Computational Structures in Data Science

Environments and Lambdas

Berkeley
UNIVERSITY OF CALIFORNIA

Functional Sequence (List) Operations

- Goal: Transform a *sequence*, and return a new result
- We'll use 3 functions that are hallmarks of functional programming
- Each of these takes in a function and a sequence as arguments

Function	Action	Input arguments	Input Fn. Returns	Output
`map`	Transform every item	1 (each item)	"Anything", a new item	**List**: same length, but possibly new values
`filter`	Return a list with fewer items	1 (each item)	A Boolean	**List:** possibly fewer items, values are the same
`reduce`	"Combine" items together	2 (current item, and the previous result)	Type should match the type each item	A "single" item

Computational Structures in Data Science

Functions That Return Functions

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Learn how to use and create higher order functions:
- Functions can be used as data
- Functions can accept a function as an argument
- **Functions can return a new function**

Review: What is a Higher Order Function?

- A function that takes in another function as an argument

OR

- **A function that returns a function as a result.**

Higher Order Functions

- **A function that returns (makes) a function**

```
def leq_maker(c):
    def leq(val):
        return val <= c
    return leq
```

```
>>> leq_maker(3)
<function leq_maker.<locals>.leq at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> [x for x in range(7) if leq_maker(3)(x)]
[0, 1, 2, 3]
```


Demo

Inner or Nested Functions

- Inner functions are *scoped* – they are not visible to the outside world
- But they can be *returned* and thus called later on.
- Like a "regular" function, they have access to all the data (including arguments) of their "parent" or "container" function.

- This can become messy!
- In the next section, we will formalize the rules.

Computational Structures in Data Science

Environment Diagrams

Berkeley
UNIVERSITY OF CALIFORNIA

Why focus on environments?

- Environments are a simplification of why Python *actually* does
- Focus on building intuition for what will happen when you run code
- Sometimes tedious, but the practice helps you solve hard questions
  - In 88C (or 61A), even our hard questions are pretty short
  - Outside of class, things can get complex quickly.
- Every programming language is a bit different, but these rules are quite common
- I understand if you don't like them now. ☺

Python Tutor Example #1

```
a = "chipotle"
b = 5 > 3
c = 8
def foo(c):
    return c - 5
def bar():
    if b:
        a = "taco bell"
result1 = foo(10)
result2 = bar()
```

- Primitives and Functions: Environment Diagram Python Tutor:

Environment Diagrams

- Organizational tools that help you understand code
- **Terminology:**
  - **Frame:** keeps track of variable-to-value bindings, each function call has a frame
  - **Global Frame:** global for short, the starting frame of all python programs, doesn't correspond to a specific function
  - **Parent Frame:** The frame of where a function is defined (default parent frame is global)
  - **Frame number:** What we use to keep track of frames, f1, f2, f3, etc
  - **Variable** vs **Value**: x = 1. x is the **variable**, 1 is the **value**

Environment Diagrams Rules

1. Always draw the global frame first
2. When evaluating assignments (lines with single equal), always evaluate right side first
3. When you **CALL** a function MAKE A NEW FRAME!
4. When assigning a primitive expression (number, boolean, string) write the value in the box
5. When assigning anything else (lists, functions, etc.), draw an arrow to the value
6. When calling a function, name the frame with the intrinsic name – the name of the function that variable points to
7. The parent frame of a function is the frame in which it was defined in (default parent frame is global)
8. If the value for a variable doesn't exist in the current frame, search in the parent frame

Python Tutor Example #2

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

n = 10
add_2 = make_adder(2)
x = add_2(5)
```

- make_adder Higher Order Function: Environment Diagram Python Tutor Link

Python Tutor Example #3

```
add_2 = make_adder(2)
add_3 = make_adder(3)
x = add_2(2)
def compose(f, g):
    def h(x):
        return f(g(x))
    return h
add_5 = compose(add_2, add_3)
z = add_5(x)
```

- Compose Python Tutor Link

Demo

Example 1:

- make_adder Higher Order Function: Environment Diagram Python Tutor Link

Example 2:

- Primitives and Functions: Environment Diagram Python Tutor:

Example 3:

- Compose Python Tutor Link

Environment Diagram Tips / Links

- NEVER draw an arrow from one variable to another.
- Useful Resources:
  - http://markmiyashita.com/cs61a/environment_diagrams/rules_of_environment_diagrams/
  - http://albertwu.org/cs61a/notes/environments.html

Why focus on environments?

- Environments are a simplification of why Python actually does
- Focus on building intuition for what will happen when you run code
- Sometimes tedious, but the practice helps you solve hard questions
  - In 88C (or 61A), even our hard questions are pretty short
  - Outside of class, things can get complex quickly.
- Every programming language is a bit different, but these rules are quite common

Computational Structures in Data Science

Lambda Expressions

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Lambda are anonymous functions, which are expressions
  - Don't use `return`, lambdas always return the value of the expression.
  - They are typically short and concise
  - They don't have an "intrinsic" name when using an environment diagram.
    - Their name is the character λ

Why Use `lambda`?

- We often can use the behavior of simple function!
- Using functions gives us flexibility
- "Inline" functions are faster/easier to write, and sometimes require less reading.
- They're not "reusable", but that's OK!

lambda

Function expression

"anonymous" function creation

lambda <arg or arg_tuple> : <expression using args>

Expression, not a statement, no return or any other statement

```
add_one = lambda v : v + 1
```

```
def add_one(v):
    return v + 1
```


Examples

```
>>> def make_adder(i):
...     return lambda x: x+i
...
>>> make_adder(3)
<function make_adder.<locals>.<lambda> at
0x10073c510>


>>> make_adder(3)(4)
7


>>> list(map(make_adder(3), [1,2,3,4]))
```

Lambda with HOFs

- **A function that returns (makes) a function**

```
def leq_maker(c):
    return lambda val: val <= c
```

```
>>> leq_maker(3)
<function leq_maker.<locals>.<lambda> at 0x1019d8c80>

>>> leq_maker(3)(4)
False

>>> filter(leq_maker(3), [0,1,2,3,4,5,6,7])
[0, 1, 2, 3]
```

More Python HOFs

- sorted – sorts a list of data
- min
- max

All three take in an optional argument called `key` which allows us to control how the function performs its action. They are more similar to filter than map.

```
max([1,2,3,4,5], key = lambda x: -x)
```

key is the name of the argument and a lambda is its value.

```
fruits = ["pear", "grape", "KIWI", "APPLE", "melon",
"ORANGE", "BANANA"]
sorted(key=lambda x: x.lower())
```


Sorting Data

- It is often useful to sort data.
- What property should we sort on?
  - Numbers: We can clearly sort.
  - What about the length of a word?
  - Alphabetically?
  - What about sorting a complex data set, but 1 attribute?
    - Image I have a list of courses: I could sort be course name, number of units, start time, etc.
- Python provides 1 function which allows us to provide a *lambda* to control its behavior

Sorting with Lambdas

```
>>> sorted([1,2,3,4,5], key = lambda x: x)
    [1, 2, 3, 4, 5]
>>> sorted([1,2,3,4,5], key = lambda x: -x)
    [5, 4, 3, 2, 1]
# Nonsensical pairing of numbers and words…
>>> sorted([(2, "hi"), (1, "how"), (5, "goes"), (7, "it")],
           key = lambda x:x[0])
[(1, 'how'), (2, 'hi'), (5, 'goes'), (7, 'it')]
>>> sorted([(2, "hi"), (1, "how"), (5, "goes"), (7, "it")],
          key = lambda x:x[1])
    [(7, 'it'), (5, 'goes'), (2, 'hi'), (1, 'how')]
>>> sorted([(2,"hi"),(1,"how"),(5,"goes"),(7,"it")],
          key = lambda x: len(x[1]))
    [(7, 'it'), (2, 'hi'), (1, 'how'), (5, 'goes')]
```


Computational Structures in Data Science

Environment Diagrams

Berkeley
UNIVERSITY OF CALIFORNIA

Revisiting Environments

```
def make_adder(n):
    return lambda k: k + n

add_2 = make_adder(2)
add_3 = make_adder(3)
x = add_2(5)
y = add_3(x)
```

Revisiting Environments: compose w/lambda

```
def make_adder(n):
    return lambda k: k + n
def compose(f, g):
    return lambda x: f(g(x))

add_2 = make_adder(2)
add_3 = make_adder(3)
add_5 = compose(add_2, add_3)

x = add_2(2)
z = add_5(x)
```

Environment Diagrams

- Organizational tools that help you understand code
- **Terminology:**
  - **Frame:** keeps track of variable-to-value bindings, each function call has a frame
  - **Global Frame:** global for short, the starting frame of all python programs, doesn't correspond to a specific function
  - **Parent Frame:** The frame of where a function is defined (default parent frame is global)
  - **Frame number:** What we use to keep track of frames, f1, f2, f3, etc
  - **Variable** vs **Value**: x = 1. x is the **variable**, 1 is the **value**

Environment Diagrams Rules

1. Always draw the global frame first
2. When evaluating assignments (lines with single equal), always evaluate right side first
3. When you **CALL** a function MAKE A NEW FRAME!
4. When assigning a primitive expression (number, boolean, string) write the value in the box
5. When assigning anything else (lists, functions, etc.), draw an arrow to the value
6. When calling a function, name the frame with the intrinsic name – the name of the function that variable points to
7. The parent frame of a function is the frame in which it was defined in (default parent frame is global)
8. If the value for a variable doesn't exist in the current frame, search in the parent frame

Demo

Example 1:

- make_adder Higher Order Function: Environment Diagram Python Tutor Link

Example 2:

- Compose Python Tutor Link