CS 61A Midterm 1 HKN Review

Soumya Basu
Brian Hou

Agenda

Hosted by HKN (hkn.eecs.berkeley.edu)

Office hours from 11AM to 5PM in 290 Cory, 345 Soda

Check our website for exam archive, course guide, course surveys, tutoring schedule.

- What Would Python Print
- Environment Diagrams
- Higher Order Functions
- Lambda Functions
- Newton's Method
- Recursion

Follow along at bit.ly/1owv0DT!

What will Python print?

```
def while_loop(n):
    i, j = 0, 1
    while i < n:
        j += 1
        while j < n:
            i += 1
            if j % i == 1:
                print(i, j)
            j += 1
        i += 1

while_loop(5)
```

```
>>> while_loop(5)
```

What will Python print?

```
def best(n):
    def pikachu():
        print('pika!')
        return n
    bulbasaur = lambda: 2
    charmander = (3, 'pika!')
    if pikachu() < bulbasaur():
        n += charmander[0]
    elif pikachu() == charmander[1]:
        print('pikachu!')
    else:
        if pikachu() % 2 == 1:
            return 'squirtle'
```

```
>>> best(4)
________
>>> ashy_boy = best(3)
________
>>> ashy_boy
________
>>> best(1)
________
```

Controlling your printing

```
def print_moar(stuff):
    i = 0
    while stuff and i < 2:
        stuff = print(stuff, print('soumya'))
        i += 1
    return stuff

>>> brian = print_moar('cs61a')
_________
>>> brian
_________
```

Boolean Expressions

BONUS!

For reference, look at lab 2 titled "Control"

- A *boolean expression* is one that evaluates to either `True`, `False`, or sometimes an `Error`.
- When evaluating boolean expressions, we follow the same rules as those used for evaluating other statements and function calls.
- The order of operations for booleans (from highest priority to lowest) is: **not, and, or**

The following will evaluate to `True`:

```
True and not False or not True and False
```

You can rewrite it using parentheses to make it more clear:

```
(True and (not False)) or ((not True) and False)
```

More Boolean Expressions

BONUS!

Short-circuiting

- Expressions are evaluated from left to right in Python.
- Expressions with **and** will evaluate to `True` only if *all* the operands are `True`. For multiple **and** expressions, Python will go left to right until it runs into the first `False` value -- then the expression will immediately evaluate to `False`.
- Expressions with **or** will evaluate to `True` if *at least one* of the operands is `True`. For multiple **or** expressions, Python will go left to right until it runs into the first `True` value -- then the expression will immediately evaluate to `True`. For example:

  ```
  5 > 6 or 4 == 2*2 or 1/0
  ```

  This evaluates to `True` because of short-circuiting.

Environment Diagrams

```
lamps = lambda you, me: you + me
def brian(mark):
    def flour(based):
        return based(mark)
    mark = 'hi'
    return flour
lamps, brian = brian, lamps
answer = lamps(brian(3, 2))
answer(print)
```

Environment Diagrams

```
fruit = lambda: apples
apples = 5
def domo(eats, apples):
        def rawr():
                return fruit()
        print(eats())
        rawr()
domo(fruit, 42)
```

Number Fun

Implement `digit_span`, a function that takes as input a positive integer and returns the difference between its largest digit and its smallest digit.

```
def digit_span(n):
    """Return the difference between the largest
    and smallest digits.

    >>> digit_span(2013)
    3
    >>> digit_span(7)
    0
    """
```

Higher-Order Functions

A function that takes in a function as an argument and/or returns a function.

```
def sum_if_even(n):
    """
    Returns the sum of the even natural numbers from 1
    to n inclusive.
    >>> sum_if_even(10)
    30
    >>> sum_if_even(2)
    2
    >>> sum_if_even(15)
    56
    """
```

Higher-Order Functions

```
def product_if_prime(n):
    """
    Returns the product of the prime natural numbers
    from 1 to n inclusive.
    >>> product_if_prime(10)
    210
    >>> product_if_prime(2)
    2
    >>> product_if_prime(6)
    30
    """
```

Higher-Order Functions

```
def sum_if_even(n):
    i = 0
    total = 0
    while i <= n:
        i += 1
        if i % 2 == 0:
            total += i
    return total
```

```
def product_if_prime(n):
    i = 0
    total = 1
    while i <= n:
        i += 1
        if is_prime(i):
            total *= i
    return total
```

```
def accumulate_if_pred(n, start, pred, combiner):
```

Higher-Order Functions

Start with a specific implementation and highlight the parts that are different.

```
def accumulate_if_pred(n, start, pred, combiner):
    i = 0
    total = 1
    while i <= n:
        i += 1
        if is_prime(i):
            total *= i
    return total
```

Higher-Order Functions

Generalize carefully and you're done!

```
def accumulate_if_pred(n, start, pred, combiner):
    i = 0
    while i <= n:
        i += 1
        if is_prime(i):
            total *= i
    return start
```

Higher-Order Functions

Generalize carefully and you’re done!

```
def accumulate_if_pred(n, start, pred, combiner):
    i = 0
    while i <= n:
        i += 1
        if pred(i):
            total *= i
    return start
```

Higher-Order Functions

Generalize carefully and you're done!

```
def accumulate_if_pred(n, start, pred, combiner):
    i = 0
    while i <= n:
        i += 1
        if pred(i):
            start = combiner(start, i)
    return start
```

Lambda Functions

- Unnamed function, no assignments

```
    "lambda <arguments>: <return value>"
>>> g = lambda y: y % 2
>>> g(4)
________
>>> g(7)
________
>>> h = lambda x: lambda y: z
>>> h(1)
________
>>> h(1000)(1)
________
```

Lambda Functions

And, we can call a lambda expression without ever giving it a name!

```
>>> f = lambda x: x + 1
>>> f(4)
```

____

```
>>> (lambda x: x + 1)(4)
```

____

```
>>> (lambda y: y(3))(lambda x: x + 4)
```

____

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
>>> x
______
>>> x(3)
______
>>> x(3)(4)
______
>>> x(3)()
______
>>> x(3)()(7)
______
```

Lambda Functions

Try on your own!

```
>>> x = lambda x: lambda: lambda y: 2 * x + 3 * y
```

We can rewrite the lambda expressions using HOFs:

```
def L1(x):
# The line above could be "def x(x):" Why?
    def L2():
        def L3(y):
            return 2 * x + 3 * y
        return L3
    return L2
```

Applications of Higher-Order Functions: Newton's Method

Newton's method is used to find (approximately) the roots (or *zeros*) of a function f, which are the input values for which the function evaluates to zero.

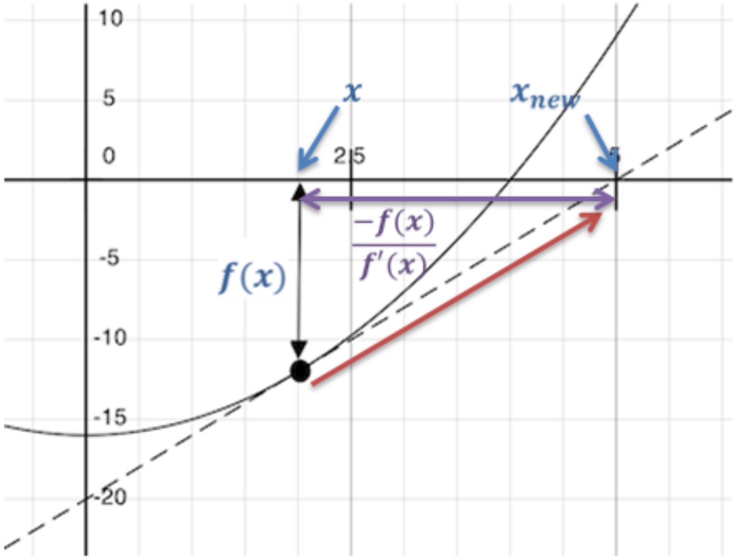
Many mathematical problems are equivalent to finding roots of specific functions.

"Square root of 2 is x, $x^2 = 2$" is equivalent to "Find a root of $x^2 - 2$."

"(Any) Angle x whose sine is 0.5" is equivalent to "Find a root of sin(x) - 0.5."

Applications of Higher-Order Functions: Newton's Method

1. Start with a function `f` and a guess x.
2. Compute the value of the function `f` at x.
3. If zero, we are done; else, compute the derivative of `f` at x, `f'(x)`.
4. Update guess to be `x - (f(x)/f'(x))`.



Animation: http://en.wikipedia.org/wiki/File:NewtonIteration_Ani.gif

Applications of Higher-Order Functions: Newton's Method

Newton's method is an instance of *iterative improvement*.

Step 1: ***Guess*** an answer to the problem.

Step 2: If the guess is ***(approximately) correct***, it is the solution; otherwise, ***update*** the guess and repeat this step.

Applications of Higher-Order Functions: Newton's Method

```
def improve(update, close, guess=1):
    while not close(guess):
        guess = update(guess)
    return guess
```

Applications of Higher-Order Functions: Newton's Method

```
def improve(update, close, guess=1,
            max_updates=100):
    k = 1
    while not close(guess) and k < max_updates:
        guess = update(guess)
        k = k + 1
    return guess
```

Applications of Higher-Order Functions: Newton's Method

```
def approx_eq(x, y, tolerance=1e-15):
    return abs(x - y) < tolerance


def newton_update(f, df):
    def update(guess):
        return guess - (f(guess) / df(guess))
    return update


def find_zero(f, df):
    # Uses Newton’s method to find a zero of a function
    # f, whose derivative is the function df.
    def near_zero(guess):
        return approx_eq(f(guess), 0)
    return improve(newton_update(f, df), near_zero)
```

Applications of Higher-Order Functions: Newton's Method

"Square root of 2 is x, $x^2 = 2$."

```
find_zero(lambda x: x**2 - 2, lambda x: 2*x)
```

"(Any) Angle x whose sine is 0.5."

```
find_zero(lambda x: sin(x) - 0.5, lambda x: cos(x))
```

"Number x that is one less than its square, or $x = x^2 - 1$."

```
find_zero(______________________________________)
```

Applications of Higher-Order Functions: Newton's Method

Not every function has a clean formula for its derivative. Write a function `derivative` that takes in a function and returns another function that approximates its derivative. Recall that

$$f'(x) \approx \frac{f(x+dx)-f(x)}{dx}$$

```
>>> derivative(lambda x: x**2)(3)    # Example usage
6.00010000012054


def derivative(f, dx=1e-4):
```

Applications of Higher-Order Functions: Newton's Method

Write a function `find_extremum` that takes a function, and finds a local extremum of that argument function. A *local extremum* is a point at which the derivative of the function is zero. (You may assume that the argument function will never have any saddle points.)

An example call is provided:

```
>>> f = lambda x: (x-1)**2 # parabola with minimum at x=1
>>> find_extremum(f)
0.999995 # very close to 1
```

Recursion

BONUS!

A function is ***recursive*** if the body calls the function itself, either directly or indirectly.

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Recursion

BONUS!

A recursive function has two important components:

1. A *base case*, where the function does not recursively call itself, and instead returns a direct answer. This is reserved for the simplest inputs.
2. A *recursive case*, where the function calls itself. The call must be made with an argument that drives the function towards the base case.

Recursion

A recursive function has two important components:

1. A *base case*.
2. A *recursive case*.

```
def factorial(n):
    if n == 1 or n == 0:
        return 1
    return n * factorial(n - 1)
```

Visualization: http://goo.gl/ux5MuQ

Recursion

Write a recursive function `add_until` that adds all of the integers from `1` to a positive integer `n`.

```
def add_until(n):
```

Recursion

Write a recursive function `eat_chocolate` that takes in a number of chocolate pieces and returns a string as follows:

```
>>> eat_chocolate(5)
"nom nom nom nom nom"
>>> eat_chocolate(2)
"nom nom"
>>> eat_chocolate(1)
"nom"
>>> eat_chocolate(0)
"No chocolate :("
```

Recursion

Write a function `count_occurrences` that takes in a number and a digit and counts the number of times the digit appears in the number.

```
def count_occurrences(num, digit):
```

Recursion

Write the function `call_until_one` recursively. Recall that it takes a function we are interested in as an argument. It returns another function that, when called on a number, will tell you how many times you can call that original function on the number until it will return a value less than or equal to 1.

```
>>> f = call_until_one(lambda x: x - 1)
>>> f(100)
99

>>> g = call_until_one(lambda x: x / 2)
>>> g(128)
7
```