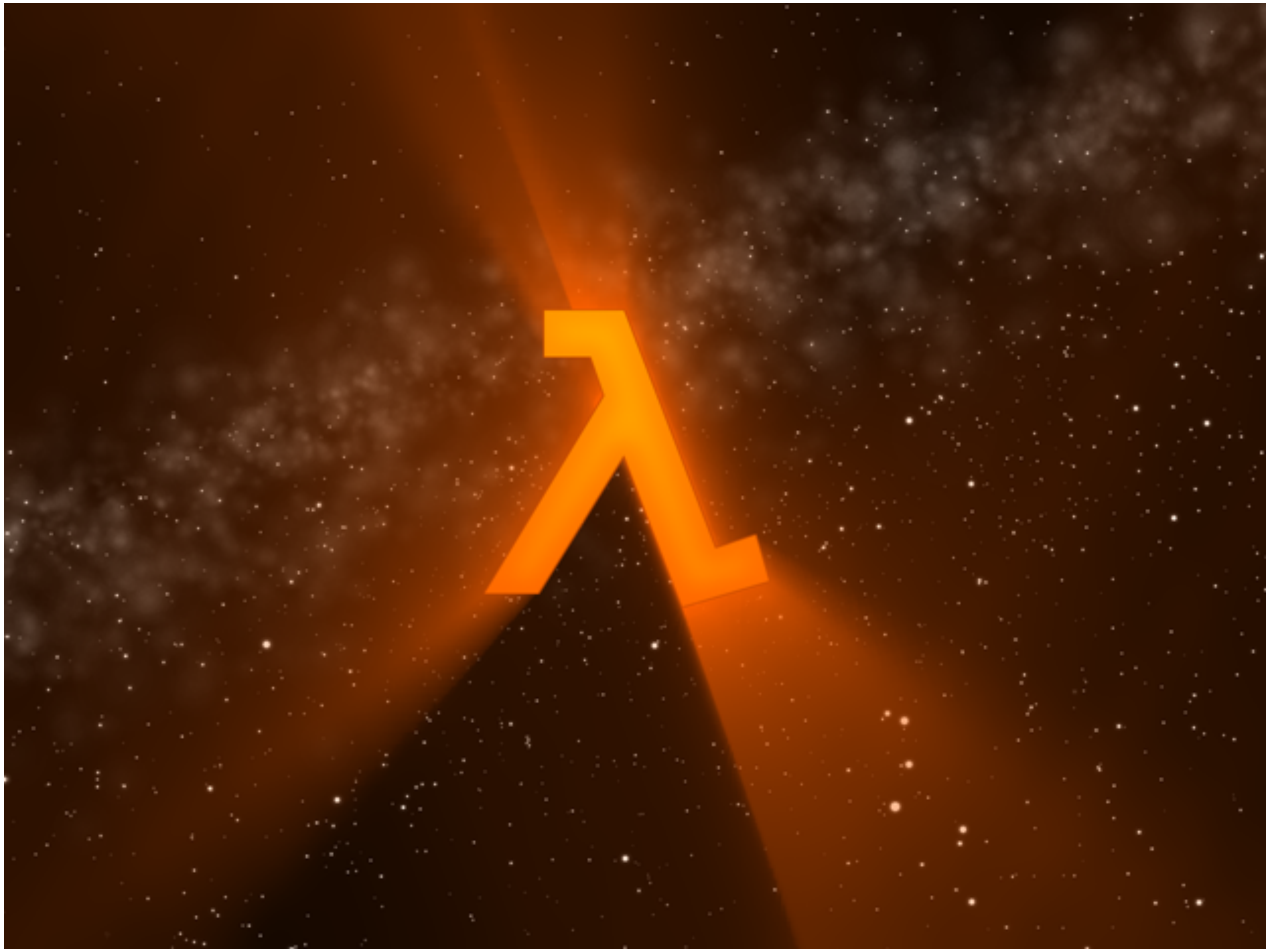
Higher Order Functions

Lambda Calculus



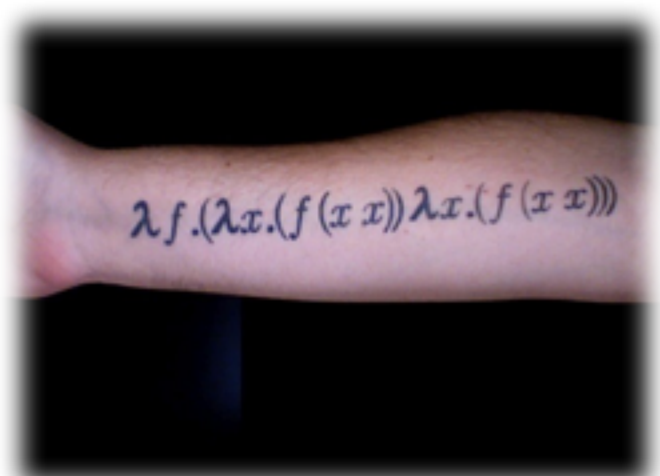
Lambda Calculus

$$(\lambda y.\ x(yz))\ (ab)$$

head

body:
an expression

another expression,
after the function



MIT

Knights of the Lambda Calculus

From Wikipedia, the free encyclopedia

The **Knights of the Lambda Calculus** is a semi-fictional organization of expert Lisp and Scheme hackers. The name refers to the lambda calculus, a mathematical formalism invented by Alonzo Church, with which Lisp is intimately connected, and references the Knights Templar.

There is no actual organization that goes by the name *Knights of the Lambda Calculus*; it mostly only exists as a hacker c MIT. For example, in the Structure and Interp e of the lecturers presents the audience with the b group. However, according to the Jargon File, a "v ons with Knights insignia on them, and some pe ts.[1]

In popular cu

A group that evolve astern Calculus, make a major appearance in the a MIT professors and other American computer . At one point in the anime, Lain is seen with co o be Lisp.[2]

The **Knights of the Lambda Calculus'** recursive emblem celebrates LISP's theoretical foundation, the lambda calculus. Y in the emblem refers to the fixed-point combinator and the reappearance of the picture in itself refers to recursion.

Print vs Return

```
def foo_print3():
    print(3)

def foo_return3():
    return 3
```

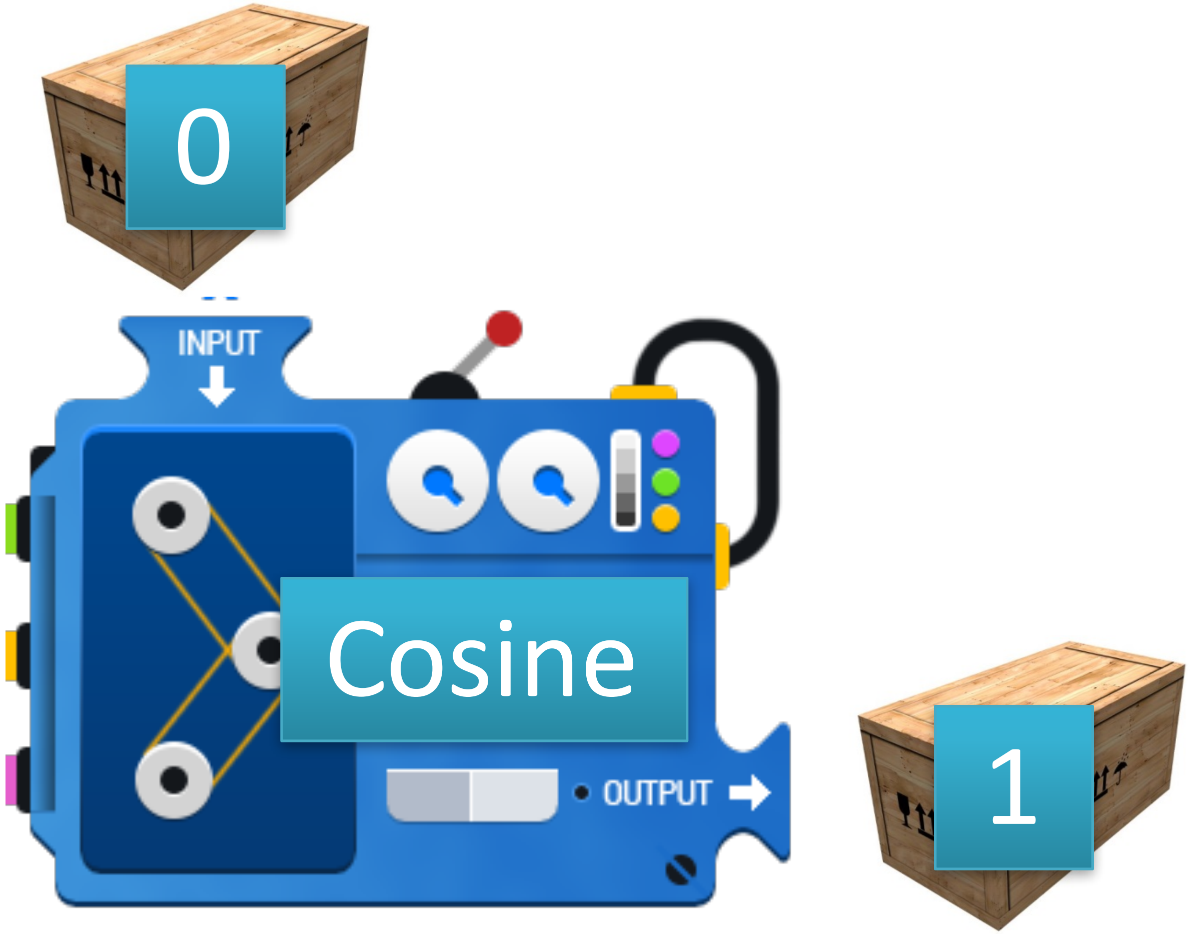
By the print function

```
>>> foo_print3()
3
>>> foo_return3()
3
>>>
```

IDLE’s echo

Function

- "Cosine" is a function
  - Input 0
  - Output 1
  - x = cos(0)
  - x = 1

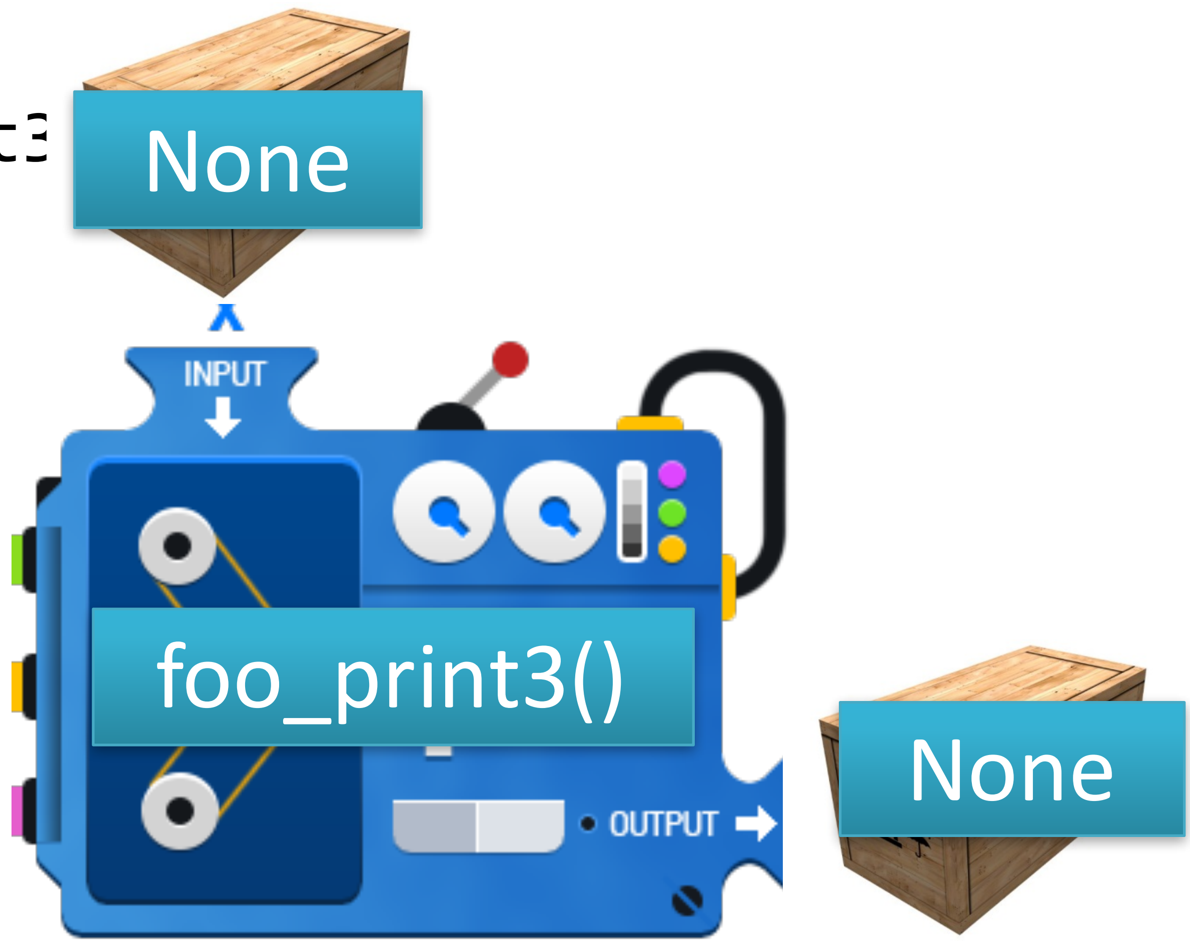


Function

- "`foo_print3()`" is a function
  - Input nothing
  - No output

```
y = foo_print3
```

None



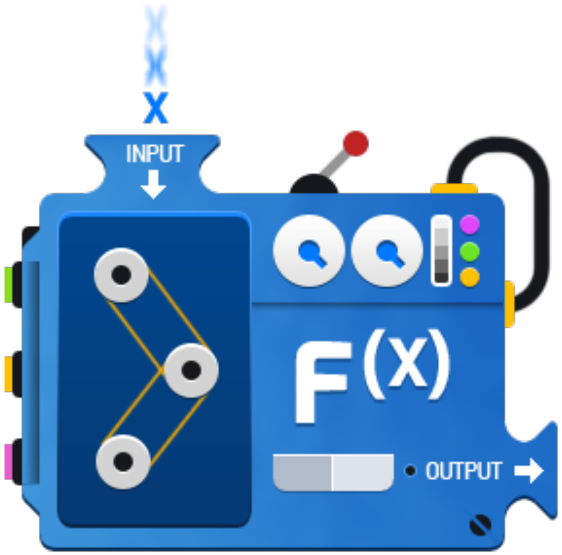
In general, we called all these "functions"

But for a function that "returns" nothing. Sometime we call it a "procedure"

If we think that

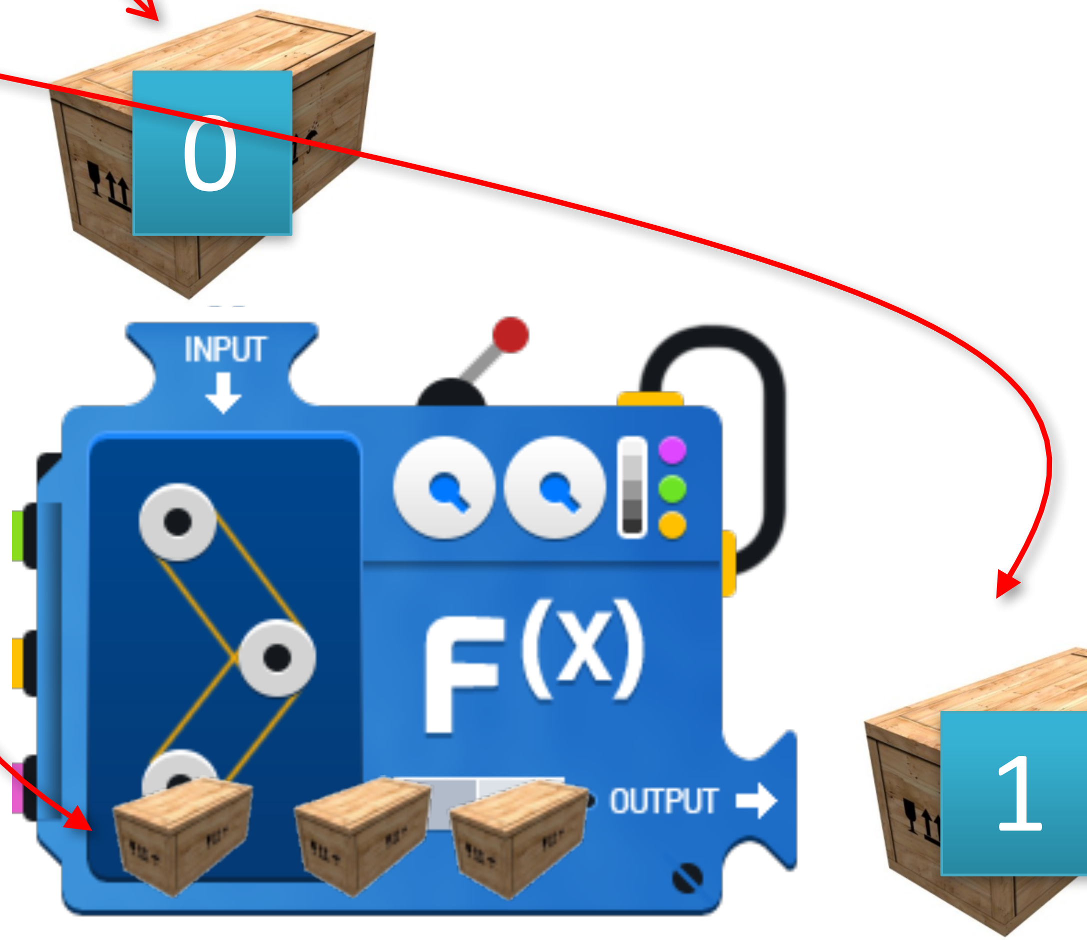
- A variable is like a crate
  - Storing a value inside

- A function is like a machine
  - Take a crate as input
  - And produce another crate
    - (For output)



Function and Variable Illustrations

- Parameters
- Return values
- Local variables



Remember how we define a function?

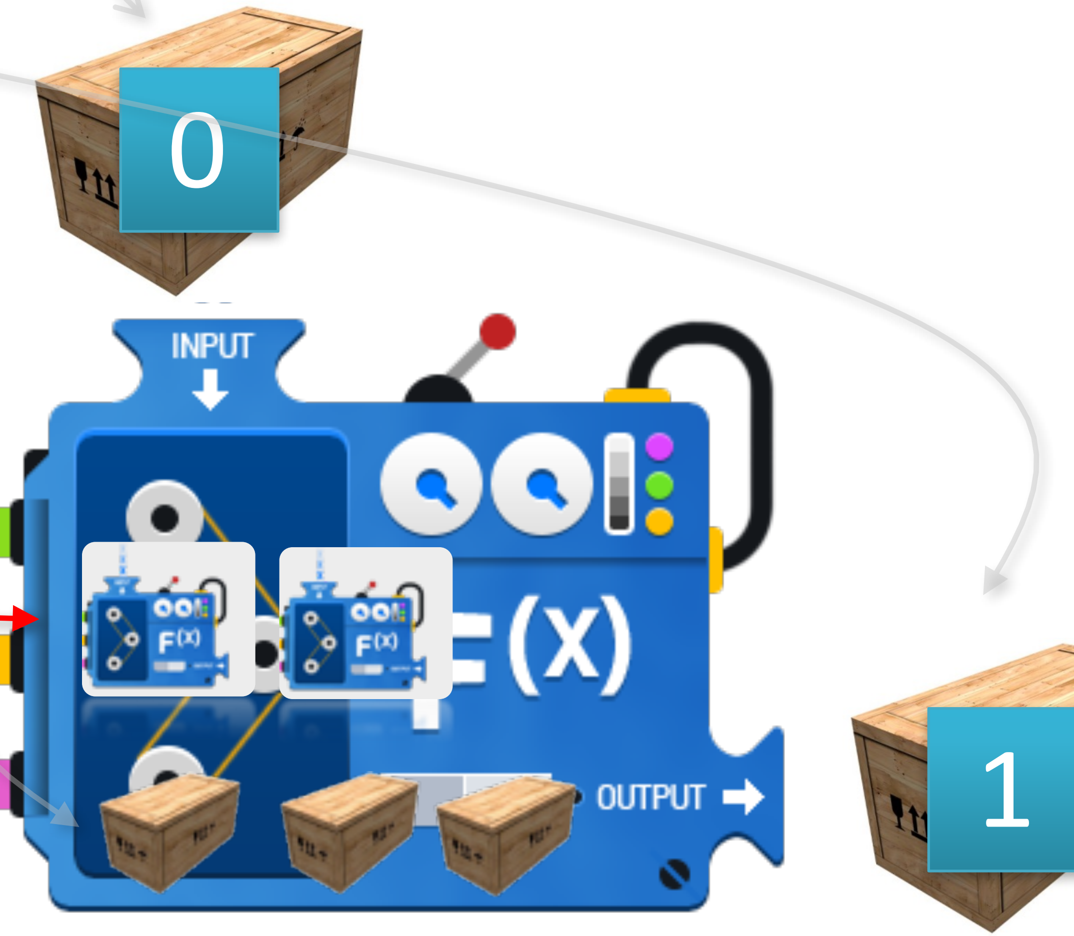
```
from math import sqrt

def distance(x1,y1,x2,y2):
    return sqrt(square(x1-x2)+square(y1-y2))

def square(x):
    return x*x
```

- But we can actually write something like this:

```
from math import sqrt

def distance(x1,y1,x2,y2):

    def square(x):
        return x*x

    return sqrt(square(x1-x2)+square(y1-y2))
```

Function and Variable Illustrations

- Parameters
- Return values
- Local variables
- **Local functions**



Remember how we define a function?

- Almost the same except
  - Outside the function **distance**, you cannot use the function **square**
  - Just like local variables

```
from math import sqrt

def distance(x1,y1,x2,y2):

    def square(x):
        return x*x

    return sqrt(square(x1-x2)+square(y1-y2))
```

Remember how we define a function?

```
from math import sqrt

def distance(x1,y1,x2,y2):
    return sqrt(square(x1-x2)+square(y1-y2))

def square(x):
    return x*x
```

Global Function

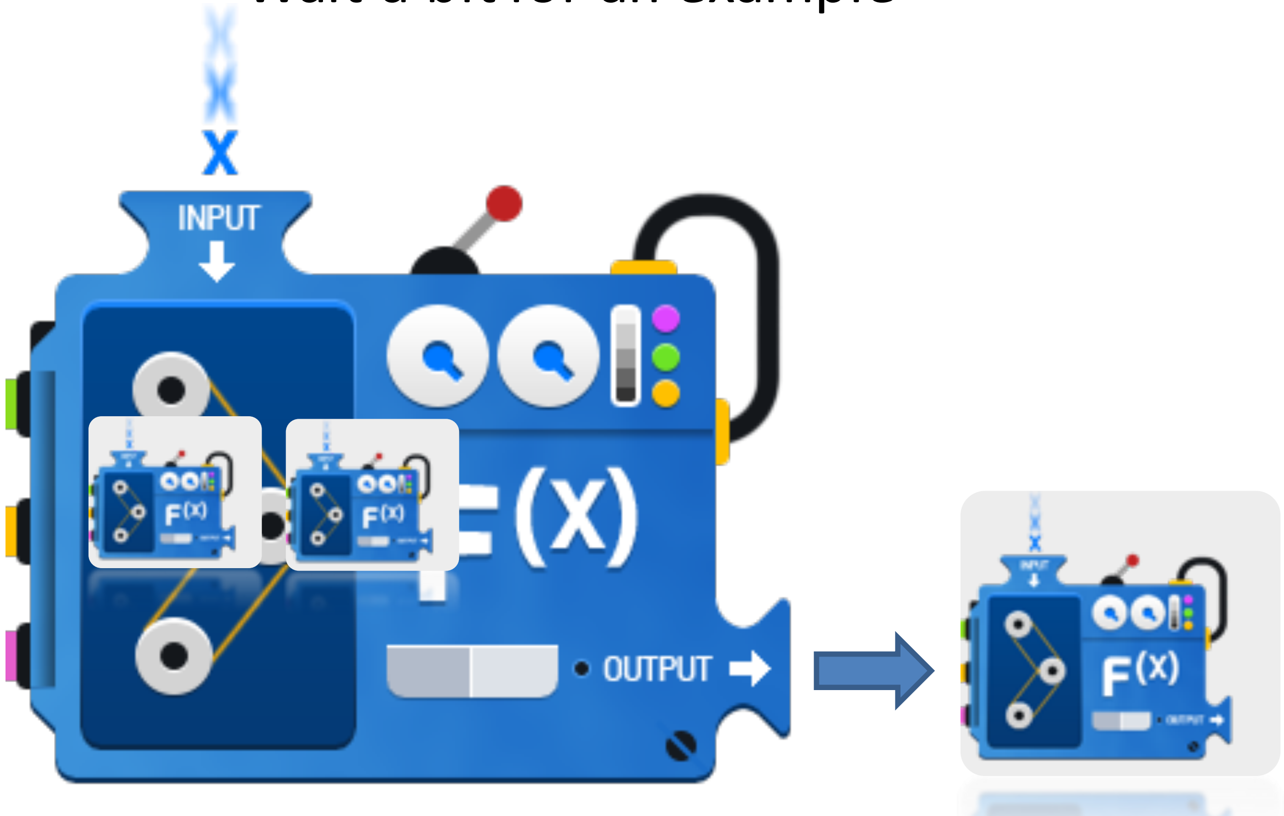
- But we can actually write like this:

```
from math import sqrt

def distance(x1,y1,x2,y2):

    def square(x):
        return x*x

    return sqrt(square(x1-x2)+square(y1-y2))
```

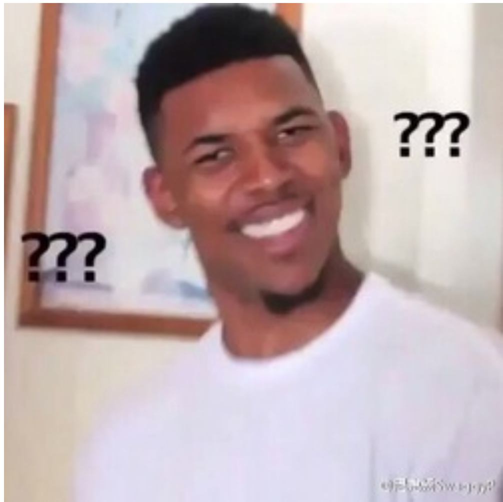
Local Function

So we can output a function!

- You can define some local functions and then return it
    - Wait a bit for an example

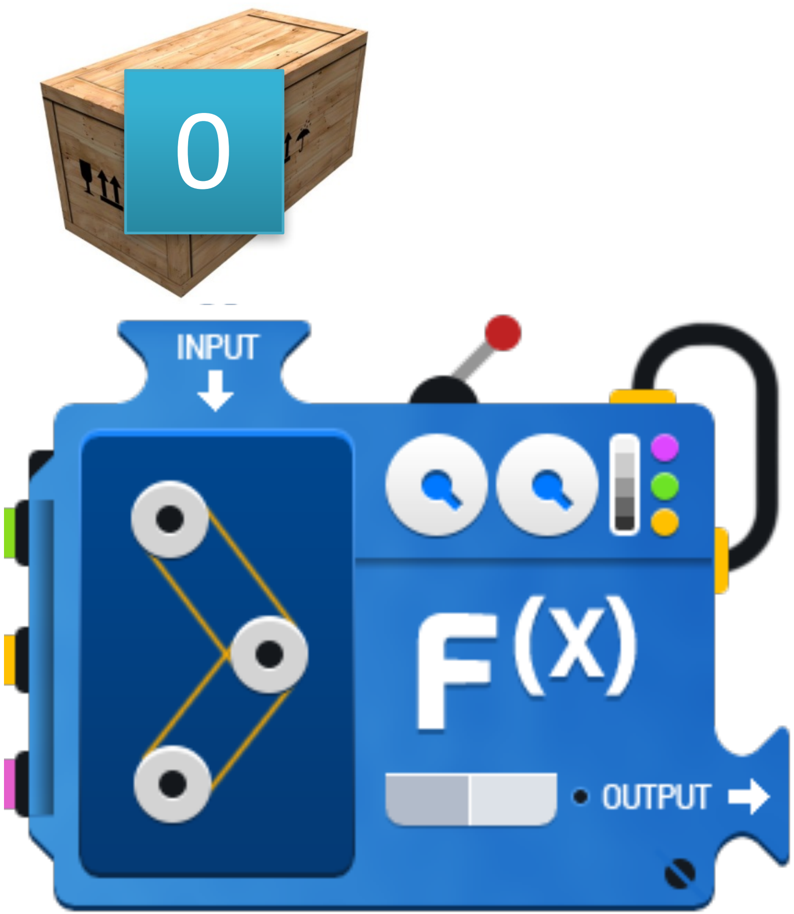


Treat a Function like a Variable

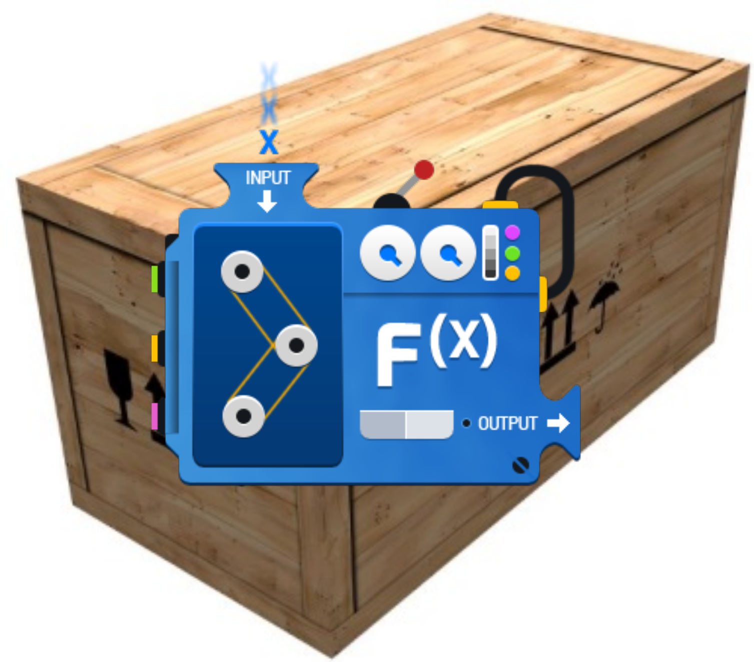


Function

- We can put a variable into a function



- Can we put a function into a variable?



“Callability”

- Normal valuables are NOT callable

```
>>> x = 1
>>> x()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    x()
TypeError: 'int' object is not callable
```

- A function is callable

```
>>> def f():
        print("Hello")


>>> f()
Hello
```

Assignments

- Normal variables can store values

```
>>> x = 1
>>> y = x
>>> x = 2
```

- Can a variable store a function?!

```
>>> def f():
        print("Hello")


>>> x = f
>>> x()
Hello
```

- Can!!!!!!

Assignments

- The function f is stored in the variable x
  - So x IS a function, same as f

```
>>> def f():
        print("Hello")


>>> x = f
>>> x()
Hello
```

See the difference

```
>>> def f2():
        return 999
```

With '()'

```
>>> x = f2()
>>> print(x)
999
>>> type(x)
<class 'int'>
```

Without '()'

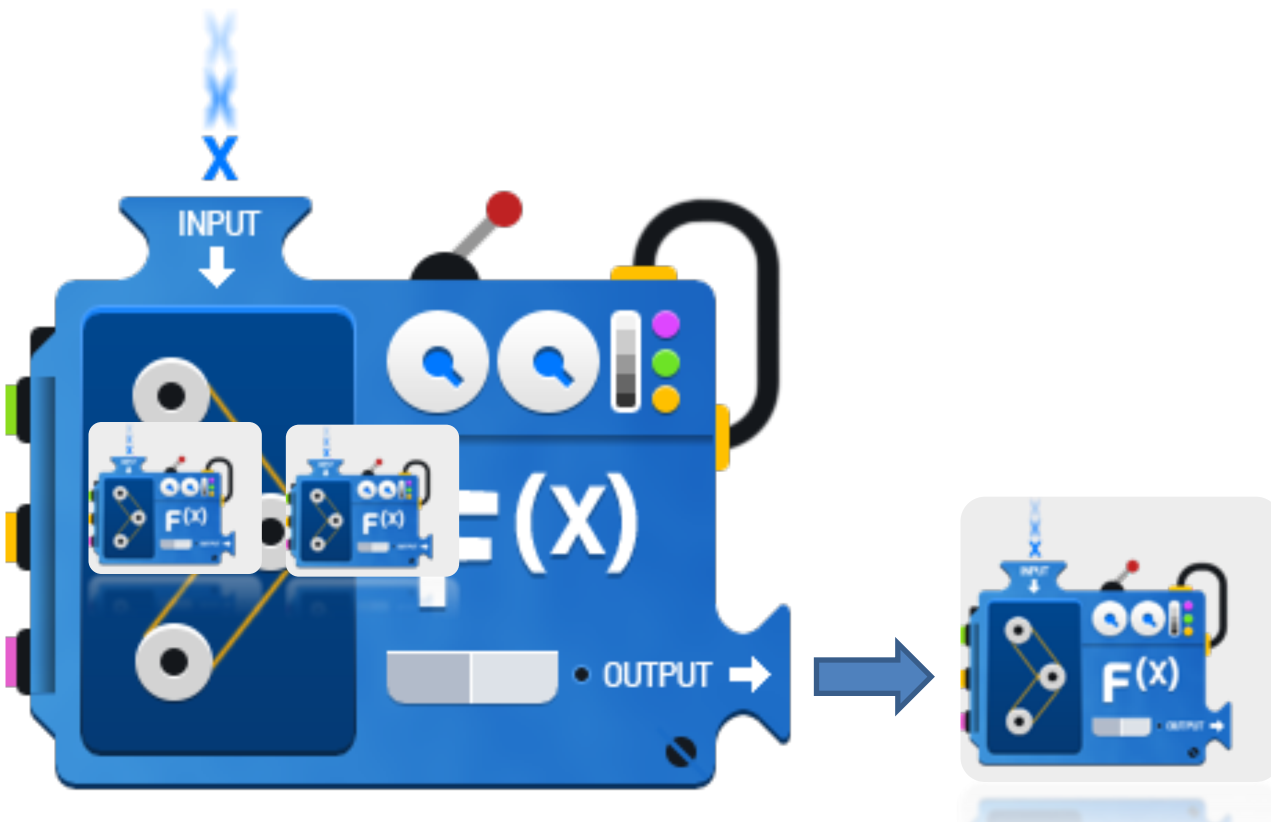
```
>>> y = f2
>>> print(y)
<function f2 at 0x0000007ACE8C5A60>
>>> type(y)
<class 'function'>
```

values

types

So we can output a function!

- You can define some local functions and then return it



A Function that Produces Another Function

```
>>> def make_yell_n(n):
        def yell_n(s):
                return s * n
        return yell_n

>>> yell_4 = make_yell_n(4)
>>> yell_4("Hello!")
'Hello!Hello!Hello!Hello!'
>>> yell_2 = make_yell_n(2)
>>> yell_2("Yay!!!")
'Yay!!!Yay!!!'
```

Returning a function

Store the returned function into "yell_4"

Use the function yell_4

Make another function yell_2

Functions can be stored in variables

```
>>> from math import cos, sin, tan
>>> f_1 = cos
>>> f_1(0)
1.0
>>> print(f_1)
<built-in function cos>
>>> def f():
        print("Hello")


>>> print(f)
<function f at 0x000000F9F93F4950>
```

Equivalent to cos(0)

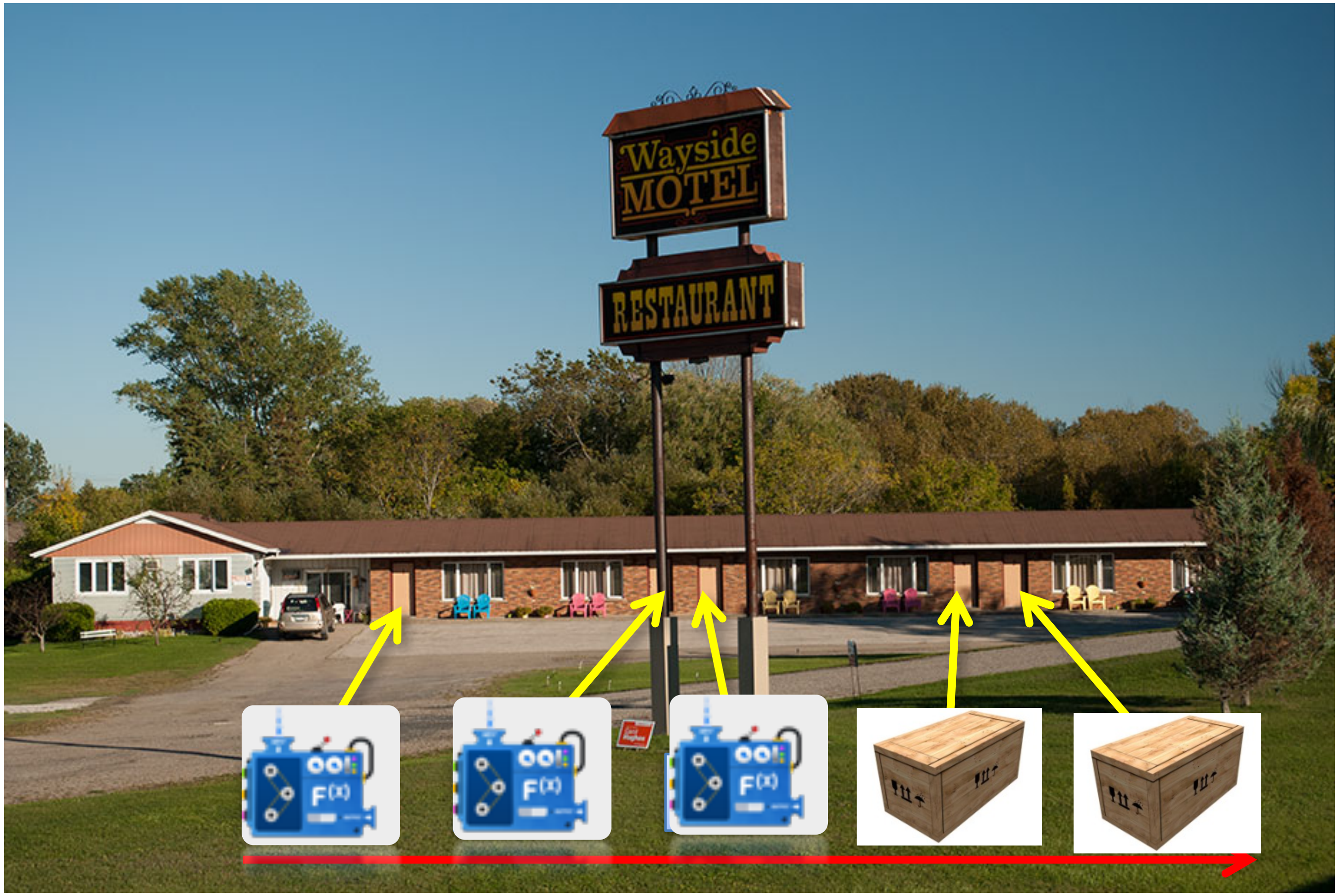
The type is "function"

- Can even store functions into a list, tuple, etc.

```
>>> my_collection = [cos, sin, tan, f, len]
>>> my_collection[4]([1,2,3])
3
```

Equivalent to `len([1,2,3])`

Motel in US



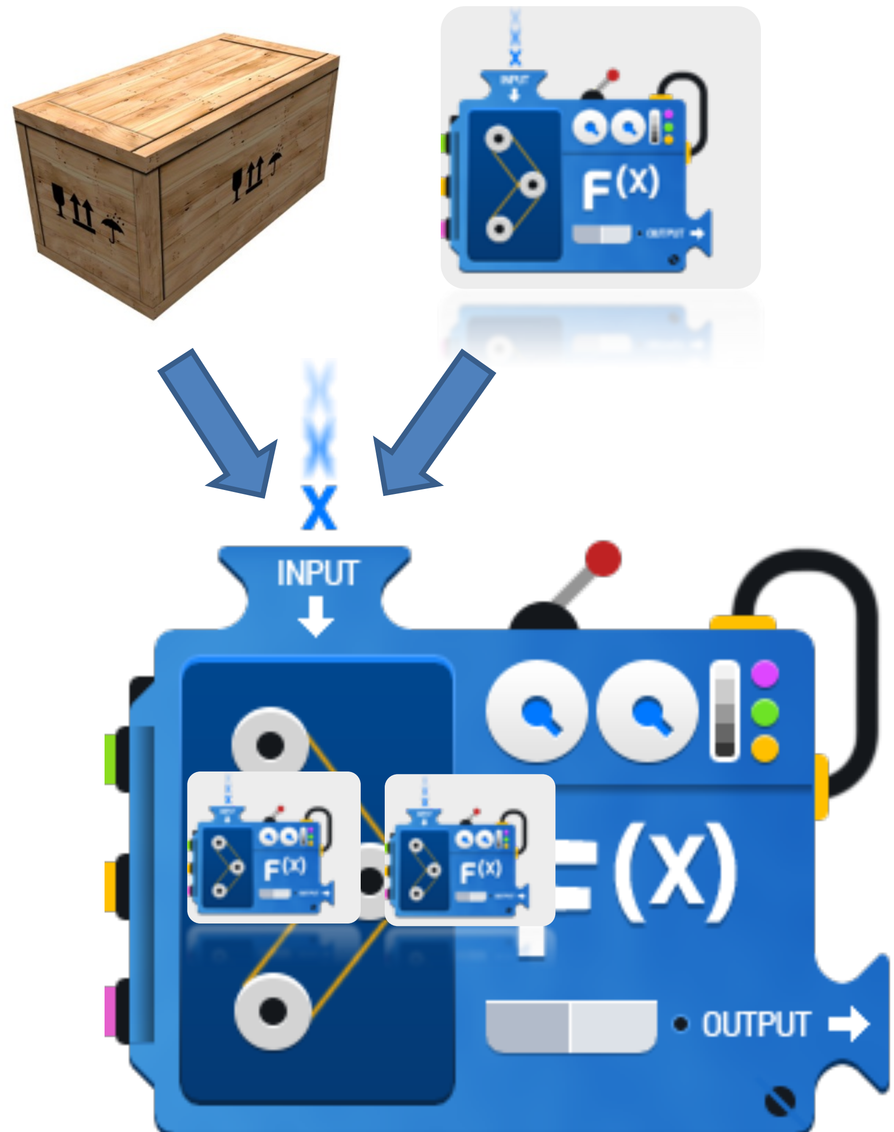
One Dimension

Hotel

My room is 02-05

And we can input a function

- We can input variables into a function
- In the same manner, we can input functions into a function



Function Composition

- In math, we can do something like

$$\log(\sin(x))$$

```
>>> def f():
        print("Hello")


>>> def do_twice(x):
        x()
        x()


>>> do_twice(f)
Hello
Hello
```

Equivalent to

```
>>> def do_twice(x):
        f()
        f()
```

Mix and Match

```
>>> def add1to(x):
        return x + 1

>>> def square(x):
        return x * x

>>> def do_3_times(f,n):
        return f(f(f(n)))

>>> do_3_times(add1to,2)
5
>>> do_3_times(square,2)
256
```

A function

A variable (can be a function too!)

Equivalent to

```
>>> def do_3_times(f,n):
        add1to(add1to(add1to(2)))
```

The Evil Lambda

The Big Evil Boss "lambda"

```
>>> def add1(x):
        return x+1

>>> add1(9)
10
>>> func = lambda x: x +1
>>> func(9)
10
```

Equivalent!!!

- difference:
  – lambda does not need a 'return'

The “Powerful” Lambda

- Apparently nothing new

```
>>> def add1(x):
        return x+1

>>> add1(9)
10
>>> func = lambda x: x +1
>>> func(9)
10
```

- But useful if you want to return a function as a result in a function

Agar Agar (Anyhow) Derivative

- We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- $\frac{d \sin x}{dx} = \cos x$
- $\frac{d\ (x^3+3x-1)}{dx} = 3x^2 + 3$

Agar Agar (Anyhow) Derivative

- We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- $\frac{d \sin x}{dx} \Rightarrow$ Derivatives $\Rightarrow \cos x$
- $\frac{d\,(x^3+3x-1)}{dx} \Rightarrow$ Derivatives $\Rightarrow 3x^2 + 3$

Agar Agar (Anyhow) Derivative

- We know that, given a function f, the derivative of f is

$$\frac{df(x)}{dx} = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

- But, if we have a very small number dx

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

Agar Agar (Anyhow) Derivative

$$\frac{d \sin x}{dx} = \cos x$$

```
>>> def deriv(f):
        dx = 0.000000001
        return lambda x: (f(x+dx)-f(x))/dx
```

Take in a function, returning another function

```
>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

- But, if we have a very small number dx

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

Agar Agar (Anyhow) Derivative

$$\frac{d\,(x^3+3x-1)}{dx} = 3x^2+3$$

```
>>> def f(x):
        return x**3+3*x-1

>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3*x**2 +3
246
```

- But, if we have very small number dx

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

Agar Agar (Anyhow) Derivative

```
>>> def deriv(f):
        dx = 0.000000001
        return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
>>> def f(x):
        return x**3+3*x-1

>>> deriv(f)(9)
246.00001324870388
>>> x = 9
>>> 3*x**2 +3
246
```

Take in a function, returning another function

Agar Agar (Anyhow) Derivative

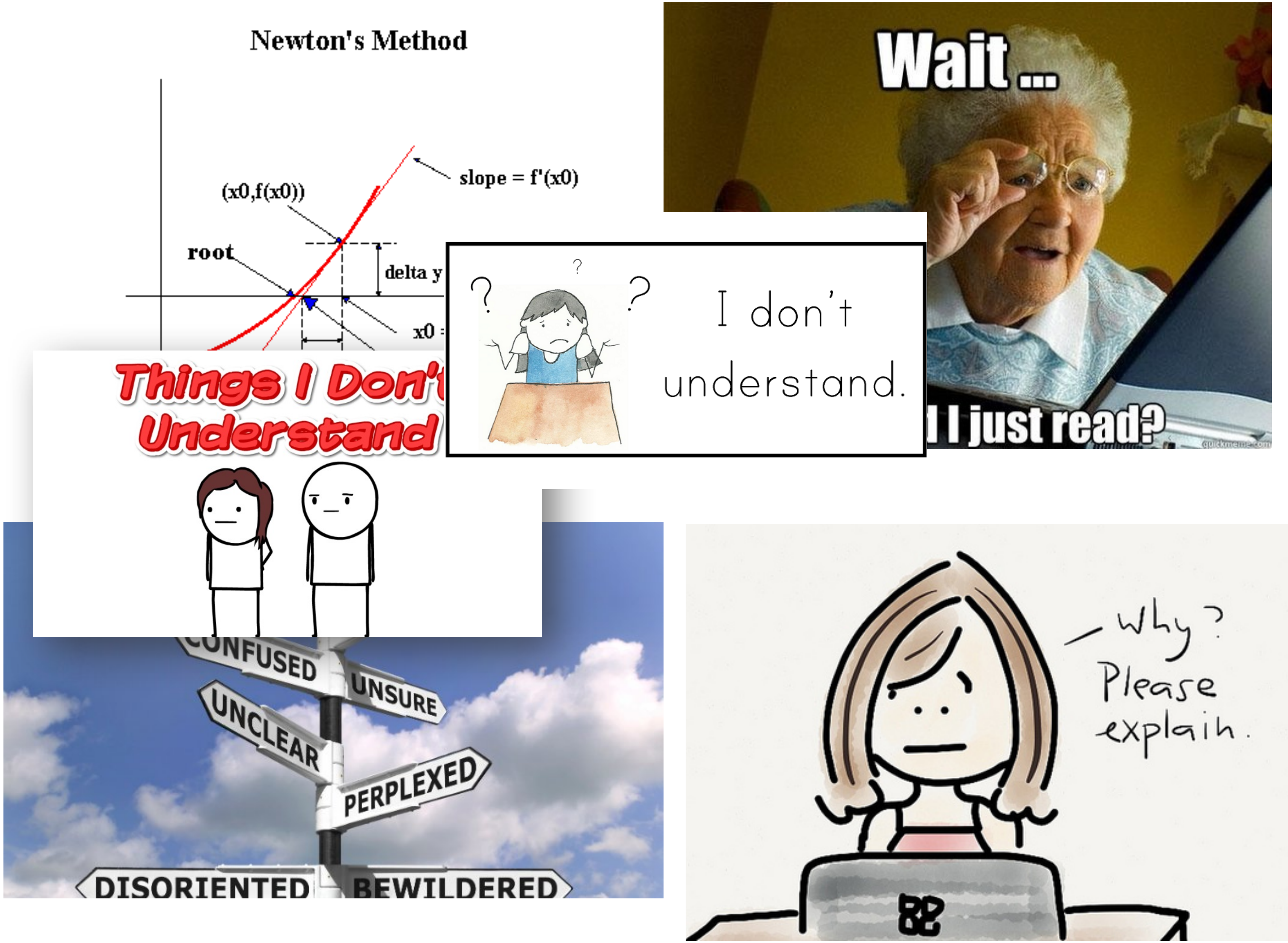
```
>>> def deriv(f):
        dx = 0.000000001
        return lambda x: (f(x+dx)-f(x))/dx

>>> cos(0.123)
0.9924450321351935
>>> func = deriv(sin)
>>> func(0.123)
0.9924450428133723
```

- $\frac{d \sin x}{dx}$ → Derivatives → $\cos x$
- $\frac{d\,(x^3+3x-1)}{dx}$ → Derivatives → $3x^2 + 3$

Example: Newton's method

- To compute root of function $g(x)$, i.e. find x such that $g(x) = 0$

1. Anyhow assume the answer x = something
2. If $g(x) \approx 0$ then stop: answer is x, return x
3. Otherwise
    - $x = x - g(x)/\texttt{deriv}(x)$
4. Go to step 2



Newton's Method
slope = f'(x0)
(x0,f(x0))
root
delta y
x0 =
Wait ...
I just read?
I don't understand.
Things I Don't
Understand
CONFUSED
UNSURE
UNCLEAR
PERPLEXED
DISORIENTED
BEWILDERED
why?
Please explain.

Example: Newton's method

```
def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

1. Anyhow assume the answer x = something
2. If $g(x) \approx 0$ then stop: answer is x, return x
3. Otherwise
   - $x = x - g(x)/\texttt{deriv}(x)$
4. Go to step 2

Example: Newton's method

- Together with the function `deriv()`

```
def deriv(f):
    dx = 0.000000001
    return lambda x: (f(x+dx)-f(x))/dx

def newtonM(g):
    x = 999 #doesnt matter
    err = 0.0000000001
    while(abs(g(x))>err):
        x = x - g(x)/deriv(g)(x)
    return x
```

Example: Newton's method

- Example: Square root of a number A
  - It's equivalent to solve the equation: $x^2 - A = 0$

```
>>> def my_own_sqrt(A):
        return newtonM(lambda x:x*x-A)

>>> x = my_own_sqrt(10)
>>> x * x
9.999999999999998
```

Example: Newton's method

- Example: Compute $\log_{10}$ (A)
  - Solve the equation: $10^x - A = 0$

```
>>> def my_own_log10(N):
        return newtonM(lambda x: 10**x - N)

>>> my_own_log10(100)
2.000000000000013
>>> x = my_own_log10(234)
>>> 10 ** x
234.0000000000892
```

What We Learn Today

- Higher Order Functions
  - We can store, pass, output functions as other variables
  - An extremely powerful feature of Python
    - Functional Programming