CS61A Lecture 3

Higher Order Functions

Jon Kotker and Tom Magrino
UC Berkeley EECS
June 20, 2012
**COMPUTER SCIENCE IN THE NEWS**

*The (Literal) Evolution of Music*

- Scientists from Imperial College London created a computer program powered by Darwinian natural selection.
  
  Theory that cultural changes in language, art, and music evolve like living things do, since consumers *choose* what is “popular”.

- Program would produce loops of random sounds and analyze opinions of musical consumers. The top loops were then “mated”.

- “DarwinTunes” (darwintunes.org) has evolved through at least 2513 generations.

def foo(n):
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k = k + 1
    return True

foo(7)? foo(9)? foo(1)?
TODAY

Domain and range of functions
... and feeding functions to functions

Wat.
**Domain and Range**

Domain is the *set of values* that the function is *defined for.*

\[\text{square}(n): \quad \text{return } n^2\]

Range (or image) is the *set of values* that the function *returns.*

Domain of square: All real *numbers.*
Range of square: All non-negative real *numbers.*
Remember the domain and range of the functions that you are writing and using!

Many bugs arise because programmers forget what a function can and cannot work with.

The domain and range are especially important when working with higher order functions.
PROBLEM: FINDING SQUARES

```python
>>> square_of_0 = 0*0
>>> square_of_1 = 1*1
>>> square_of_2 = 2*2
>>> square_of_3 = 3*3
...
>>> square_of_65536 = 65536*65536
>>> square_of_65537 = 65537*65537
...
```
**Problem: Finding Squares**

```python
>>> square_of_0 = 0*0
>>> square_of_1 = 1*1
>>> square_of_2 = 2*2
>>> square_of_65536 = 65536*65536
>>> square_of_65537 = 65537*65537
...```

There has to be a better way!
PROBLEM: FINDING SQUARES

>>> square_of_0 = 0*0
>>> square_of_1 = 1*1
>>> square_of_2 = 2*2
>>> square_of_3 = 3*3

... >>> square_of_65536 = 65536*65536
>>> square_of_65537 = 65537*65537

... 

Can we generalize?
FINDING SQUARES: GENERALIZATION

THE SQUARE OF A NUMBER

def square(n):
    return n * n

IS THE NUMBER MULTIPLIED BY ITSELF
PROBLEM: SUMS OF SERIES

def sum_of_n_squares(n):
    '''Returns $1^2 + 2^2 + 3^2 + \ldots + n^2.$'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + square(k), k + 1
    return sum

def sum_of_n_cubes(n):
    '''Returns $1^3 + 2^3 + 3^3 + \ldots + n^3.$'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + cube(k), k + 1
    return sum
**Problem: Sums of Series**

```python
from math import sin, sqrt
def sum_of_n_sines(n):
    '''Returns \( \sin(1) + \sin(2) + \sin(3) + \ldots + \sin(n) \).'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + sin(k), k + 1
    return sum

def sum_of_n_sqrts(n):
    '''Returns \( \sqrt{1} + \sqrt{2} + \ldots + \sqrt{n} \).'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + sqrt(k), k + 1
    return sum

and so on...
```
**PROBLEM: SUMS OF SERIES**

```
from math import sin, sqrt
def sum_of_n_sines(n):
    '''Returns \( \sin(1) + \sin(2) + \sin(3) + \cdots + \sin(n) \).'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + sin(k), k + 1
    return sum

def sum_of_n_sqrts(n):
    '''Returns \( \sqrt{1} + \sqrt{2} + \cdots + \sqrt{n} \).'''
    sum, k = 0, 1
    while k <= n:
        sum, k = sum + sqrt(k), k + 1
    return sum

and so on...
```
**PROBLEM: SUMS OF SERIES**

```python
def sum_of_n_squares(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + square(k)
        k = k + 1
    return sum

def sum_of_n_cubes(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + cube(k)
        k = k + 1
    return sum

def sum_of_n_sines(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + sin(k)
        k = k + 1
    return sum

def sum_of_n_sqrts(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + sqrt(k)
        k = k + 1
    return sum
```

Can we generalize?
ANNOUNCEMENTS

• HW2 is released, due **Tuesday, June 26**.
• Discussions earlier held in 320 Soda will now be held in **310 Soda** for the next six weeks. We will move back to 320 Soda for the last two weeks.
• Groups for studying and midterms will be assembled on **Thursday, June 21** in discussion section.
EVALUATION OF PRIMITIVE EXPRESSIONS

>>> 3
3 evaluates to 3

>>> x = 5
Statement

>>> x
5 evaluates to 5

>>> 'CS61A'
'CS61A' evaluates to 'CS61A'
FUNCTION NAMES

>>> square
<function square at 0x0000000002279648>

Python’s representation of

square(n):

return n*n

square is the name of the function, or “machine”, that squares its input

Address where the function is stored in the computer.
It is not fixed.
FUNCTION NAMES

>>> square
<function square at 0x00000000002279648>
>>> cube
<function cube at 0x000000000022796C8>
>>> max
<built-in function max>

Function names are *primitive expressions* that evaluate to the corresponding “machines”
Using Primitive Expressions

We have seen primitive expressions used as arguments to functions:

```python
>>> square(3)
9
>>> x = 4
>>> square(x)
16
```

Can we then use function names as arguments?
**Problem: Sums of Series**

```python
def sum_of_n_squares(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + square(k)
        k = k + 1
    return sum

def sum_of_n_cubes(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + cube(k)
        k = k + 1
    return sum

def sum_of_n_sines(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + sin(k)
        k = k + 1
    return sum

def sum_of_n_sqrts(n):
    sum, k = 0, 1
    while k <= n:
        sum = sum + sqrt(k)
        k = k + 1
    return sum
```
FUNCTIONS AS ARGUMENTS

def summation(n, term):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = k + 1
    return sum

def sum_of_n_squares(n):
    return summation(n, square)

term(1) + term(2) + ... + term(n)
FUNCTIONS AS ARGUMENTS

def summation(n, \textit{square}):  
    \textit{sum}, \textit{k} = 0, 1  
    \textbf{while} \textit{k} \leq \textit{n}:  
        \textit{sum} = \textit{sum} + \textit{square}(\textit{k})  
        \textit{k} = \textit{k} + 1  
    \textbf{return} \textit{sum}

def sum_of_n_squares(n):  
    \textbf{return} \text{summation}(n, \textit{square})
**FUNCTIONS AS ARGUMENTS**

1. **square(n):**
   
   ```python
   return n*n
   ```

2. **summation(n, term):**
   
   ```python
   sum, k = 0, 1
   while k <= n:
       sum = sum + term(k)
       k = k + 1
   return sum
   ```

summation takes a function as an argument, and is thus a higher order function.
Higher Order Functions

http://cdn.memegenerator.net/instances/400x/13712469.jpg
**HIGHER ORDER FUNCTIONS: PRACTICE**

```python
def summation(n, term):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = k + 1
    return sum

def sum_of_n_cubes(n):
    return summation(n, lambda k: k**3)

def sum_of_n_sines(n):
    return summation(n, lambda k: math.sin(k))

def sum_of_n_positive_ints(n):
    return summation(n, lambda k: k)**2
```
def summation(n, term):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = k + 1
    return sum

def sum_of_n_cubes(n):
    return summation(n, lambda x: x ** 3)

def sum_of_n_sines(n):
    return summation(n, sin)

def sum_of_n_positive_ints(n):
    return summation(n, lambda x: x)
HIGHER ORDER FUNCTIONS

Functions that can take other functions as arguments are considered *higher order functions*.

The *domains* of these functions now include *other functions*.

*Functions* can be treated as *data*!
Why “higher order”? 

*First-order functions* only take non-function values, like numbers and strings, as arguments.  
*Second-order functions* can take first-order functions as arguments.  
*Third-order functions* can take second-order functions as arguments.  

... and so on.
HOFs ELSEWHERE

\[ \int_{a}^{b} f(x) \, dx \]

Integral \( f \) is a **function** that takes three arguments: lower limit \( a \), upper limit \( b \), and a **function** \( f \).
HOFs ELSEWHERE

YOU (KIND OF) are a function (KIND OF)

Recipe (Function) + Ingredients → Food
HOFs ELSEWHERE

Basketball

Playing

NOUN (Data)

VERB (Function)

Interests: Playing Basketball

NOUN ("Gerund") (Function as data)
FUNCTIONS AS ARGUMENTS

def summation2(n, \textit{term}, \textit{next}): 
    \texttt{sum, k = 0, 1}
    \texttt{while k <= n:}
        \texttt{sum = sum + term(k)}
        \texttt{k = next(k)}
    \texttt{return sum}
FUNCTIONS AS ARGUMENTS: PRACTICE

def summation2(n, term, next):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = next(k)
    return sum

def summation(n, term):
    return summation2(n, term, ________________)
FUNCTIONS AS ARGUMENTS: PRACTICE

```python
def summation2(n, term, next):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = next(k)
    return sum

def summation(n, term):
    return summation2(n, term,
                      lambda x: x + 1)
```

def summation(n, term):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = k + 1
    return sum

def sum_of_n_even(n):
    """Returns the sum of the first n even numbers.""
    return summation(n, ________________)

def sum_of_n_odd(n):
    """Returns the sum of the first n odd numbers.""
    return summation(n, ________________)

def sum_of_n_starting_from_m(m, n):
    """Returns the sum of the first n numbers starting from m.""
    return summation(n, ________________)
def summation(n, term):
    sum, k = 0, 1
    while k <= n:
        sum = sum + term(k)
        k = k + 1
    return sum

def sum_of_n_even(n):
    '''Returns the sum of the first n even numbers.'''
    return summation(n, lambda x: 2 * x)

def sum_of_n_odd(n):
    '''Returns the sum of the first n odd numbers.'''
    return summation(n, lambda x: 2 * x + 1)

def sum_of_n_starting_from_m(m, n):
    '''Returns the sum of the first n numbers starting from m.'''
    return summation(n, lambda x: x + m)
**Higher Order Functions: Practice**

The value of the *derivative* of a function $f$ at a point $x$ can be approximated as

\[
\frac{f(x + \Delta x) - f(x)}{\Delta x}
\]

if $\Delta x$ is really small.

Write the function `deriv` that takes as arguments a function $f$, a point $x$, and a really small value `delta_x`, and returns the approximate value of the derivative at $x$.

```python
def deriv(f, x, delta_x):
    return __________________
```
The value of the derivative of a function $f$ at a point $x$ can be approximated as

$$\frac{f(x + \Delta x) - f(x)}{\Delta x}$$

if $\Delta x$ is really small.

Write the function `deriv` that takes as arguments a function $f$, a point $x$, and a really small value `delta_x`, and returns the approximate value of the derivative at $x$.

```python
def deriv(f, x, delta_x):
    return (f(x + delta_x) - f(x)) / delta_x
```
CONCLUSION

- It is important to remember the domain and range of functions!
- Functions can take functions as arguments.
- **Preview**: Functions can also *return* functions.