Introduction to Functional Programming (Python)

John R. Woodward

Functional Programming

1. Programming paradigm? Object oriented, imperative
2. Immutable state (referential transparency)
3. **Higher order functions, partial functions, anonymous functions (lambda)**
4. **Map, filter, reduce (fold)**
5. List data structures and recursion feature heavily
6. Type inference (type signatures)
7. Lazy and eager evaluation, list comprehension

State & Referential Transparency

1. In maths, a function depends **ONLY ON ITS INPUTS** e.g. root(4) = +/-2
2. This is a “stateless function”
3. A stateful function (method, procedure) stores some data which changes i.e. **mutable data**
4. E.g. a function which returns the last argument it was given
5. F(4) = -1, F(55) = 4, F(3) = 55, F(65) = 55,
6. (**side effects**, e.g. accessing a file)
7. Harder to debug stateful function than stateless (why).

Can Your Programming Language Do This? Motivating Example

- http://www.joelonsoftware.com/items/2006/08/01.html

```
// A trivial example:

alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Moose!");
```

You only need to write code for the “bit that changes”

Can Your Programming Language Do This? Motivating Example

- http://www.joelonsoftware.com/items/2006/08/01.html

```
// A trivial example:

alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Moose!");
```

The repeated code looks wrong, of course, so you create a function:

```
function SwedishChef( food )
{
    alert("I'd like some " + food + "!");
}

SwedishChef("Spaghetti");
SwedishChef("Chocolate Moose");
```

You only need to write code for "**the bit that changes**"

What is repeated here?
What is the abstraction?

```
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```

The Abstraction

1. We are
2. - getting an “object” (lobster/chicken)
3. - **repeating an action twice**
    1. Put in pot, put in pot
    2. Boom boom, boom boom.
4. The first action is with the object.
5. The second action is with another object (water, coconut)

We repeat the 2nd action TWICE

```
function Cook( i1, i2, f )
{
     alert("get the " + i1);
     f(i1);
     f(i2);
}

Cook( "lobster", "water", PutInPot );
Cook( "chicken", "coconut", BoomBoom );
```

```
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```

Higher Order Functions (HOF)

1. In most programming languages we pass around integers, Booleans, strings, as argument to function and return types
2. For example **Boolean isPrime(Integer n)**
3. Takes an integer and returns true/false depending on if it is prime or not.
4. **With functional languages we can pass around functions**. Type signatures

Examples of higher order functions

1. At college you had to differentiate y=mx+c
2. Also integration (returns a function).
3. In physics…law of refraction/reflection.
4. Fermat's principle states that light takes a path that (locally) minimizes the optical length between its endpoints. (Lagrangian Mechanics)
5. Mechanics…hanging chain (a ball rolls down a slope – a chain takes on a shape).
6. Droplets minimize surface area, aerodynamics (wing shape).
7. CALCULUS OF VARIATION

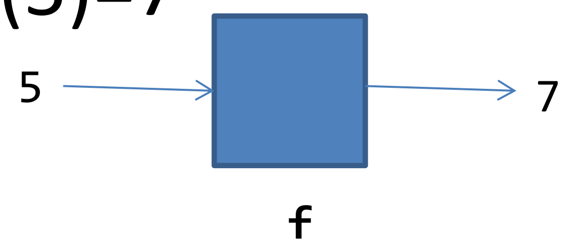
Example: A linear function

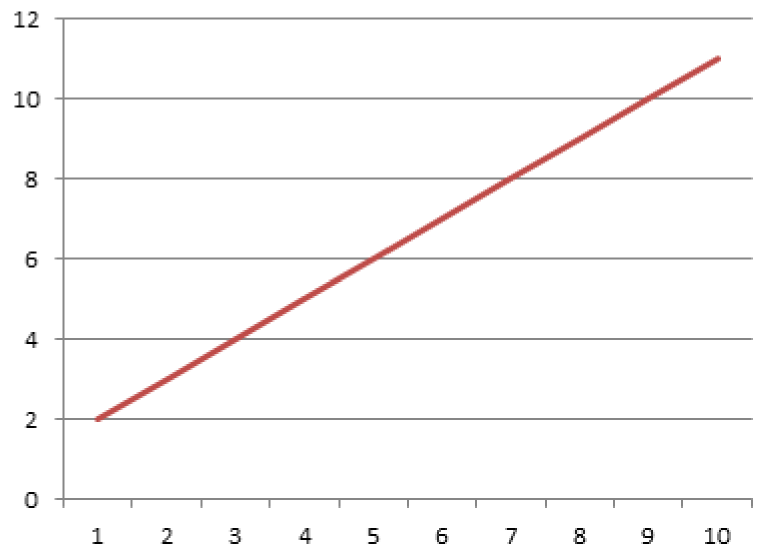
```
#return a function
#note return result, not result(x)
def linear(gradient, intercept):
    def result(x):
        return gradient*x + intercept
    return result #not result(x)
myLinearFunction = linear(4,3)
#make a function 4*x+3
print myLinearFunction(8)
#evaluate the function at point 8
print linear(4,3)(8)
#or do directly
```

<u>Inputs</u>, <u>outputs</u> and <u>functions</u>

- Input **x**, to a function **f**, outputs **f(x)**
- f is a "<u>process</u>"

1. x=5,
2. f(x) = x+2,
3. f(5)=7

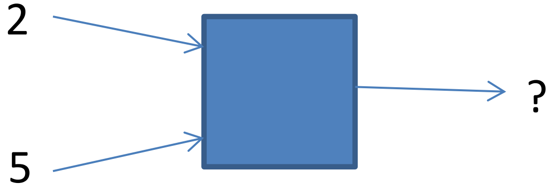




- Typically inputs/outputs are basic data types (int, float, double, Boolean, char, String)

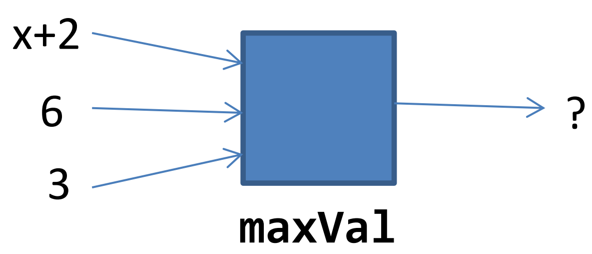
Maximum of two numbers

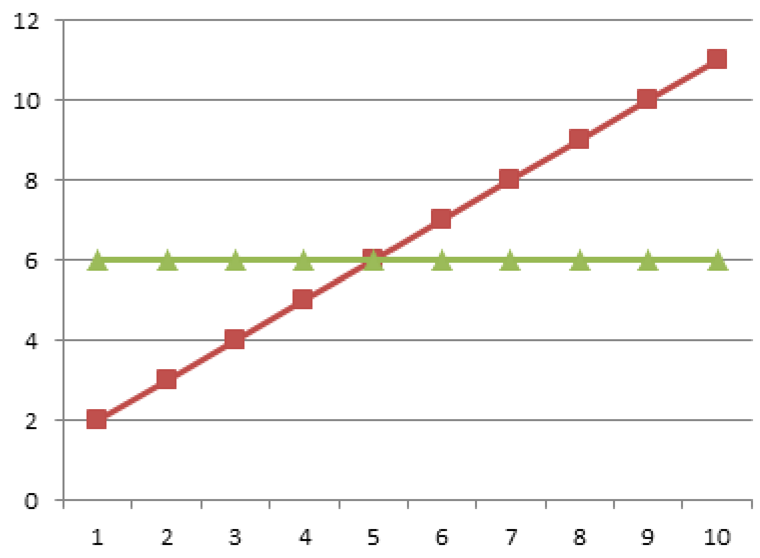
```
def max(x,y):
    if (x>y):
        return x
    else:
        return y
#what is the input and output data-type
e.g. max(2,5)=???
```



Maximum of two functions at x

```
def maxVal(f, g, x):
    return max(f(x), g(x))
#what is the input and output data-type
example
f(x)=x+2
g(x)=6
maxVal(f,g,3)
```





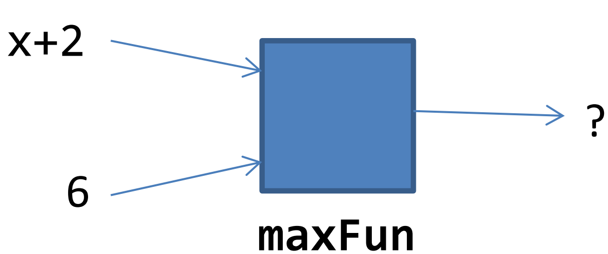
Returning the Function

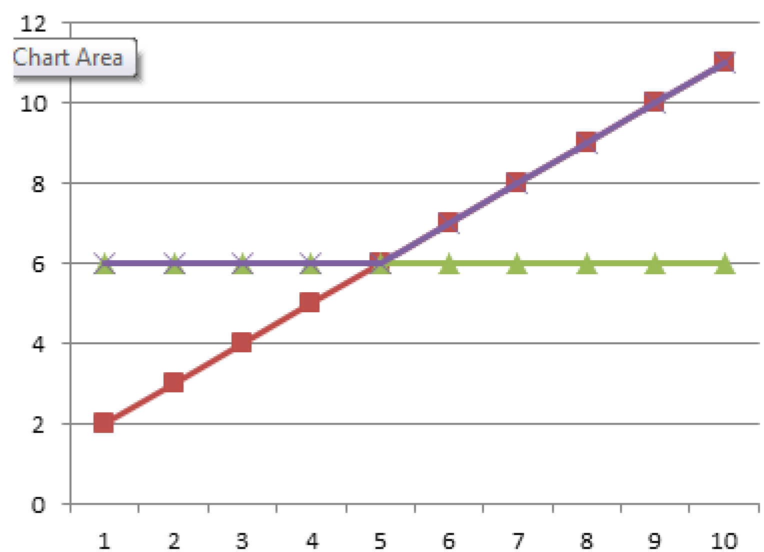
```
def maxFun(f, g):
    def maximumFunction(x):
        return max(f(x), g(x))
    return maximumFunction
```

example

f(x)=x+2   g(x)=6

maxFun(f,g)=?





f(x) and f

1. In maths, f and f(x) are different.
2. f is a function!
3. f(x) is the value of f at value x
4. Never say, “the function f(x)”
5. Say - The function f that takes a variable x (i.e. f takes variable x)
6. Or – the function f at the point x (i.e. f given x has the value ??)

Summation

1. In maths $\sum_{i=1}^{i=10} f(x) =$ f(1)+f(2)+…+f(10)
2. $\sum$ takes 3 arguments, and upper and lower bound and a function to sum over.
3. e.g. $\sum_{i=1}^{i=3} 2x =$ 2.1+2.2+3.2=2+4+6=12

```
4. def sum(f, a, b):
5.     total = 0
6.     for i in range(a, b+1):
7.         total += f(i)
8.     return total
```

Product Symbol in Maths

1. In mathematics we often use the product symbol . $\prod_{i=1}^{i=3} f(x)$ means f(1)*f(2)*f(3).
2. You can do this for the labs ☺
3. Hint: is its almost “cut and paste”, but you need to think a little.

Your turn…in the lab

1. Write a function f which returns a function which is the **minimum** of functions f1, f2
2. i.e. f(x)=min {f1(x) and f2(x)}
3. Write a function which returns a function which is the **average** of two functions
4. i.e. f(x)={f1(x) + f2(x)}/2
5. Can you generalize this to a **list of functions**

1 Simple example: Higher order Function

- **#Takes a function f and value x and evaluates f(x), returning the result.**
- `def` **apply(f, x):**
- `    return f(x)`
- `#f is a function, x is input to f`

2 Composition as HOF

- `def compositionValue(f, g, x):`
- `    return f(g(x))`
- This take two functions f and g.
- And a value x
- And calculates the values f(g(x))
- Picture this
- **What restrictions are there?**
- (lets write primitive recursion)

3 Returning a Function fg

```
def compositionFunction(f, g):
    def result(x):
        return f(g(x))
    return result
myFunction = compositionFunction(timesThree, timesTwo)
print myFunction(4)
print apply(compositionFunction(timesThree, timesTwo) , 4)
print (lambda x, y :compositionFunction(x,y)) (timesThree, timesTwo) (4)
```

Variable as Strings, and “literal strings”

1. `name = "john r woodward"`
2. `print name`
3. #Or we can just print the name directly
4. `print " john r woodward "`
5. If we only use “John r woodward” once – we could use a string literal (a one-off use, do not need variable name).
6. If we use “John” multiple times – define a variable – name – and use that.

Anonymous Functions (lambda)

```
1. def f (x): return x**2
2. print f(8)
3. #Or we can do
4. g = lambda x: x**2
5. print g(8)
6. #Or just do it directly
7. print (lambda  x: x+2) (4)
8. y = (lambda  x: x*2) (4)
9. print y # the value is stored in y
```

Lambda – filter, map, reduce

```
foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
print filter(lambda x: x % 3 == 0, foo)
#[18, 9, 24, 12, 27]
print map(lambda x: x * 2 + 10, foo)
#[14, 46, 28, 54, 44, 58, 26, 34, 64]
print reduce(lambda x, y: x + y, foo)
#139
#(more detail in a few slides)
```

What are the following types

```
• def inc(x):
•     return x+1

1. print inc(5)
2. print type(inc)
3. print type(inc(5))
4. print type("inc(5)")
5. print type(eval ("inc(5)" ))
6. print type((lambda  x: x*2))
7. print type((lambda  x: x*2) (4))
```

And the types are…

1. 6
2. <type 'function'>
3. <type 'int'>
4. <type 'str'>
5. <type 'int'>
6. <type 'function'>
7. <type 'int'>

What does the following do

1. `def f(a, b): return a+b`
2. `def inc(x): return x+1`
3. `def double(x): return x*2`
4. `print f(1,2)`
5. `print f("cat","dog")`
6. `print f(inc, double)`

output

1. 3
2. catdog
3. Traceback (most recent call last):

```
  File "C:\Users\jrw\workspace\Python1\addition1.py", line 6, in <module>
    print f(inc, double)
  File "C:\Users\jrw\workspace\Python1\addition1.py", line 1, in f
    def f(a, b): return a+b
TypeError: unsupported operand type(s) for +: 'function' and 'function'
```

What does the following print

- `foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]`

1. `print filter(lambda x: x % 3 == 0, foo)`
2. `print map(lambda x: x * 2 + 10, foo)`
3. `print reduce(lambda x, y: x + y, foo)`

output

1. [18, 9, 24, 12, 27]
2. [14, 46, 28, 54, 44, 58, 26, 34, 64]
3. 139

What does the following print

- `foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]`

1. `print type(filter(lambda x: x % 3 == 0, foo))`
2. `print type(map(lambda x: x * 2 + 10, foo))`
3. `print type(reduce(lambda x, y: x + y, foo))`

output

1. <type 'list'>
2. <type 'list'>
3. <type 'int'>

Lists & operations

- Functional programming uses LISTs as its primary data structure. E.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
1. listNumbers = [1,2,3] #[1,2,3]
2. listNumbers.append(4)#[1, 2, 3, 4]
3. listNumbers.insert(2, 55)#[1, 2,
   55, 3, 4]
4. listNumbers.remove(55)#[1, 2, 3, 4]
5. listNumbers.index(4)#3
6. listNumbers.count(2)#1
```

Map – (a transformation)

- **Map** takes a function and a list and applies the function to each elements in the list
- `def cube(x): return x*x*x`
- `print map(cube, range(1, 11))`
- `print map(lambda x :x*x*x, range(1, 11))`
- `print map(lambda x :x*x*x, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])`
- `# [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]`
- `#what is the type signature`

filter

- **Filter** takes a function (what type???) and a list, and returns items which pass the test
- `def f1(x): return x % 2 != 0`
- `def f2(x): return x % 3 != 0`
- `def f3(x): return x % 2 != 0 and x % 3 != 0`

1. `print filter(f1, range(2, 25))`
2. `print filter(f2, range(2, 25))`
3. `print filter(f3, range(2, 25))`

filter 2

- `def f1(x): return x % 2 != 0`
- `def f2(x): return x % 3 != 0`
- `def f3(x): return x % 2 != 0 and x % 3 != 0`

- `print filter(f1, range(2, 25))`
- # [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]#odd
- `print filter(f2, range(2, 25))`
- # [2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23]
- `#not divisible by 3`
- `print filter(f3, range(2, 25))`
- # [5, 7, 11, 13, 17, 19, 23]#not divisible by 2 and 3

A list can contain different datatypes

- `list1 = [0, 1, 0.0, 1.0, True, False, "True", "False", "", None, [True], [False]]`
- `def isTrue(x):`
- `    return x`
- `print filter(isTrue, list1)`

output

- [1, 1.0, True, 'True', 'False', [True], [False]]

Reduce – try this

- `myList = [1, 2, 3, 4]`

1. `reduce((lambda x, y: x + y), myList)`
2. `reduce((lambda x, y: x - y), myList)`
3. `reduce((lambda x, y: x * y), myList)`
4. `reduce((lambda x, y: x / y), myList)`

answers

- `reduce( (lambda x, y: x + y), [1, 2, 3, 4] )`
- `reduce( (lambda x, y: x - y), [1, 2, 3, 4] )`
- `reduce( (lambda x, y: x * y), [1, 2, 3, 4] )`
- `reduce( (lambda x, y: x / y), [1, 2, 3, 4] )`

- 10
- -8
- 24
- 0

The last one

- `print reduce( (lambda x, y: x / y), [1.0, 2.0, 3.0, 4.0] )`

The last one

- `print reduce( (lambda x, y: x / y), [1.0, 2.0, 3.0, 4.0] )`
- 0.0416666666667
- # what is the type signature?

Map, Reduce, Filter

- **Map, reduce and filter** all take a function as an argument.
- What type is the function in each case

1. **Map,**
2. **Reduce,**
3. **Filter**

Data Types of Function Taken.

1. Map: takes an argument of type T and returns a type S.
2. It returns a list of S, denoted [S]
3. Reduce, takes two arguments of type T, and returns an argument of type T
4. Filter: take an argument of type T and returns a Boolean (True/False)
5. It returns a list of T, denoted [T]

Partial Application

1. Motivating example
2. Addition (+) takes two args (arg1 + arg2)
3. What if we only supply one arg?
4. We cannot compute e.g. (1+?)
5. But it is still a function of one arg
6. (1+? could be called what???)

Inc as partial add

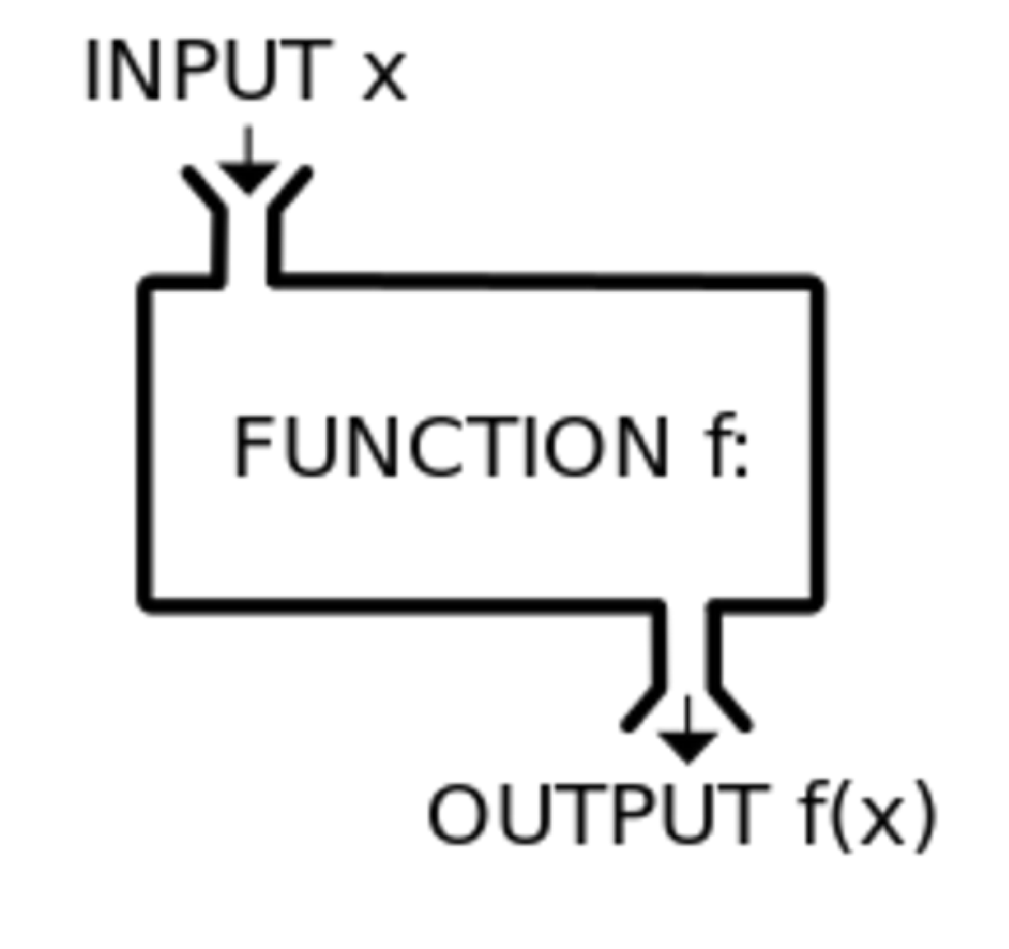
- #add (2 args), inc(x)=add(1,x)
- #inc is a special case of add
- from functools import partial
- def **add(a,b)**:
-     return a+b
- inc = partial(add, 1)
- print inc(4)

Inc defined with add

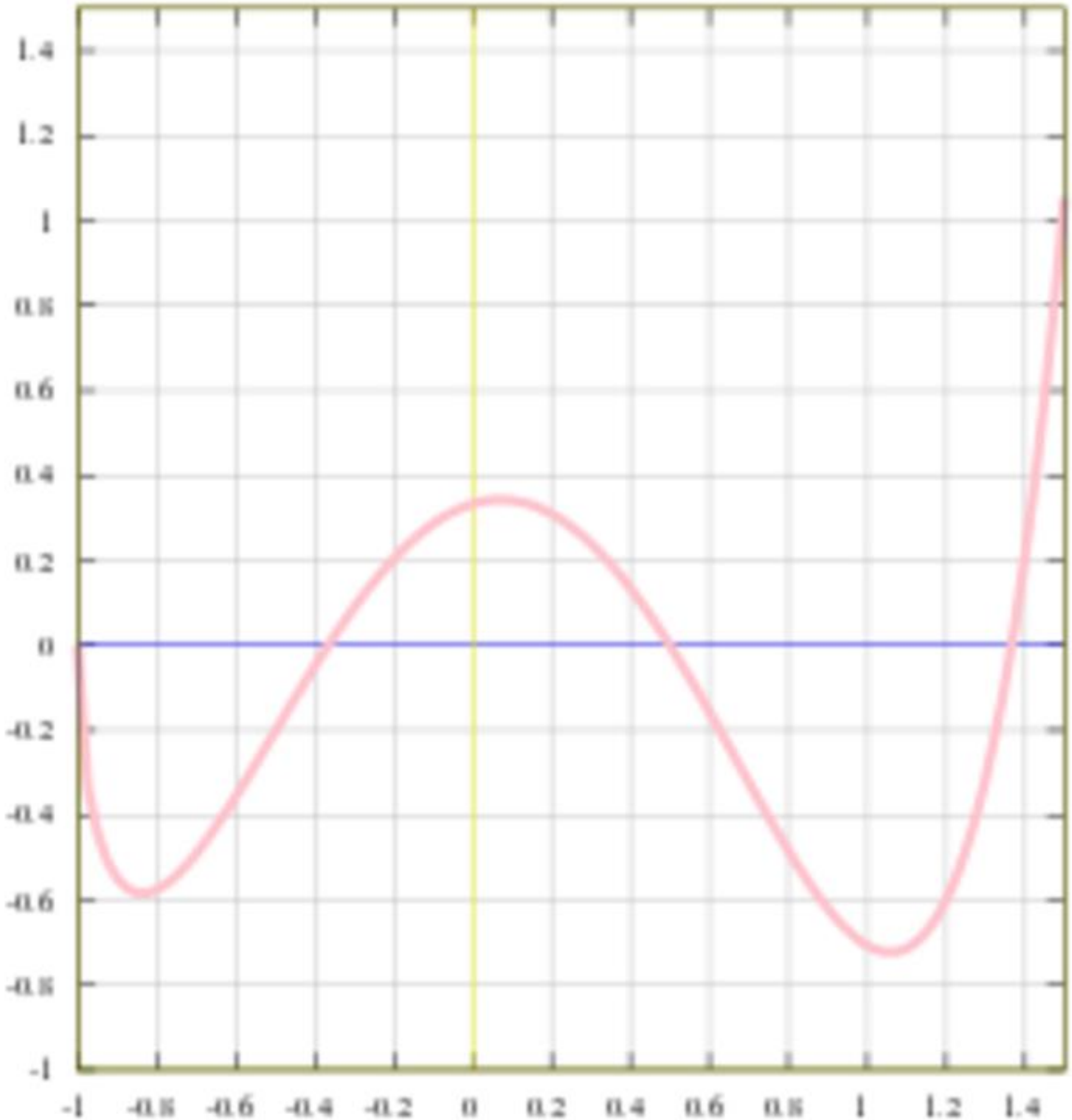
- #add takes 2 arguments
- def **add(x,y):**
-     return x+y
- #we can define a new function by hardcoding one variable
- def **inc(x):**
-     return add(1,x)
- print inc(88)

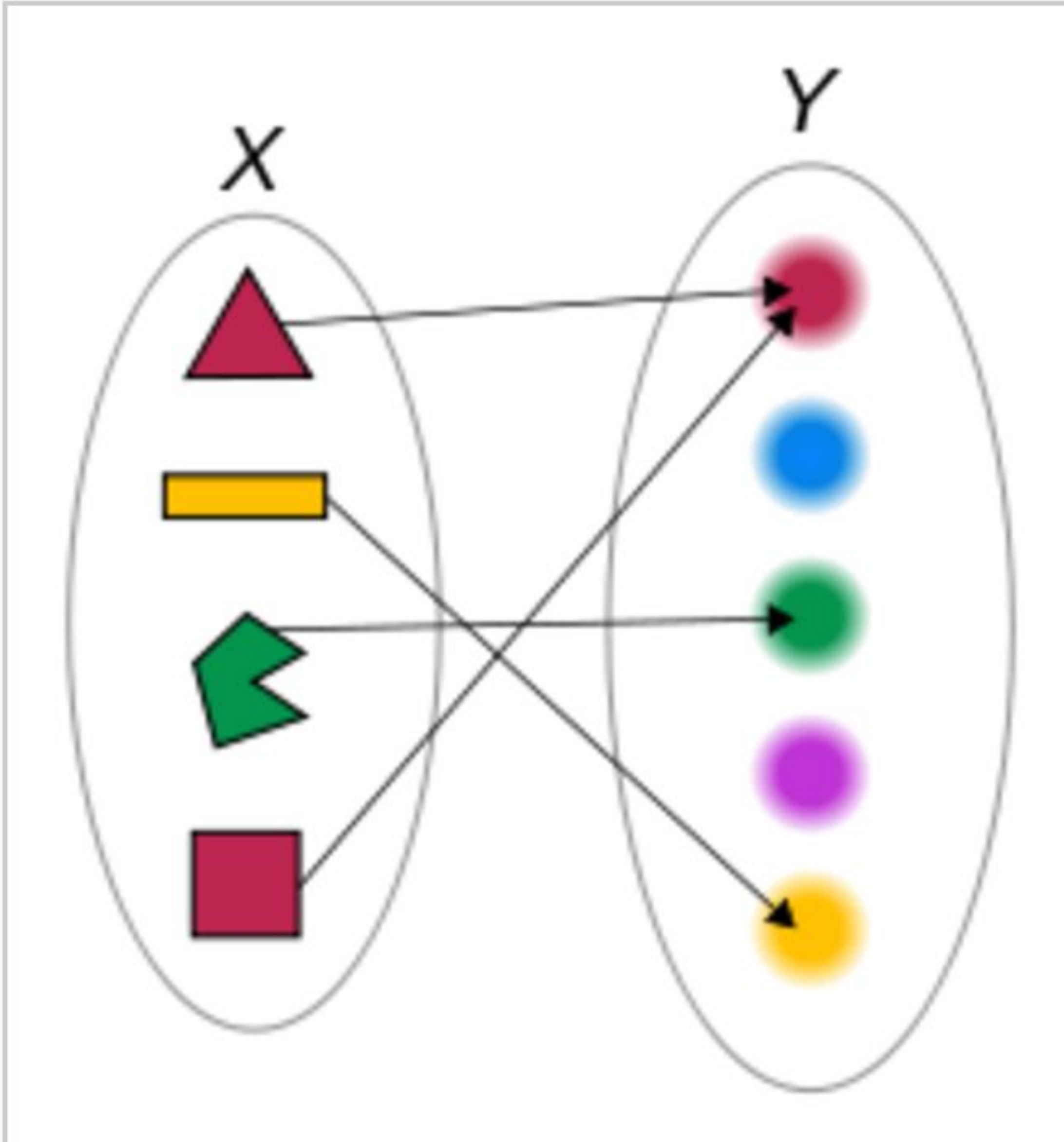
double as partial mul

- #mul(2 args), double(x)=mul(2,x)
- #double is a special case of mul
- from functools import partial
- def **mul(a,b)**:
-     return a*b
- double = partial(mul, 2)
- print double(10)



A function *f* takes an input *x*, and returns a single output *f*(*x*). One metaphor describes the function as a "machine" or "black box" that for each input returns a corresponding output.


1.4
1.2
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-1
-0.8
-0.6
-0.4
-0.2
0
0.2
0.4
0.6
0.8
1
1.2
1.4



A function that associates to any of the four colored shapes its color.