Functional Programming in Python

Programming 2
01219116

Solution: mysum

```
def mysum(lst):
    """
    >>> mysum([10, 20, 30])
    60
    >>> mysum([1,0,-10,5,9])
    5
    """

    if lst == []:
        return 0
    else:
        return lst[0] + mysum(lst[1:])
```

Solution mymax

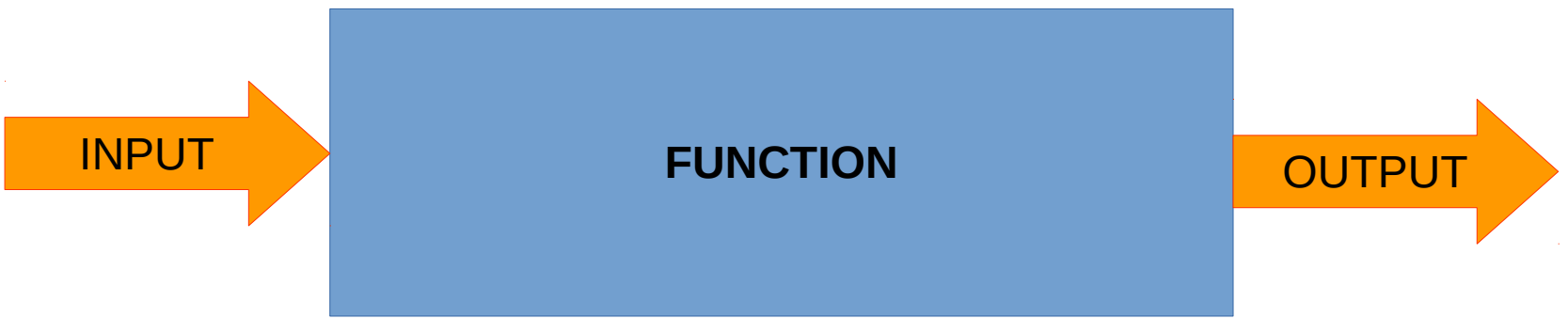
```
def mymax(lst):
    """
    >>> mymax([1])
    1
    >>> mymax([5,6,7,100,200])
    200
    """
    if len(lst)==1:
        return lst[0]
    else:
        mx = mymax(lst[1:])
        if mx > lst[0]:
            return mx
        else:
            return lst[0]
```

Solution mymerge

```
def mymerge(lst1, lst2):
    """
    >>> mymerge([1,2,5,6,10],[2,3,4,5,8,9,12])
    [1, 2, 2, 3, 4, 5, 5, 6, 8, 9, 10, 12]
    """
    if lst1 == []:
        return lst2
    if lst2 == []:
        return lst1
    if lst1[0] <= lst2[0]:
        return [lst1[0]] + mymerge(lst1[1:], lst2)
    else:
        return [lst2[0]] + mymerge(lst1, lst2[1:])
```

What is functional programming?

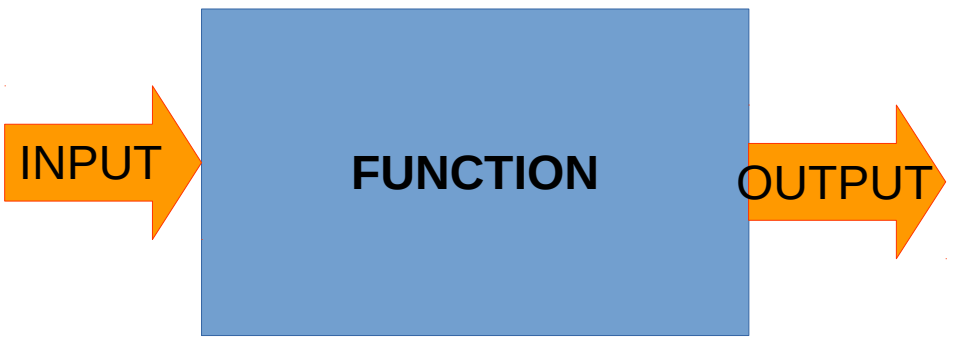
- Computation = function
  - Think of functions as *mathematical* functions, where there is no state changes or ***mutable*** data.
  - Declarative programming



What is functional programming?

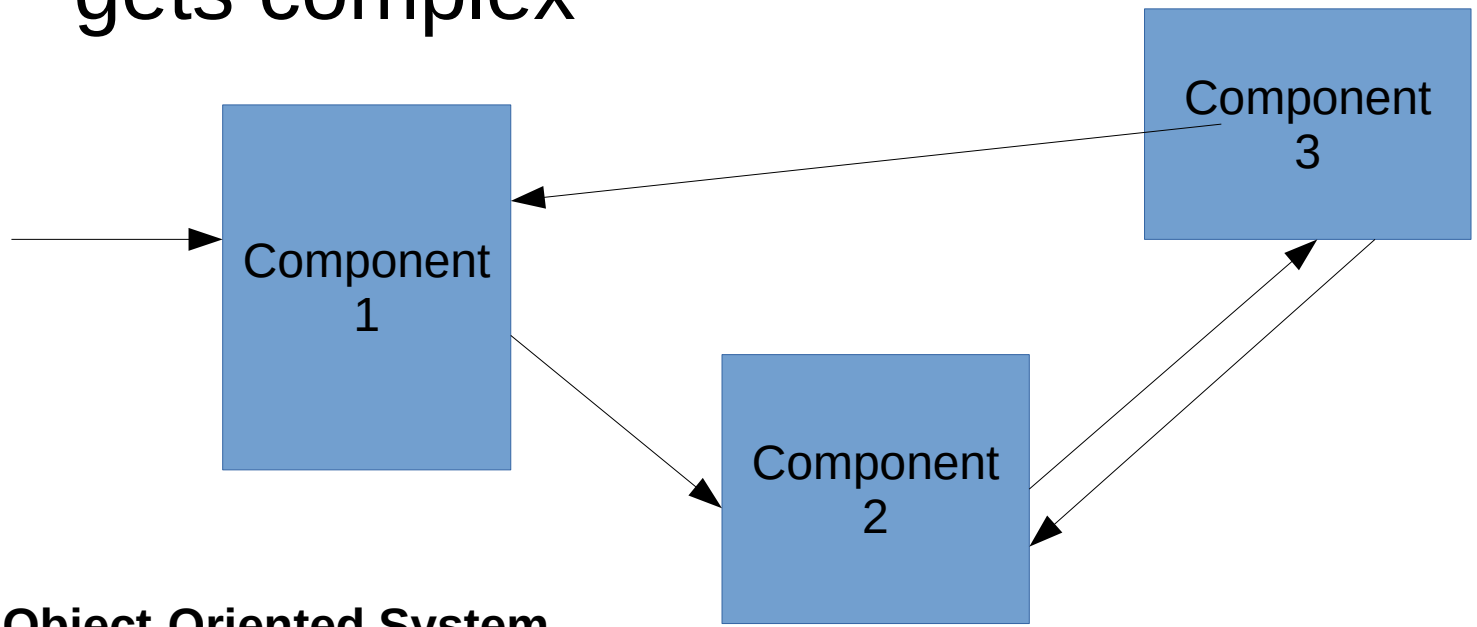
- No side effects.
- The same input produces the same output every time.

```
>>> f(5)
12
>>> f(5)
12
>>> f(6)
15
>>> f(5)
12
```



Why?

- Much simpler to think about when your system gets complex



Object-Oriented System

```
>>> c.reset()
0
>>> c.add(1)
1
>>> c.add(1)
2
>>> c.add(1)
3
```

Higher-order functions

- Functions that
  - takes functions as arguments, or
  - returns a function

compose

```
def square(x):
    return x*x
def plus_one(x):
    return x+1

def compose(f,g):
    def h(x):
        return f(g(x))

    return h
```

```
>>> square(10)
100
>>> plus_one(100)
101
>>> k = compose(plus_one, square)
>>> k(10)
101
>>> compose(square, plus_one)(10)
121
```

Lambda

```
def compose(f,g):
    return lambda x: f(g(x))
```

```
>>> square = lambda x: x*x
>>> square(10)
100
>>> (lambda x: x+1)(100)
101
>>> k = compose(lambda x: x+1, square)
>>> k(10)
101
```

Important higher-order functions

- map
- filter
- reduce (in functools)

Exercises 1

- occurs_in(x, lst)
  - returns True if **x** is in **lst**.
- myzip
  - myzip([1,2,3],[100,200,300]) => [(1,100), (2,200), (3,300)]
- are_same_list(lst1, lst2) --- use **“zip”**
  - are_same_list([1,2,3],[1,2,3]) => True
  - are_same_list([1,2],[1]) => False

Exercises 2

- apply_n_times(f, n)
  - takes function f and returns a function that takes x and apply f for n times.
  - e.g. apply_n_times(lambda x: x + 1, 10) returns a function that takes a number and add 10 to it
  - try to write with reduce

Exercise 3

- factorial(n)
  - returns n!
  - hint: you may want to use range(1,n+1) = [1,2,3,..,n] with reduce

Exercise 4

- polyev(f,x) – evaluate polynomial f at x, f(x)
  - f is list of coefficient of a polynomial, e.g., 3x^3 + 10x + 5 is written as [3,0,10,5]
  - e.g., polyev([2,1,1], 5) evaluate polynomial 2x^2 +x+1 at x=5

Partial functions

```
def add(x,y):
    return x+y
```

```
>>> from functools import partial
>>> g = partial(add, b=100)
>>> g(50)
150
```