

# Computational Structures in Data Science

---

Lists & Higher Order Functions

UC Berkeley

# Reminders

Reminders:

<https://go.c88c.org/qa5/> - Use during lecture!

<https://go.c88c.org/5> - self check (after lecture)

**Select / Change your attendance option by next week, 2/11.**

# List Comprehensions vs for Loops

- List comprehensions always return a list!
- For loops do not return anything.

```
my_data = []  
for item in range(10):  
    my_data.append(item)  
my_data
```

```
# or  
my_data = [ item for item in range(10) ]
```

# Computational Structures in Data Science

---

## List Comprehensions

UC Berkeley



# Learning Objectives

- List comprehensions let us build lists "inline".
- List comprehensions are an *expression that returns a list*.
- We can easily "filter" the list using a conditional expression, i.e. `if`

# Data-driven iteration

- describe an expression to perform on each item in a sequence
- let the data dictate the control
- In some ways, nothing more than a concise for loop.

```
[ <expr with loop var> for <loop var> in <sequence expr > ]
```

```
[ <expr with loop var> for <loop var> in <sequence expr >  
if <conditional expression with loop var> ]
```

# Putting the Pieces together.

```
# Let's say we we are taking 4 courses.  
# How many data courses are we taking?  
courses = ['DATA C88C', 'DATA 8', 'POLSCI 2', 'MATH  
54']  
courses.count('DATA') # What's this?  
courses[0]  
courses[0].split()  
  
# Complete me!  
departments = [ _____ ]  
departments.count('DATA')
```

# Putting the Pieces together.

```
# Let's say we we are taking 4 courses.  
# How many data courses are we taking?  
courses = ['DATA C88C', 'DATA 8', 'POLSCI 2', 'MATH  
54']  
courses.count('DATA') # What's this?  
courses[0]  
courses[0].split()  
  
# Complete me!  
departments = [ c.split()[0] for c in courses ]  
departments.count('DATA')
```

# Why use list comprehensions?

- Transforming elements in a list
- Filtering or removing elements from a list
- Combining the two

This is a *surprising* number of tasks!

Next up, do this in a *functional* way

# Demo!

# Computational Structures in Data Science

---

Higher Order Functions

UC Berkeley

# 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

# Code is a Form of Data

- Numbers, Strings: All kinds of data
- Code is its own kind of data, too!
- Why?
  - More expressive programs, a new kind of abstraction.
  - "Encapsulate" logic and data into neat packages.
- This will be one of the trickier concepts in CS88.

# 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.

# Brief Aside: `import`

- Python organizes code in modules
    - These functions come with Python, but you need to "import" them.
  - `import module_name`
    - gives us access to `module_name` and `module_name.x`
  - `import module_name as my_module`
    - can access `my_module` and `my_module.x` (same code, just a different name)
  - `from module_name import x, y, z`
    - can only access the functions we import. `x` is `my_module.x`
- ```
from math import pi, sqrt
from operator import mul
```

# An Interesting Example

$$\sum_{k=1}^5 k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^5 k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^5 \frac{8}{(4k-3) \cdot (4k-1)} = \frac{8}{3} + \frac{8}{35} + \frac{8}{99} + \frac{8}{195} + \frac{8}{323} = 3.04$$

# Why Higher Order Functions?

- We can sum 1 to  $N$  easily enough.
- We can sum 1 to  $N^2$  easily enough too.
- Or we can sum, 1 to  $N^3$ ...
- But why write so many functions?

**Why not write *one function(!)* which allows us flexibility in solving many problems?**

# Summing Multiple Ways

```
def sum_to_n(n):  
    total = 0  
    for i in range(n + 1):  
        total = total + 1  
    return total
```

```
def sum_to_by_twos(n):  
    total = 0  
    for i in range(n + 1):  
        total = total + 2  
    return total
```

```
def sum_by_squares(n):  
    total = 0  
    for i in range(n + 1):  
        total = total + i * i  
    return total
```

```
def sum_to_by_cubes(n):  
    total = 0  
    for i in range(n + 1):  
        total = total + i ** 3  
    return total
```

# A Generic Sum Function

```
def summation(n, _____):  
    """Sum the first N terms of a sequence.  
    >>> summation(5, cube)  
    225  
    >>> summation(5, identity)  
    15  
    >>> summation(10, identity)  
    55  
    """"  
  
    total = 0  
    for i in range(n + 1):  
        total = total + _____  
    return total
```

# A Generic Sum Function

```
def summation(n, term):  
    """Sum the first N terms of a sequence.  
    >>> summation(5, cube)  
    225  
    >>> summation(5, identity)  
    15  
    >>> summation(10, identity)  
    55  
    """  
    total = 0  
    for i in range(n + 1):  
        total = total + term(i)  
    return total
```

# Computational Structures in Data Science

---

Lambda Expressions

UC Berkeley

# 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  $\lambda$

# Why Use $\lambda$ 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

```
>>> add_3 = lambda x: x + 3
```

```
>>> add_3(1)
```

```
4
```

```
>>> square = lambda x: x * x
```

```
>>> square(4)
```

```
16
```

```
# Next Up
```

```
>>> list(map(lambda x: x + 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)
```

`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())
```

# 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.

```
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

UC Berkeley

# 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                                                 |
|---------------|--------------------------------|-------------------------------------------|--------------------------------------|--------------------------------------------------------|
| <b>map</b>    | Transform every item           | 1 (each item)                             | "Anything", a new item               | <b>List:</b> same length, but possibly new values      |
| <b>filter</b> | Return a list with fewer items | 1 (each item)                             | A Boolean                            | <b>List:</b> possibly fewer items, values are the same |
| <b>reduce</b> | "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

UC Berkeley

# 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

```
>>> add_3 = lambda x: x + 3
```

```
>>> list(map(add_3, [1, 2, 3, 4]))  
[4, 5, 6, 7]
```

```
>>> list(map(lambda x: x+3, [1, 2, 3, 4]))  
[4, 5, 6, 7]
```

# Computational Structures in Data Science

---

Lists & Higher Order Functions:  
Filter

UC Berkeley

# 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]
```

# Computational Structures in Data Science

---

Lists & Higher Order Functions  
Reduce

UC Berkeley

# 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

UC Berkeley

# 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(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!

# Acronym With HOFs

What if we want to control the filtering method?

```
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)))
```

# Three super important HOFs

\* 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.

# 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                                                 |
|---------------|--------------------------------|-------------------------------------------|--------------------------------------|--------------------------------------------------------|
| <b>map</b>    | Transform every item           | 1 (each item)                             | "Anything", a new item               | <b>List:</b> same length, but possibly new values      |
| <b>filter</b> | Return a list with fewer items | 1 (each item)                             | A Boolean                            | <b>List:</b> possibly fewer items, values are the same |
| <b>reduce</b> | "Combine" items together       | 2 (current item, and the previous result) | Type should match the type each item | A "single" item                                        |