HIGHER ORDER FUNCTIONS AND ENVIRONMENT DIAGRAMS 2

COMPUTER SCIENCE 61A

June 25, 2015

1 Higher Order Functions

A **higher order function** (HOF) is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

1.1 Functions as Arguments

One way a higher order function can exploit other functions is by taking functions as input. Consider this higher order function called `negate`.

```
def negate(f, x):
    return -f(x)
```

`negate` takes in a function `f` and a number `x`. It doesn't care what exactly `f` does, as long as `f` takes in a number and returns a number. Its job is simple: call `f` on `x` and return the negation of that value.

1.2 Questions

1. Here are some possible functions that can be passed through as `f`.

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

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

   What will the following Python statements output?

```
>>> negate(square, 5)

>>> negate(double, -19)

>>> negate(double, negate(square, -4))
```

2. Implement a function `keep_ints`, which takes in a function `cond` and a number `n`, and only prints a number from `1` to `n` if calling `cond` on that number returns `True`:

```
def keep_ints(cond, n):
    """Print out all integers 1..i..n where cond(i) is true

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> keep_ints(is_even, 5)
    2
    4
    """
```

1.3 Functions as Return Values

Often, we will need to write a function that returns another function. One way to do this is to define a function inside of a function:

```
def outer(x):
    def inner(y):
        ...
    return inner
```

The return value of `outer` is the function `inner`. This is a case of a function returning a function. In this example, `inner` is defined inside of `outer`. Although this is a common pattern, we can also define `inner` outside of `outer` and still use the same `return` statement.

```
def inner(y):
    ...
def outer(x):
    return inner
```

1.4 Questions

1. Use this definition of `outer` to fill in what Python would print when the following lines are evaluated.

```
def outer(n):
    def inner(m):
        return n - m
    return inner
>>> outer(61)

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

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

2. Implement a function `keep_ints` like before, but now it takes in a number `n` and returns a function that has one parameter `cond`. The returned function prints out all numbers from 1..i..n where calling `cond(i)` returns True.

```
def keep_ints(n):
    """Returns a function which takes one parameter cond and
    prints out all integers 1..i..n where calling cond(i)
    returns True.

    >>> def is_even(x):
    ...     # Even numbers have remainder 0 when divided by 2.
    ...     return x % 2 == 0
    >>> keep_ints(5)(is_even)
    2
    4
    """
```

2 Lambda Functions

Lambda expressions are one-line functions that specify two things: the parameters and the return expression.

A lambda expression that takes in no arguments and returns 8:

```
lambda:  8
```
8: return value

A lambda expression that takes two arguments and returns their product:

```
lambda  x, y :  x * y
```
x, y: parameters; x * y: return expression

Unlike functions created by a `def` statement, the function object that a lambda expression creates has no intrinsic name and is not bound to any variable. In fact, nothing changes in the current environment when we evaluate a lambda expression unless we do something with this expression, such as assign it to a variable or pass it as an argument to a higher order function.

2.1 Questions

1. What would Python print?

```
>>> a = lambda: 5
>>> a()

>>> a(5)

>>> b = lambda: lambda x: 3
>>> b()(15)

>>> c = lambda x, y: x + y
>>> c(4, 5)

>>> d = lambda x: lambda y: x * y
>>> d(3)

>>> d(3)(3)

>>> e = d(2)
>>> e(5)

>>> f = lambda: print(1)

>>> g = f()
```

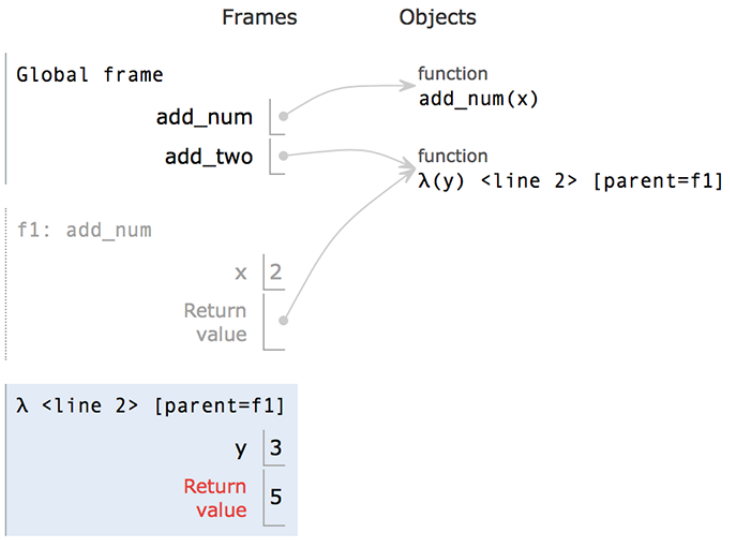
2. Fill in the blanks with one-line lambda expressions so that each call expression that follows returns 3.

```
>>> f1 = ____________________
>>> f1()
3

>>> f2 = ____________________
>>> f2()()
3

>>> f3 = ____________________
>>> f3()(3)
3

>>> f4 = ____________________
>>> f4()()(3)()
3
```

3 Environment Diagrams

Recall that an **environment diagram** keeps track of all the variables that have been defined and the values they are bound to. However, values are not necessarily only integers and strings. Environment diagrams can model more complex programs that utilize higher order functions.

```
def add_num(x):
    return lambda y: x + y

add_two = add_num(2)
add_two(3)
```



Lambdas are represented similiarly to functions in environment diagrams, but since they lack an instrinsic name, the lambda symbol (λ) is used instead.

The parent of a function, including lambdas, is always the frame in which the function is defined. It is useful to include the parent in environment diagrams in order to find variables that are not defined in the current frame. In the previous example, when we call `add_two` (which is really the lambda function), we need to know what `x` is in order to compute `x + y`. Since `x` is not in the frame `f2`, we look at the frame's parent, which is `f1`. There, we find `x` is bound to `2`.

As illustrated above, higher order functions that return a function are represented with a pointer to the function object.

3.1 Questions

1. Draw the environment diagram that results from executing the code below.

```
from operator import add

six = 1

def ty(one, a):
    spring = one(a, six)
    return spring

six = ty(add, 6)
spring = ty(add, 6)
```

2. Draw the environment diagram for the following code:

```
from operator import add
def curry2(h):
    def f(x):
        def g(y):
            return h(x, y)
        return g
    return f

make_adder = curry2(add)
add_three = make_adder(3)
five = add_three(2)
```

3.2 Extra Questions

1. Draw the environment diagram for the following code: (Note that using the + operator with two strings results in the second string being appended to the first. For example `"C" + "S"` concatenates the two strings into one string `"CS"`)

```
y = "y"
h = y
def y(y):
    h = "h"
    if y == h:
        return y + "i"
    y = lambda y: y(h)
    return lambda h: y(h)
y = y(y)(y)
```