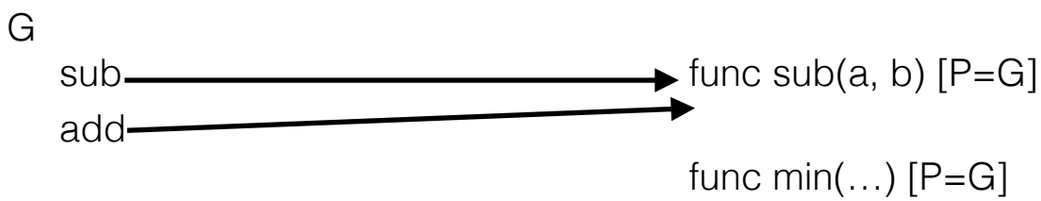
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?

Solution



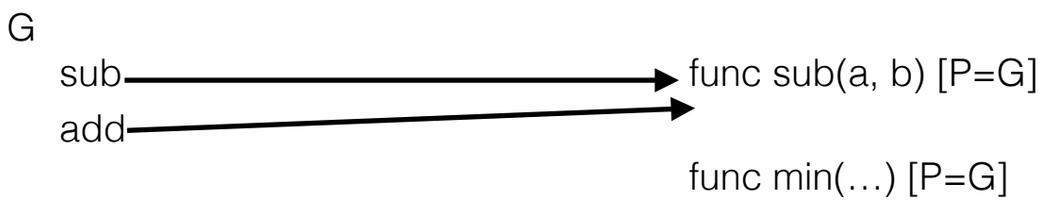
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**

Solution



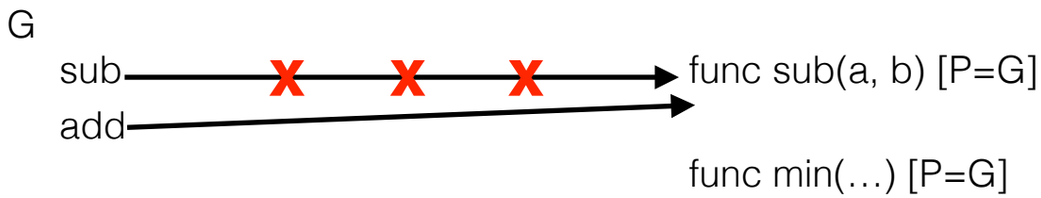
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**

Solution



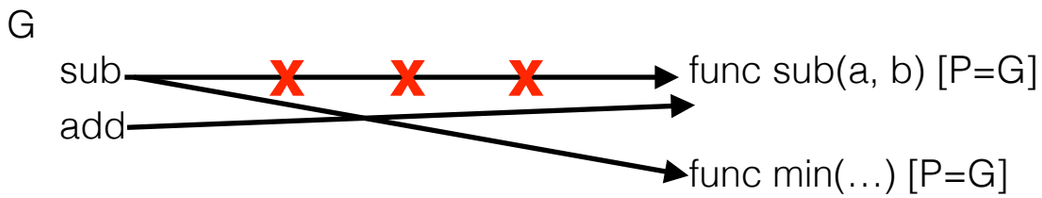
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**

Solution



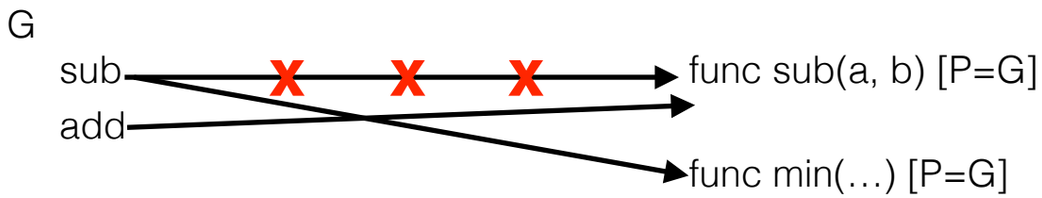
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



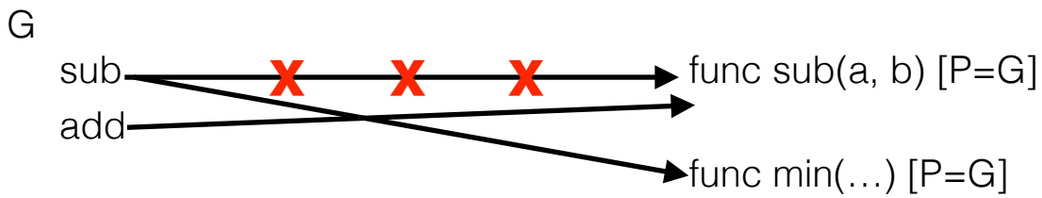
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



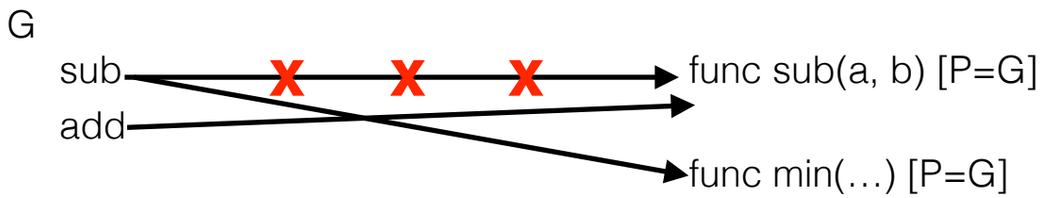
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



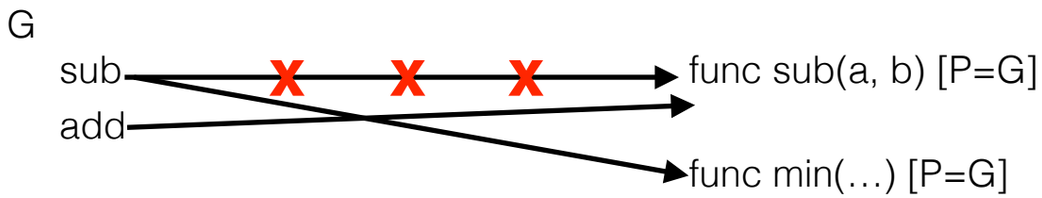
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



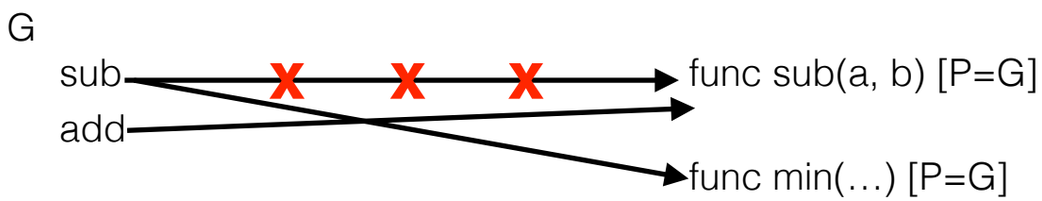
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



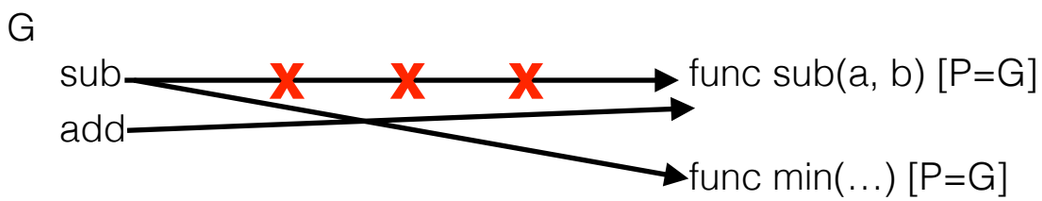
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



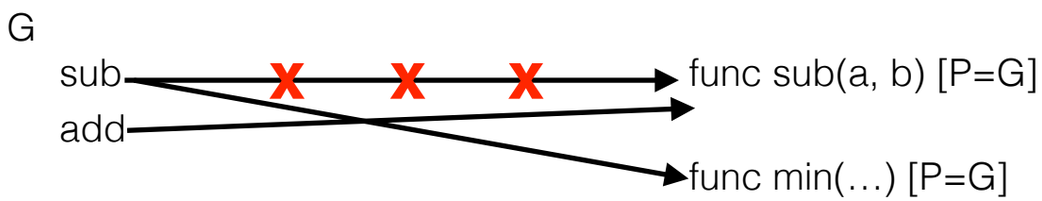
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!

Solution



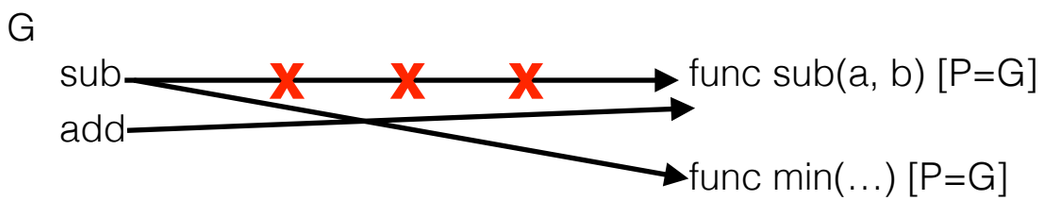
Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

Solution



Draw the environment diagram that results from running the following code.

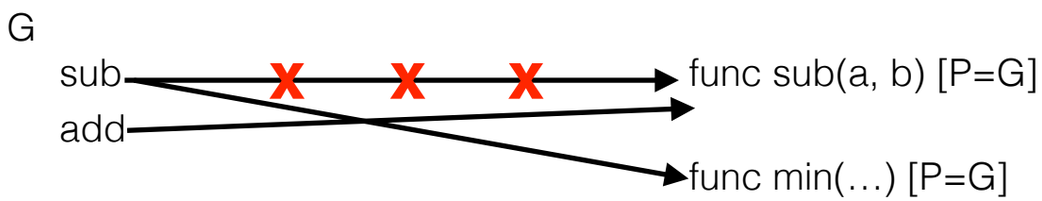
```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2, sub(2, 3)))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

the first function we call is sub, which is really min.
min is built in so we do not need to draw a frame.

Solution



Draw the environment diagram that results from running the following code.

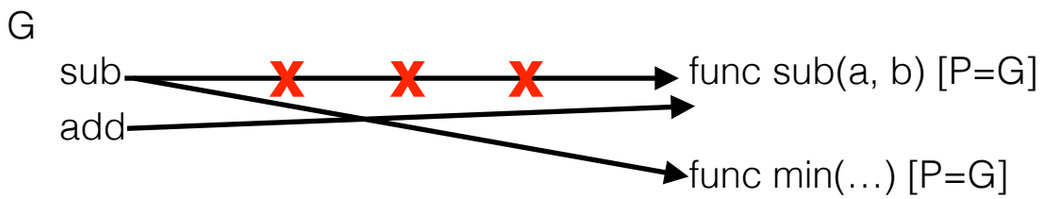
```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

the first function we call is sub, which is really min.
min is built in so we do not need to draw a frame.

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

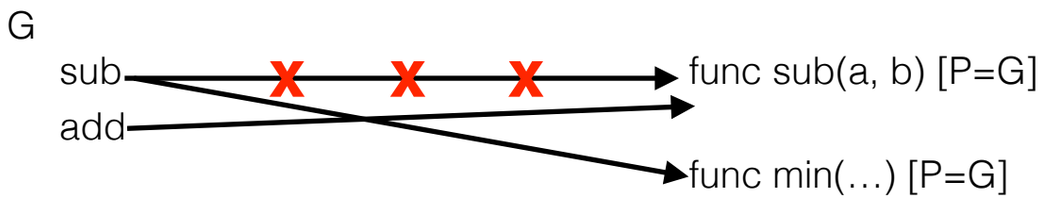
Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

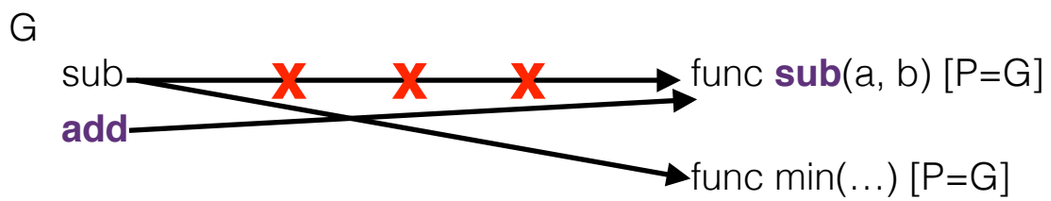
- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

  next is add, which is really sub. we pass in 2 and 2

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

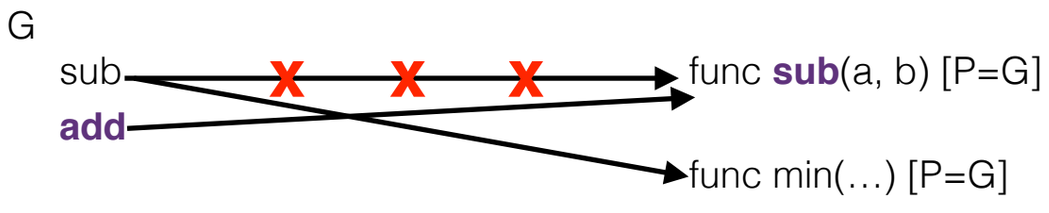
- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

  next is add, which is really sub. we pass in 2 and 2

Solution



f1: **sub** [P=G]
  a: 2
  b: 2

Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

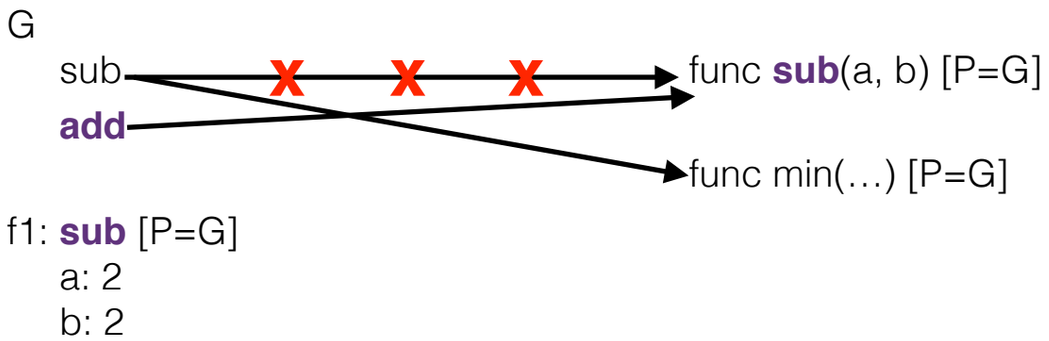
  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

Solution



f1: **sub** [P=G]
  a: 2
  b: 2

Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

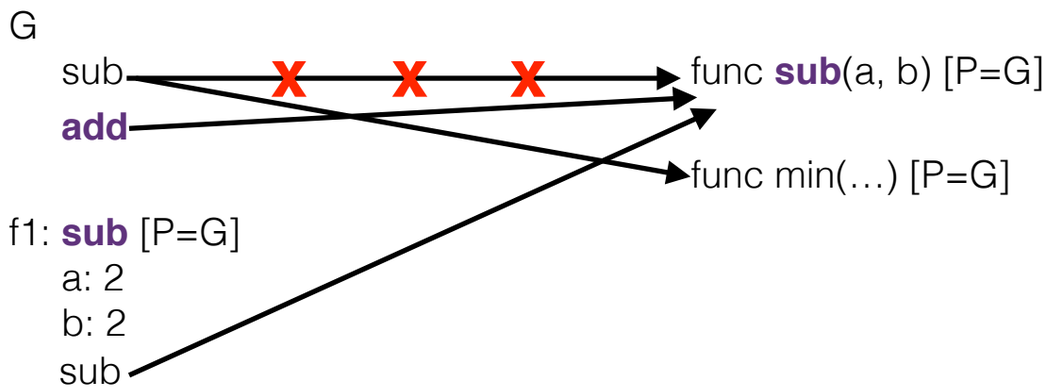
- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

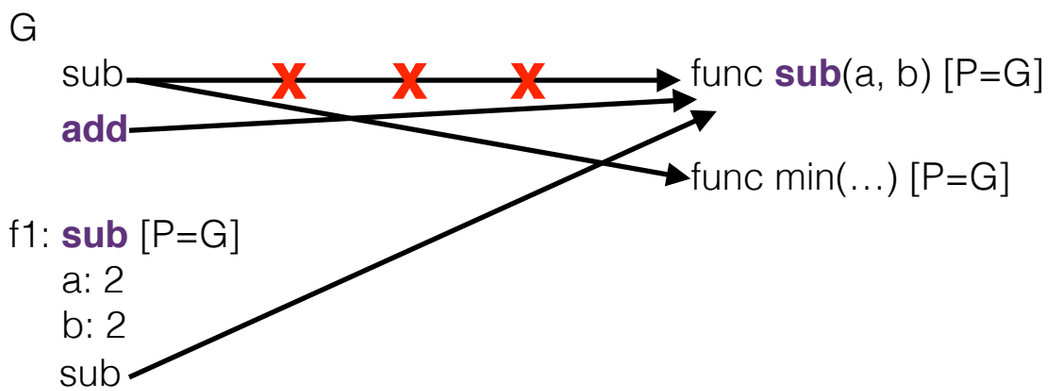
  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

- what do we return from sub? whats a? whats b?

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(add(2,    2    ))
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- `add = sub` is an assignment. whats the value of the LHS? what name do we assign to this value?
- `sub = min` is another assignment. whats the **value**? whats the **name**? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

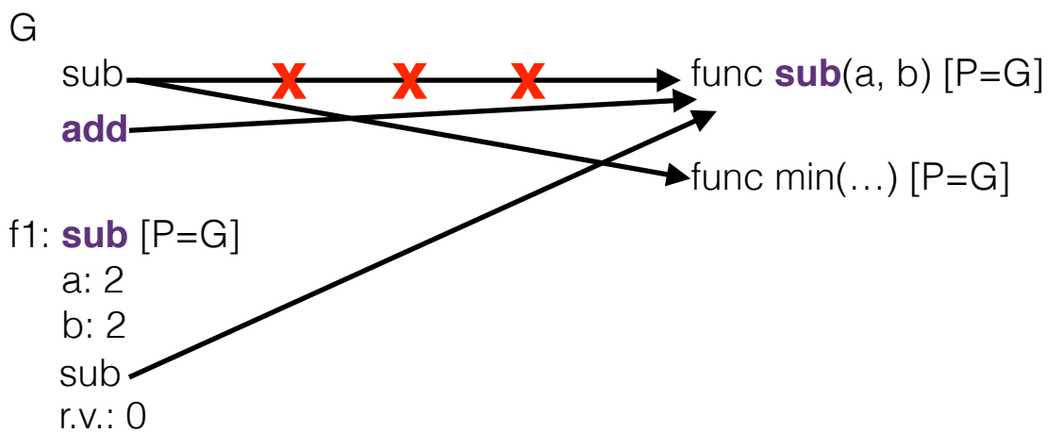
  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

- what do we return from sub? whats a? whats b? a=2, b=2 —> a - b = 0

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(        0          )
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- add = sub is an assignment. whats the value of the LHS? what name do we assign to this value?
- sub = min is another assignment. whats the value? whats the name? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

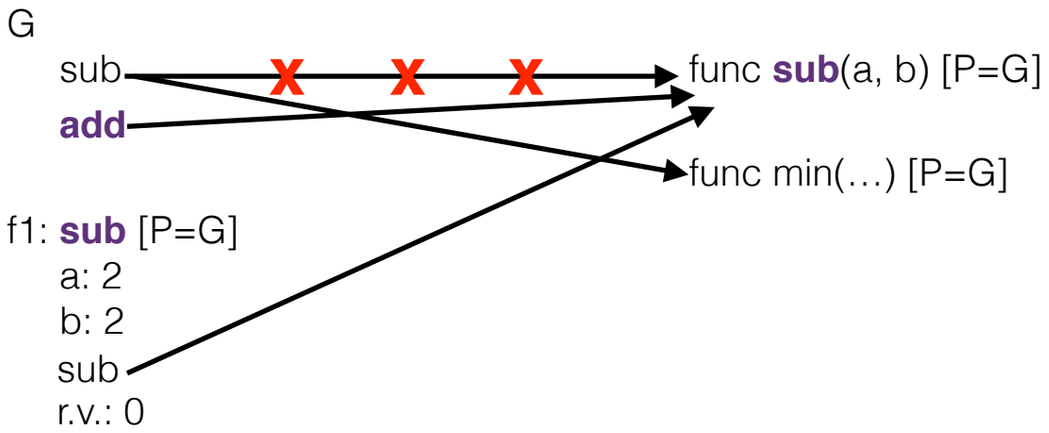
  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

- what do we return from sub? whats a? whats b? a=2, b=2 —> a - b = 0

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(        0          )
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- add = sub is an assignment. whats the value of the LHS? what name do we assign to this value?
- sub = min is another assignment. whats the value? whats the name? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

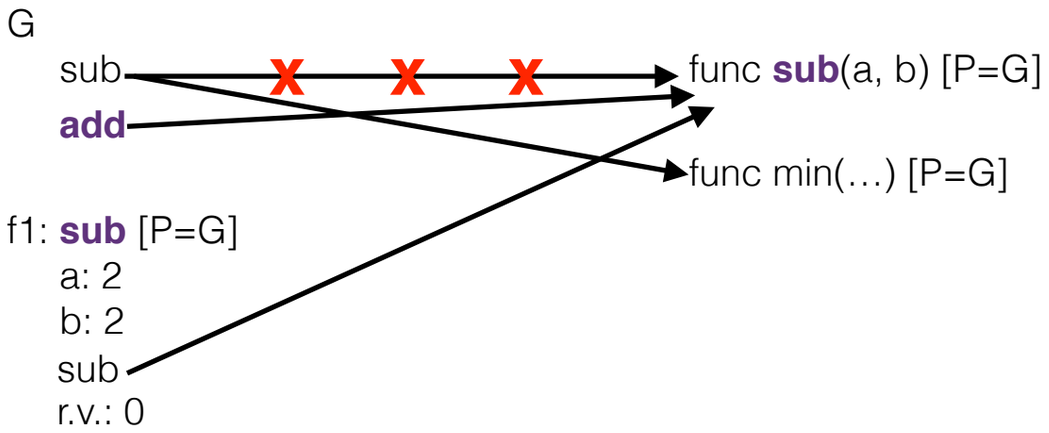
  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

- what do we return from sub? whats a? whats b? a=2, b=2 —> a - b = 0
- finally, what do we print?

Solution



Draw the environment diagram that results from running the following code.

```
from operator import add
def sub(a, b):
    sub = add
    return a - b
add = sub
sub = min
print(        0          )
```

Reasoning

- ignore the import
- first line is a def. (write the name and point it at the function signature)
- add = sub is an assignment. whats the value of the LHS? what name do we assign to this value?
- sub = min is another assignment. whats the value? whats the name? recall that we cannot have the same name bound to 2 values in the same **frame**
- now we will walk through the order of function calls. remember, operator then operands!
- what's the first function we call? what's that functions intrinsic name? do we need to draw a frame for this function?

  the first function we call is sub, which is really min.
  min is built in so we do not need to draw a frame.

- what's the next function we call? what's its intrinsic name? what arguments do we pass in?

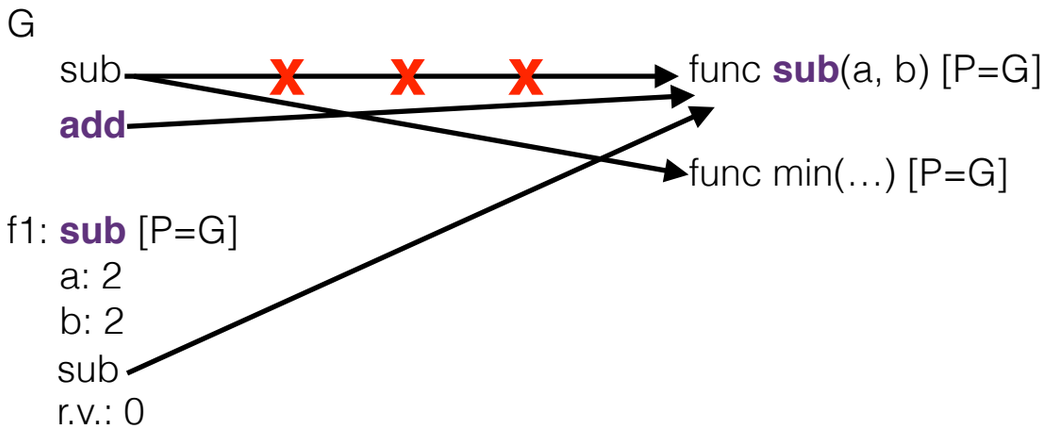
  next is add, which is really sub. we pass in 2 and 2

- now we are ready to execute the body of the function. `sub = add` is an assignment statement. whats the value of the RHS? what name do we bind?

  we look for add in f1. its not there so we look in G. add points to sub.
  the LHS says sub, so we must bind the name sub

- what do we return from sub? whats a? whats b? a=2, b=2 —> a - b = 0
- finally, what do we print? 0

Solution



Higher Order Functions

What is a higher order function?

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

```
x = 2
def inner(y):
        print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

1

```
x = 2
def inner(y):
        print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

inner is defined inside outer, so its parent is outer

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

1

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

inner is defined inside outer, so its parent is outer

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

1

3

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

1

3

inner is defined inside outer, so its parent is outer

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

1 1 3

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

1 3 1

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

1 3 1

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

1

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

1 3 1

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

3 1

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

when inner looks for x, it will look in its parent, which is G
note this is different from the code on the left!

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

when inner looks for x, it will look in its parent, which is G
note this is different from the code on the left!

Higher Order Functions

What is a higher order function?

Any function that manipulates other functions.

How can we *manipulate* other functions?

Pass a function in as an argument.
Return a function
Both.

You already worked with higher order functions in Lab01! Remember **repeated**?

Ex: What does the code on the left print? right?

```
x = 2
def outer(x):
    def inner(y):
        print(x)
    return inner

outer(1)(3)
```

inner is defined inside outer, so its parent is outer

inner is defined in G, so its parent is G

so when inner looks for x, it will look in its parent (outer)

```
x = 2
def inner(y):
    print(x)
def outer(x):
    return inner

outer(1)(3)
```

when inner looks for x, it will look in its parent, which is G
note this is different from the code on the left!

Code on the left will print 1. Code on the right will print 2.

3.4 #1

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)


>>> f = outer(10)
>>> f(4)


>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition
- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

  no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

  no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

  no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

  we just defined inner as a function. so the value of the name inner is the function called inner

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

61

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

  no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

  we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

What will Python display?

```
def outer(m):
    def inner(n):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

10

4

10 - 4

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

  we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
  is this a function call?

  no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

  we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

  4

- what is n - m?

  n is 10, m is 4. n - m = 10 - 4 = 6

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

10

4

10 - 4

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

10

4

10 - 4

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

5

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(m):
    def inner(n):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

What will Python display?

```
def outer(n):
    def inner(m):
        return n - m
    return inner
```

```
>>> outer(61)
```

<func …>

```
>>> f = outer(10)
>>> f(4)
```

6

```
>>> outer(5)(4)
```

1

Reasoning

- first we call `outer`. what do we pass in as the argument?

we pass in 61.

- now we are ready to execute the body of `outer`. the first thing we encounter is another function definition

- after defining `inner`, we encounter the line: `return inner`
is this a function call?

no, inner is not a function call. there are no parenthesis! another way to tell its not a function call is by noticing there are no arguments

- since `inner` is not a function call, we must be trying to return the **value** of the **name** `inner`. what is the value of `inner`?

we just defined inner as a function. so the value of the name inner is the function called inner

- therefore we return the function called `inner`. Python will display this as something gross, but we got the important fact: calling `outer` will return another function
- in the next line (`f = outer(10)`), we again call `outer`, which we know returns the function `inner`, but this time we **bind** the returned function to the **name** `f`
- now we do the function call `f(4)`. we know that f is really just `inner`, and we are passing in 4. what does m get bound to inside `inner`?

4

- what is n - m?

n is 10, m is 4. n - m = 10 - 4 = 6

- try the last line! keep track of what n and m are

3.2 #1

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

- since we want to iterate over all values from `1` to `n`, we need something to keep our place

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

- since we want to iterate over all values from `1` to `n`, we need something to keep our place

- we want to check if calling `cond` on each number from `1` to `n` returns `true`. so we need to do the following somewhere in our solution: `cond(#)`

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

- since we want to iterate over all values from `1` to `n`, we need something to keep our place

- we want to check if calling `cond` on each number from `1` to `n` returns `true`. so we need to do the following somewhere in our solution: `cond(#)`

- if `cond(#)` returns `true`, we want to print #

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

- since we want to iterate over all values from `1` to `n`, we need something to keep our place

- we want to check if calling `cond` on each number from `1` to `n` returns `true`. so we need to do the following somewhere in our solution: `cond(#)`

- if `cond(#)` returns `true`, we want to print #

use these facts to help you write a solution

Write a function that takes in a function `cond` and number `n` and prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(cond, n):
    k = 1
    while k <= n:
        if cond(k):
            print(k)
        k += 1
```

Reasoning

- we need to do some action for every number from `1` to `n`. what python tool should we use?

  while loop!

- since we want to iterate over all values from `1` to `n`, we need something to keep our place

- we want to check if calling `cond` on each number from `1` to `n` returns `true`. so we need to do the following somewhere in our solution: `cond(#)`

- if `cond(#)` returns `true`, we want to print #

use these facts to help you write a solution

3.4 #2

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

- this is very similar to what we just did!

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do?
does this sound familiar?

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?

  the returned function does exactly what `keep_ints` used to do

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?

  the returned function does exactly what `keep_ints` used to do

- instead of keep_ints doing the work, `keep_ints` will define a function that does that work for us

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?

  the returned function does exactly what `keep_ints` used to do

- instead of keep_ints doing the work, `keep_ints` will define a function that does that work for us

use these facts to help you write a solution

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
    def do_work(cond):
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?

  the returned function does exactly what `keep_ints` used to do

- instead of keep_ints doing the work, `keep_ints` will define a function that does that work for us

use these facts to help you write a solution

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
    def do_work(cond):
```

```
    return do_work
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?
  - the returned function does exactly what `keep_ints` used to do

- instead of keep_ints doing the work, `keep_ints` will define a function that does that work for us

use these facts to help you write a solution

Write a function that takes in a function a number `n` and returns another function that takes in one parameter `cond`. The returned function prints the numbers from `1` to `n` for which calling `cond` on that number returns `true`.

```
def keep_ints(n):
    def do_work(cond):
        k = 1
        while k <= n:
            if cond(k):
                print(k)
            k += 1
    return do_work
```

Reasoning

- this is very similar to what we just did!

- what does the returned function do? does this sound familiar?

  the returned function does exactly what `keep_ints` used to do

- instead of keep_ints doing the work, `keep_ints` will define a function that does that work for us

use these facts to help you write a solution