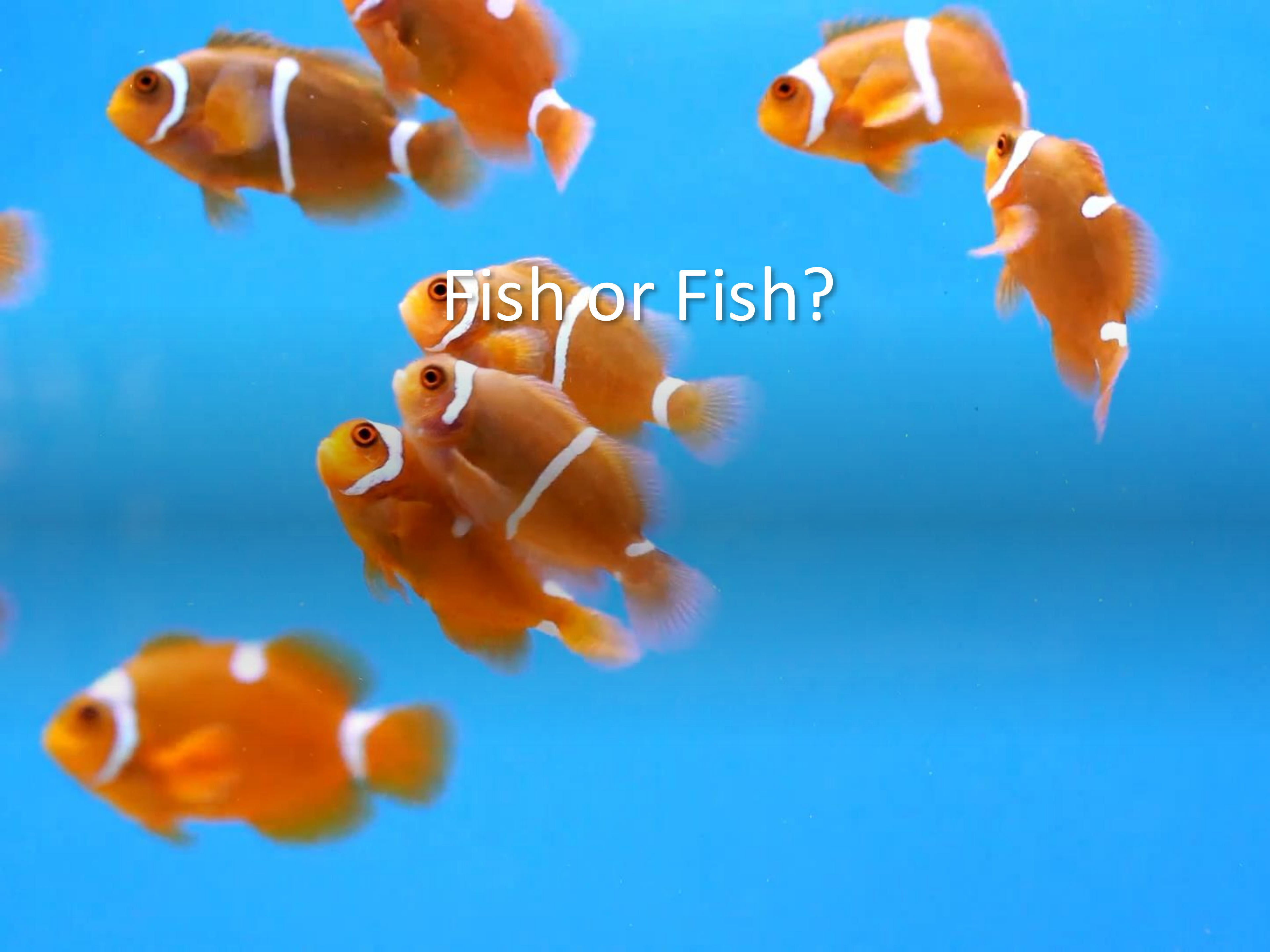
Introduction

- x = 1
- What is x + 2?

- What is 1 + 2?

Fish or Fish?



The Old Proverbs Say

- 'Give a man a fish and he will eat for a day. Teach a man how to fish and you feed him for a lifetime.'
- Chinese:
  - 授人以鱼不如授人以渔

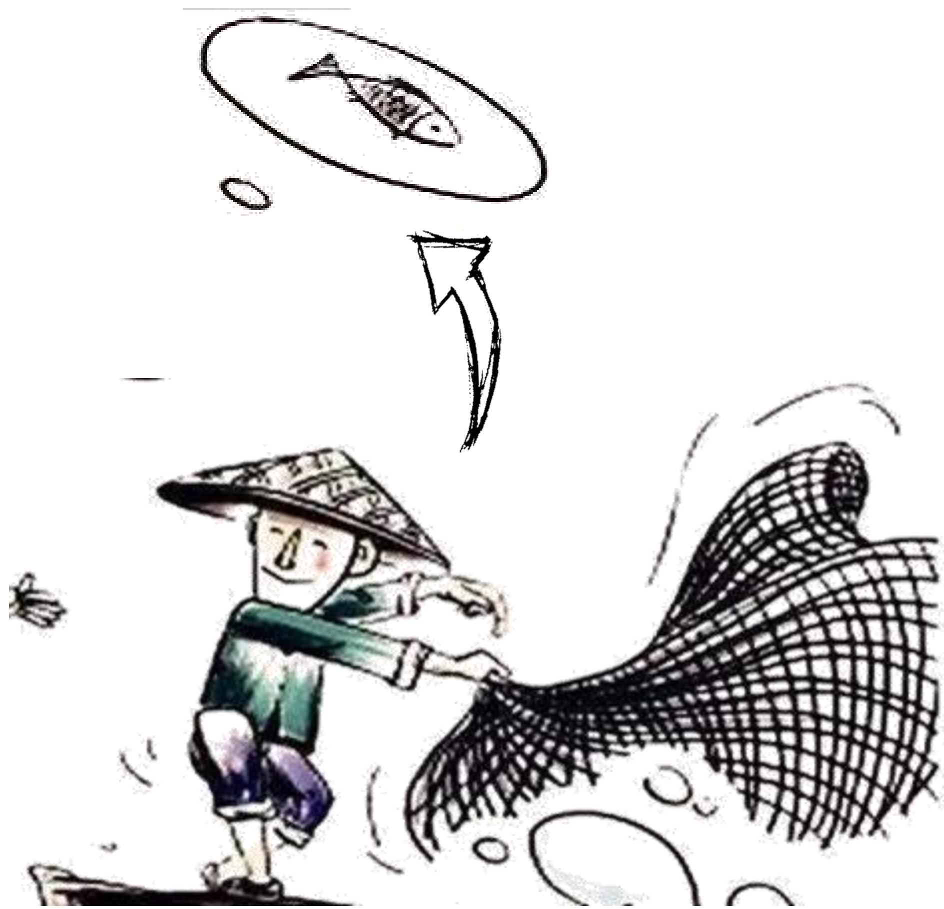
A method

An object

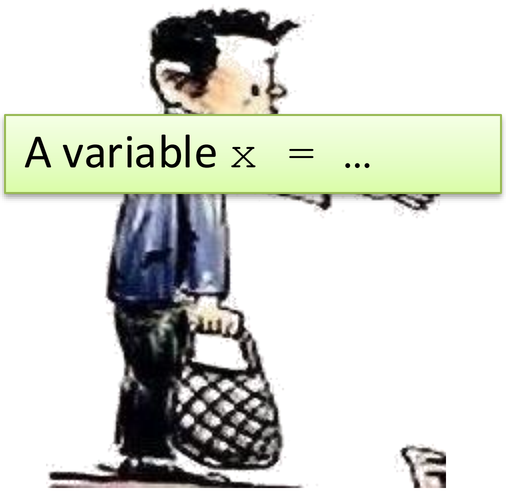
```python
def func():
    return 'fish'
```

```
x = func()
```

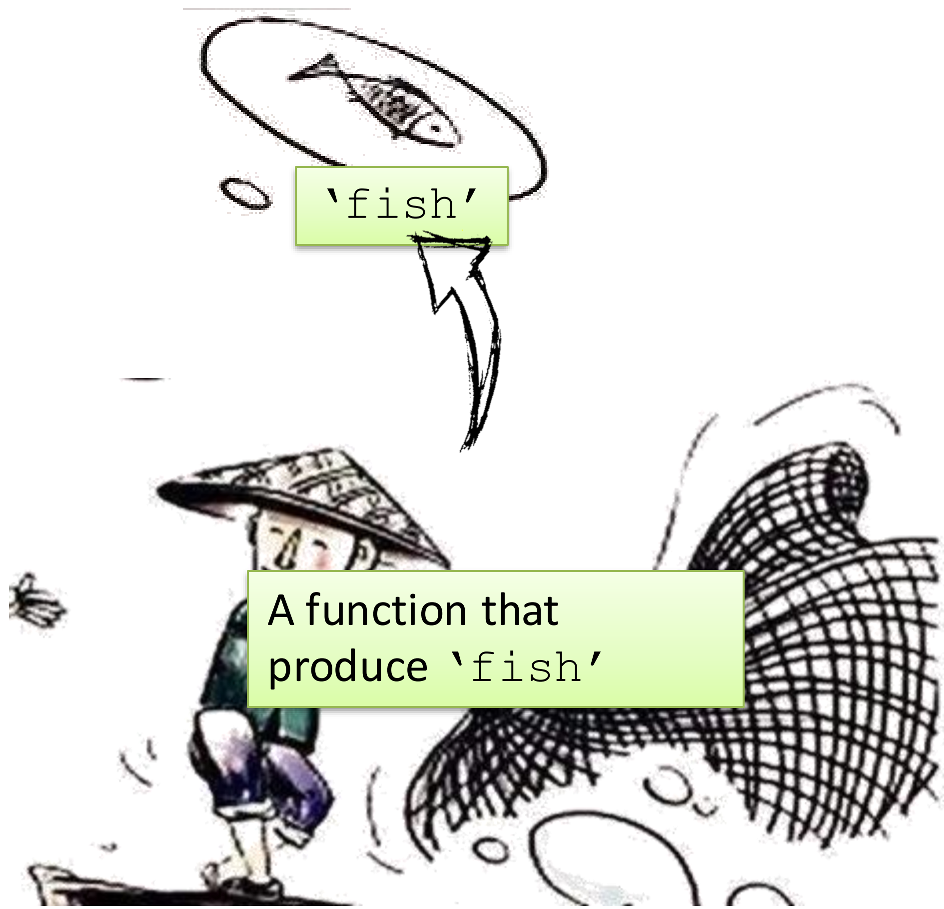
Mr. X

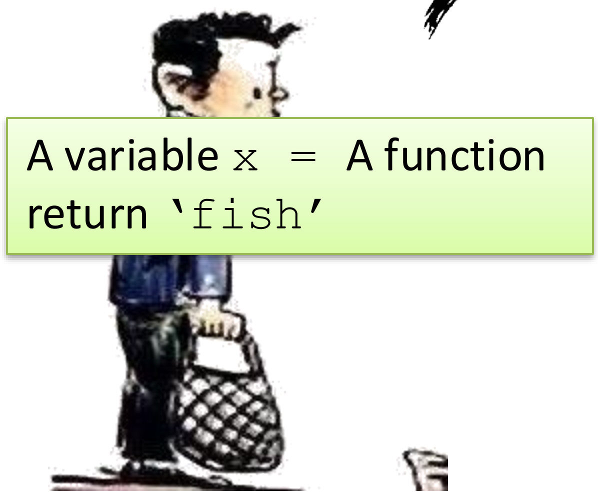


`x = func()`



Mr. X





Mr. X

Higher Order Functions

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))
```

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
```

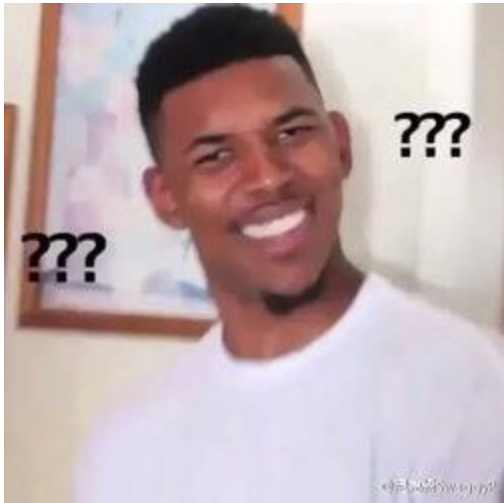
Global Function

- 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

Treat a Function like a Variable



“Callability”

- Normal variables 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'>
```

values

Without '()'

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

types

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])

Function Composition

- In math, we can do something like

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

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

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

Equivalent to

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

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

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
```

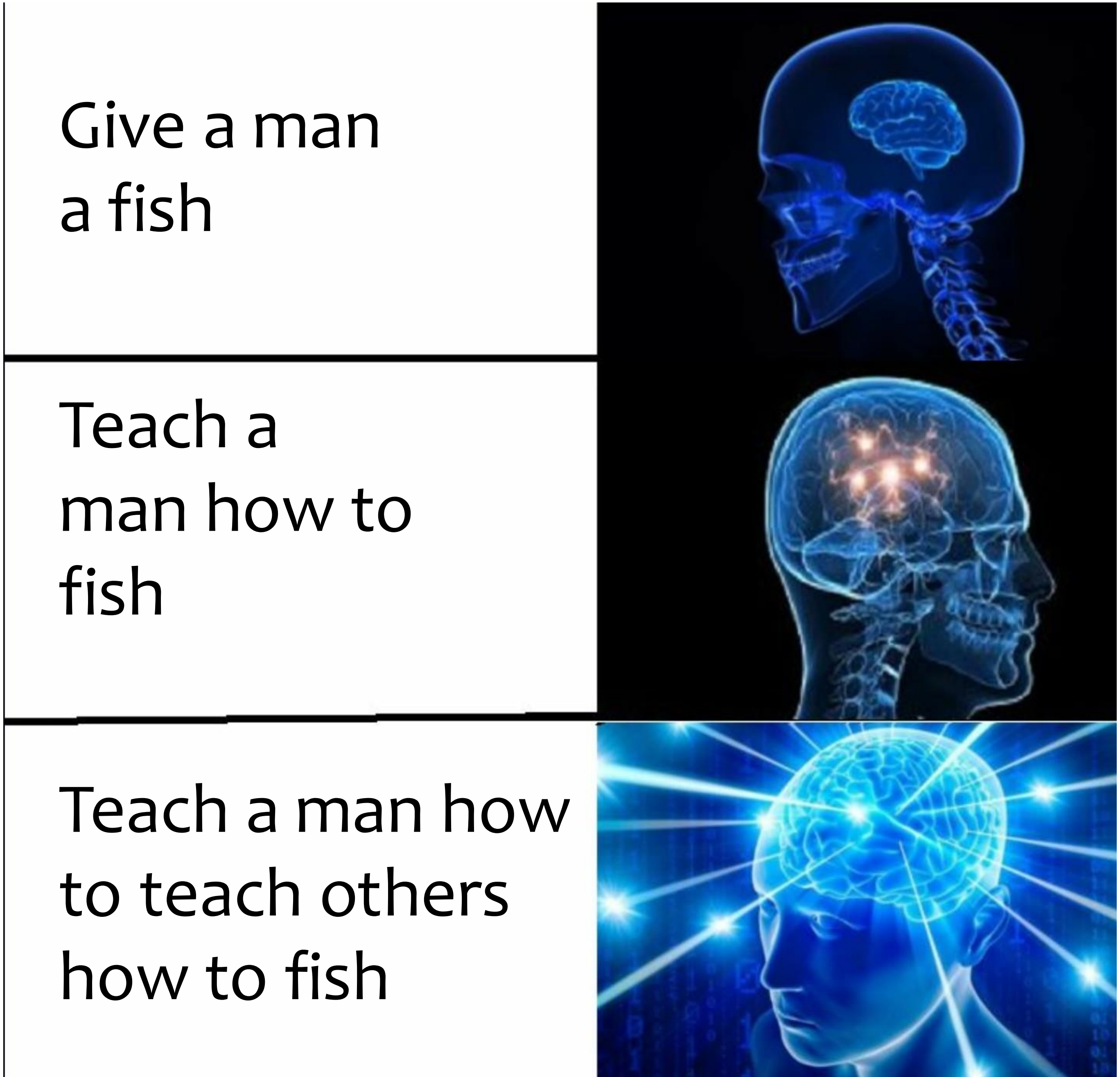
A function

A variable (can be a function too!)

Equivalent to

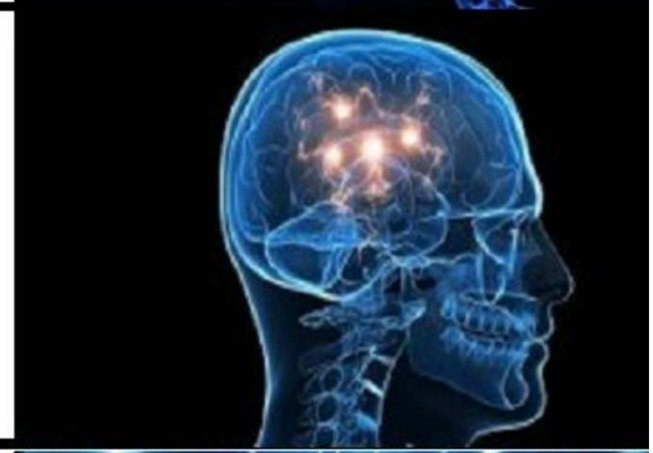
```
>>> def do_3_times(f,n):
        add1to(add1to(add1to(2)))
```

Mr. X

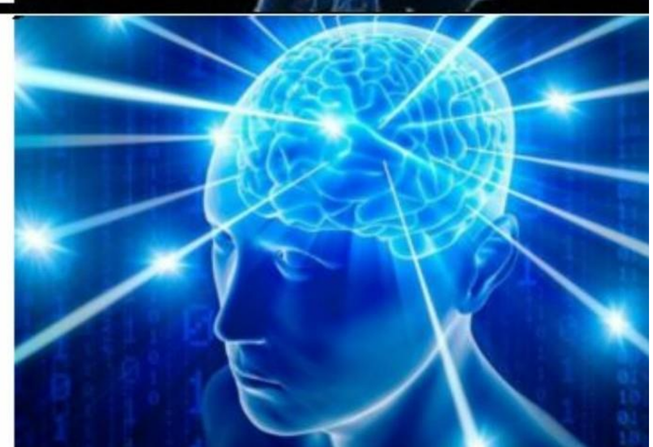

Give a man a fish
Teach a man how to fish
Teach a man how to teach others how to fish

A variable x =
an object

A variable x =
a function
that returns
an object



A variable x = a
function that returns
a function that
returns an object



lambda

Integers

```
x = 5
```

- x is the *name* of the variable that contains an integer 5
- 5 is an integer object
- The object 5 can be used without the name x
- e.g.

```
print(5+1)
```

Functions

```
def square(x):
    return x**2
```

- "square" is the name of the *function* that can square an input
- The *function* to square is an independent object
- The *function* can be used without the name square
- e.g.

```
print((lambda x:x**2)(2))
```

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

The “Powerful” Lambda

- Apparently nothing nev

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

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

```
>>> def aFunctionAddN(n):
    return lambda x: x + n

>>> f1 = aFunctionAddN(10)
>>> f1(1)
11
>>> f1(2)
12
>>> f2 = aFunctionAddN(99)
>>> f2(1)
100
>>> f2(f1(3))
112
```

Why do we want that?!

- Because we can create/output an object!
- e.g. for integers

```
def times_two(x):
    return x*2
```

- Same for function

```
def make_a_function(x):
    return ***some function***
```

Create Functions to Power a Number

```
def make_power_func(n):
    return lambda x:x**n

square = make_power_func(2)
cube = make_power_func(3)
square_root = make_power_func(0.5)

>>> print(square(3))
9
>>> print(cube(2))
8
>>> print(square_root(16))
4.0
```

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 very small number dx

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

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 $\Rightarrow$ $3x^2 + 3$

The Derivative is a function!

- Its input is a function
  - And output another function

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

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
```

Take in a function, returning another function

- 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 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 ➡ $3x^2 + 3$

Application Example of `deriv()`

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) ≈ 0 then stop: answer is x, return x
3. Otherwise
    - x = x – g(x)/deriv(x)
4. Go to step 2

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) ≈ 0 then stop: answer is x, return x
3. Otherwise
   - x = x – g(x)/deriv(x)
4. Go to step 2

Example: Newton's method

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

```
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
```

```
>>> 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
```

You can solve any equation!

…. that Newton Method can solve.