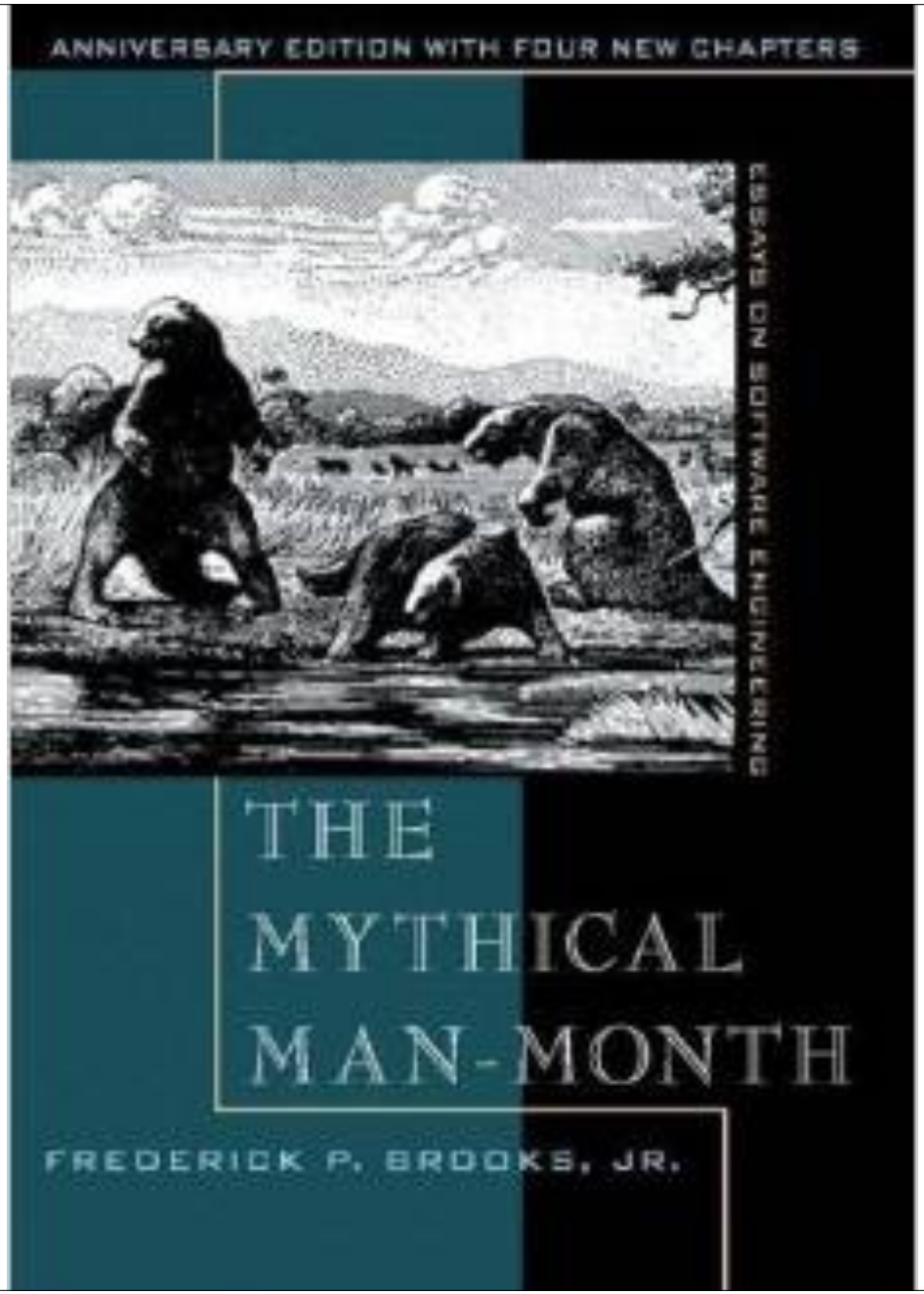
Python functional programming

Reuven M. Lerner, PhD
reuven@lerner.co.il

Functional programming

- A different way of thinking about programming
- A different way of writing programs
- A different way of approaching problems
- Often provides new, clean, elegant solutions…
- … but it requires that you think about things very differently.


ANNIVERSARY EDITION WITH FOUR NEW CHAPTERS
ESSAYS ON SOFTWARE ENGINEERING
THE MYTHICAL MAN-MONTH
FREDERICK P. BROOKS, JR.

Mathematical functions

- We know about mathematical functions:

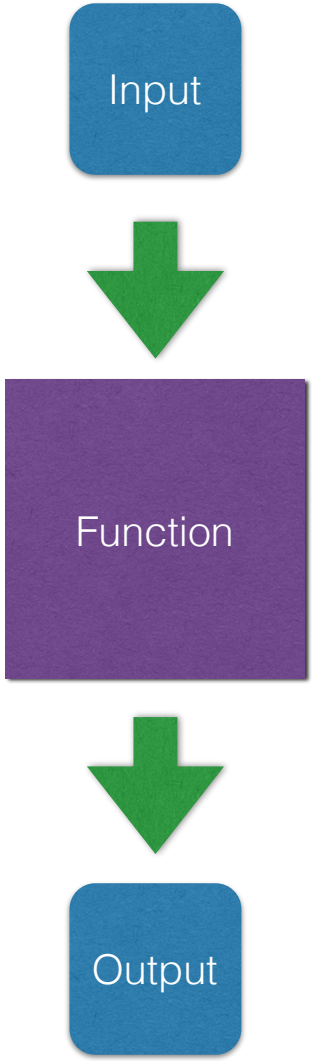
```
f = 2x
```

- What if we want to do something more complex? We write two functions, and nest them:

```
f = 2x

g = sin(x)

f(g(x))
```

How is this different?

- We don't change the input data
  - Rather, we produce new outputs based on those inputs
- We don't use assignment in our functions
  - Rather, we use more, small functions and nest them



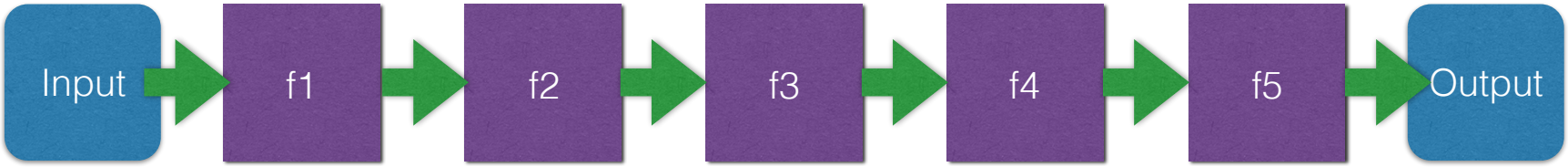
No external state changes in this function…

… and if you can avoid internal state, even better!

Why work this way?

- Simpler functions — easier to write and maintain
- Easer to test functions, and predict their results
- Easier to combine functions in new, different ways
- You can split work across threads, processors, or even machines without any ill effects
- Certain things can be expressed easily, elegantly

The FP perspective



OO or functional?

- Python supports both styles
- It's common to mix them in one program
- My current approach: Use objects in general, but functional techniques inside of methods

Sequences and iterables

- There are three built-in sequences in Python: strings, tuples, and lists
- Many other objects are "iterable," meaning that you can stick them into a "for" loop and get one item at a time
  - Examples: files and dicts
- This discussion mainly has to do with iterables

Where are iterables?

- Strings, lists, and tuples (obviously)
- Dicts, sets, and files
- Complex data structures: Lists of lists, lists of dicts, dicts of dicts (and others)
- Results from database queries
- Information from a network feed
- Many, many objects return iterables

Transformation

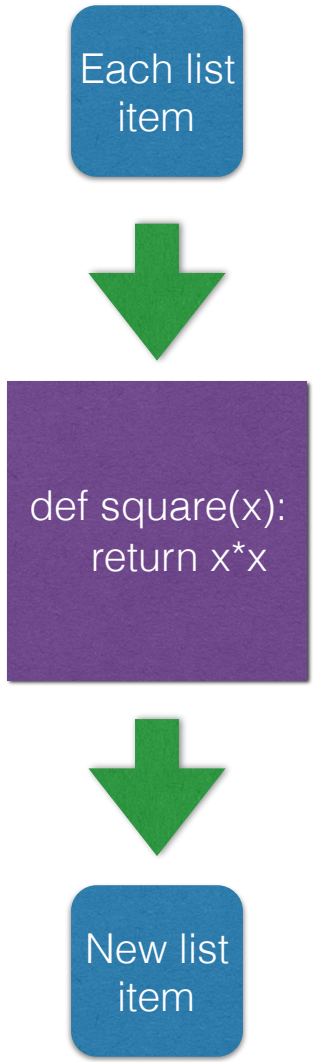
- You often want to turn one iterable into another
- For example, you might want to turn the list

```
[0,1,2,3,4]
```

- into the list

```
[0,1,4,9,16]
```

- We can *transform* the first list into the other by applying a Python function.


Each list item
def square(x):
return x*x
New list item

The usual solution

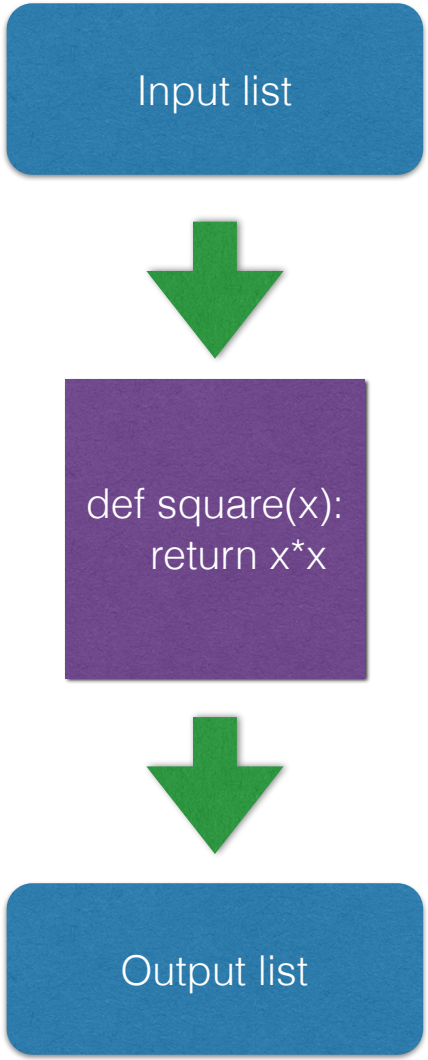
```
>>>> input = range(5)

>>>> def square(x):

        return x*x


>>> output = [ ]

>>> for x in input:

        output.append(square(x))


>>> output

[0, 1, 4, 9, 16]
```

What's wrong with this?

- Nothing is *wrong*.
- But functional programming looks at this, and says:
  - Why create a new variable ("output")?
  - Why are you building it, one step at a time?
  - Why are you modifying the state of "output"?
  - Why do it in such a clumsy way?

The elegant way

- "I want to apply my function, *square*, to each and every element of the input list, and get a list back."


Input list
def square(x):
return x*x
Output list

List comprehensions

- In Python, we do this with a "list comprehension":

```
[square(x) for x in range(5)]
```

- Or if you prefer:

```
[x*x for x in range(5)]
```

List comprehensions

- This expresses what we previously said:
  - I want a new list.
  - This new list should have the same number of elements as the input list.
  - Each element of the new list should be the result of applying square(x) to each element.

Many, many uses

- List comprehensions are powerful because they expression this idea in a compact, elegant form
- (And yes, it's a bit hard to read. I admit it!)
- Any time you have an iterable, and want to do something with each element, you likely want to use a list comprehension.

Ints to strings

- I can't say

```
', '.join(range(5))
```

- because str.join's input must contain strings.
- Solution:

```
', '.join([str(x) for x in range(5)])
```

Lowercase all words

- I can transform a sentence into all lowercase:

```
words = 'This is a sentence for my Python
class'.split()

[word.lower() for word in words]
```

- Or even:

```
' '.join([word.lower() for word in words])
```

Filenames to files

- I can get a list of filenames from os.listdir:

```
os.listdir('/etc')
```

- I can get a file object for each of these:

```
[open('/etc/' + filename)
 for filename in os.listdir('/etc')]
```

File contents

- If I want to get the names of users on my Unix system, I can say

```
[ line.split(":")[0]

  for line in open('/etc/passwd') ]
```

Pig Latin!

```
def plword(word):
    vowels = 'aeiou'
    if word[0] in vowels:
        return word + 'way'
    else:
        return word[1:] + word[0] + 'ay'
```

Translation

```
' '.join([plword(word)

          for word in open('column-215') ])
```

List of dicts

- If I have a list of dicts ("people), each of which looks like:

```
p1 = {'first_name':'Reuven',
'last_name':'Lerner', 'phone':'054-496-8405'}
```

- I can get each person's full name as follows:

```
[ person['first_name']+' '+person['last_name']

 for person in people ]
```

Comprehensions

- Once you start to use list comprehensions, you'll see opportunities for this kind of transformation, or "mapping," just about everywhere.
- Python uses iterables in a lot of places, which means that you have many, many opportunities to do this
- It's often worth turning your data into an iterable, so that you can put it inside of a list comprehension!

Creating sets

- We can create sets by passing set() an iterable
- So we can create a set with:

```
set([x*x for x in range(5)])
```

- In modern versions of Python, we can also say:

```
{x*x for x in range(5)}
```

- Curly braces give us a set — a *set comprehension*

Set comprehensions

- Create a set, based on any iterable
- Lots of uses:
  - Usernames
  - Filenames
- Anything you get that's non-unique, which you want to make unique, is a perfect candidate!

Dict comprehensions

- Why let sets have all of the fun?
- Use curly braces, just like a set comprehension — but then separate the two values with a colon (:), just like in a dictionary definition.

```
{ line.split(':')[0] : line.split(':')[2]

  for line in open('/etc/passwd')

  if line[0] != '#' }
```

You can…

```
>>> query_string = 'a=1&b=2&c=xyz'

>>> [item.split('=')

    for item in query_string.split('&')]

[['x', '1'], ['y', '2'], ['z', 'abc']]


>>> dict([item.split('=')

    for item in query_string.split('&')])

{'a': '1', 'b': '2', 'c': 'xyz'}
```

… but even better

```
>>> query_string = 'a=1&b=2&c=xyz'

>>> { item.split('=')[0] : item.split('=')[1]

    for item in query_string.split('&') }

{'a': '1', 'b': '2', 'c': 'xyz'}
```

Filtering

- By default, a comprehension returns a collection with the same number of elements as its input.
- However, we can add an "if" statement to the end, which filters the output.
- Only those items for which the expression returns True" will be output

Filtering

- I can say:

```
[x*x for x in range(10) if x%2]
```

- That allows

```
[1, 9, 25, 49, 81]
```

Loops vs. comprehensions

- Many people ask me why they should use list comprehensions, when we already have "for" loops.
- The answer: These are completely different things!

Who cares?

- Comprehensions let you create lists, dictionaries, and sets quickly and easily.
- Moreover, they let you *map* the values from one collection to another
- Indeed, comprehensions are the modern incarnations of two very old functions, "map" and "filter"

Immutable data

- We know that Python has both mutable and immutable data structures
- In functional programming, we pretend that our data structures are immutable, even if they aren't
- But if we want to enforce immutable data, we can do it — typically using tuples

Dictionary comprehensions

- Just like a list comprehension, but with curly braces and name:value as the output

```
{ word:word.lower() for word in 'ABC DEF GHI'.split() }

{ 'ABC': 'abc', 'DEF': 'def', 'GHI': 'ghi'}
```

Set comprehensions

- Set comprehensions!

```
{ word.lower() for word in 'ABC DEF GHI'.split() }

    set(['abc', 'ghi', 'def'])
```

Getting fancier

- Sometimes we want to perform more than simple list transformations
- For that to work, it'll sometimes come in handy to use functions. But why define a new function if I'm just going to use it once?

How does sort work?

- Sorting via list.sort()
- The function that “sort” uses to compare items is called “cmp”, and is built into Python.
- Returns -1, 0, or 1 to show sort order
- Any function that returns -1, 0, or 1 can also be used

Changing sort's behavior

- You can change sort's behavior by passing it a parameter (named "cmp").
- Pass it a function that takes two arguments, and returns -1, 0, or 1, depending on which takes precedence

Sort example

```
mylist = ['aaa', 'AAA', 'bbb', 'BBB',

          'ccc', 'CCC']

print "Without sorting:"

print mylist

print "With sorting:"

mylist.sort()

print mylist
```

Sort example, continued

```
def insensitive_compare(x, y):

    return cmp(x.lower(), y.lower())



print "With sorting, ignoring case:"

mylist.sort(cmp=insensitive_compare)

print mylist
```

Why define a function?

- In the previous example, I defined a function (insensitive_comparison) only to pass it to sort.
- I could instead define an anonymous function, one that is a function in every respect, but doesn't have a name.

Anonymous functions?

```
lambda x: x * x
```

- A function object, but without a name
- We can also say:

```
square = lambda x: x*x

square(5)
```

lambda

- Declares the start of an anonymous function definition
- Can take one or more parameters
- Must return an expression
- Must fit within one line

Case-insensitive sort

- “sort” in modern Python versions supports an optional “key” parameter
- This is a function (named or anonymous) that is invoked on each element. So:

```
mylist.sort(key=str.lower)

mylist.sort(key=lambda item: item['a'])
```

What about immutables?

- Functional sort: sorted(), which works on tuples and strings, as well as lists:

```
>>> sorted('abcdef')

['a', 'b', 'c', 'd', 'e', 'f']

>>> sorted('fgsadfaggz')

['a', 'a', 'd', 'f', 'f', 'g', 'g', 'g',
's', 'z']
```

Use lambda with sorted

```
>>> t = (346, 23, 40, 1000)

>>> sorted(t, cmp=lambda x,y: cmp(y,x))

[1000, 346, 40, 23]
```

Functional programming!

- Work with lists and other collections
- Apply a function to the entire collection
- Get a new collection (perhaps smaller) in return, based on applying the function
- Use lambda to define short, throwaway functions
- As few side effects (i.e., assignments) as possible

reversed

```
tup = (80, 3, 2, 200, 15, 9)

tup.reverse()    # Error

reversed(tup)    # Works!

print reversed(tup)   # Huh?

tuple(reversed(tup))  # Um, OK
```

zip

```
x = range(10)

y = range(10,20)

zip(x,y)

  [(0, 10), (1, 11), (2, 12),

   (3,  13), (4, 14), (5, 15),

   (6, 16), (7, 17), (8, 18),

   (9, 19)]
```