HKN CS 61A
Midterm 1 Review

Spring 2015

Austin Le
J Lee
Sherdil Niyaz

Agenda

Hosted by HKN (hkn.eecs.berkeley.edu)

Office hours from 11AM to 5PM in 290 Cory, 345 Soda.

Check our website for exam archive, course guide, course surveys, tutoring schedule.

- What Will Python Print
- Environment Diagrams
- Higher Order Functions
- Lambda Functions
- Recursion

What will Python print?

```
def while_loop(n):
    i, j = 0, 1
    while i < n:
        j += 1
        while j < n:
            i += 1
            if j % i == 1:
                print(i, j)
            j += 1
        i += 1

while_loop(5)
```

```
>>> while_loop(5)
```

What will Python print?

```
def while_loop(n):
    i, j = 0, 1
    while i < n:
        j += 1
        while j < n:
            i += 1
            if j % i == 1:
                print(i, j)
            j += 1
        i += 1

while_loop(5)
```

```
>>> while_loop(5)
2 3
3 4
```

http://goo.gl/ixcbEm

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
_______
>>> oak
_______
>>> ash = best(3)
_______
>>> ash
_______
>>> best(1)
_______
```

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
pika!
pika!
pika!
>>> oak
_______
>>> ash = best(3)
_______
>>> ash
_______
>>> best(1)
_______
```

Note that there are no quotes around a **printed** string.

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
pika!
pika!
pika!
>>> oak
(nothing is printed)
>>> ash = best(3)
_______
>>> ash
_______
>>> best(1)
_______
```

Note that there are no quotes around a **printed** string.

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
pika!
pika!
pika!
>>> oak
(nothing is printed)
>>> ash = best(3)
pika!
pika!
pika!
>>> ash
_______
>>> best(1)
_______
```

Note that there are no quotes around a **printed** string.

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
pika!
pika!
pika!
>>> oak
(nothing is printed)
>>> ash = best(3)
pika!
pika!
pika!
>>> ash
'squirtle'
>>> best(1)
_____
```

Note that there are no quotes around a **printed** string.

But we do have quotes around the **returned** string!

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = lambda p: pikachu
    if pikachu() < bulbasaur():
        print('mew')
    elif pikachu() == charmander(1):
        print('pikachu!')
    if pikachu() % 2 == 1:
        return 'squirtle'
```

```
>>> oak = best(4)
pika!
pika!
pika!
>>> oak
(nothing is printed)
>>> ash = best(3)
pika!
pika!
pika!
>>> ash
'squirtle'
>>> best(1)
pika!
mew
pika!
'squirtle'
```

Note that there are no quotes around a **printed** string.

But we do have quotes around the **returned** string!

What will Python print?

```
def print_moar(stuff):
    i = 0
    while stuff and i < 100:
        if not stuff:
            print('best champion')
        stuff = print(stuff, print('worst champion'))
        i += 1
    return stuff

>>> truth = print_moar('teemo')
________
>>> truth
________
```

What will Python print?

```
def print_moar(stuff):
    i = 0
    while stuff and i < 100:
        if not stuff:
            print('best champion')
        stuff = print(stuff, print('worst champion'))
        i += 1
    return stuff

>>> truth = print_moar('teemo')
worst champion
teemo None
>>> truth
_______
```

What will Python print?

```
def print_moar(stuff):
    i = 0
    while stuff and i < 100:
        if not stuff:
            print('best champion')
        stuff = print(stuff, print('worst champion'))
        i += 1
    return stuff

>>> truth = print_moar('teemo')
worst champion
teemo None
>>> truth
(nothing is printed)
```

Boolean Expressions

BONUS!

For reference, look at lab 2 titled "Control"

- A *boolean expression* is one that evaluates to either `True`, `False`, or sometimes an `Error`.
- When evaluating boolean expressions, we follow the same rules as those used for evaluating other statements and function calls.
- The order of operations for booleans (from highest priority to lowest) is: **`not`, `and`, `or`**

  The following will evaluate to `True`:

  ```
  True and not False or not True and False
  ```

  You can rewrite it using parentheses to make it more clear:

  ```
  (True and (not False)) or ((not True) and False)
  ```

More Boolean Expressions

BONUS!

Short-circuiting

- Expressions are evaluated from left to right in Python.
- Expressions with **and** will evaluate to `True` only if *all* the operands are `True`. For multiple **and** expressions, Python will go left to right until it runs into the first `False` value -- then the expression will immediately evaluate to `False`.
- Expressions with **or** will evaluate to `True` if *at least one* of the operands is `True`. For multiple **or** expressions, Python will go left to right until it runs into the first `True` value -- then the expression will immediately evaluate to `True`. For example:

```
>>> 5 > 6 or 4 == 2*2 or 1/0
>>> (5 > 6) or (4 == 2*2) or 1/0        [rewritten]
True
```

Number Fun

Write a function that prints out the first `n` Fibonacci prime numbers AFTER 1 (i.e. the first valid number is 2). A Fibonacci prime number is a prime number that is also a Fibonacci number. Assume that you have a function `is_prime(x)` that returns `True` if `x` is prime and `False` if not.

```
def nth_fib_prime(n):
    """
    >>> nth_fib_prime(4)
    2
    3
    5
    13
    """
```

Number Fun

Write a function that prints out the first `n` Fibonacci prime numbers AFTER 1 (i.e. the first valid number is 2). A Fibonacci prime number is a prime number that is also a Fibonacci number. Assume that you have a function `is_prime(x)` that returns `True` if `x` is prime and `False` if not.

```
def nth_fib_prime(n):
    count = 0
    curr, next = 2, 3
    while count < n:
        if is_prime(curr):
            print(curr)
            count += 1
        curr, next = next, curr + next
```

Higher-Order Functions

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

Higher-Order Functions

Write the function `commutative` that takes in a function `f` and returns another function. Both `f` and the returned function should take 2 arguments. The returned function returns `True` if the two arguments called could be swapped and still have the same return value when called by `f` and `False` otherwise.

```
def commutative(f):
    """
    >>> commutative(add)(1, 2)
    True # 1+2 == 2+1
    >>> commutative(lambda x, y: x*x+y)(2, 5)
    False # 2*2+5 != 5*5+2
    """
```

Higher-Order Functions

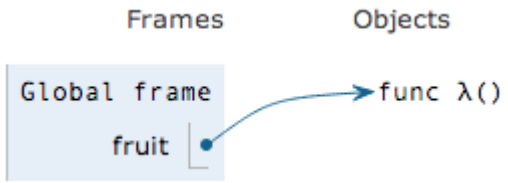
Write the function `commutative` that takes in a function `f` and returns another function. Both `f` and the returned function should take 2 arguments. The returned function returns `True` if the two arguments called could be swapped and still have the same return value when called by `f` and `False` otherwise.

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

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

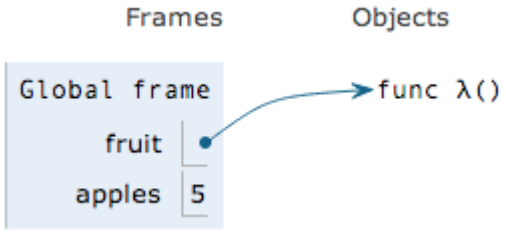
HOF's & Environment Diagrams

```
1 fruit = lambda: apples
2 apples = 5
3 def domo(eats, apples):
4     def rawr():
5         return fruit()
6     print(eats())
7     rawr()
8 domo(fruit, 42)
```



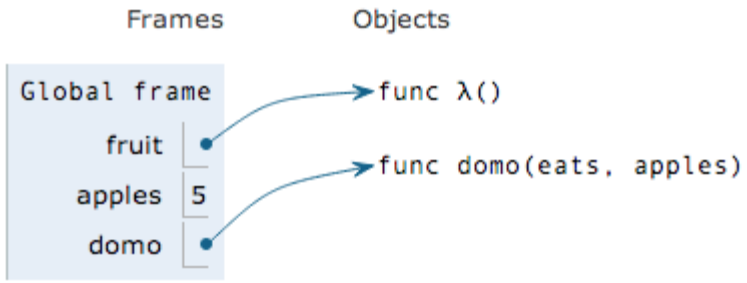
HOF's & Environment Diagrams

```
1 fruit = lambda: apples
2 apples = 5
3 def domo(eats, apples):
4     def rawr():
5         return fruit()
6     print(eats())
7     rawr()
8 domo(fruit, 42)
```



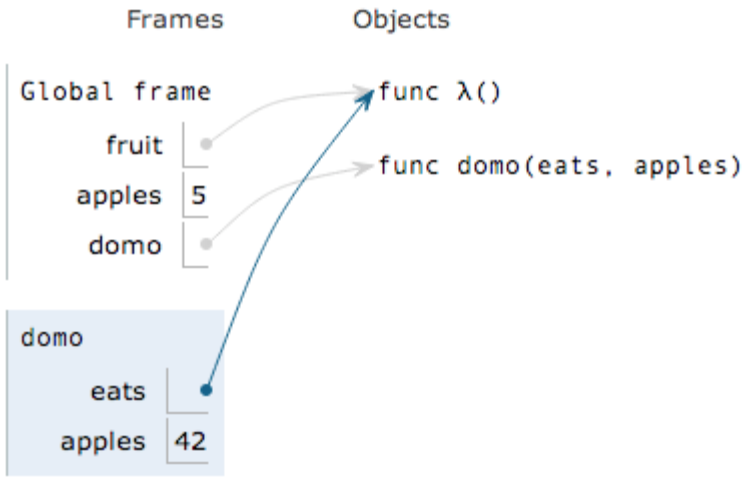
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



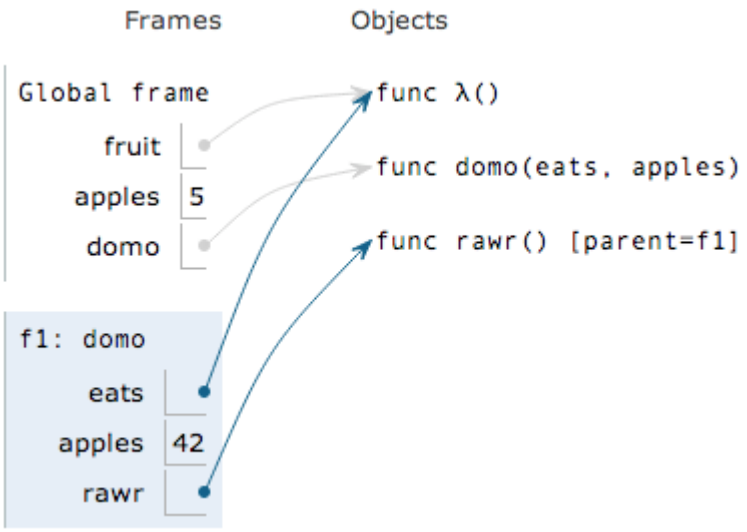
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



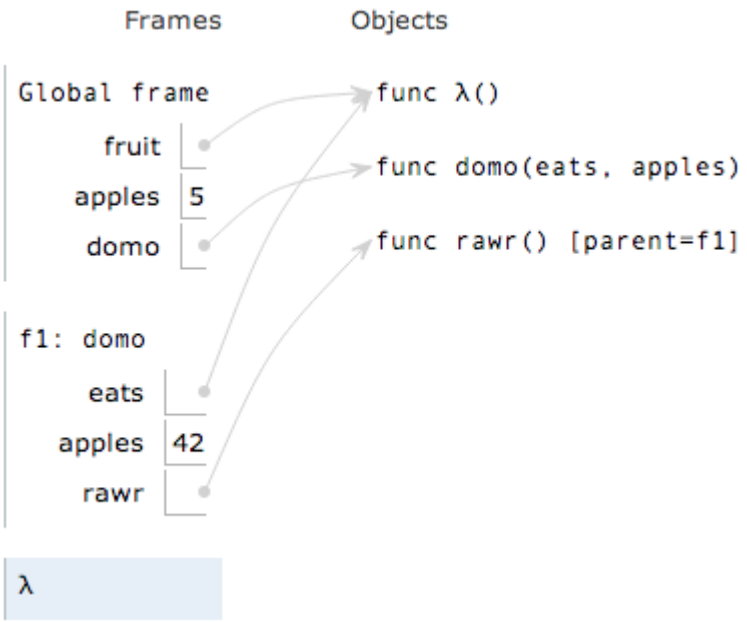
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



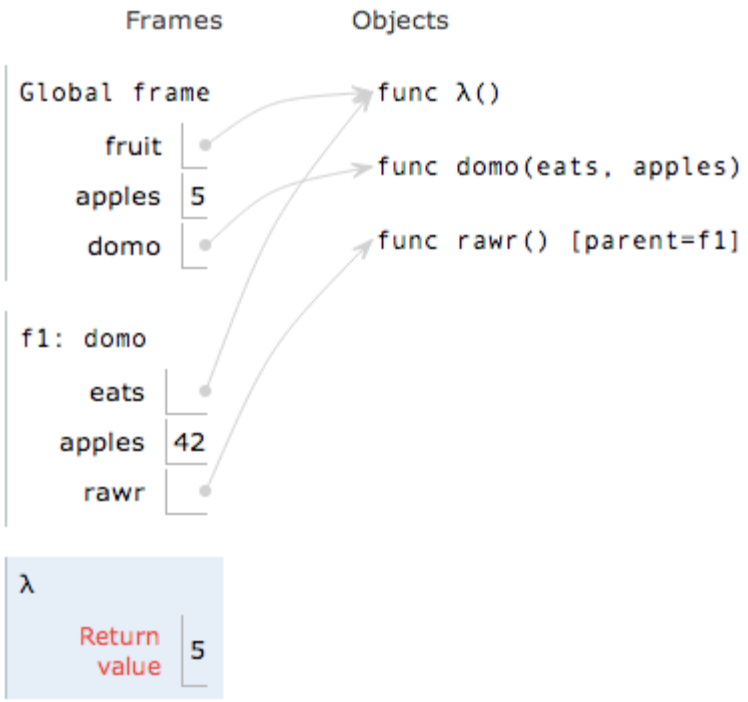
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```

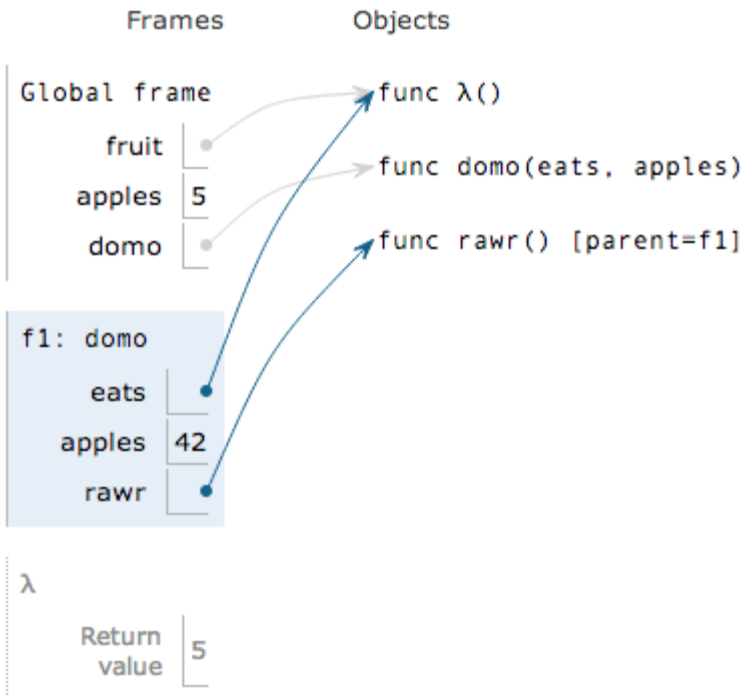


HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```

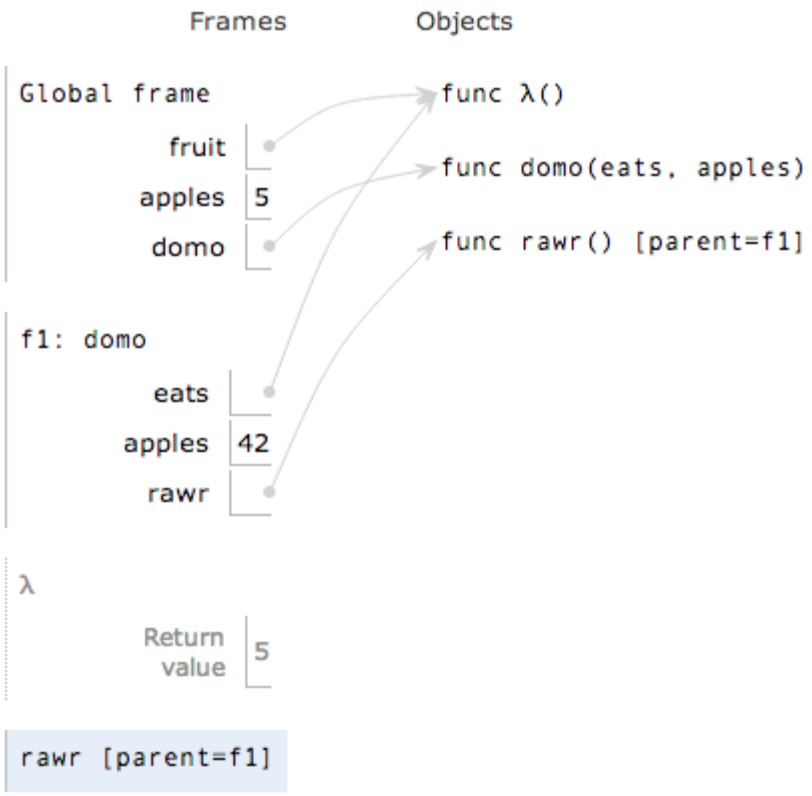
Program output:

```
5
```



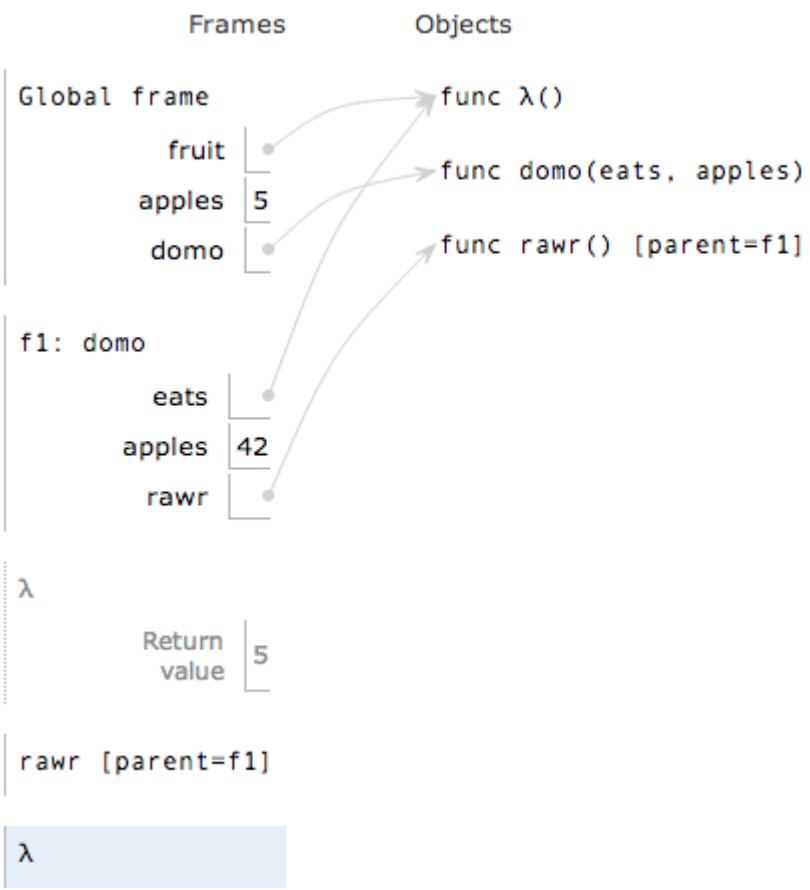
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



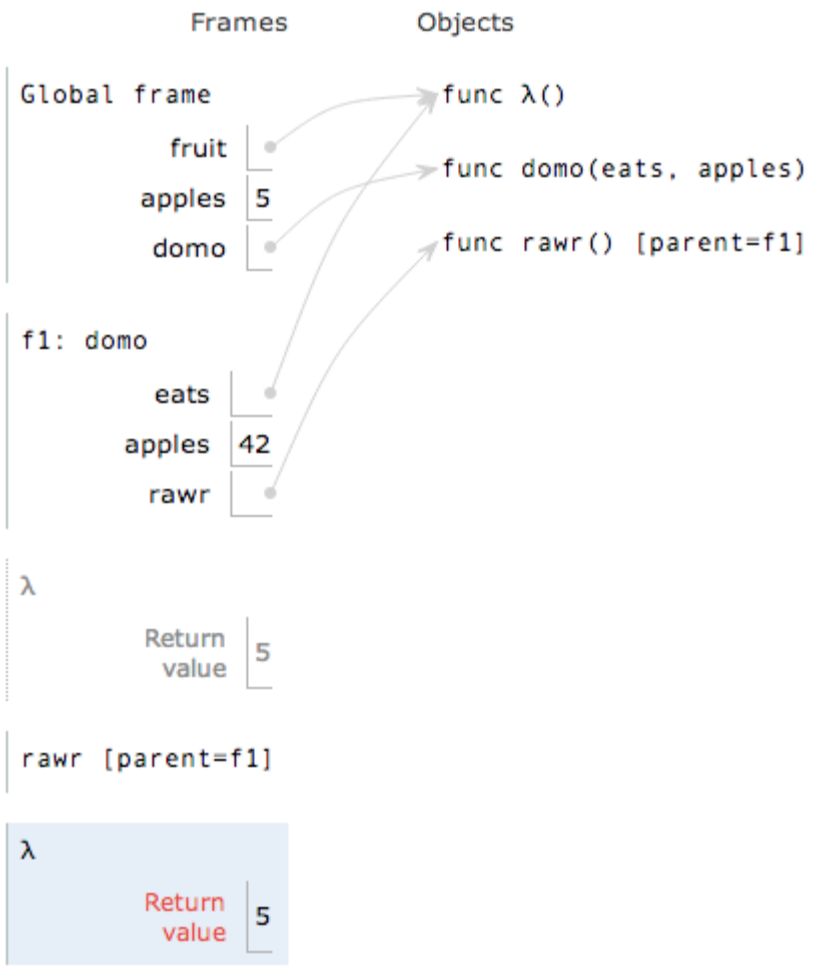
HOF's & Environment Diagrams

```
1 fruit = lambda: apples
2 apples = 5
3 def domo(eats, apples):
4     def rawr():
5         return fruit()
6     print(eats())
7     rawr()
8 domo(fruit, 42)
```



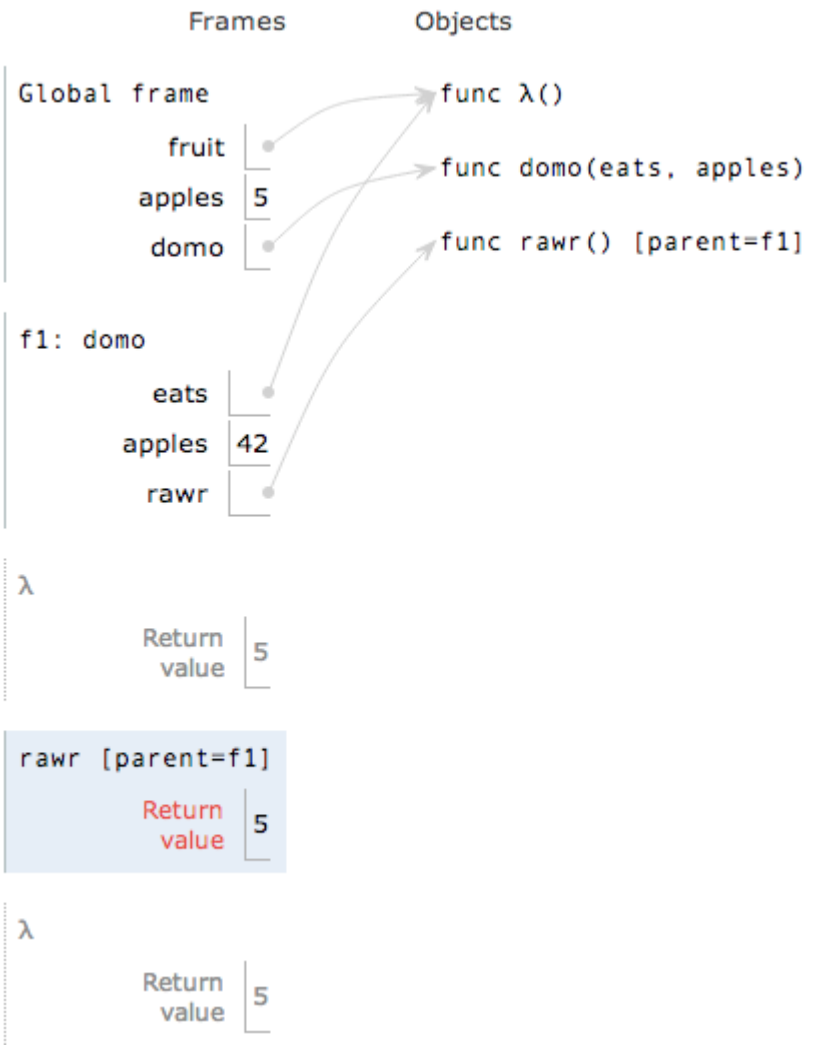
HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```



HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```

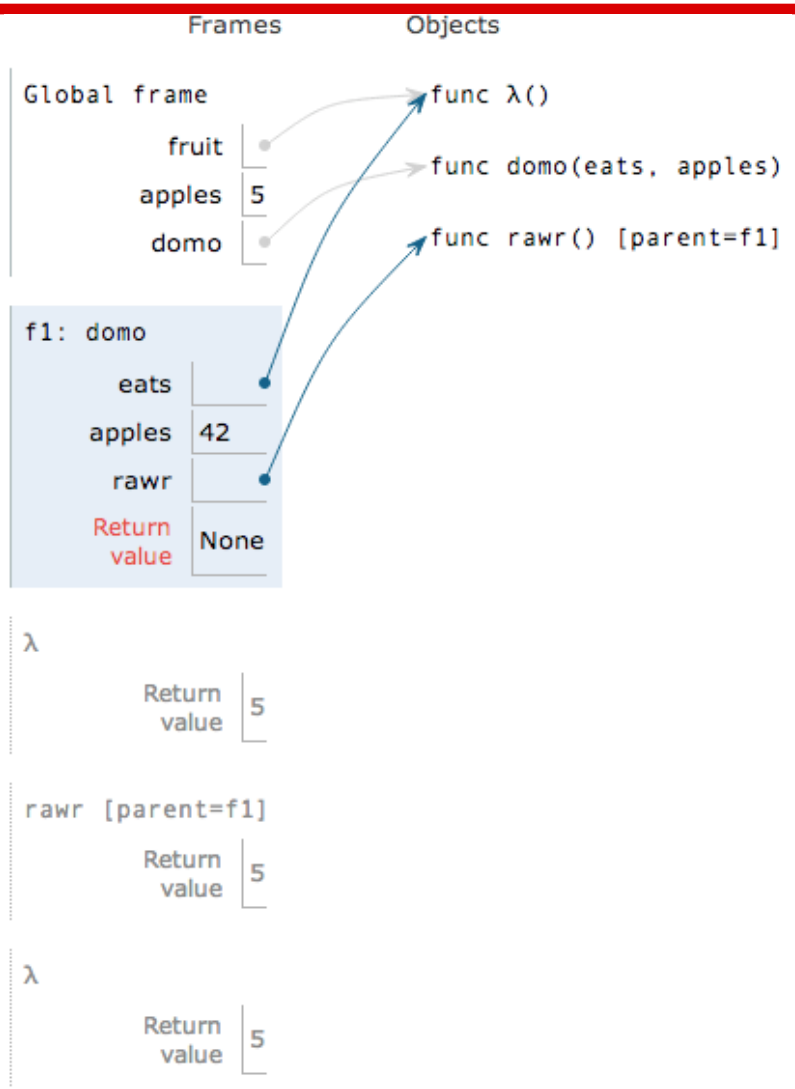


HOF's & Environment Diagrams

```
1  fruit = lambda: apples
2  apples = 5
3  def domo(eats, apples):
4      def rawr():
5          return fruit()
6      print(eats())
7      rawr()
8  domo(fruit, 42)
```

Program output:

```
5
```



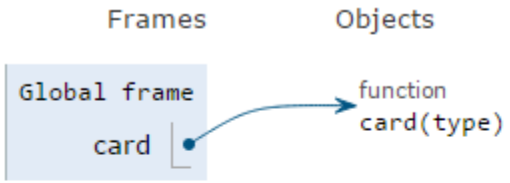
More HOF's & Environment Diagrams! Challenge problem!

```
def card(type):
    card = 5
    def spell(type, secret):
        if type:
            return spell(not type, secret)
        return secret(card, type)
    return spell

new_card = card(2)
my_card = new_card(7, lambda x, y: lambda: x and y)
my_card
_______
my_card()
_______
```

More HOF's & Environment Diagrams! Challenge problem!

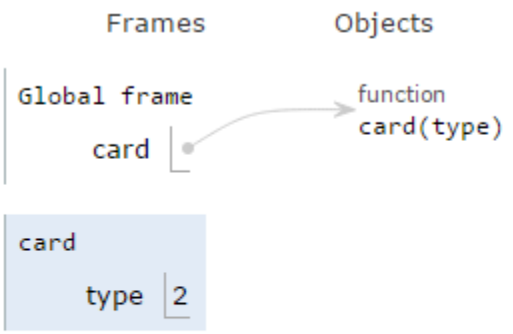
```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```




```
my_card
_______
my_card()
_______
```

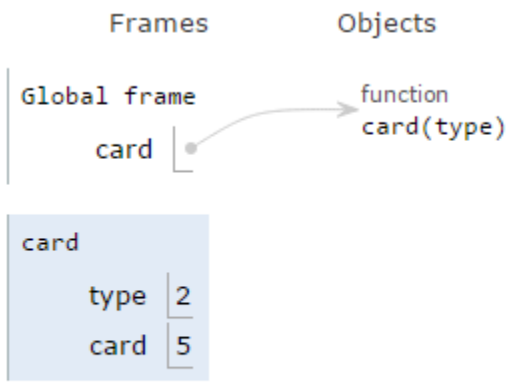
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



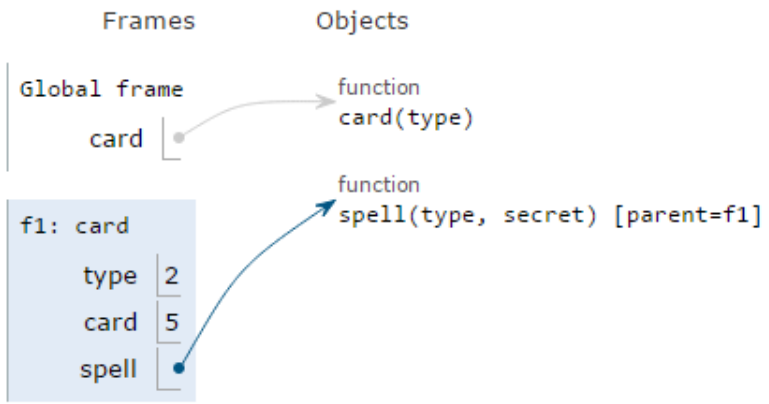
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



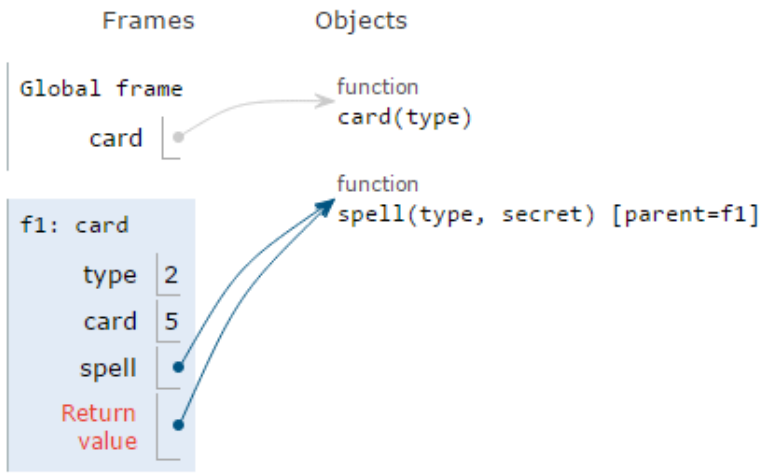
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



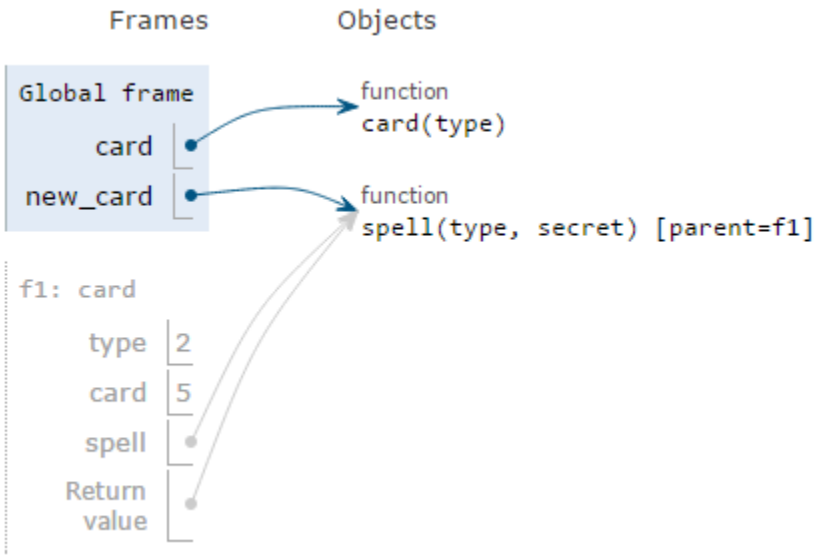
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



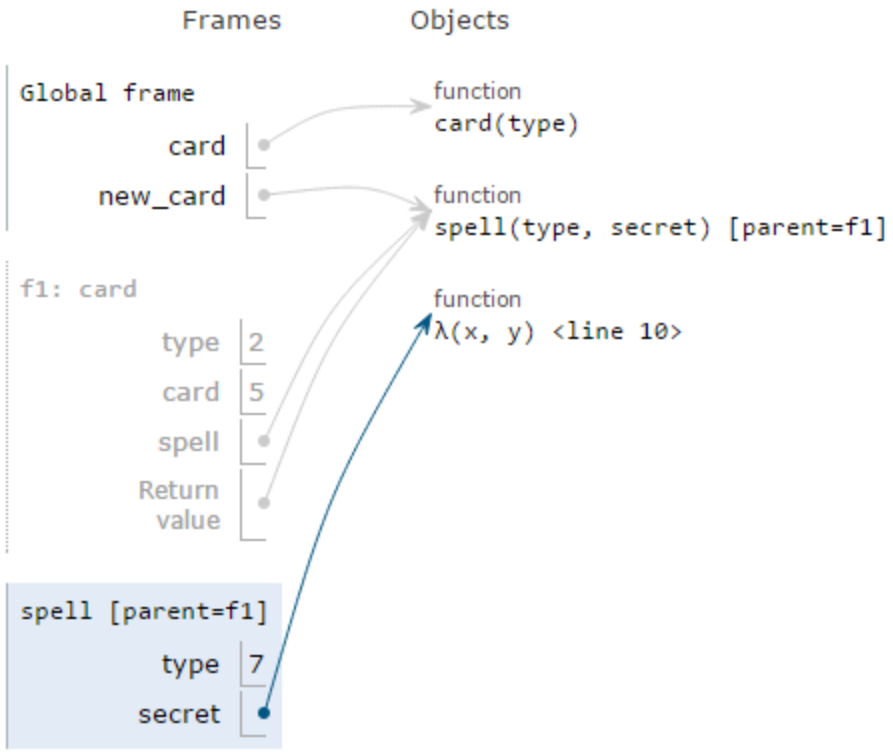
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



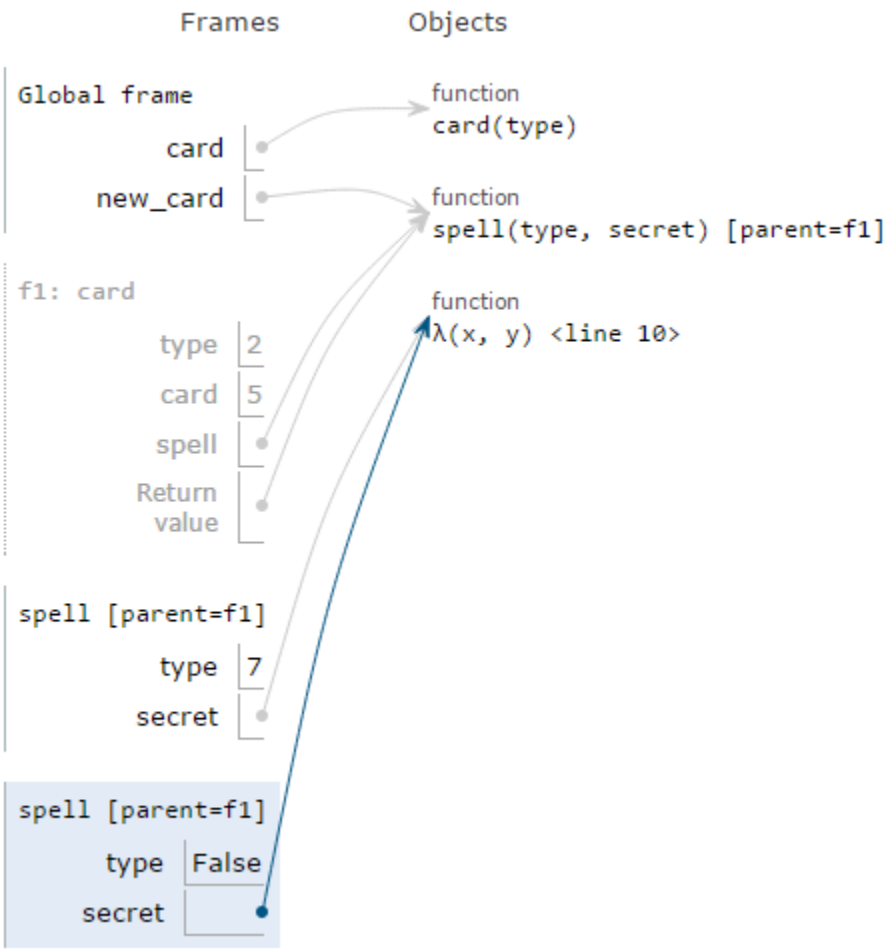
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



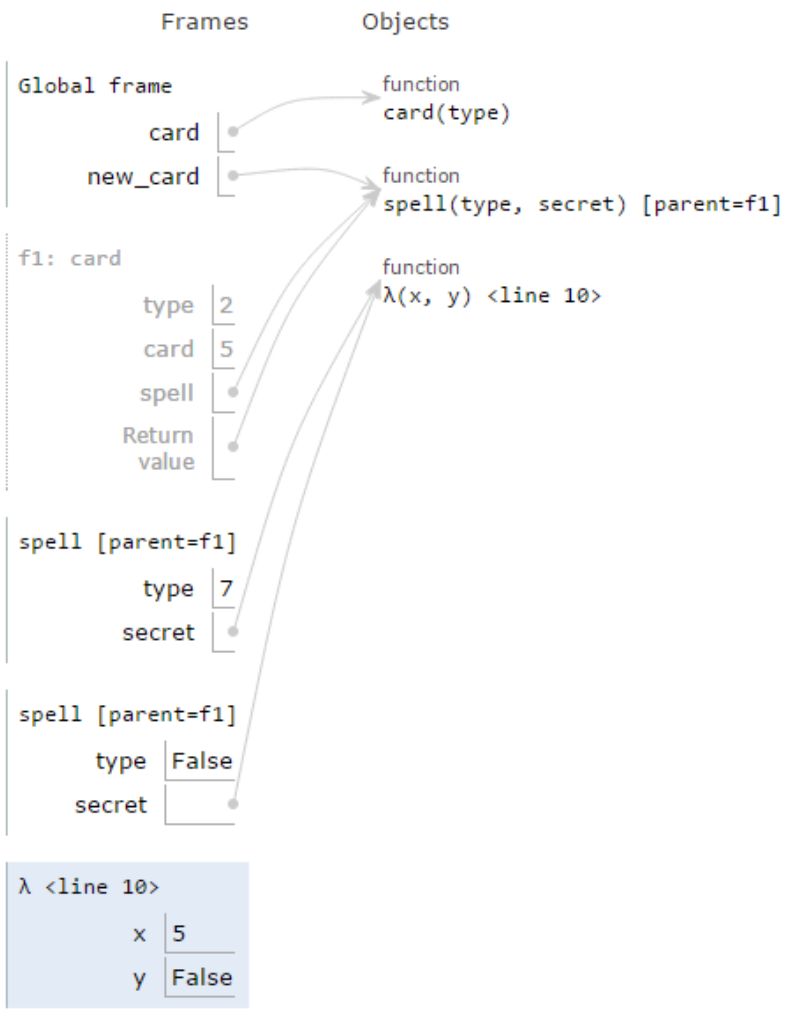
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



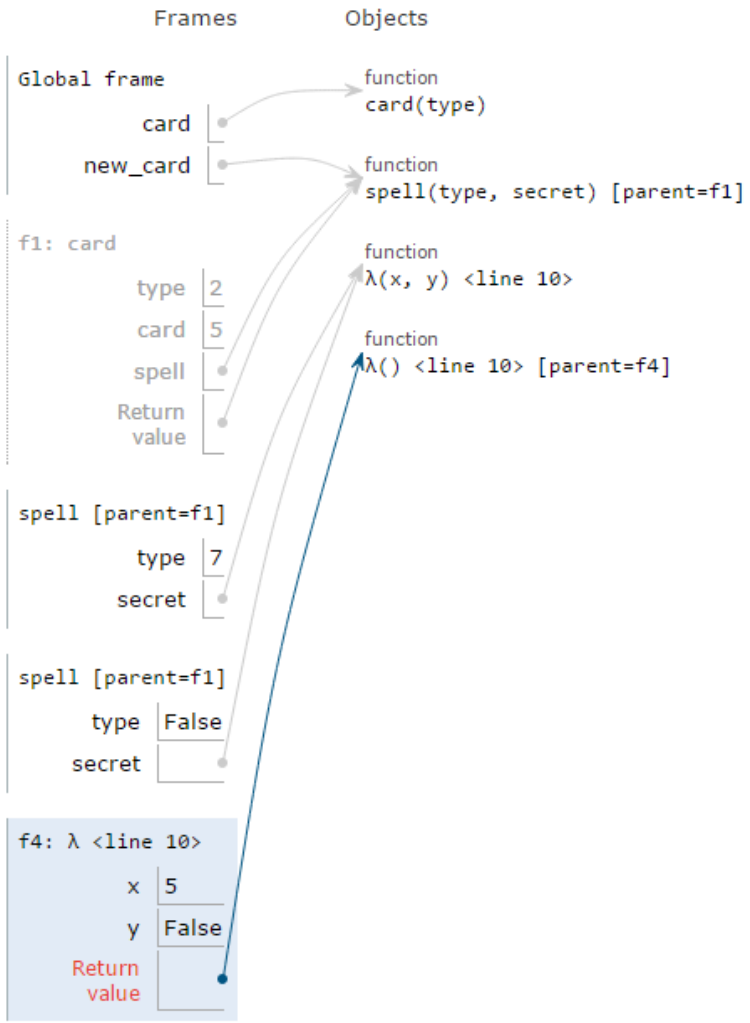
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



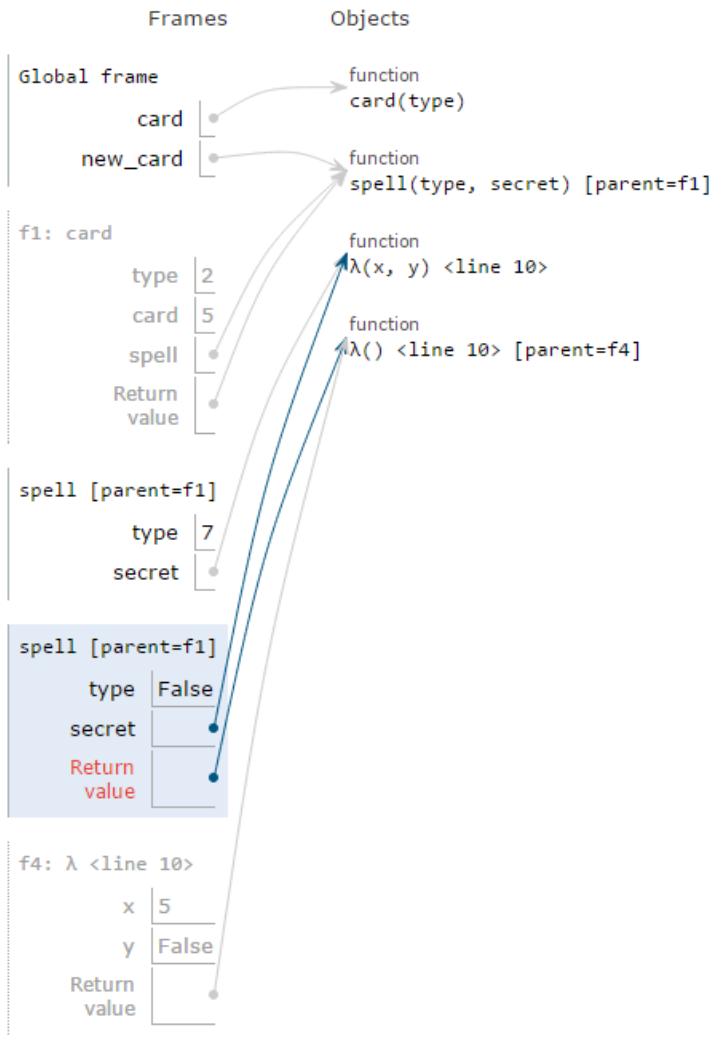
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



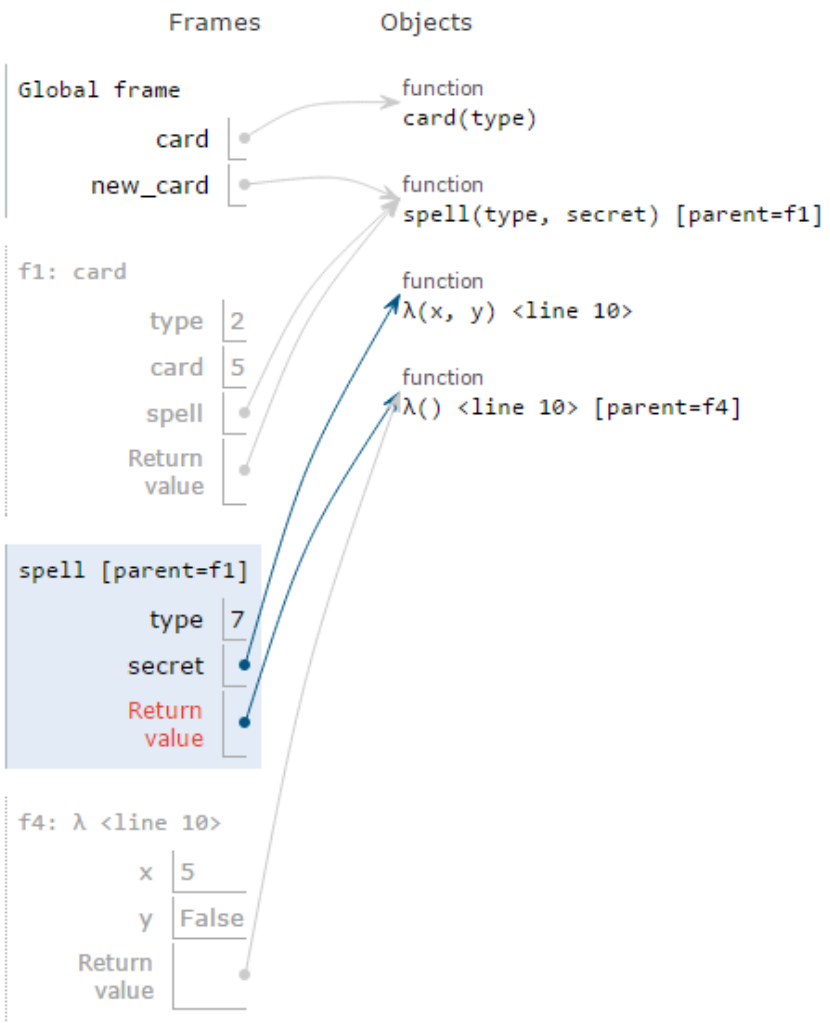
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



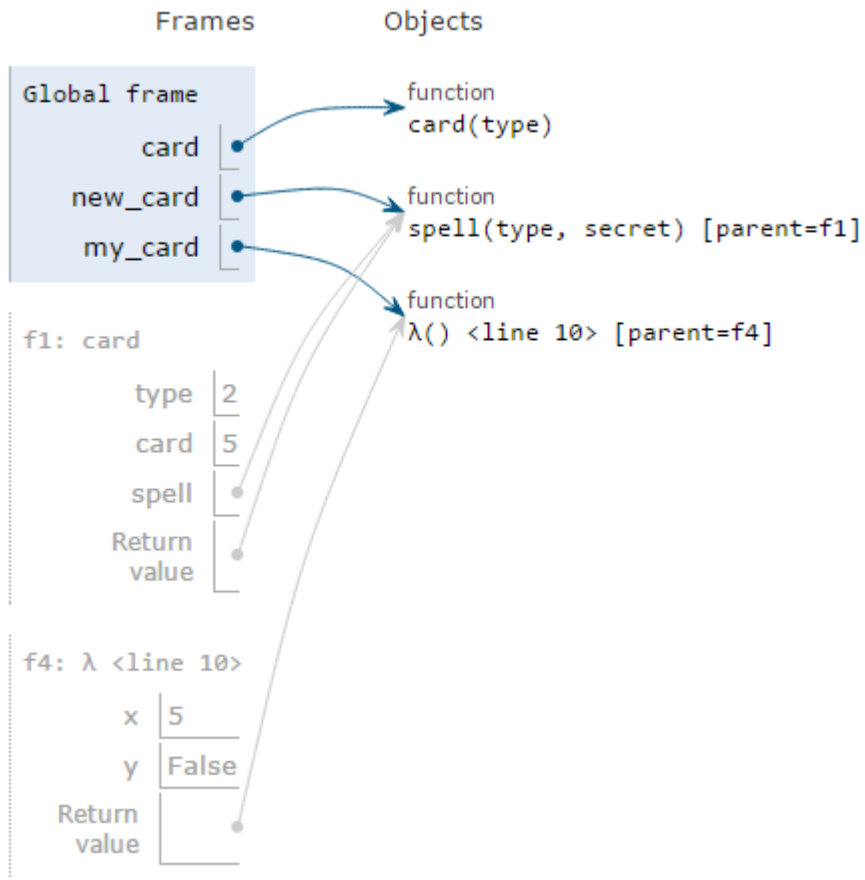
More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```



More HOF's & Environment Diagrams!

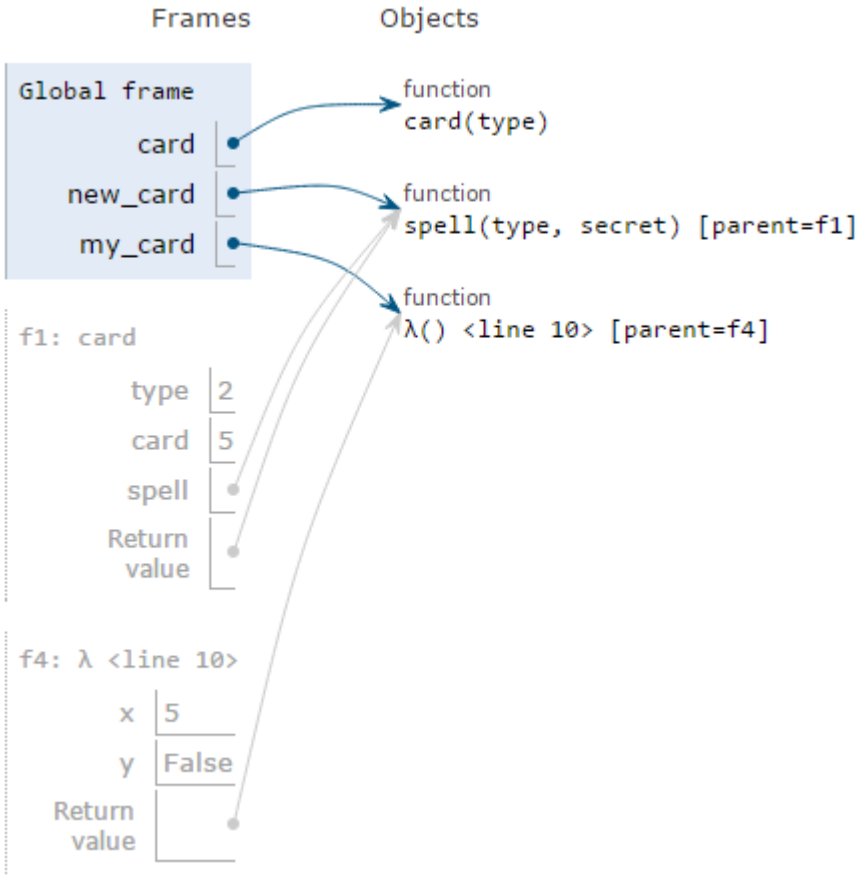
```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```

my_card

_____

my_card()

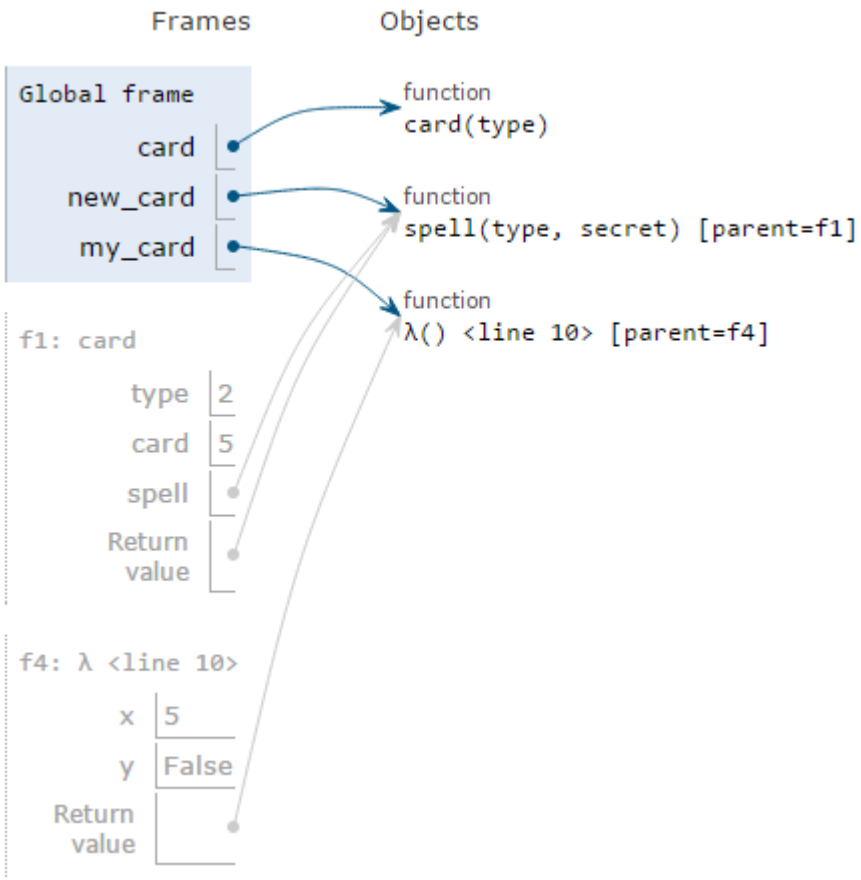
_____



More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```

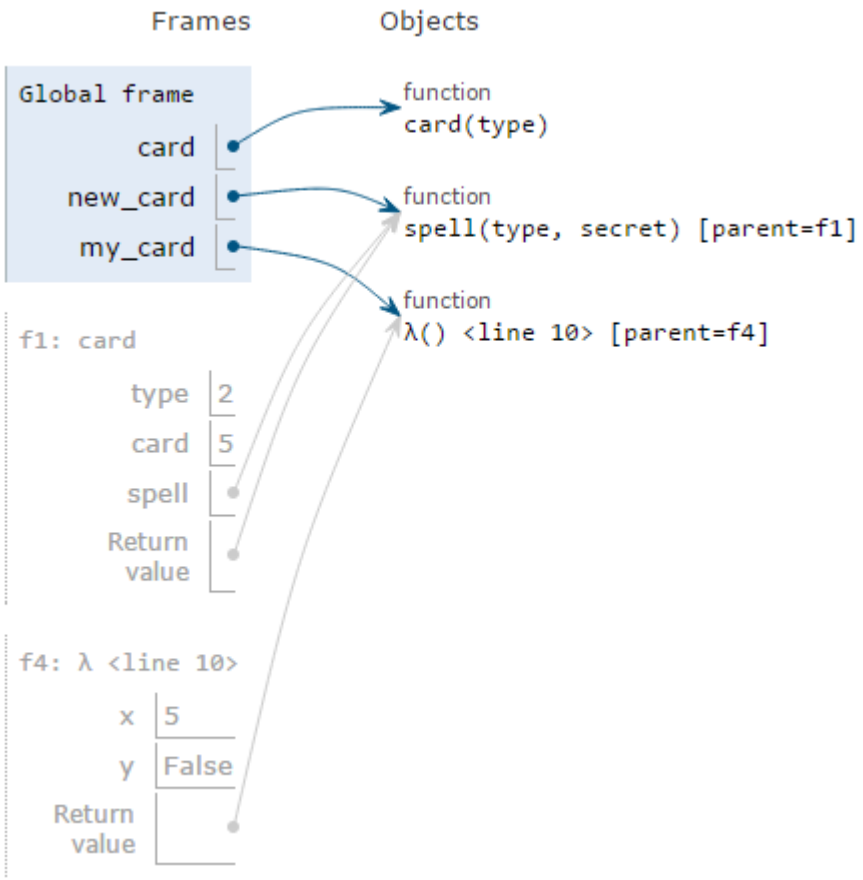
```
my_card
<function <lambda> at ... >
my_card()
_______
```



More HOF's & Environment Diagrams!

```
1  def card(type):
2      card = 5
3      def spell(type, secret):
4          if type:
5              return spell(not type, secret)
6          return secret(card, type)
7      return spell
8
9  new_card = card(2)
10 my_card = new_card(7, lambda x, y: lambda: x and y)
```

```
my_card
<function <lambda> at ... >
my_card()
False
```

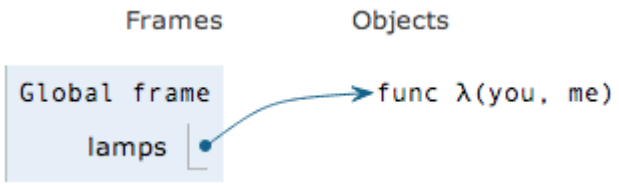


PythonTutor: http://tinyurl.com/cardenvdia

Extra practice!

HOF's & Environment Diagrams

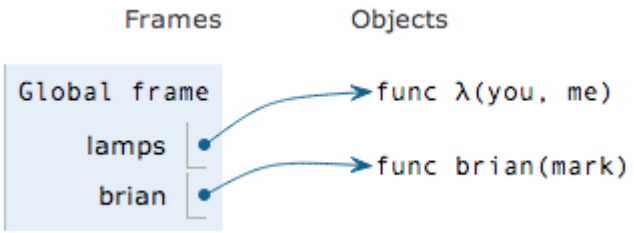
```
1 lamps = lambda you, me: you + me
2 def brian(mark):
3     def flour(based):
4         return based(mark)
5     mark = 'hi'
6     return flour
7 lamps, brian = brian, lamps
8 answer = lamps(brian(3, 2))
9 answer(print)
```



Extra practice!

HOF's & Environment Diagrams

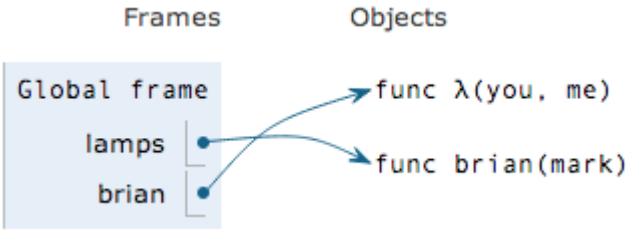
```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```



Extra practice!

HOF's & Environment Diagrams

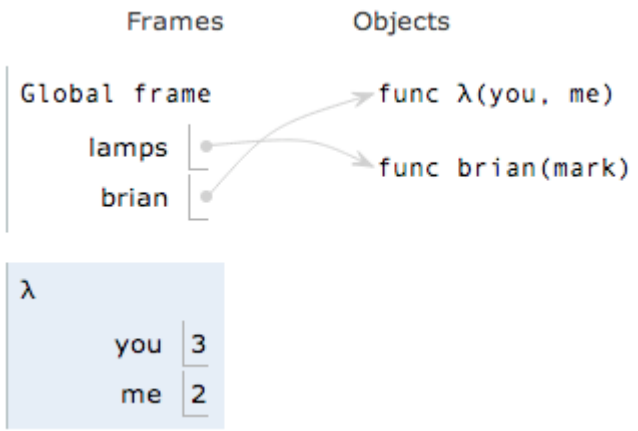
```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```



Extra practice!

HOF's & Environment Diagrams

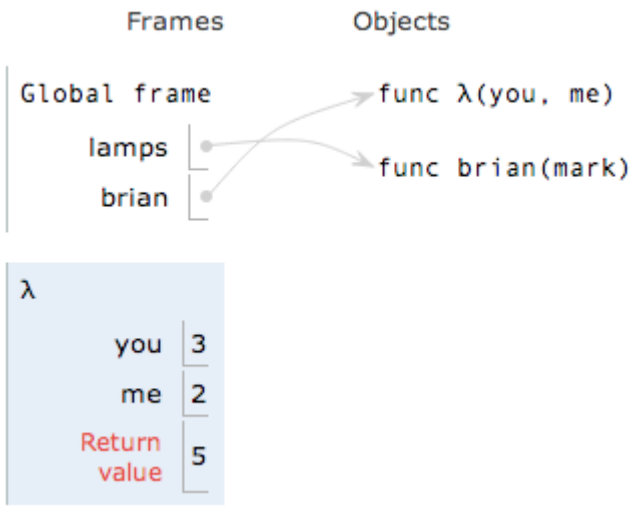
```
1 lamps = lambda you, me: you + me
2 def brian(mark):
3     def flour(based):
4         return based(mark)
5     mark = 'hi'
6     return flour
7 lamps, brian = brian, lamps
8 answer = lamps(brian(3, 2))
9 answer(print)
```



Extra practice!

HOF's & Environment Diagrams

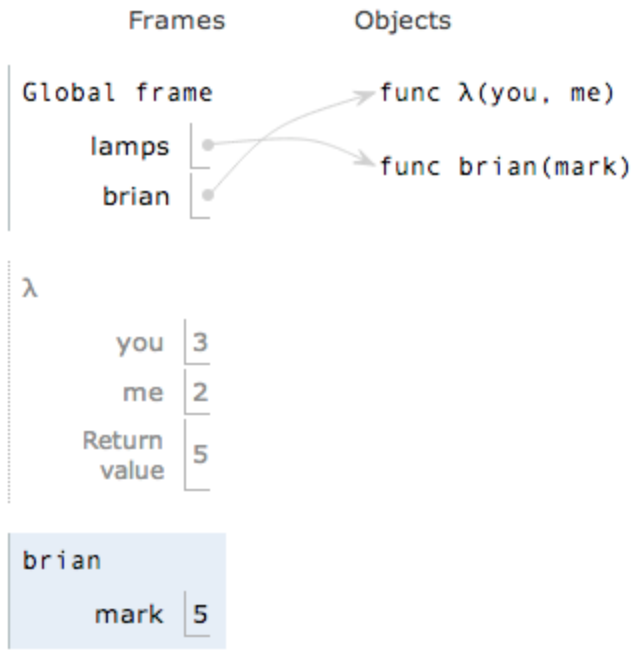
```
1 lamps = lambda you, me: you + me
2 def brian(mark):
3     def flour(based):
4         return based(mark)
5     mark = 'hi'
6     return flour
7 lamps, brian = brian, lamps
8 answer = lamps(brian(3, 2))
9 answer(print)
```



Extra practice!

HOF's & Environment Diagrams

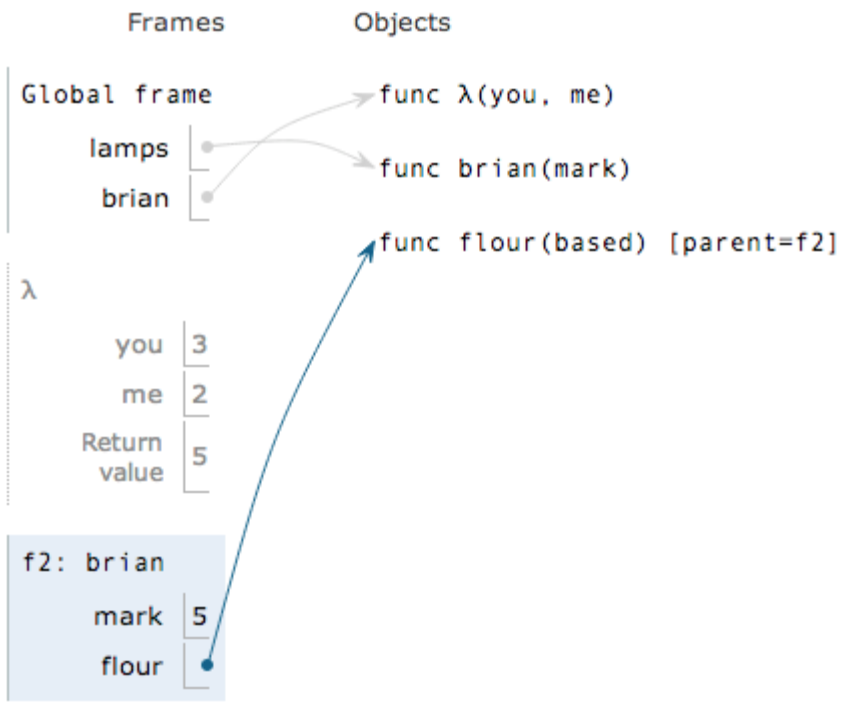
```
1 lamps = lambda you, me: you + me
2 def brian(mark):
3     def flour(based):
4         return based(mark)
5     mark = 'hi'
6     return flour
7 lamps, brian = brian, lamps
8 answer = lamps(brian(3, 2))
9 answer(print)
```



Extra practice!

HOF's & Environment Diagrams

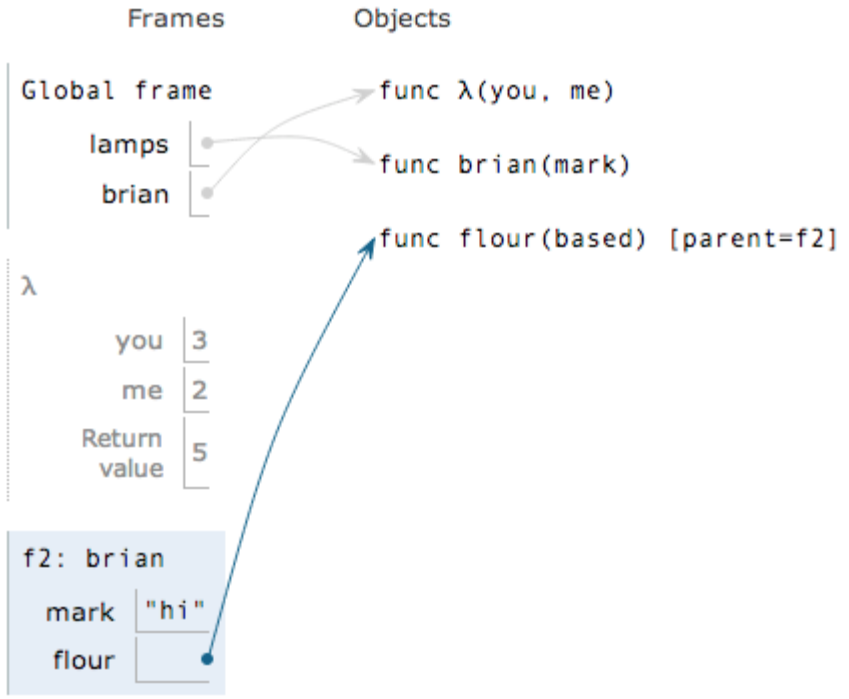
```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```



Extra practice!

HOF's & Environment Diagrams

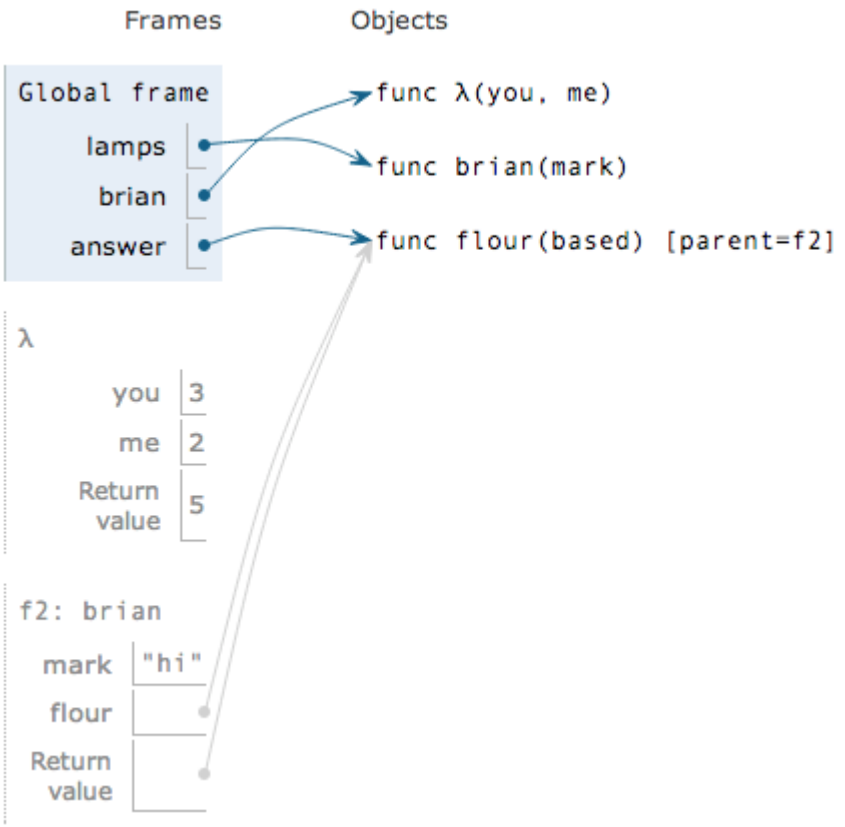
```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```



Extra practice!

HOF's & Environment Diagrams

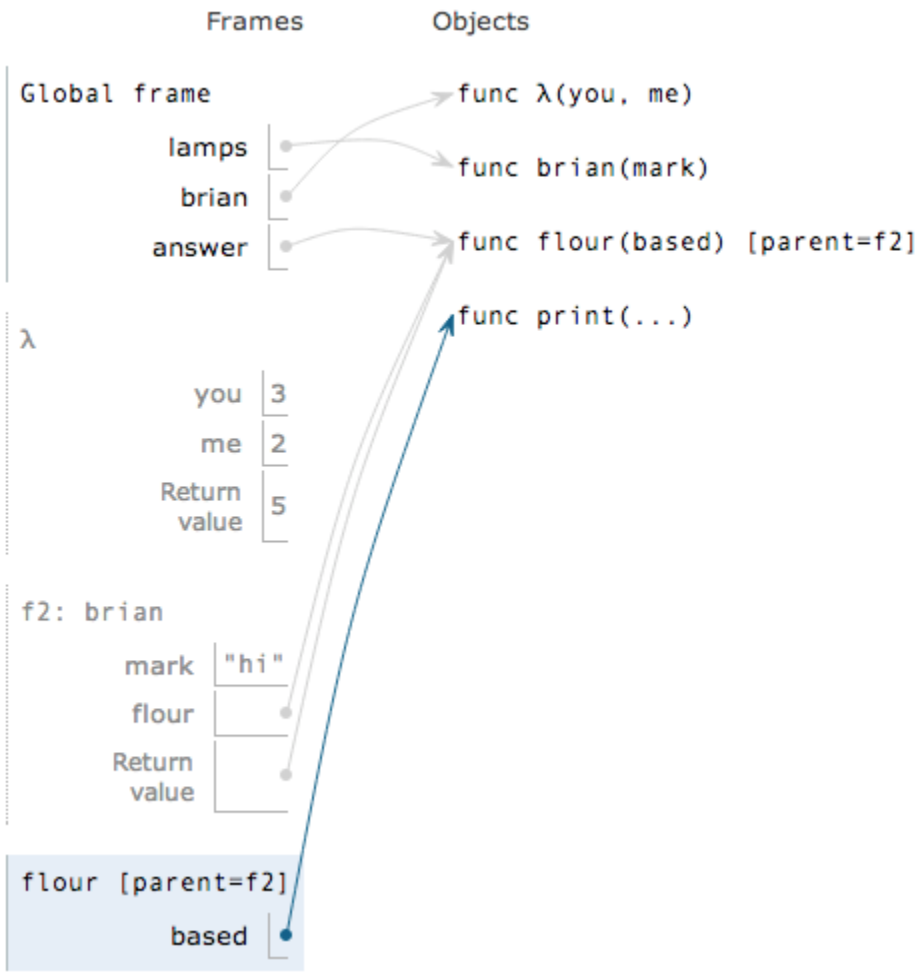
```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```



Extra practice!

HOF's & Environment Diagrams

```
1 lamps = lambda you, me: you + me
2 def brian(mark):
3     def flour(based):
4         return based(mark)
5     mark = 'hi'
6     return flour
7 lamps, brian = brian, lamps
8 answer = lamps(brian(3, 2))
9 answer(print)
```



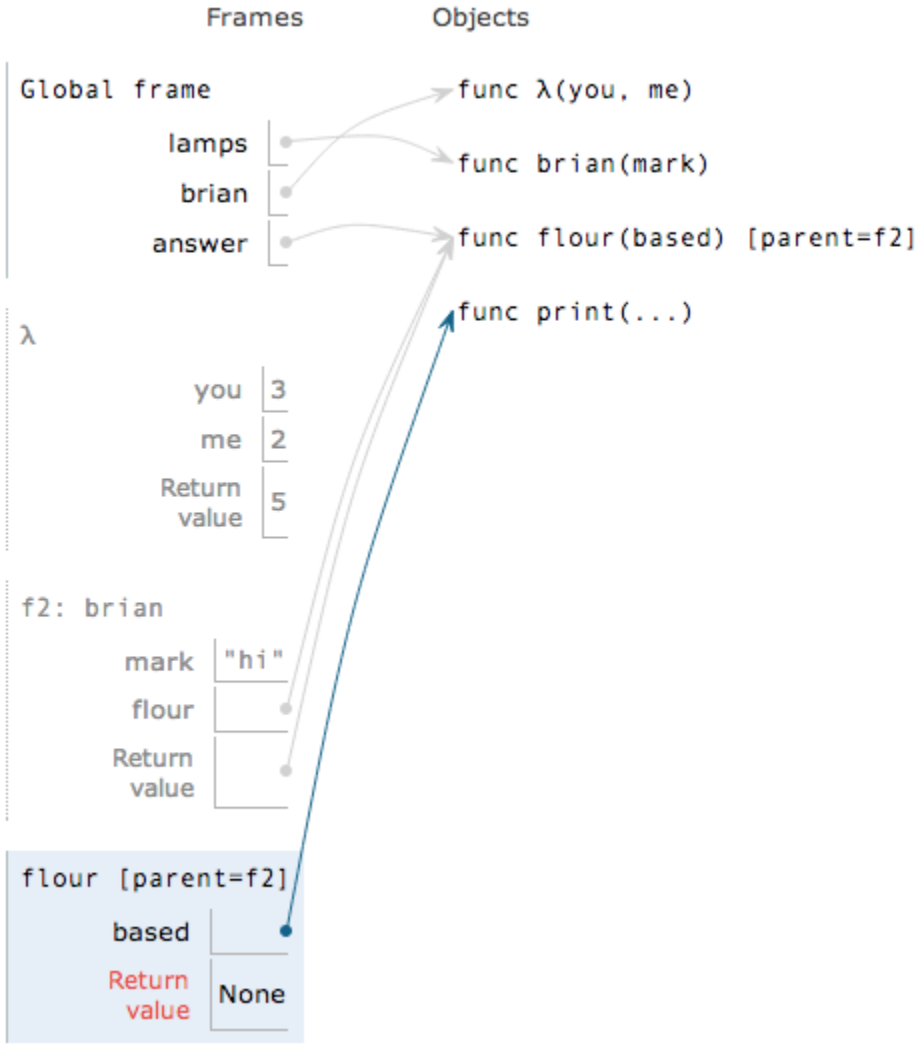
Extra practice!

HOF's & Environment Diagrams

```
1  lamps = lambda you, me: you + me
2  def brian(mark):
3      def flour(based):
4          return based(mark)
5      mark = 'hi'
6      return flour
7  lamps, brian = brian, lamps
8  answer = lamps(brian(3, 2))
9  answer(print)
```

Program output:

```
hi
```



Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)

_________
>>> g(7)

_________
>>> h = lambda x: lambda y: z
>>> h(1)

_________
>>> h(1000)(1)

_________
```

Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)
0
>>> g(7)
_________
>>> h = lambda x: lambda y: z
>>> h(1)
_________
>>> h(1000)(1)
_________
```

Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)
0
>>> g(7)
1
>>> h = lambda x: lambda y: z
>>> h(1)

________
>>> h(1000)(1)

________
```

Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)
0
>>> g(7)
1
>>> h = lambda x: lambda y: z
>>> h(1)
FUNCTION
>>> h(1000)(1)
```

Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)
0
>>> g(7)
1
>>> h = lambda x: lambda y: z
>>> h(1)
FUNCTION
>>> h(1000)(1)
ERROR (NameError)
```

Lambda Functions

And, we can call a lambda expression without ever giving it a name!

```
>>> f = lambda x: x + 1
>>> f(4)

_________

>>> (lambda x: x + 1)(4)

_________

>>> (lambda y: y(3))(lambda x: x + 4)

_________
```

Lambda Functions

And, we can call a lambda expression without ever giving it a name!

```
>>> f = lambda x: x + 1
>>> f(4)
5
>>> (lambda x: x + 1)(4)
```

```
>>> (lambda y: y(3))(lambda x: x + 4)
```

Lambda Functions

And, we can call a lambda expression without ever giving it a name!

```
>>> f = lambda x: x + 1
>>> f(4)
5
>>> (lambda x: x + 1)(4)
5
>>> (lambda y: y(3))(lambda x: x + 4)
```

Lambda Functions

And, we can call a lambda expression without ever giving it a name!

```
>>> f = lambda x: x + 1
>>> f(4)
5
>>> (lambda x: x + 1)(4)
5
>>> (lambda y: y(3))(lambda x: x + 4)
7
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x

_____
>>> x(3)

_____
>>> x(3)(4)

_____
>>> x(3)()

_____
>>> x(3)()(7)

_____
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
```

We can rewrite the lambda expressions using HOFs:

```
def L1(x):
# The line above could be "def x(x):" Why?
    def L2():
        def L3(y):
            return 2 * x + 3 * y
        return L3
    return L2
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
FUNCTION
>>> x(3)
_____
>>> x(3)(4)
_____
>>> x(3)()
_____
>>> x(3)()(7)
_____
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
FUNCTION
>>> x(3)
FUNCTION
>>> x(3)(4)
_____
>>> x(3)()
_____
>>> x(3)()(7)
_____
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
FUNCTION
>>> x(3)
FUNCTION
>>> x(3)(4)
ERROR (TypeError)
>>> x(3)()
_____
>>> x(3)()(7)
_____
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
FUNCTION
>>> x(3)
FUNCTION
>>> x(3)(4)
ERROR (TypeError)
>>> x(3)()
FUNCTION
>>> x(3)()(7)
_____
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
FUNCTION
>>> x(3)
FUNCTION
>>> x(3)(4)
ERROR (TypeError)
>>> x(3)()
FUNCTION
>>> x(3)()(7)
27
```

Recursion

Recursion

A recursive function has two important components:

1. A *base case*.
2. A *recursive case*.

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Visualization: http://goo.gl/ux5MuQ

Recursion

```
def johnlovespuppies(n):
   if n == 0:
      print “Announcements!”
   print (“I like puppies ^___^”)
   return johnlovespuppies(n - 1)
```

What will the following function call print?

```
>>> johnlovespuppies(10)
```

Recursion

```
def johnlovespuppies(n):
   if n == 0:
      print “Announcements!”
   print (“I like puppies ^___^”)
   return johnlovespuppies(n - 1)
```

What will the following function call print?

```
>>> johnlovespuppies(10)
```

Error (INFINITE PUPPY SO WOW)

Recursion

```
#correct this time
def johnlovespuppies(n):
   if n == 0:
      print “Announcements!”
      return #WE NEED THIS!
   print (“I like puppies ^___^”)
   return johnlovespuppies(n - 1)
```

Recursion

```
#correct this time
def johnlovespuppies(n):
   if n == 0:
      print “Announcements!”
      return #WE NEED THIS!
   print (“I like puppies ^___^”)
   return johnlovespuppies(n - 1)
```

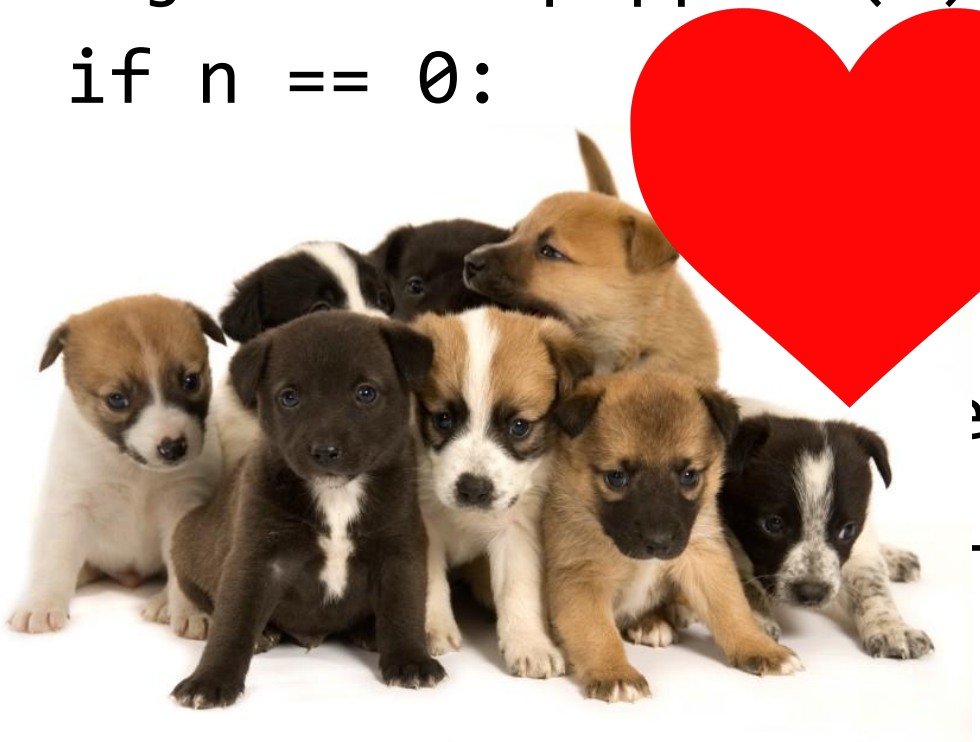
Recursion

```
#correct this time
def johnlovespuppies(n):
    if n == 0:
                                    ts!”


                                    es ^___^”)
                                    .es(n - 1)
```

Recursion

```
#correct this time
def johnlovespuppies(n):
    if n == 0:
                                  !”
                                  es ^___^”)
                                  .es(n - 1)
```



Recursion

Write a recursive function `log` that takes a base `b` and a number `x`, and returns $\log_b(x)$, the power of `b` that is `x`. (Assume that x is some power of b.)

```
def log(b, x):
    """
    >>> log(2, 8):
    3
    """
```

Recursion

Write a recursive function `log` that takes a base `b` and a number `x`, and returns $\log_b(x)$, the power of `b` that is `x`. (Assume that x is some power of b.)

```
def log(b, x):
    if x == 1:
        return 0
    return 1 + log(b, x/b)
```

Recursion

Write a recursive function `eat_chocolate` that takes in a number of chocolate pieces and **returns** a string as follows:

```
def eat_chocolate(num_pieces):
    """
    >>> eat_chocolate(5)
    "nom nom nom nom nom"
    >>> eat_chocolate(1)
    "nom"
    >>> eat_chocolate(0)
    "No chocolate :("
    """
```

Recursion

Write a recursive function `eat_chocolate` that takes in a number of chocolate pieces and **returns** a string as follows:

```
def eat_chocolate(num_pieces):
    if num_pieces == 0:
        return "No chocolate :("
    elif num_pieces == 1:
        return "nom"
    return "nom " + \
           eat_chocolate(num_pieces - 1)
```

Recursion

Write a function `count_occurrences` that takes in a number `num` and a number `digit` and counts the number of times `digit` appears in `num`.

```
def count_occurrences(num, digit):
    """
    >>> count_occurences(3403, 3)
    2
    >>> count occurences(8940, 2)
    0
    """
```

Recursion

Write a function `count_occurrences` that takes in a number `num` and a number `digit` and counts the number of times `digit` appears in `num`.

```
def count_occurrences(num, digit):
    if num < 10:
        if num == digit:
            return 1
        return 0
    last_digit = num % 10
    if last_digit == digit:
        return 1 + count_occurrences(num // 10, digit)
    return count_occurrences(num // 10, digit)
```

Recursion

Write the function `call_until_one` that...

- takes a function `func` as an argument
- returns another function that takes a number `x` that returns the number of times you can call `func` on `x` until it returns a value less than or equal to 1.

```
def call_until_one(func):
    """
    >>> f = call_until_one(lambda x: x - 1)
    >>> f(100)
    99

    >>> g = call_until_one(lambda x: x / 2)
    >>> g(128)
    7
    """
```

Recursion

Write the function `call_until_one` that...

- takes a function `func` as an argument
- returns another function that takes a number `x` that returns the number of times you can call `func` on `x` until it returns a value less than or equal to 1.

```
def call_until_one(func):
    def count_calls(x):
        if x <= 1:
            return 0
        return 1 + count_calls(func(x))
    return count_calls
```

Feedback

We would like your feedback on this review session, so that we can improve for future review sessions.

We'd love to hear your suggestions, complaints or comments!

Thank you, and best of luck on your midterm!