Computational Structures in Data Science

HOFs & Environment Diagrams

Week 2, Summer 2024. 6/27 (Thurs)

Berkeley
UNIVERSITY OF CALIFORNIA

Announcements

- Upcoming due dates
  - HW02, Lab02: Due June 29th (2 days from now!)
- Released today
  - HW03, Lab03 (Due: July 1st)

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

- HOF’s + sequences review/practice
- Environment diagrams practice

Computational Structures in Data Science

HOFs and Sequences

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

Review: Acronym + “exclude words”

```
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: suppose we had a list of (small) words that we know we want to exclude from the acronym builder (eg `[“the”, “of”, “at”, …]`). How could we modify this code to use this list of exclude words?

```
EXCLUDE_WORDS = ["the", "of", "at", "an", "a"]
```

Solution 1

```
Input: "The University of California at Berkeley"
Output: "UCB"

# V1: (straightforward) hardcode EXCLUDE_WORDS in should_keep_word()
def acronym_v1(sentence):
    words = sentence.split()
    return reduce(add, map(first_letter, filter(should_keep_word, words)))

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

def should_keep_word(word):
    EXCLUDE_WORDS = ["the", "of", "at", "an", "a"]
    return word not in EXCLUDE_WORDS and is_long_word(word)
```

Solution 2: HOF's

```
def acronym_v2(sentence, should_keep_word_fn):
    words = sentence.split()
    return reduce(add, map(first_letter, filter(should_keep_word_fn, words)))

def create_filter_fn(exclude_words, min_len):
    def inner_fn(word):
        return word not in exclude_words and len(word) > min_len
    return inner_fn

filter_fn_orig = create_filter_fn(["the", "of", "at", "an", "a"], 3)
filter_fn_silly = create_filter_fn(["Berkeley"], 0)

>>> acronym_v2('The University of California at Berkeley', filter_fn_orig)
UCB
>>> acronym_v2('The University of California at Berkeley', filter_fn_silly)
```

QUESTION: What does this output?

Solution 2: HOF's

```
def acronym_v2(sentence, should_keep_word_fn):
    words = sentence.split()
    return reduce(add, map(first_letter, filter(should_keep_word_fn, words)))

def create_filter_fn(exclude_words, min_len):
    def inner_fn(word):
        return word not in exclude_words and len(word) > min_len
    return inner_fn

filter_fn_orig = create_filter_fn(["the", "of", "at", "an", "a"], 3)
filter_fn_silly = create_filter_fn(["Berkeley"], 0)

>>> acronym_v2('The University of California at Berkeley', filter_fn_orig)
UCB
>>> acronym_v2('The University of California at Berkeley', filter_fn_silly)
TUoCa
```

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

Exercise: reduce

Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:

- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
# 1 + (2**2) + 3 + (4**2) = 24
>>> sum_reduce_weird([1, 2, 3, 4])
24
# 2 + 3 = 5
>>> sum_reduce_weird([2, 3])
5
```

```
def sum_reduce_weird(nums):
    # FILL ME IN
    return reduce(SOMETHING, nums)
```


Exercise: reduce

Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:
- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
# 1 + (2**2) + 3 + (4**2) = 24
>>> sum_reduce_weird([1, 2, 3, 4])
24
# 2 + 3 = 5
>>> sum_reduce_weird([2, 3])
5
```

```
def sum_reduce_weird(nums):
    def reduce_fn(a, b):
        if (b % 2) == 0:
            return a + (b ** 2)
        else:
            return a + b
    return reduce(reduce_fn, nums)
```


Exercise: reduce

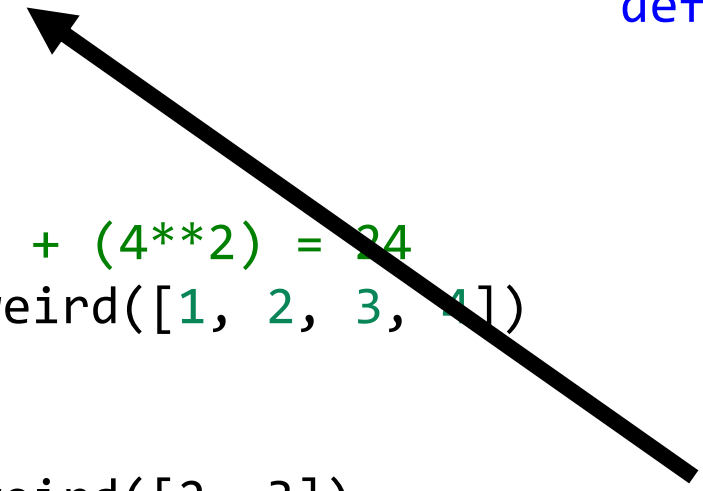
Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:

- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
# 1 + (2**2) + 3 + (4**2) = 24
>>> sum_reduce_weird([1, 2, 3, 4])
24
# 2 + 3 = 5
>>> sum_reduce_weird([2, 3])
5
```



```
def sum_reduce_weird(nums):
    def reduce_fn(a, b):
        if (b % 2) == 0:
            return a + (b ** 2)
        else:
            return a + b
    return reduce(reduce_fn, nums)
```

Question: Why do we need this caveat?

Exercise: reduce

Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:
- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
# 1 + (2**2) + 3 + (4**2) = 24
>>> sum_reduce_weird([1, 2, 3, 4])
24
# 2 + 3 = 5
>>> sum_reduce_weird([2, 3])
5
```

```
def sum_reduce_weird(nums):
    def reduce_fn(a, b):
        if (b % 2) == 0:
            return a + (b ** 2)
        else:
            return a + b
    return reduce(reduce_fn, nums)
```

Question: Why do we need this caveat?
Answer: because `reduce()` starts the initial value as the first sequence element.

Exercise: reduce

Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:
- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
>>> sum_reduce_weird_v2([1, 2, 3, 4])
# FILL ME IN
>>> sum_reduce_weird_v2([2, 3])
# FILL ME IN
```

```
def sum_reduce_weird_v2(nums):
    def reduce_fn(a, b):
        if (b % 2) == 0:
            return b ** 2
        else:
            return b
    return reduce(reduce_fn, nums)
```

Question: for this `sum_reduce_weird_v2()` implementation, what would python print?

Exercise: reduce

Question: given a list of integers, fill in the `sum_reduce_weird()` function so that it sums the input integers according to the following rule:
- If the integer is even: add its **value squared** to the output
- If the integer is odd: add its **unmodified value** to the output

Caveat: for the first integer in the list, add its **unmodified value** to the output.

Example:

```
>>> sum_reduce_weird_v2([1, 2, 3, 4])
16
>>> sum_reduce_weird_v2([2, 3])
3
```

```python
def sum_reduce_weird_v2(nums):
    def reduce_fn(a, b):
        if (b % 2) == 0:
            return b ** 2
        else:
            return b
    return reduce(reduce_fn, nums)
```

Question: for this `sum_reduce_weird_v2()` implementation, what would python print?

Intuition: the first arg to the reduce_fn, `a`, is the "accumulator" (or "total-so-far"). Omitting it means `reduce()` can't keep track of the total output!

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 commom

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

Environment Diagrams Rules

- Tip: Use Python Tutor to visualize the environment diagram rules on your own code!
  - https://tutor.cs61a.org/
  - (found on course page by clicking “Python Tutor” at top bar)

Python Tutor Example #1 (LINK)

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

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

Main takeaway from this:

- Function calls create a new frame
- Variable resolution order always starts at current frame, and if the variable is not found in current frame, try to find it in the parent frame(s).

Python Tutor Example #2 (LINK)

```
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()
```

Main takeaway from this:

- Variable resolution order always starts at current frame, and if the variable is not found in current frame, try to find it in the parent frame(s).
  - Global frame is the parent-most frame!

Python Tutor Example #3 (LINK)

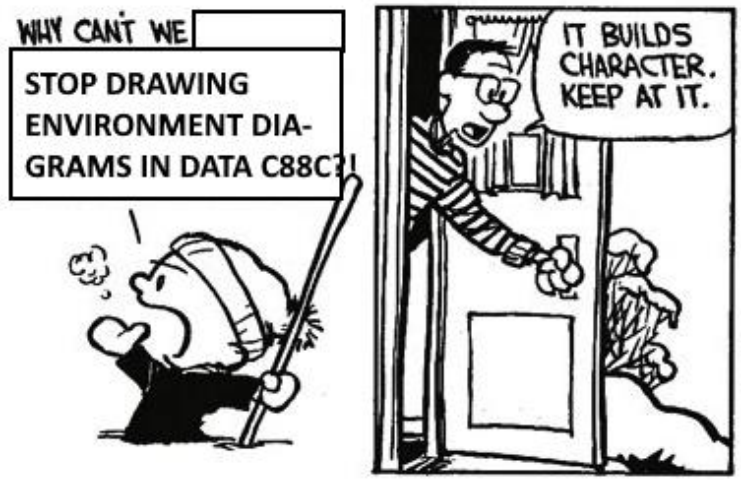
```
def make_adder(a):
    return lambda x: x + a

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)
```

Main takeaway from this:

- A function's parent frame is the frame in which it was created, NOT where it is called from!
  - Aka "lexical/static" scoping.
- Lambdas behave just like functions, but with "no name" (anonymous fn)
- Variable "shadowing": the `x` in `h()` frame "**shadows**" the `x` in the global frame

Environment Diagram Tips / Links

- NEVER draw an arrow from one variable to another.
- **Practice tip**: recreate the environment diagram pictures manually with pencil + paper, then check your work with Python Tutor!
  - Tedious, but it's helpful (and it builds character!)
- Useful Resources:
  - http://markmiyashita.com/cs61a/environment_diagrams/rules_of_environment_diagrams/
  - http://albertwu.org/cs61a/notes/environments.html
  - https://tutor.cs61a.org/



Credit: Bill Waterson, "Calvin and Hobbes"

Today’s overview. Any questions?

- HOF’s + sequences review/practice
- Environment diagrams practice