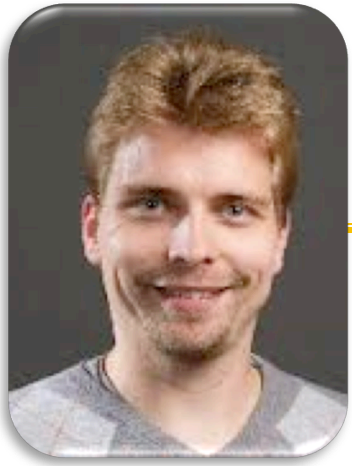

Computational Structures in Data Science

UC Berkeley EECS
Adj. Ass. Prof.
Dr. Gerald Friedland

Lecture #3: Control Recap & Higher Order Functions

February 11, 2019

http://inst.eecs.berkeley.edu/~cs88

Solutions for the Wandering Mind

- **Could we build a complete computer that has no instructions, only data?**

Yes! A computer that only uses a single instruction doesn't have to distinguish between instructions. The program is a sequence of arguments to that instruction.

One Instruction Computer: https://en.wikipedia.org/wiki/One_instruction_set_computer

Generalization: Cellular Automaton (Rule F110) https://en.wikipedia.org/wiki/Cellular_automaton

Is this how the universe works?

Administrative issues

- **Tutoring**
  – To help you prepare for exams, we will be hosting small group tutoring we will also be having guerrilla section.
  – Pay attention on Piazza and ask TAs for details.

- **Midterm Thursday 3/7. DSP and make-up details TBD.**

Computational Concepts Toolbox

- Data type: values, literals, operations,
  - e.g., int, float, string
- Expressions, Call expression
- Variables
- Assignment Statement
- Sequences: tuple, list
- Data structures
- Tuple assignment
- Call Expressions
- Function Definition Statement
- Conditional Statement
- Iteration:
  - data-driven (list comprehension)
  - control-driven (for statement)
  - while statement

Computational Concepts today

- Recap: Control structures
- Higher Order Functions
- Functions as Values
- Functions with functions as argument
- Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions

Big Idea: Software Design Patterns

`for` statement – iteration control

- **Repeat a block of statements for a structured sequence of variable bindings**

```
<initialization statements>
for <variables> in <sequence expression>:
  <body statements>

<rest of the program>
```

```
def cum_OR(lst):
   """Return cumulative OR of entries in lst.
   >>> cum_OR([True, False])
   True
   >>> cum_OR([False, False])
   False
   """
   co = False
   for item in lst:
          co = co or item
   return co
```

`while` statement – iteration control

- **Repeat a block of statements until a predicate expression is satisfied**

```
<initialization statements>
while <predicate expression>:
    <body statements>

<rest of the program>
```

```
def first_primes(k):
    """ Return the first k primes.
    """
    primes = []
    num = 2
    while len(primes) < k :
        if prime(num):
            primes = primes + [num]
        num = num + 1
    return primes
```

Data-driven iteration

- **describe an expression to perform on each item in a sequence**
- **let the data dictate the control**

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

```
def dividers(n):
    """Return list of whether numbers greater than 1 that divide n.

    >>> dividers(6)
    [True, True]
    >>> dividers(9)
    [False, True, False]
    """
    return [divides(n,i) for i in range(2,(n//2)+1) ]
```

iClicker Fun

- **My favorite color is?**

  A) Green
  B) Blue
  C) Red
  D) Yellow
  E) Pink

- **Hint: Go bears!**

Solution:
G) Gold

Control Structures Review

- **A *while* loop is superior to a *for* loop?**

  A) Correct
  B) Wrong

Solution:
A) Everything that a *for* loop can do can be implemented with a *while* loop. But not everything that a *while* loop can do is implementable in a *for* loop. Example: *while not key_pressed():*

Control Structures Review

- **List comprehension is superior to a *for* loop?**

  A) Correct
  B) Wrong

Solution:
B) No. They are just two different constructs.

Control Structures Review

- **A function should…**

  A) implement as many features as possible
  B) have a short name (Occam's Razor)!
  C) implement one thing well
  D) A & B
  E) B & C

Solution:
C) Make the function as short as possible but not shorter to do <u>one thing well</u>.

Control Structures Review

- **The result of** *range(0,10)* **is…**

  A) *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]*
  B) *[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*
  C) *[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]*
  D) *[1, 2, 3, 4, 5, 6, 7, 8, 9]*
  E) an error

Solution:
A) *range(m,n)* **creates a list with elements from m to n-1.**

Control Structures Review

- **The result of** *[i for i in range(3,9) if odd(i)]* **is…**

  A) *[3, 4, 5, 6, 7, 8, 9]*
  B) *[3, 4, 5, 6, 7, 8]*
  C) *[1, 3, 5, 7, 9]*
  D) *[3, 5, 7, 9]*
  E) *[3, 5, 7]*

Solution:
E) *[3, 5, 7]*

Control Structures Review

- **The result of** *len([i for i in range(1,10) if even(i)])* **is…**

  A) 5
  B) 4
  C) 3
  D) 2
  E) 1

Solution:
B) *len([2, 4, 6, 8])=4*

Iteration Review

- **When should we use a *for* loop, rather than list comprehension?**

  A) Always
  B) On the midterm/final
  C) When the Prof/TA tells me so
  D) When I am not creating a list
  E) C & D

Solution:
D) if no list is needed, a *for* loop is more efficient

Higher Order Functions

- **Functions that operate on functions**
- **A function**

```
def odd(x):
    return (x%2==1)

>>> odd(3)
True
```

Why is this not ‘odd’ ?

- **A function that takes a function arg**

```
def filter(fun, s):
    return [x for x in s if fun(x)]

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

Higher Order Functions (cont)

- **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

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

One more example

- **What does this function do?**

```
def split_fun(p, s):
    """ Returns <you fill this in>."""
    return [i for i in s if p(i)], [i for i in s if not p(i)]
```

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

Three super important HOFS

`map(function_to_apply, list_of_inputs)`

Applies function to each element of the list

`filter(condition, list_of_inputs)`

Returns a list of elements for which the condition is true

`reduce(function, list_of_inputs)`

Reduces the list to a result, given the function

Function Factories

```
def linemaker(m, b):
    def linefun(x):
# Create a function that embeds the parameters of the line
        return m*x + b
# Return that dynamically created function
return linefun
```

```
def make_decoder(code_map):
    """Make a decoder function specified by a map"""
    def decode(code):
        for (code_num, desc) in code_map:
            if code == code_num:
                return desc
        return "unknown"
    return decode
```

Computational Concepts today

- Higher Order Functions
- Functions as Values
- Functions with functions as argument
- Assignment of function values
- Higher order function patterns
  - Map, Filter, Reduce
- Function factories – create and return functions

Big Idea: Software Design Patterns

Thoughts for the Wandering Mind (Holiday Edition)

- How many answers can be maximally responded to by 20 questions (how much data do I need on my game device)?

- How can a 20-questions game get away with less?

- How can you make a 20-questions game fail (adversarial attack)?