

Computational Structures in Data Science

Lecture 6: Lists & Higher Order Functions

Week 2, Summer 2024. 6/26 (Wed)

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

Exam dates

Midterm: Wednesday July 17th, 3PM – 5PM PST

Final: Wednesday August 7th, 3PM – 5PM PST

Exams will be **administered online**, and **proctored via Zoom**. You may need to present your ID (eg student CalID card, or any ID with your name + photo) during the Zoom call to proctors.

Important: for those that can't make the above exam times, we will have **alternate exam times**. Stay tuned for details here!

Today's overview

- Lambda expressions
- Higher-order functions + sequences
 - map, filter, reduce

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 λ ambda?

- Utility in simple functions! No "state", no need to "def"ine something
- 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]))  
[4, 5, 6, 7]
```


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)
```

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

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

Computational Structures in Data Science

HOFs That Operate on Sequences

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Learn three new common Higher Order Functions:
 - map, filter, reduce
- These each apply a function to a sequence (list) of data
- They are "lazy" so we may need to call `list()`

Functional List Operations

- Goal: Transform a list, 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

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

Why Learn HOFs this way?

- Break a complex task into many smaller parts
 - Small problems are easier to solve
 - They're easier to understand and debug
- Directly maps to transforming data in lists and tables
 - **map**: transformations, **apply**
 - **filter**: selections, **where**
 - **reduce**: aggregations, **groupby**

Learning Objectives

- Map: Transform each item
 - Input: A function and a sequence
 - Output: A sequence of the same length. The items may be different.

Computational Structures in Data Science

Higher Order Functions:
map

Berkeley
UNIVERSITY OF CALIFORNIA

map(function, sequence)

```
list(map(function_to_apply, list_of_inputs))
```

Transform each of items by a function.

e.g. square()

Inputs (Domain):

- Function
- Sequence

Output (Range):

- A sequence

Simplified Implementation

```
def map(function, sequence):
```

```
    return [ function(item) for item in sequence ]
```

```
list(map(square, range(10)))
```


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

Important: this is a function!



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

```
[4, 5, 6, 7]
```

Computational Structures in Data Science

Lists & Higher Order Functions: Filter

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Learn three new common Higher Order Functions:
 - map, filter, reduce
- These each apply a function to a sequence (list) of data
- map/filter are "lazy" so we may need to call `list()`

- Filter: Keeps items matching a condition.
 - Input: A function and sequence
 - Output: A sequence, possibly with items removed. The items don't change.

filter(function, sequence)

```
list(filter(function, list_of_inputs))
```

Keeps each of item where the function is true.

Inputs (Domain):

- Function
- Sequence

Output (Range):

- A sequence

Simplified implementation

```
def filter(function, sequence):
```

```
    return [ item for item in sequence if function(item) ]
```

```
filter(is_even, range(10))
```

Lambda with HOFs

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

```
def less_than_5(c):  
    return c < 5
```

```
>>> less_than_5  
<function less_than_5... at 0x1019d8c80>
```

```
>>> filter(less_than_5, [0,1,2,3,4,5,6,7])  
[0, 1, 2, 3, 4]
```

```
>>> filter(lambda x: x < 3, [0,1,2,3,4,5,6,7])  
[0, 1, 2]
```

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]
```

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

Computational Structures in Data Science

Lists & Higher Order Functions
Reduce

Berkeley
UNIVERSITY OF CALIFORNIA

Learning Objectives

- Learn three new common Higher Order Functions:
 - map, filter, reduce
- These each apply a function to a sequence (list) of data

- Reduce: “Combines” items together, probably doesn’t return a list.
 - Input: A 2 item function and a sequence
 - A single value

reduce(function, list_of_inputs)

Successively **combine** items of our sequence

- function: add(), takes 2 inputs gives us 1 value.

Inputs (Domain):

- Function, with 2 inputs
- Sequence

Output (Range):

- An item, the type is the output of our function.

Note: We must import reduce from functools!

Simplified implementation

```
def reduce(function, sequence):  
    result = function(sequence[0], sequence[1])  
    for index in range(2, len(sequence)):  
        result = function(result, sequence[index])  
    return result
```

Reduce is an aggregation!

- Reduce aggregates or combines data
- This is commonly called "group by"
- In Data 8:
 - sum over a range of values
 - joining multiple cells into 1 array
 - calling `max()`, `min()` etc. on a column
- We'll revisit aggregations in SQL

Computational Structures in Data Science

Lists & Higher Order Functions Acronym

Berkeley
UNIVERSITY OF CALIFORNIA

Today's Task: Acronym

Input: "The University of California at Berkeley"

Output: "UCB"

```
def acronym(sentence):  
    """YOUR CODE HERE"""
```

P.S. Pedantry alert: This is really an *initialism* but that's rather annoying to say and type. 😊 (However, the code we write is the same, the difference is in how you pronounce the result.) The more you know!

Today's Task: Acronym

Input: "The University of California at Berkeley"

Output: "UCB"

```
def acronym(sentence):  
    """ (Some doctests)  
    """  
    words = sentence.split()  
    return reduce(add, map(first_letter, filter(is_long_word, words)))
```

P.S. Pedantry alert: This is really an *initialism* but that's rather annoying to say and type. 😊 (However, the code we write is the same, the difference is in how you pronounce the result.) The more you know!

Today's Task: Acronym

Input: "The University of California at Berkeley"

Output: "UCB"

```
def acronym(sentence):  
    """ (Some doctests)  
    """  
    words = sentence.split()  
    return reduce(add, map(first_letter, filter(is_long_word, words)))
```

Interpretation:

First, keep only the long words from `words` (aka “**filter()**”)

=> [“University”, “California”, “Berkeley”]

Then, for each surviving word, take just the first letter (aka “**map()**”)

=> [“U”, “C”, “B”]

Finally, concatenate the first letters together (aka “**reduce()**”)

=> “UCB”

Today's Task: Acronym

Input: "The University of California at Berkeley"

Output: "UCB"

```
def acronym(sentence):  
    """ (Some doctests)  
    """  
    words = sentence.split()  
    return reduce(add, map(first_letter, filter(is_long_word, words)))
```

Question: what would be reasonable implementations of `first_letter()` and `is_long_word()`?

```
def first_letter(word):  
    # ...?
```

```
def is_long_word(word):  
    # ...?
```

Today's Task: Acronym

Input: "The University of California at Berkeley"

Output: "UCB"

```
def acronym(sentence):  
    """ (Some doctests)  
    """  
    words = sentence.split()  
    return reduce(add, map(first_letter, filter(is_long_word, words)))
```

```
def first_letter(word):  
    # edge case: empty string -> return  
    empty  
    return word[0] if len(word) > 0 else ""
```

```
def is_long_word(word):  
    # heuristic: a long word has more than  
    3 letters  
    return len(word) > 3
```

Question: what would be reasonable implementations of `first_letter()` and `is_long_word()`? (ANSWER ABOVE)

Acronym With HOFs

What if we want to control the filtering method, `is_long_word()`?

```
def keep_words(word):  
    specials = ['Los']  
    return word in specials or long_word(word)
```

```
def acronym_hof(sentence, filter_fn):  
    words = sentence.split()  
    return reduce(add, map(first_letter,  
filter(filter_fn, words)))
```

HOF Summary

* For the builtin filter/map, you need to then call list on it to get a list.

If we define our own, we do not need to call list

```
list(map(function_to_apply, list_of_inputs))
```

Applies function to each element of the list

```
list(filter(condition, list_of_inputs))
```

Returns a list of elements for which the condition is true

```
reduce(function, list_of_inputs)
```

Applies the function, combining items of the list into a "single" value.

(Rehash) Functional Sequence Operations

- Goal: Transform a list, 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

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 Make 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 – leq_maker

- [PythonTutor Link](#)

Demo - compose

[Python Tutor Link](#)

```
def compose(f, g):
```

```
    def h(x):
```

```
        return f(g(x))
```

```
    return h
```

```
add_5 = compose(add_2, add_3)
```

```
y = add_5(7)
```