Higher Order Functions in Python

Remark

- Techniques from F#.

About speaker

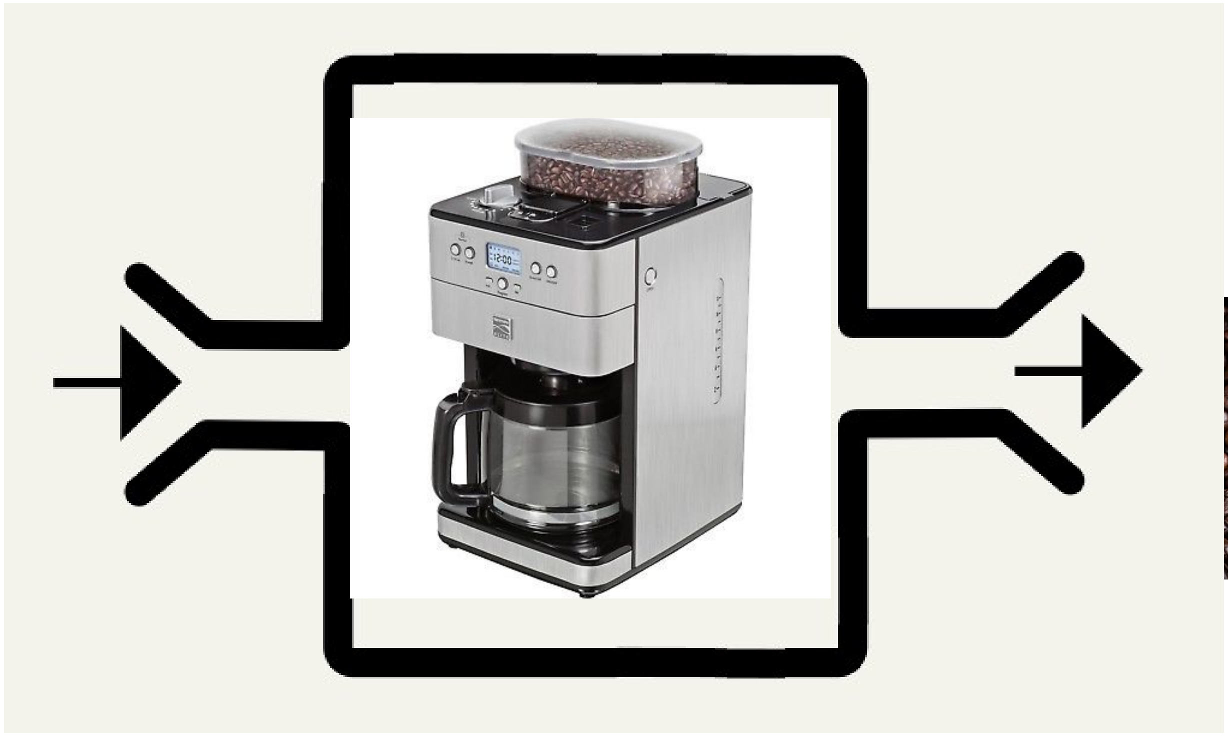
- Chang HaiBin
- Data Engineer (Hedge Fund)
- M.Sc. Uni. of Michigan (Math)

- Financial Engineer (Fintech Company)
- Business Analyst (e-commerce company)

Minimum requirement

Function

- Function is a machine that **take input(s)**, and **returns an output**



3 concepts

3 concepts:

- Functions as input
- Functions as output
- Partial Application

Functions as inputs

-
  ```
  def f(x,y,z):
      return x + y + z
  ```

- `def f(x,y,z):`

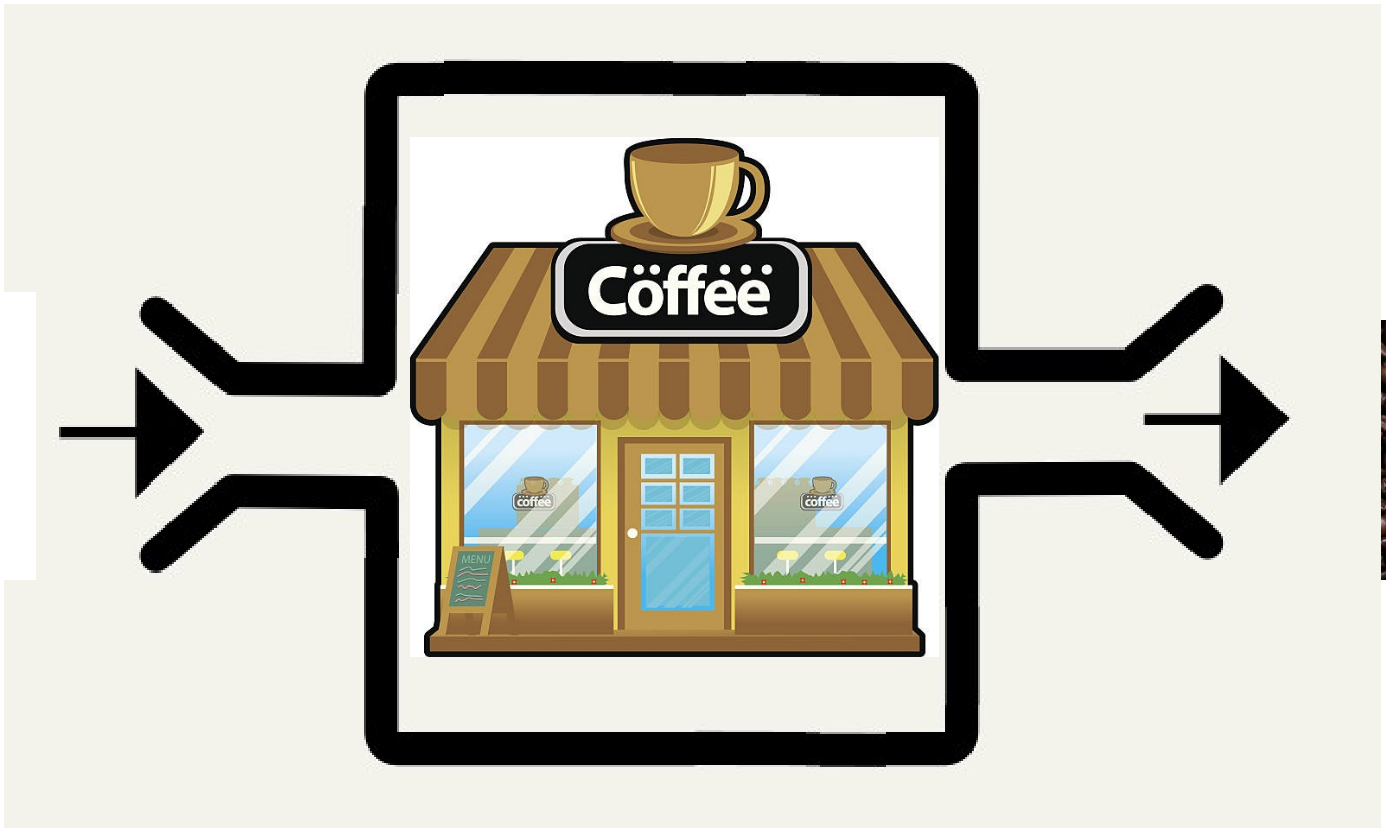
Most of the time, x, y, z are simple data types

- e.g. string, int, float, date, List, Set, Dictionary, etc.

- `def f(x,y,z):`

  Most of the time, x, y, z are simple data types

- But they can actually be functions as well!

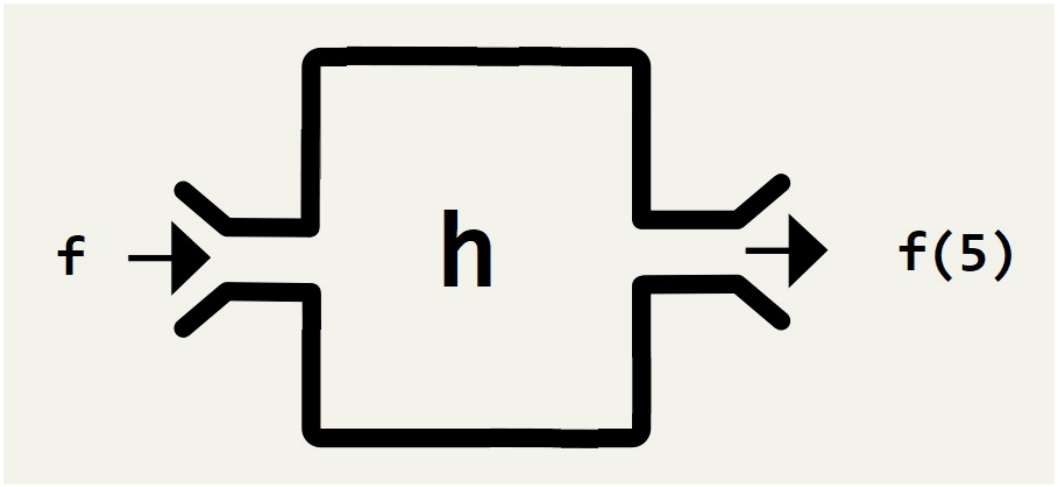

Cöffëë
coffee
coffee
MENU

Example

-
  ```
  def h(f):
      return f(5)
  ```

Example

-
```
def h(f):
        return f(5)
```



- h is a bigger function that:
  - Accepts a smaller function f
  - Returns a value f(5)

Example 1

-
  ```
  def h(f):
      return f(5)
  ```

-
  ```
  def g(x):
      return x + 1
  ```

Example 1

-
  ```
  def h(f):
      return f(5)
  ```

-
  ```
  def g(x):
      return x + 1
  ```

- `Then h(g) = 6`

Example 2

-
  ```
  def h(f):
      return f(5)
  ```

-
  ```
  def k(x):
      return x * 100
  ```

Example 2

-
  ```
  def h(f):
      return f(5)
  ```

-
  ```
  def k(x):
      return x * 100
  ```

- `Then h(k) = 500`

How is this useful?

Newton’s Method

Newton’s method

- Newton’s method helps you find the (approximate) solution of a function.
- It is available in “scipy” library.

Newton’s method

- $f(x) = x^2 - 3$
- $f(x) = 0$ when $x = \sqrt{3} \approx 1.732$

Newton’s method

```python
from scipy import optimize

def f(x):
    return x * x - 3

solution = optimize.newton(f,5)




print(solution)
# 1.7320508075688772
```

Newton’s method

```
from scipy import optimize

def f(x):
    return x * x - 3

solution = optimize.newton(f,5)
```

Smaller function “f” accepted by a bigger function!

```
print(solution)

# 1.7320508075688772
```

Designing Insurance Product

Pricing Insurance Product

- How much should I charge for insurance?

-
```
def price():
    ....
```

Pricing Insurance Product

- e.g. Depends on age

-
  ```
  def price(age):
      ....
  ```
- age: int

Pricing Insurance Product

- `e.g. Depends on probability of injury`

-
  ```
  def price(age, prob):
      ....
  ```
- `age: int`
- `prob: float`

Pricing Insurance Product

- `What if prob depends on time?`
- `def price(age, `**`prob`**`):`

  ```
      ....
  ```

- `age: int`
- **`prob`**`: ???`

Pricing Insurance Product

- Input a function instead!
- `def price(age, probFunc):`
  ```
      ....
  ```
- `age: int`
- `probFunc: datetime -> float`

Functions as outputs

-
  ```
  def f(x,y,z):
      return x + y + z
  ```

-
  ```
  def f(x,y,z):
      return ......
  ```

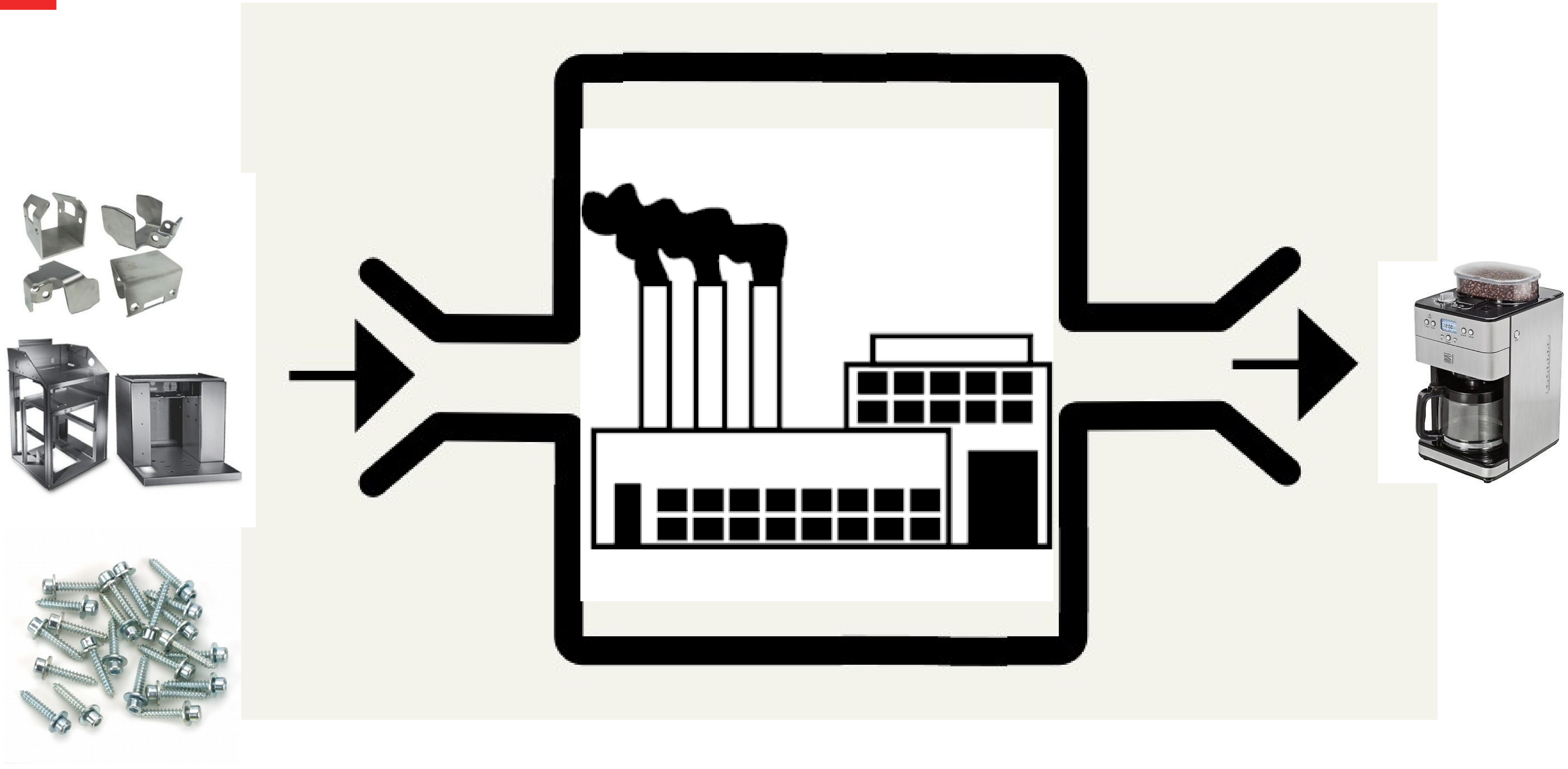
  Most of the time, we also return simple data types

- e.g. string, int, float, date, List, Set, Dictionary, etc.

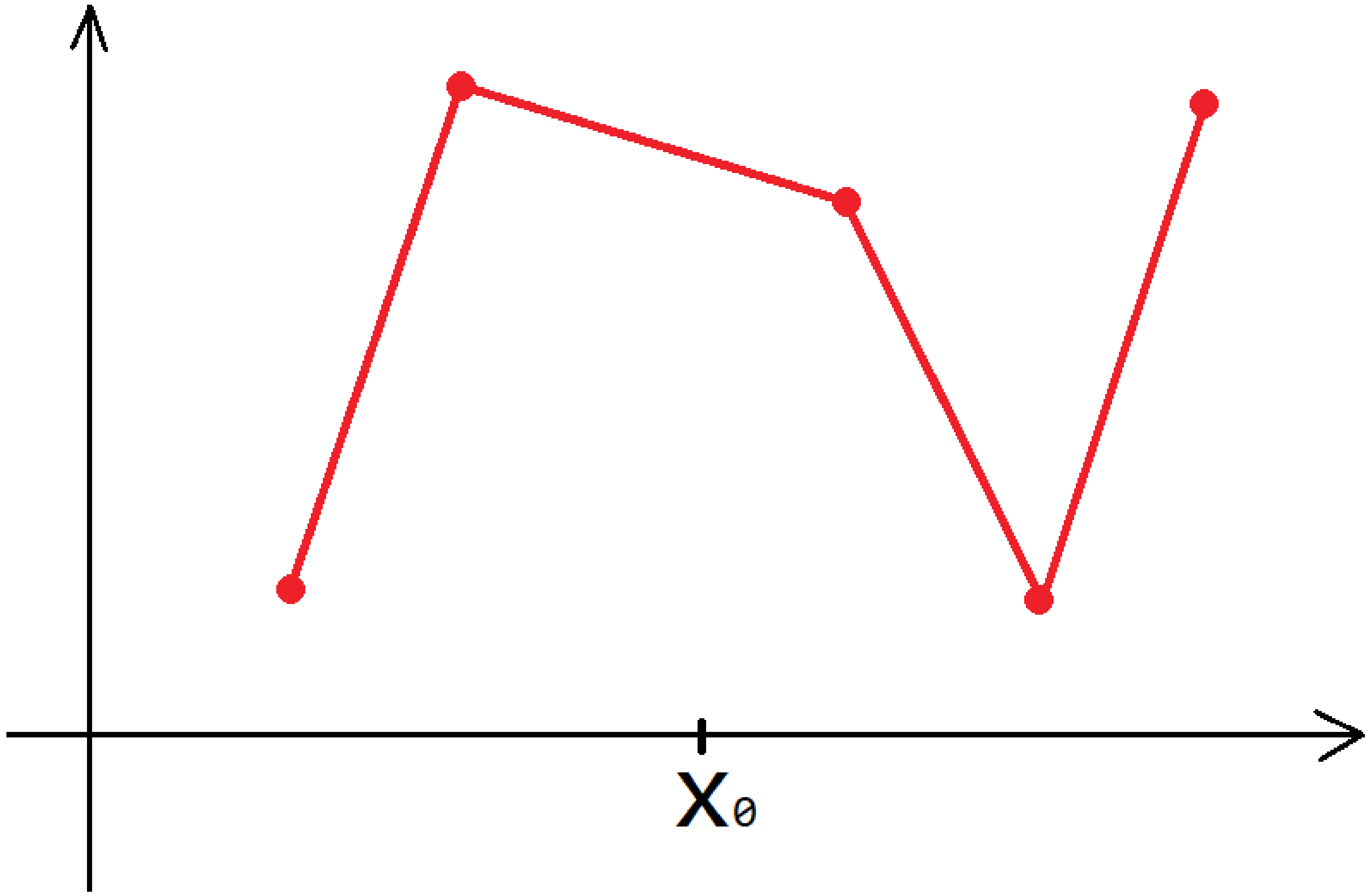
-
  ```
  def f(x,y,z):
      return ......
  ```

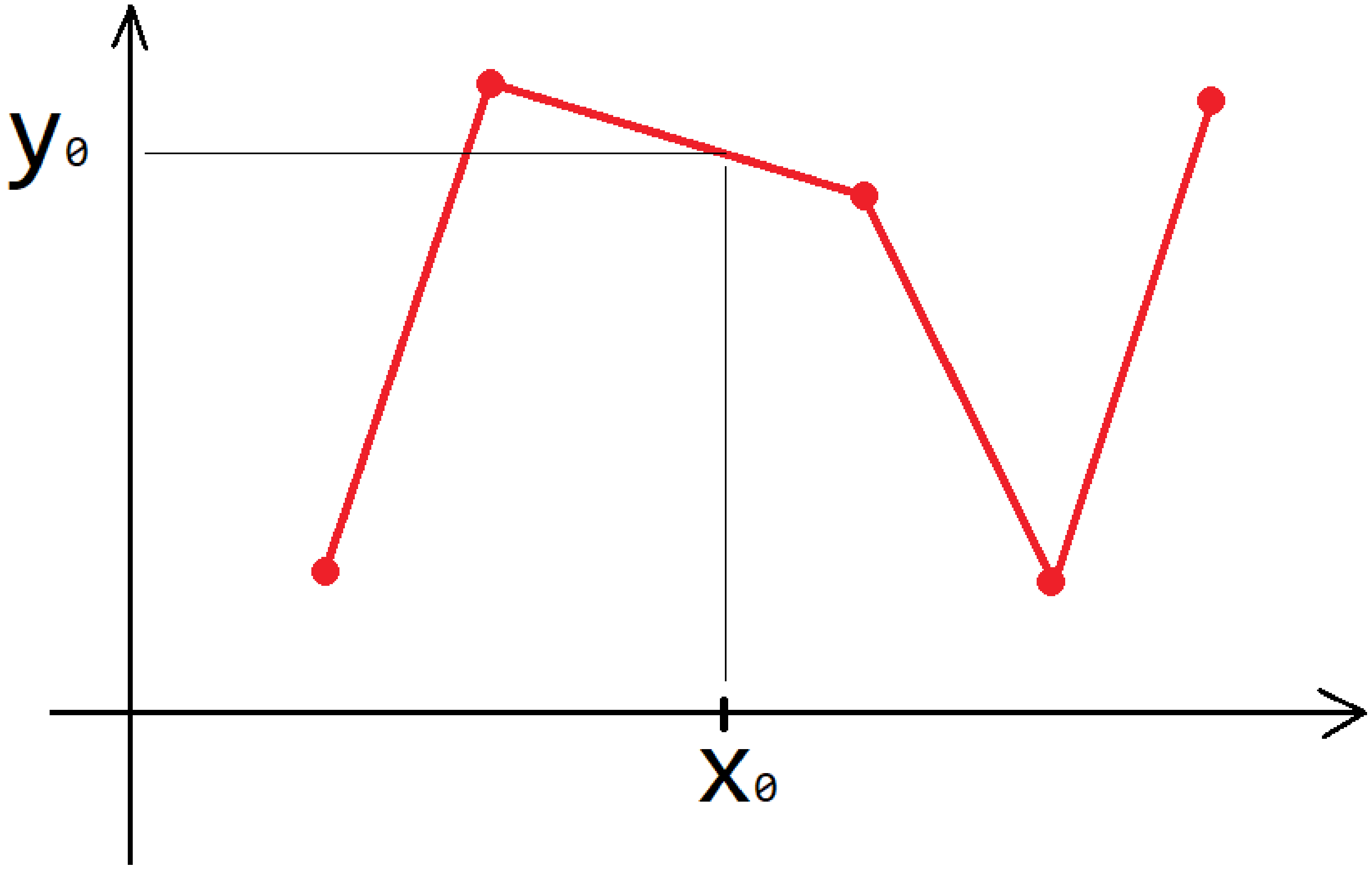
  Most of the time, we also return simple data types

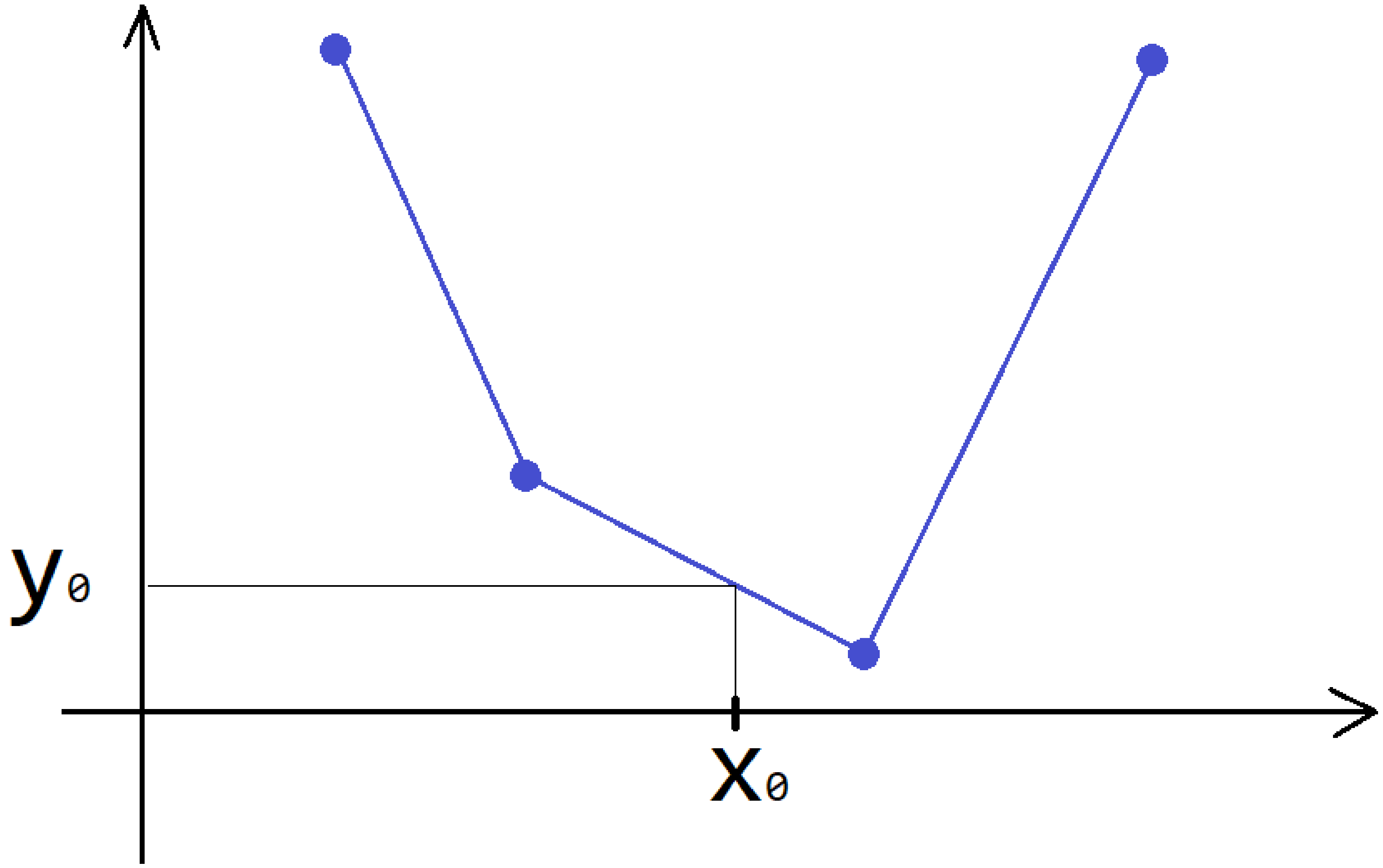
- Again, you can also return a function!



Interpolation


X0


y₀
x₀



y_0
x_0

Interpolate

- Interpolate function depends on:
  - Original Dataset
  - Value being queried x_0

Implementation 1

- `def interpolate(dataset, x0):`

  ```
      .......
  ```

- `dataset = [(-1,4),(1,7),(5,3)]`

Implementation 2

-
  ```
  class Interpolate:
      def __init__(self,dataset):
          ......

      def get_value(self,x0):
          .......
  ```

Implementation 2

-
  ```
  class Interpolate:
      def __init__(self,dataset):
      def get_value(self,x0):
  ```

- `dataset = [(-1,4),(1,7),(5,3)]`

- `inter_obj = Interpolate(dataset)`
- `result = inter_obj.get_value(2)`

Implementation 3

-
  ```
  def interpolate(dataset):
      def get_value(x0):
          ......
      return get_value
  ```

Implementation 3

```
• def interpolate(dataset):
      def get_value(x0):
          ......
      return get_value
```

“get_value” is a smaller function that is returned by a bigger function “interpolate” !

Implementation 3

-
  ```
  def interpolate(dataset):
      def get_value(x0):
          ......
      return get_value
  ```

- `dataset = [(-1,4),(1,7),(5,3)]`

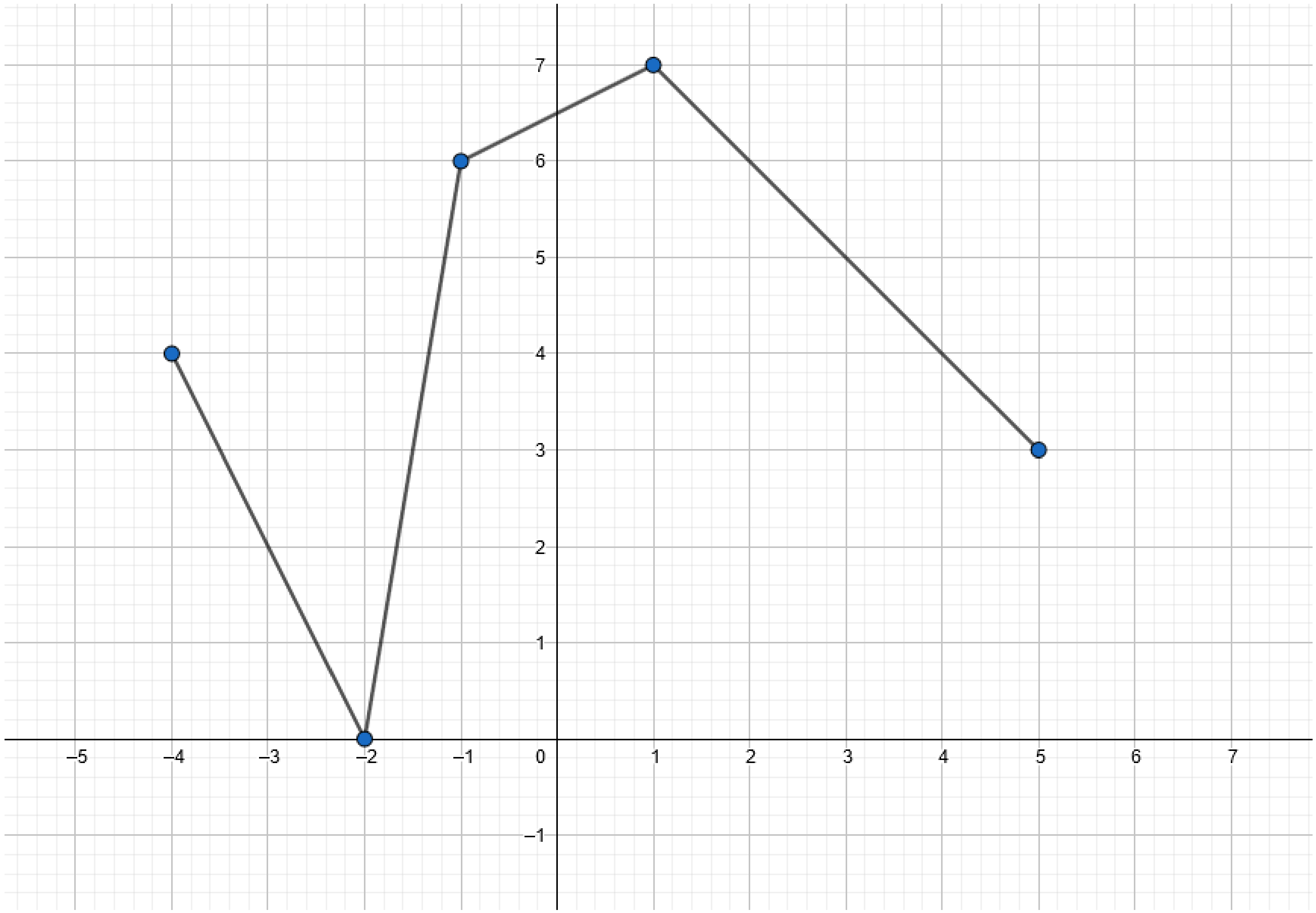
- `inner_func = interpolate(dataset)`
- `result = inner_func(2)`

Implementation 3

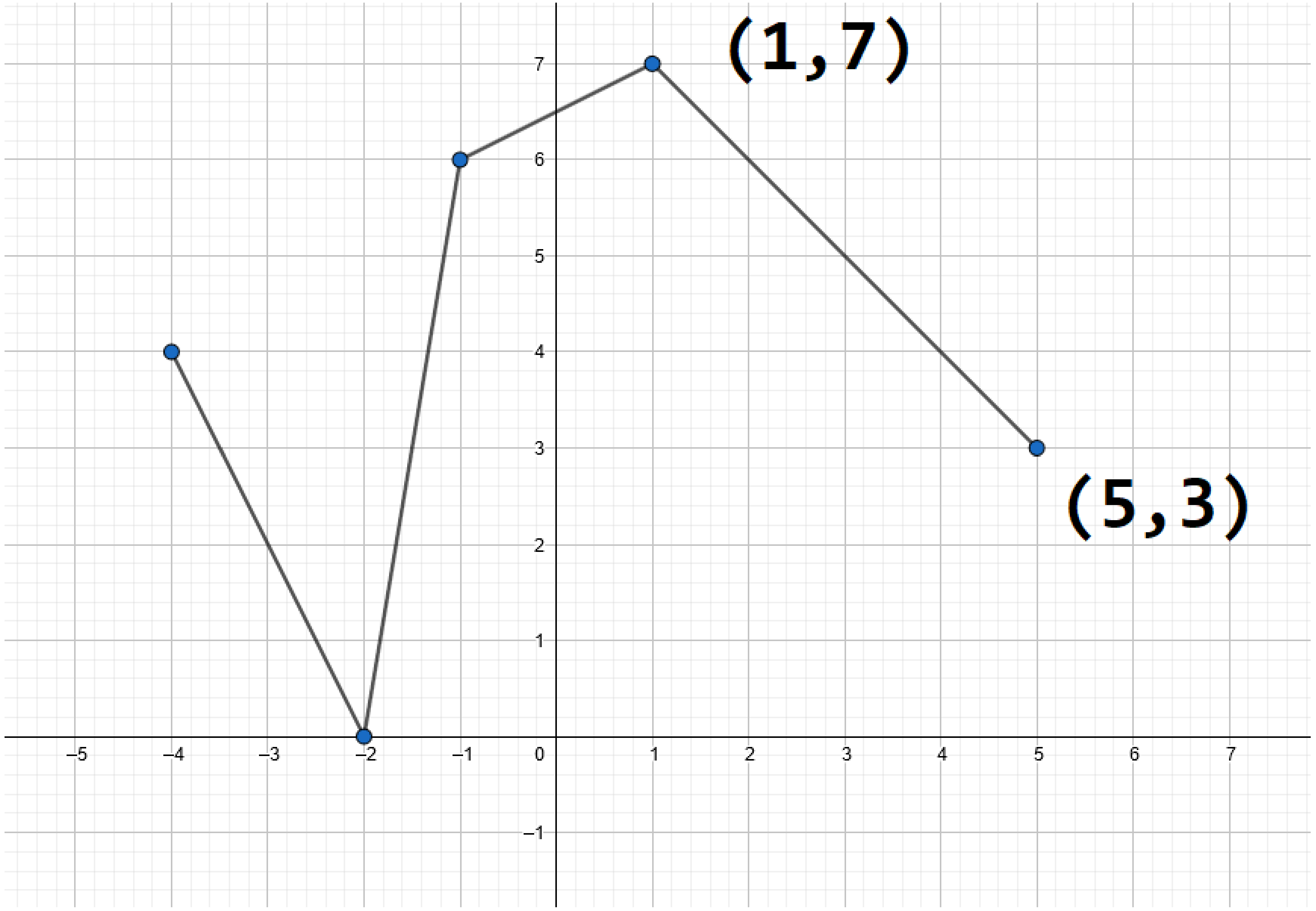
-
  ```
  def interpolate(dataset):
      def get_value(x0):
          ......
      return get_value
  ```
- `dataset = [(-1,4),(1,7),(5,3)]`
- `inner_func = interpolate(dataset)`
- `result = inner_func(2)`

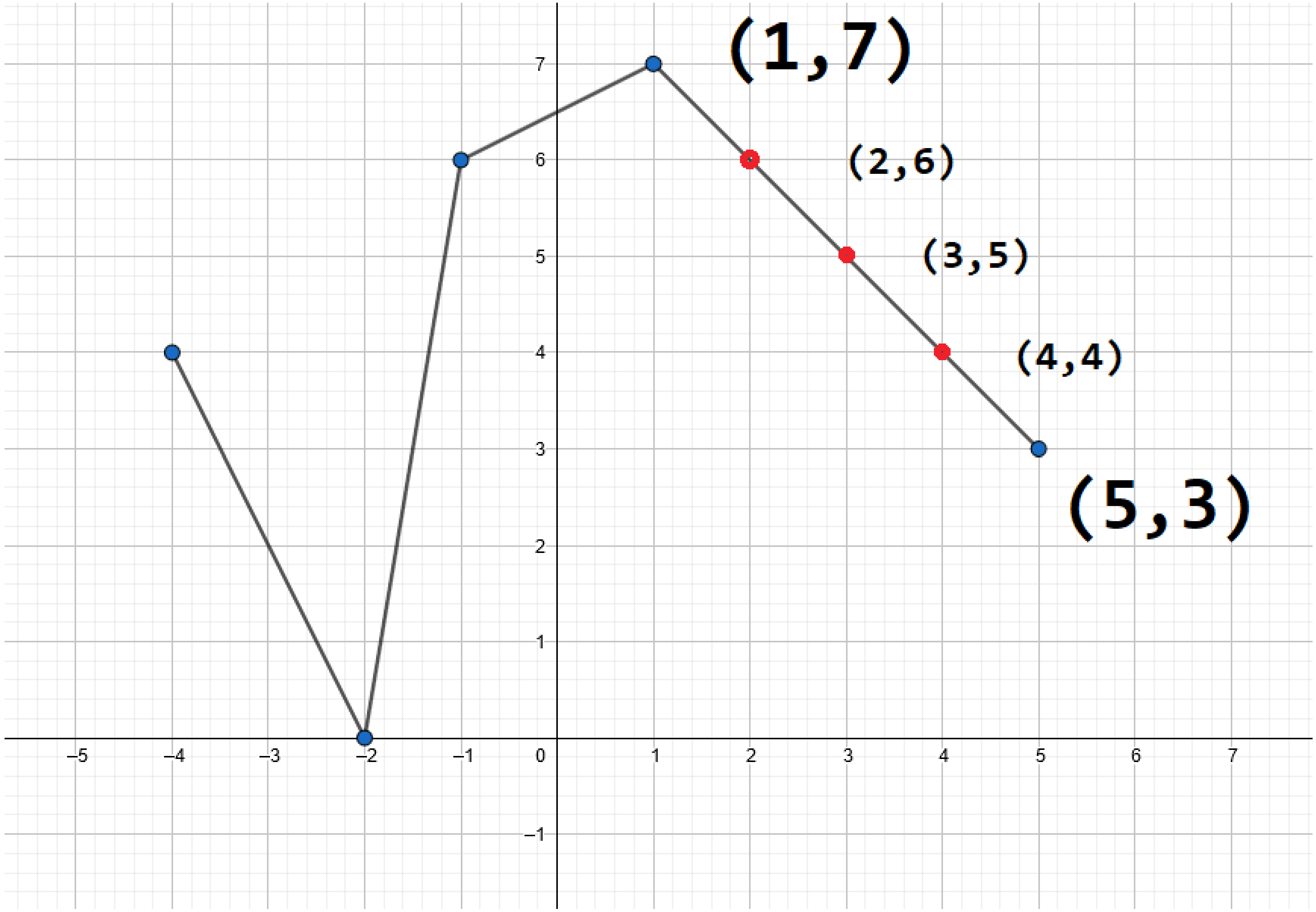
Directly use inner_func! No need to find the method hidden inside an object.

Interpolation Demo



7
6
5
4
3
2
1
0
−1
−5
−4
−3
−2
−1
1
2
3
4
5
6
7


(1,7)
(5,3)


(1,7)
(2,6)
(3,5)
(4,4)
(5,3)
−5
−4
−3
−2
−1
0
1
2
3
4
5
6
7
−1

Partial Application

Analogy

- If a function/ machine
  - Needs 3 inputs
  - But only 2 inputs provided

  - Still needs additional 1 inputs.

Analogy

- If a function/ machine
  - Needs 3 inputs
  - But only 2 inputs provided
  - Becomes a brand new function/machine that needs 1 inputs.

Example

-
  ```
  def f(x,y,z):
      return x + y + z
  ```

Example

-
  ```
  def f(x,y,z):
      return x + y + z
  ```

-
  ```
  result = f(1,2,3)
  ```

-
  ```
  # result = 6
  ```

Missing Variable

-
  ```
  def f(x,y,z):
      return x + y + z
  ```
- `result = f(1,2)`
- `# TypeError: f() missing 1 required positional argument: 'z'`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```

Lambda Definition

- `f = lambda x: lambda y: lambda z: \`
  ```
          x + y + z
  ```
- `result = f(1)(2)(3)`
- `# result = 6`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```
- `result = f(1)(2)`
- `# <function <lambda>.<locals>.<lambda>.<locals>.<lambda> at 0x013737C8>`
- Valid code!

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```

- `f : X -> Y -> Z -> result`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```

- `f : X -> Y -> Z -> result`
- `f(x)  -> Y -> Z -> result`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
           x + y + z
  ```

- `f : X -> Y -> Z -> result`
- `f(x)  -> Y -> Z -> result`
- `f(x)(y)    -> Z -> result`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
            x + y + z
  ```

- `f : X -> Y -> Z -> result`
- `f(x)  -> Y -> Z -> result`
- `f(x)(y)     -> Z -> result`
- `f(x)(y)(z)       -> result`

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
            x + y + z
  ```

-
  ```
  def f(x):
      def inner_1(y):





      return inner_1
  ```

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```

-
  ```
  def f(x):
      def inner_1(y):
          def inner_2(z):

          return inner_2
      return inner_1
  ```

Lambda Definition

-
  ```
  f = lambda x: lambda y: lambda z: \
          x + y + z
  ```

-
  ```
  def f(x):
      def inner_1(y):
          def inner_2(z):
              return x + y + z
          return inner_2
      return inner_1
  ```

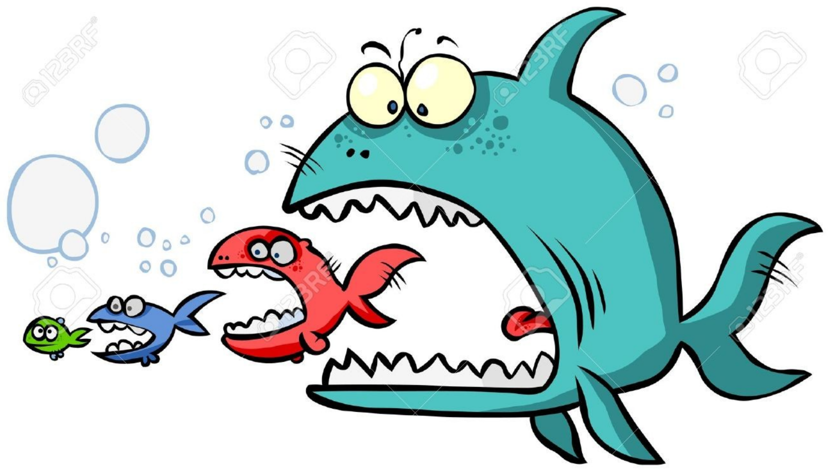
SKI Combinators

Lambda Calculus

-
  ```
  def B(x,y,z):
      return x(y(z))
  ```

-
  ```
  def D(w,x,y,z):
      return w(x,y(z))
  ```

-
  ```
  def O(x,y):
      return y(x(y))
  ```

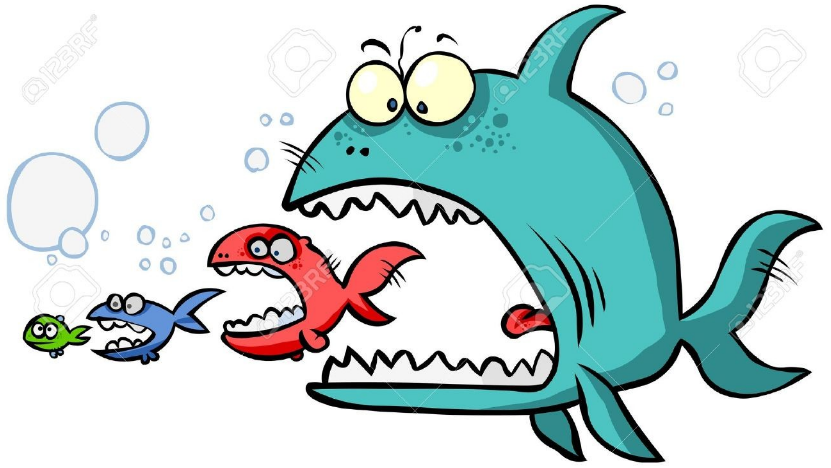


Lambda Calculus

-
  ```
  def B(x,y,z):
        return x(y(z))
  ```

-
  ```
  def D(w,x,y,z):
        return w(x,y(z))
  ```

-
  ```
  def O(x,y):
        return y(x(y))
  ```



Are these valid expressions?

What are the dependencies between w,x,y,z?

SKI

-
  ```
  def S(x,y,z):
      return x(z,y(z))
  ```

-
  ```
  def K(x,y):
      return x
  ```

-
  ```
  def I(x):
      return x
  ```

SKI

-
  ```
  S = lambda x: lambda y: lambda z: \
      x(z, y(z)) ★
  ```

-
  ```
  K = lambda x: lambda y:
      x
  ```

-
  ```
  I = lambda x:
      x
  ```

★Written in easier form

SKI

You can express any lambda expression using only S, K, I.

```
S = lambda x: lambda y: lambda z: \
    x(z, y(z)) ★
K = lambda x: lambda y:
    x
I = lambda x:
    x
```

★Written in easier form

Number Game Analogy

Given a number:

- 1. Take the first digit
- 2. Multiply by the rest of the digits
- 3. Repeat

- 927583 -> 9 x 27583 -> 248247

Number Game Analogy

- 927583 -> 9 x 27583 -> 248247

Number Game Analogy

- 927583 -> 9 x 27583 -> 248247
- 248247 -> 2 x 48247 -> 96494

Number Game Analogy

- 927583 -> 9 x 27583 -> 248247
- 248247 -> 2 x 48247 -> 96494
- 96494 -> 9 x 6494 -> 58446

Number Game Analogy

```
927583   ->   9 x 27583   ->   248247
248247   ->   2 x 48247   ->    96494
 96494   ->   9 x  6494   ->    58446
 58446   ->   5 x  8446   ->    42230
```

Example

```
T(x,y) = y(x)
```

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

- `T(x) = S(K(S(I)))(K)(x)`

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

- `T(x) = S(K(S(I)))(K)(x)`
  `     = K(S(I))(x)[K(x)]`
- `Because S(x,y,z) = x(z)(y(z))`

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

- `T(x) = S(K(S(I)))(K)(x)`
  `       = K(S(I))(x)[K(x)]`
  `       = S(I)[K(x)]`

- Because `K(x,y) = x`

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

-
  ```
  T(x) = S(K(S(I)))(K)(x)
       = K(S(I))(x)[K(x)]
       = S(I)[K(x)]
  ```

-
  ```
  T(x)(y) = S(I)[K(x)](y)
  ```

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

- `T(x) = S(K(S(I)))(K)(x)`
  `     = K(S(I))(x)[K(x)]`
  `     = S(I)[K(x)]`
- `T(x)(y) = S(I)[K(x)](y)`
  `        = I(y)[K(x)(y)]`

- `Because S(x,y,z) = x(z)(y(z))`

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

-
  ```
  T(x) = S(K(S(I)))(K)(x)
       = K(S(I))(x)[K(x)]
       = S(I)[K(x)]
  ```

-
  ```
  T(x)(y) = S(I)[K(x)](y)
          = I(y)[K(x)(y)]
          = y[K(x)(y)]
  ```
- Because `I(x) = x`

Example

```
T(x,y) = y(x)
Goal: T = S(K(S(I)))(K)
```

-
  ```
  T(x) = S(K(S(I)))(K)(x)
       = K(S(I))(x)[K(x)]
       = S(I)[K(x)]
  ```

-
  ```
  T(x)(y) = S(I)[K(x)](y)
          = I(y)[K(x)(y)]
          = y[K(x)(y)] = y[x]
  ```
- Because K(x,y) = x

Another example

Can you do this?

```
def g(y):
    return y(y)
```

Can you do this?

```
def g(y):
    return y(y)

def I(x):
    return x

result = g(I)
print(result)
```

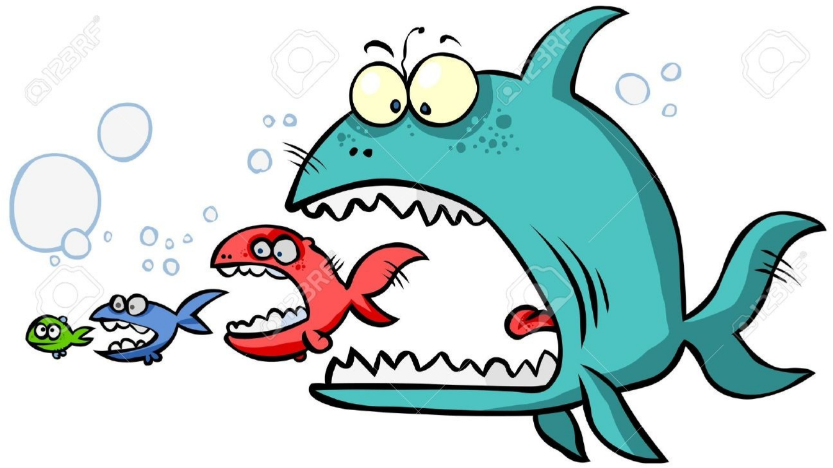
Can you do this?

```
def g(y):
    return y(y)

def I(x):
    return x

result = g(I)
print(result)
```

- `# <function I at 0x033209C0>`

Take away

- Functions accepting/returning functions can be a powerful tool
- Implementation at the language level is messy
- It encourages you to think in terms of dependencies.



Summary

3 concepts:

- Functions as input
- Functions as output
- Partial Application

Q&A