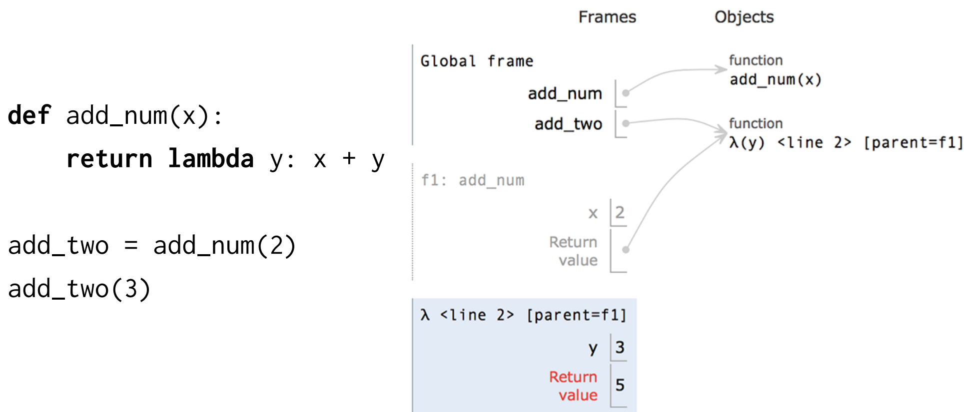
CS 61A Higher Order Functions and Sequences
Summer 2018 Discussion 2: June 26, 2018

1 Higher Order Functions

HOFs in 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.



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

The parent of any 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 have their return value represented with a pointer to the function object.

A Note on Lambda Expressions

A lambda expression evaluates to a function, called a lambda function. In the code above, `lambda y: x + y` is a lambda expression, and can be read as a function that takes in one parameter `y` and returns `x + y`.

A lambda expression by itself evaluates to a function but does not bind it to a name. Also note that the return expression of this function is not evaluated until the lambda is called. This is similar to how defining a new function using a def statement does not execute the functions body until it is later called.

```
>>> what = lambda x : x + 5
>>> what
<function <lambda> at 0xf3f490>
```

Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that evaluate to functions.

```
>>> (lambda y: y + 5)(4)
9
>>> (lambda f, x: f(x))(lambda y: x + 1, 10)
11
```

Questions

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

```
1 def curry2(h):
2     def f(x):
3         def g(y):
4             return h(x, y)
5         return g
6     return f

7 make_adder = curry2(lambda x, y: x + y)
8 add_three = make_adder(3)
9 five = add_three(2)
```

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

```
1  n = 7

2  def f(x):
3      n = 8
4      return x + 1

5  def g(x):
6      n = 9
7      def h():
8          return x + 1
9      return h

10 def f(f, x):
11     return f(x + n)

12 f = f(g, n)
13 g = (lambda y: y())(f)
```

1.3 *The following question is extremely difficult. Something like this would not appear on the exam. Nonetheless, it's a fun problem to try.*

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

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"`

```
1  y = "y"
2  h = y
3  def y(y):
4      h = "h"
5      if y == h:
6          return y + "i"
7      y = lambda y: y(h)
8      return lambda h: y(h)
9  y = y(y)(y)
```

Writing Higher Order Functions

1.4 Write a function that takes in a function `cond` and a number `n` and prints numbers from `1` to `n` where 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
    """
```

1.5 Write a function similar to `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 numbers from 1 to `n` where calling `cond` on that number 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 Sequences and Lists

A *sequence* is an ordered collection of values. It has two fundamental properties: length and element selection. In this discussion, we'll explore one of Python's data types, the *list*, which implements this abstraction.

In Python, we can have lists of whatever values we want, be it numbers, strings, functions, or even other lists! Furthermore, the types of the list's contents need not be the same. In other words, the list need not be homogenous.

Lists can be created using square braces. Their elements can be accessed (or *indexed*) with square braces. Lists are zero-indexed: to access the first element, we must index at 0; to access the ith element, we must index at $i - 1$.

We can also index with negative numbers. These begin indexing at the end of the list, so the index -1 is equivalent to the index `len(list) - 1` and index -2 is the same as `len(list) - 2`.

Let's try out some indexing:

```
>>> fantasy_team = ['aaron rodgers', 'desean jackson']
>>> print(fantasy_team)
['aaron rodgers', 'desean jackson']
>>> fantasy_team[0]
'aaron rodgers'
>>> fantasy_team[len(fantasy_team) - 1]
'desean jackson'
>>> fantasy_team[-1]
'desean jackson'
```

If we have two lists, we can use the + operator to create a new list with the values of the original two lists, concatenated together.

```
>>> fish_names = ['Dory', 'Flounder']
>>> rabbit_names = ['Bugs Bunny', 'Officer Hopps']
>>> animal_names = fish_names + rabbit_names
>>> animal_names
['Dory', 'Flounder', 'Bugs Bunny', 'Officer Hopps']
```

Sequences also have a notion of length, the number of items stored in the sequence. In Python,we can check how long a sequence is with the `len` built-in function.

We can also check if an item exists within a list with the `in` statement.

```
>>> poke_team = ['Meowth', 'Mewtwo']
>>> len(poke_team)
2
>>> 'Meowth' in poke_team
True
>>> 'Pikachu' in poke_team
False
```

Questions

2.1 What would Python display?

```
>>> a = [1, 5, 4, [2, 3], 3]
>>> print(a[0], a[-1])
```

```
>>> len(a)
```

```
>>> 2 in a
```

```
>>> 4 in a
```

```
>>> a[3][0]
```

Slicing

If we want to access more than one element of a list at a time, we can use a *slice*. Slicing a sequence is very similar to indexing. We specify a starting index and an ending index, separated by a colon. Python creates a new list with the elements from the starting index up to (but not including) the ending index.

We can also specify a step size, which tells Python how to collect values for us. For example, if we set step size to 2, the returned list will include every **other** value, from the starting index until the ending index. A negative step size indicates that we are stepping backwards through a list when collecting values.

You can also choose not to specify any/all of the slice arguments. Python will perform some default behaviour if this is the case:

- If the step size is left out, the default step size is 1.
- If the start index is left out, the default start index is the beginning of the list.
- If the end index is left out, the default end index is the end of the list.
- If the step size is negative, the default start index becomes the end of the list, and the default end index becomes the beginning the of the list.

Thus, `lst[:]` creates a list that is identical to `lst` (a copy of `lst`). `lst[::-1]` creates a list that has the same elements of `lst`, but reversed. Those rules still apply if more than just the step size is specified e.g. `lst[3::-1]`.

```
>>> directors = ['jenkins', 'spielberg', 'bigelow', 'kubrick']
>>> directors[:2]
['jenkins', 'spielberg']
>>> directors[1:3]
['spielberg', 'bigelow']
>>> directors[1:]
['spielberg', 'bigelow', 'kubrick']
>>> directors[0:4:2]
['jenkins', 'bigelow']
>>> directors[::-1]
['kubrick', 'bigelow', 'spielberg', 'jenkins']
```

Questions

2.2 What would Python display?

```
>>> a = [3, 1, 4, 2, 5, 3]
>>> a[1::2]
```

```
>>> a[:]
```

```
>>> a[1:-2]
```

```
>>> a[2:4:2]
```

```
>>> a[::2]
```

```
>>> a[::-1]
```

3 List Comprehensions

A **list comprehension** is a compact way to create a list whose elements are the results of applying a fixed expression to elements in another sequence.

```
[<map exp> for <name> in <iter exp> if <filter exp>]
```

It might be helpful to note that we can rewrite a list comprehension as an equivalent for statment. See the example to the right.

```
new_lst = []
for <name> in <iter exp>:
    if <filter exp>:
        new_lst += [<map_exp>]
return new_lst
```

Let's break down an example:

```
[x * x - 3 for x in [1, 2, 3, 4, 5] if x % 2 == 1]
```

In this list comprehension, we are creating a new list after performing a series of operations to our initial sequence `[1, 2, 3, 4, 5]`. We only keep the elements that satisfy the filter expression `x % 2 == 1` (1, 3, and 5). For each retained element, we apply the map expression `x*x - 3` before adding it to the new list that we are creating, resulting in the output `[-2, 6, 22]`.

Note: The `if` clause in a list comprehension is optional.

Questions

3.1 What would Python display?

```
>>> [i + 1 for i in [1, 2, 3, 4, 5] if i % 2 == 0]
```

```
>>> [i * i - i for i in [5, -1, 3, -1, 3] if i > 2]
```

```
>>> [[y * 2 for y in [x, x + 1]] for x in [1, 2, 3, 4]]
```