“My CS 5 Black Lecture Notes Were Stolen By Aliens” claims distraught CS 5 Professor.

Claremont, CA: A Harvey Mudd CS Professor claims that his lecture notes were stolen by five-eyed aliens early on Thursday morning. “I wrote up some notes for my lecture and, while I went out to get a Spam sandwich before class, aliens snuck into my office and stole my notes to eat for breakfast,” claimed the distraught professor. “We’re obligated to investigate every claim,” said a campus security officer, “but we’re skeptical of this one. Aliens are not known to eat class notes for breakfast. They much prefer Spam-flavored Pop Tarts and Red Bull. We’re investigating, but we suspect that squirrels were the actual perpetrators of this terrible crime.” Why are you reading this drivel? Don’t you have anything better to do?!
A true Story of my friend
Arthur Benjamin

That’s crArizonay!
A true Story of my friend
Arthur BeNJamin

That’s crArizonay!
Date: Wed, 10 Feb 2010 10:39:45 -0800 (PST)
From: Arthur Benjamin <benjamin@math.hmc.edu>
To: Tina Straley <tstraley@maa.org>
Cc: Arthur Benjamin <benjamin@math.hmc.edu>
Subject: amusing (?) typo

Dear Tina (or should that be TIndianaa?):

Yesterday I received the attached letter from MAA membership with my "new" name, "Arthur BeNew Jerseyyamin". Apparently the NJ in the middle of my last name was replaced by New Jersey.

I laughed it off, but I worry that this could be a bigger problem, and thought you should know.

Art (or should that be Arkansast?)
lambda (aka “anonymous functions”)

def f(x):
    return 2 * x

f = lambda(x): 2 * x

>>> f(5)
10

We’re now in Chapter 3 of the textbook!
Writing map “from scratch”

def dbl(x):
    return 2*x

>>> list(map(dbl, range(5)))
[0, 2, 4, 6, 8]

>>> list(map(lambda x: 2*x, range(5)))
[0, 2, 4, 6, 8]
Writing map “from scratch”

```python
def dbl(x):
    return 2 * x

>>> mymap(dbl, range(5))
[0, 2, 4, 6, 8]

>>> mymap(lambda x: 2*x, range(5))
```

```python
def mymap(myFunc, myList):
    if myList == []: return []
    else:
        return
```

How does Python know that myFunc is a function and myList is a list?
Writing map “from scratch”

```python
def dbl(x):
    return 2 * x

>>> mymap(dbl, range(5))
[0, 2, 4, 6, 8]

>>> mymap(lambda x: 2*x, range(5))
```

```python
def mymap(myFunc, myList):
    if myList == []: return []
    else:
        return [myFunc(myList[0])] + mymap(myFunc, myList[1:])
```

How does Python know that myFunc is a function and myList is a list?
def f1(x):
    return 5 * x + 1

def f2(x):
    return x ** 2

>>> newf = upper(f1, f2)
>>> newf(2)
11

>>> newf(10)
100

def upper(f1, f2):
    return lambda x: \[\text{max} ( \quad ) \]
    I’m returning a function (that takes one input)
def f1(x):
    return 5 * x + 1

def f2(x):
    return x ** 2

>>> newf = upper(f1, f2)
>>> newf(2)
11

>>> newf(10)
100

def upper(f1, f2):
    return lambda x: max(f1(x), f2(x))

I'm returning a function (that takes one input)
def times10(x):
    return 10 * x

def add42(x):
    return x + 42

def ttt(x):
    return 2 ** x

flist = [times10, add42, ttt]

>>> f = se(flist)
>>> f(1)
>>> f(2)
>>> f(5)
>>> f(10)

def se(funcList):
    return lambda x:
        I'm returning a
        function (that takes
        one input)
```python
def times10(x):
    return 10 * x

def add42(x):
    return x + 42

def ttt(x):
    return 2 ** x

flist = [times10, add42, ttt]

>>> f = se(flist)
>>> f(1)
>>> f(2)
>>> f(5)
>>> f(10)

def se(funcList):
    return lambda x: max(map(lambda f: f(x), funcList))

I'm returning a function (that takes one input)

On input x, here's what I return
```
def times10(x):
    return 10 * x

def add42(x):
    return x + 42

def ttt(x):
    return 2 ** x

flist = [times10, add42, ttt]

>>> f = se(flist)
>>> f(1)
>>> f(2)
>>> f(5)
>>> f(10)

def se(funcList):
    return lambda x: max(map(lambda f: f(x), funcList))

I’m returning a function (that takes one input)

On input x, here’s what I return
def times10(x):
    return 10 * x

def add42(x):
    return x + 42

def ttt(x):
    return 2 ** x

flist = [times10, add42, ttt]

>>> f = se(flist)
>>> f(1)
>>> f(2)
>>> f(5)
>>> f(10)

def se(funcList):
    return lambda x: max(map(lambda f: f(x), funcList))

I'm returning a function (that takes one input) On input x, here's what I return
Derivatives

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

>>> def f(x):
    return x ** 2

>>> f = lambda x: x ** 2

>>> fprime = derivative(f, 0.001)

>>> fprime(10)
20.0001

>>> import math

>>> h = derivative(math.cos, 0.001)

>>> h(math.pi)
0.0000...

def derivative(f, h):
    return
Derivatives

\[ f'(x) \approx \frac{f(x + h) - f(x)}{h} \]

```python
>>> def f(x):
    return x ** 2
>>> f = lambda x: x ** 2
>>> fprime = derivative(f, 0.001)
>>> fprime(10)
20.0001
>>> import math
>>> h = derivative(math.cos, 0.001)
>>> h(math.pi)
0.0000...

def derivative(f, h):
    return lambda x: (f(x+h)-f(x))/h
```
\[ f'(x) \approx \frac{f(x+h) - f(x)}{h} \]

```python
>>> def f(X):
    return X ** 4
>>> g = nthDerivative(f, 2, 0.001)
>>> g(10)
1200-ish
```

```python
def nthDerivative(f, n, h):
    if n == 0:
        return
    else:
        return derivative(nthDerivative(f, n-1, h), h)
```

You can use the derivative function that we just wrote!

In your worksheet...
The text is about the $n^{th}$ derivative of a function $f(x)$. The derivative is approximated by:

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

The code snippet demonstrates how to compute the second derivative of a function. The function $f(X)$ is defined as $X^4$. The $n^{th}$ derivative of $f$ is computed using a recursive function `nthDerivative(f, n, h)`. The function is called with $n = 2$ and $h = 0.001$ to compute the second derivative at $x = 10$, which is approximately 1200.
What does filter do?

```python
>>> list(filter(lambda x: x%2 == 0, range(10)))
[0, 2, 4, 6, 8]

>>> list(filter(lambda x: x%3 != 0, range(10)))
[1, 2, 4, 5, 7, 8]
```
Listing Primes...

Objective: Find all primes less than or equal to some given n.

Approach 1: Test 2, 3, ..., n for primality

Approach 2: The Sieve of Eratosthenes...

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...

2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
>>> sieve([2, 3, 4, 5, 6, 7, 8, 9, 10])
[2, 3, 5, 7]

def sieve(L):
    if L == []: return []
    else: return [L[0]] +
        sieve(
)

def primes(n):
    return sieve(range(2, n+1))

This is called a “wrapper” function for sieve.
>>> sieve([2, 3, 4, 5, 6, 7, 8, 9, 10])
[2, 3, 5, 7]

def sieve(L):
    if L == []: return []
    else: return [L[0]] +
        sieve(list(filter(lambda x: x % L[0] != 0, L[1:] )))

def primes(n):
    return sieve(range(2, n+1))

This is called a “wrapper” function for sieve.
Why map, reduce, and filter are preferred to for loops…
import turtle

def draw(repeats, side, angle):
    if repeats == 0:
        return # stop drawing here
    else:
        turtle.forward(side)
        turtle.left(angle)
        draw(repeats-1, side, angle)
    return # optional to put return here
The Sierpinski Fractal Triangle!
The 200 refers to the length of one side of the triangle!
import turtle

def fractal(length, level):
    if level == 0:
        turtle.forward(length)
        turtle.left(120)
        turtle.forward(length)
        turtle.left(120)
        turtle.forward(length)
        turtle.left(120)
        return
    else:
        fractal(length/2, level-1)  # First recursive call
        turtle.forward(length/2)
        fractal(length/2, level-1)  # Second recursive call
        turtle.backward(length/2)
        turtle.left(60)
        fractal(length/2, level-1)  # Third recursive call
        turtle.left(60)
        turtle.back(length/2)
        turtle.right(60)
        return

# Fill this in... (6 lines of code are missing)
import turtle

def fractal(length, level):
    if level == 0:
        turtle.forward(length)
        turtle.left(120)
        turtle.forward(length)
        turtle.left(120)
        turtle.forward(length)
        turtle.left(120)
        return
    else:
        fractal(length/2, level-1)  # First recursive call
        turtle.forward(length/2)
        fractal(length/2, level-1)  # Second recursive call
        turtle.backward(length/2)
        turtle.left(60)
        turtle.forward(length/2)
        turtle.right(60)
        fractal(length/2, level-1)   # Third recursive call
        turtle.left(60)
        turtle.back(length/2)
        turtle.right(60)
        return
This week’s lab...

svtree(100, 6)
Snowflakes!
Snowflakes!