Functional Programming

- PoPl

What is functional programming?

- (Yet another) programming paradigm
- In functional programming:
  - Output of functions depend only on input (no global variables)
  - Minimize use of while loops (for loops are easily converted to recursion)
  - Minimize use of variables
- Motivation:
  - Functional programs are easier verified for correctness
  - Functional programs can be more scalable
  - Functional programs are much better parallelizable
    - Map/Reduce framework is based on FP

Generating Lists (List Comprehensions)

- Recall the expression

  for x in X   #goes through all the elements of X which is list, tuple, set,…

- Python allows us to "materialize" the results in a list

  X = (3,2,1)

  [x for x in X] #generates a new list

  >> [3,2,1]

- Can be used with conditions

  [x for x in X if x >=2]

  >> [3,2]

- Can be used with multiple collections (constructs list of tuples)

  Y = ['a', 'b', 'c']

  [(x,y) for x in X for y in Y]

  >> [(3, 'a'), (3, 'b'), (3, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (1, 'a'), (1, 'b'), (1, 'c')]

Generating Lists (List Comprehensions)

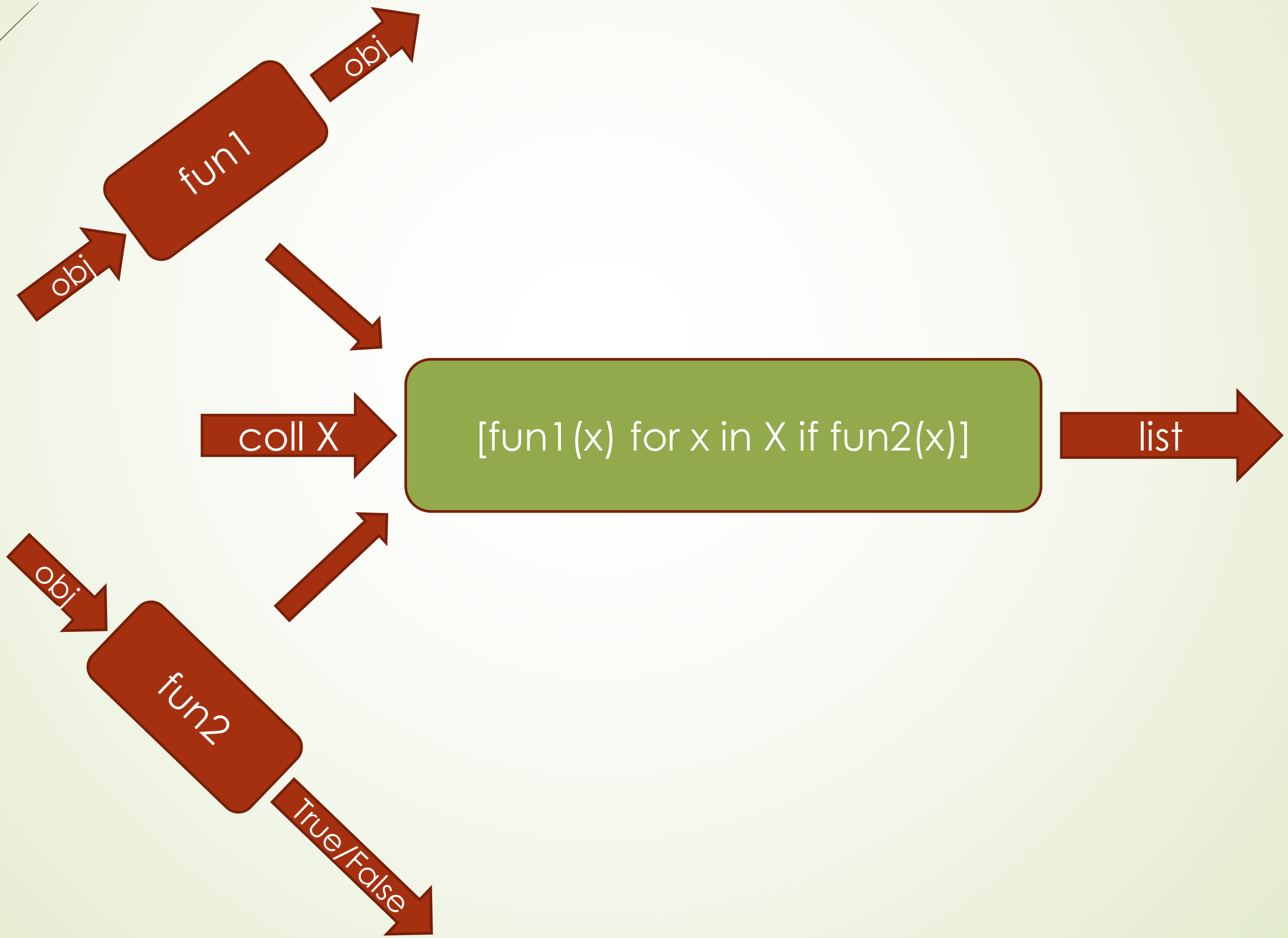
- Can use arbitrary expression (not just x)

```
x = [3,2,1]
[x**2 for x in X]
>> [9,4,1]
[fun(x) for x in X]                          #where fun(x) is a (scalar) function
nums = [int(x) for x in input().split()]  #used for reading input and converting to int
[x.method() for x in X]    #where method() is of a class x belongs to
```

- Can use any function that returns True/False under “if” part:

```
def is_even(x): return x%2 == 0
[x for x in X if is_even(x)]
>> [2]
```

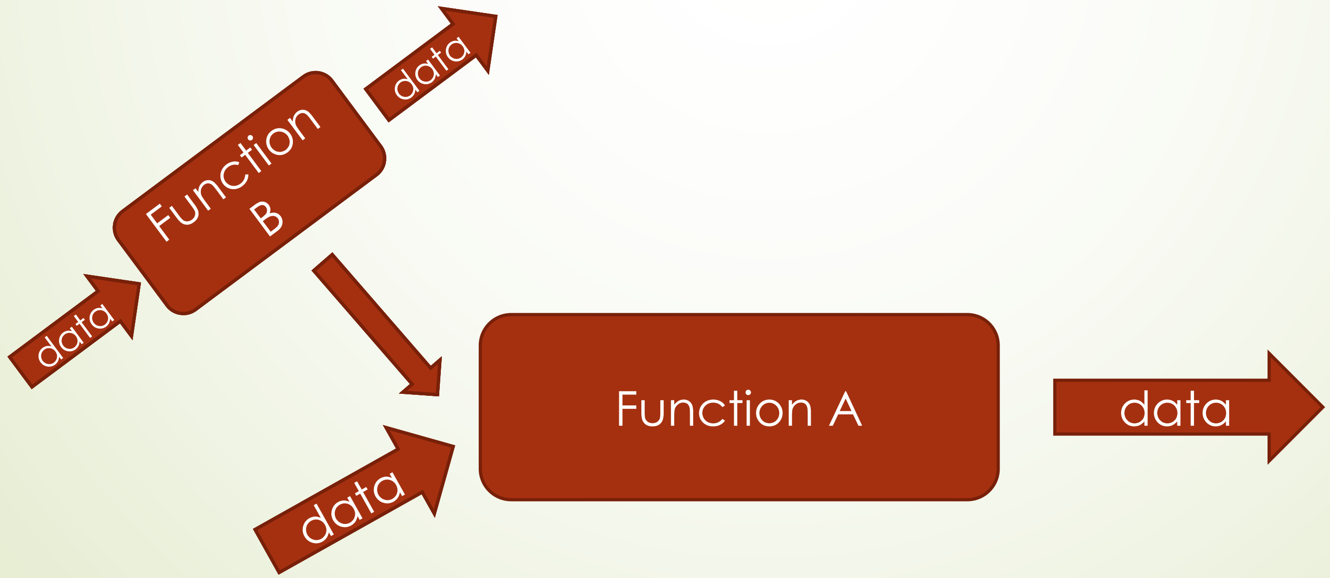
Generating Lists: General View



Higher Order Functions

- Classical functions

- Higher order functions



Higher Order Functions (cont.)

- Functions that take functions as arguments. Why?

```
def double(x):     #works on floats
    return 2*x
```

- If I need a function that sums doubles of the elements of vector:

```
def sum_double_vector(X):  #works on float lists
    sum = 0
    for x in X:
        sum+=double(x)
    return sum
```

- Suppose also

```
def square(x):   #works on floats
    return x*x
```

- If I need vector version, I similarly implement sum_square_vector(X)
- Redundancy! (later half(x), etc)

- Python supports HO-functions and allows an elegant non-redundant solution:

```
def sum_function_vector(f, X):
    sum = 0
    for x in X:
        sum+= f(x)
    return sum
Y = [1,2,3,4]
s1 = sum_function_vector(double,Y)
s2 = sum_function_vector(square,Y)
s3 = sum_function_vector(math.sqrt,Y)
….
```

Python tutor demo: https://goo.gl/4AfWwu

Higher Order Functions (cont.)

- Classical functions

- Higher order functions that return functions

Higher Order Functions (cont.)

- Functions that return functions. Why?

```
def double(x):     #works on floats
    return 2*x
```

- If I need vector version:

```
def vec_double(X):  #works on vects
    Y = []
    for x in X:
        Y.append(double(x))
    return Y
```

- Suppose also

```
def square(x):   #works on floats
    return x*x
```

- If I need vector version, I similarly implement vec_square(X)
- Redundancy! (later half(x), etc)

- Python supports HO-functions and allows an elegant non-redundant solution:

```
def vectorize(f):
    def new_func(X):
        Y = []
        for x in X:
            Y.append(f(x))
        return Y
    return new_func
vec_double = vectorize(double)
vec_square = vectorize(square)
…
Y = vec_double([1,2,3])
```

Iterators: Motivation

- Consider

  X = [1,3,7,8,…] #1B items

- Function

  def square(x): return x*x

- Tasks: print(True) if 9 is in [square(x) for x in X]
- Obvious solution

```
Y = [square(x) for x in X]
i = 0
while i < len(Y):
    if Y[i]==9:
        print(True)
        break
    i +=1
```

- Problem: waste time/memory to create (materialize) Y = [1,9,49,…] to access only first 2 elements of it

Iterators

- Allows to create elements of a collection 1 by 1 when they are needed instead of materializing the whole collection

```
X = [1,3,7,8,…]          #1B items
it_squares_X = (square(x) for x in X)     #creates an iterator instead of a list
while True:
    next_value = next(it_squares_X)
    if next_value == 9:
        print(True)
        break
```

- next(it_squares_X) says "get me the next item of Y = [square(x) for x in X]"
  - the next item is created in memory only when requested
  - Y is virtual

Iterators (cont)

- Iterators have limited functionality. Apart from next(), we what we can do with them is testing whether we are at the end by catching StopIteration exception:

```
it = (x in [0,1])
next(it)
>>0
next(it)
>>1
next(it)
>> Error: StopIteration Exception
```

- Also, Python provides convenient access to iterators via a for loop:

```
X = [1,3,7,8,…]            #1B items
it_squares_X = (square(x) for x in X)
for y in it_squares_X:
        if y == 9:
                print(True)
                break
```

Iterators (cont)

- From an iterator, we can materialize (almost) any form of collection (it takes time and memory)

```
Y = list(it_squares_X)
Y = set(it_squares_X)
Y = tuple(it_squares_X)
```

- Vice versa, we can get an iterator over any (materialized) collection

```
X = {1,2,3}
It_over_X = iter(X)
```

- Conversion between types of collections is done by rematerializing iterators

```
Y = list(X)      #converts set to list
#equivalent to
Y = list(iter(X))
```

- For collections of large size, convert as little as possible

Map, filter, and other built in functions

- There are frequently used forms of list comprehension/generation that are implemented via built in functions
- map(fun, collectionA, collectionB, …)
  - creates the sequence

    fun(collectionA[0], collectionB[0],…), fun(collectionA[0], collectionB[0],…),….

    and returns its iterator
  - def plus(x,y): return x+y

    X,Y = [1,2,3], [5,6,7]

    it = map(plus,X,Y)

    list(it)

    >> [6,8,10]
  - Instead of collection, can take an iterator
- filter(fun, collection)
  - def is_even(x): return x % 2 == 0

    X = [1,2,3,4,5]

    it = filter(is_even, X)

    list(it)

    >> [2,4]

Map, filter, and other built in functions (cont)

- enumerate(collection, start = 0)

  X = [‘a’, ‘b’, ‘c’]

  it = enumerate(X)

  list(it)

  >> [(0, ‘a’), (1,’b’), (2, ‘c’)]

- zip(collection1, collection2,…)

  X, Y = [‘a’, ‘b’, ‘c’], [0,1,2]

  list(zip(X,Y))

  >> [(‘a’, 0), (‘b’,1), (‘c’, 2)]

- sorted(collection), equivalent to

  iter(list(collection).sort())