STATS 701
Data Analysis using Python

Lecture 10: Functional Programming II: `functools`

What about reduce?

Saw map and filter last lecture, but we can't have MapReduce without **reduce**

```
sum(range(1,11))
```

55

```
import functools
functools.reduce(lambda x,y: x+y, range(1,11))
```

55

```
functools.reduce(lambda x,y: x*y, range(1,11))
```

3628800

Reduce operations **reduce** an iterator (i.e., a sequence) to a single element. Sum is a good example of a reduce function.

`functools` contains a bunch of useful functional programming functions, including reduce.

`functools.reduce` takes a function and and iterator and performs a reduce operation on the iterator using the function.

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

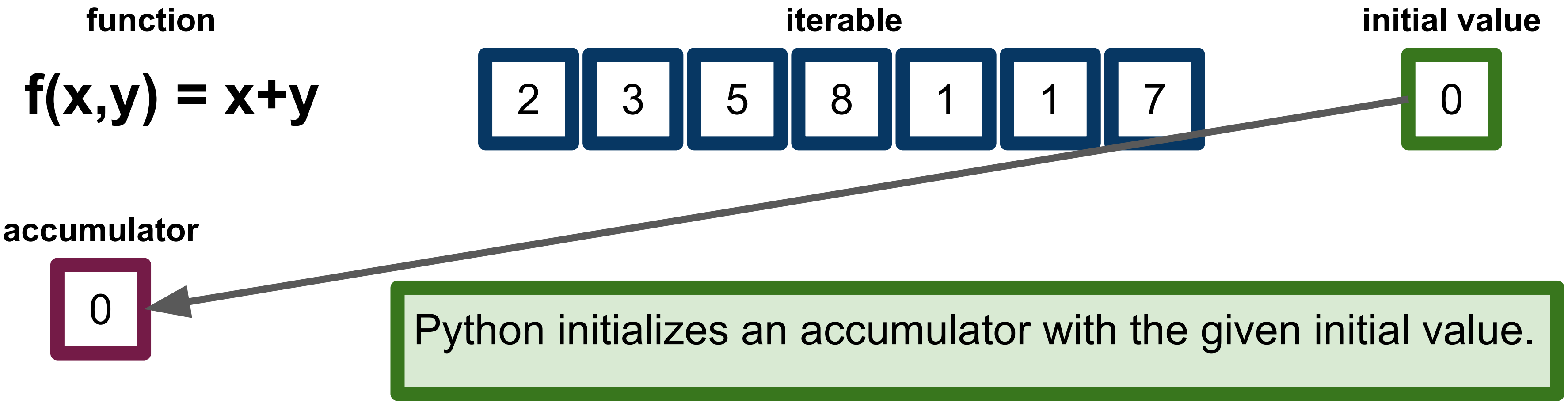
2	3	5	8	1	1	7

initial value

0

Reduce operations

Three fundamental pieces:



Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

0

Now, Python repeatedly updates the accumulator, with
`acc += f(acc,y)`
where `y` traverses the iterable

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

0

$$f(0,2) = 2$$

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

2

f(0,2) = 2

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

2

$$f(2,3) = 5$$

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

5

f(2,3) = 5

Reduce operations

Three fundamental pieces:

function

$$f(x,y) = x+y$$

iterable

2	3	5	8	1	1	7

initial value

0

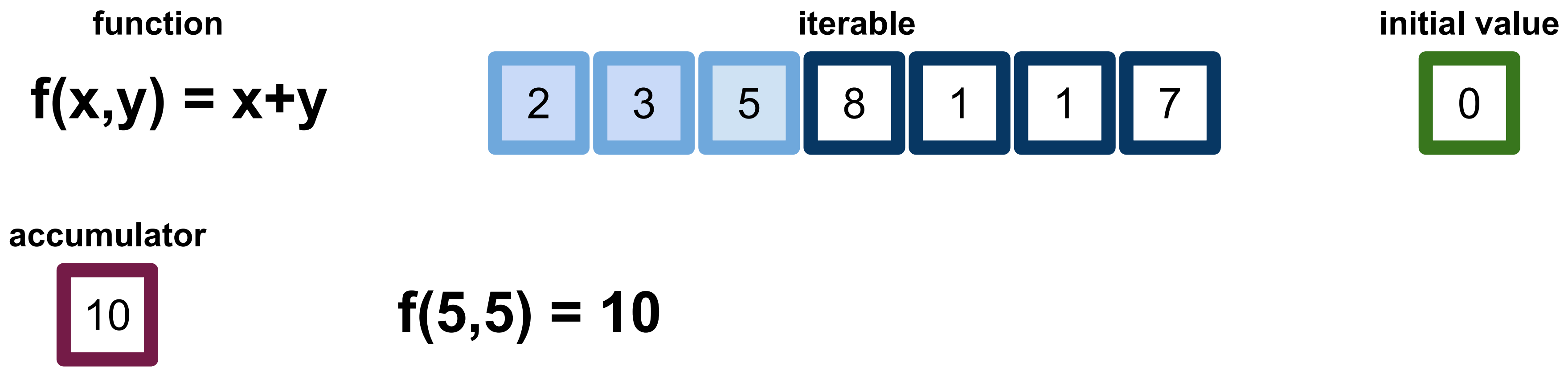
accumulator

5

$$f(5,5) = 10$$

Reduce operations

Three fundamental pieces:



Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2 3 5 8 1 1 7

initial value

0

accumulator

10

...and so on.

Reduce operations

Three fundamental pieces:

function

f(x,y) = x+y

iterable

2	3	5	8	1	1	7

initial value

0

accumulator

27

Once Python gets a `StopIteration` error indicating that the iterator has no more elements, it returns the value in the accumulator.

Reduce operations

Three fundamental pieces:

function	iterable	initial value
f(x,y) = x+y	2 3 5 8 1 1 7	0

```
functools.reduce(lambda x,y:x+y, range(10), 0)
```

```
45
```

```
functools.reduce(lambda x,y:x+y, range(10))
```

```
45
```

```
functools.reduce(lambda x,y:x+y, [2.71828])
```

```
2.71828
```

If the initial value isn't supplied, Python initializes the accumulator as `acc = f(x,y)` where `x` and `y` are the first two elements of the iterator. If the iterator is length 1, it just returns that element. All told, it's best to always specify the initial value, except in very simple cases (like these slides).

Reduce operations

Three fundamental pieces:

function

$f(x,y) = x+y$

iterable

2	3	5	8	1	1	7

initial value

0

Warning: if the iterator supplied to `reduce` is empty, then we really do need the initial value!

```
1  functools.reduce(lambda x,y:x+y,[])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-116-a9df9cc43559> in <module>()
----> 1 functools.reduce(lambda x,y:x+y,[])

TypeError: reduce() of empty sequence with no initial value
```

Reduce in Python

`reduce` is not included as a built-in function in Python, unlike `map` and `filter`

Because developers felt that reduce is not “Pythonic”

The argument is that reduce operations can always be written as a for-loop:

```
import functools
functools.reduce(lambda x,y: x+y, range(10))
```

```
45
```

```
acc = 0
for i in range(10):
    acc += i
acc
```

```
45
```

Reduce in Python

`reduce` is not included as a built-in function in Python, unlike `map` and `filter`

- Because developers felt that reduce is not "Pythonic"

The argument is that reduce operations can always be written as a for-loop:

```
import functools
functools.reduce(lambda x,y: x+y, range(10))
```

```
45
```

```
acc = 0
for i in range(10):
    acc += i
acc
```

```
45
```

This criticism is mostly correct, but we'll see later in the course when we cover MapReduce that there are cases where we really do want a proper reduce function.

Reduce in Python

```
import random
numbers = [random.randint(1,1000) for _ in range(100)]
functools.reduce(lambda x,y:x if x > y else y, numbers)
```

```
989
```

All of the standard reduce-like functions are easily reimplemented with reduce statements, like this example, with `max`. Note the use of Python's in-line conditional statement.

```
def tuple_add(x,y):
    if len(x) != len(y):
        raise TypeError('Tuple lengths must agree')
    r = len(x)*[0]
    for i in range(len(x)):
        r[i] = x[i] + y[i]
    return tuple(r)
functools.reduce(tuple_add, [(1,2),(1,3),(2,5),(3,7),(5,11)])
```

```
(12, 28)
```

More often, one has to implement the pairwise function. For example, here we have implemented a function for entrywise addition of tuples.

Note: there are "more functional" ways to do this. Since tuples are themselves iterable, we could write a clever "function of functions" to do this more gracefully. More on this soon.

Related: `itertools.accumulate`

```
1 itertools.accumulate(range(1,10), lambda x,y:x+y)
```

```
<itertools.accumulate at 0x10a6aa348>
```

`itertools.accumulate` performs a reduce operation, but it returns an iterator over the partial “sums” of its argument. Returns an empty iterator if argument is empty.

```
1 list(itertools.accumulate(range(1,10), lambda x,y:x+y))
```

```
[1, 3, 6, 10, 15, 21, 28, 36, 45]
```

```
1 list(itertools.accumulate([(1,2),(1,3),(2,5),(3,7),(5,11)], tuple_add))
```

```
[(1, 2), (2, 5), (4, 10), (7, 17), (12, 28)]
```

I put “sums” in quotes, because of course the function need not be addition. The point is that we get an iterator over the values of the accumulator at each step of the reduce operation.

Python `operator` module

It's awfully annoying to have to write `lambda x,y:x+y` all the time

```
functools.reduce(lambda x,y:x*y, range(1,10))
```

```
362880
```

Here is what we'd *like* to write, but of course it's a syntax error.

```
functools.reduce(*,range(1,10))
```

```
  File "<ipython-input-90-461403074121>", line 1
    functools.reduce(*,range(1,10))
                     ^
SyntaxError: invalid syntax
```

`operator.mul` gives us `*`, but as a function, just as though we wrote a lambda expression.

```
import operator
functools.reduce(operator.mul,range(1,10))
```

```
362880
```

`operator` includes many other functions:

Math: `add()`, `sub()`, `mul()`, `abs()`, etc.

Logic: `not_()`, `truth()`.

Bitwise: `and_()`, `or_()`, `invert()`.

Comparison: `eq()`, `ne()`, `lt()`, `le()`, etc.

Identity: `is_()`, `is_not()`.

https://docs.python.org/3/library/operator.html#module-operator

More functional patterns: `functools`

`functools` module provides a number of functional programming constructions

```
import functools
pow2 = functools.partial(math.pow, 2)
pow2
```

```
functools.partial(<built-in function pow>, 2)
```

`functools.partial` takes a function and a set of arguments to pass to the function. Returns a function with some of its arguments “fixed”.

```
list(map( pow2, range(10) ))
```

```
[1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0, 512.0]
```

So in this case, it’s like we got a new function, `pow2(x) == math.pow(x,2)`

```
def my_pow(x=1, y=1):
    return math.pow(x,y)
my_square = functools.partial(my_pow, y=2)
list(map( my_square, range(10) ))
```

```
[0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0]
```

`functools.partial` also lets us pass keyword arguments.

Higher-order functions and currying

`functools.partial` takes a function (and other stuff), returns a function

Called a **higher-order function**

In most other languages, Python's `functools.partial` is called **currying**

```
f = lambda x,y,z : x*y*z
curry1 = functools.partial(f,2)
curry2 = functools.partial(curry1,3)
curry2(4)
```

```
24
```

`curry1` takes two arguments, returns their product times 2.

`curry2` takes one argument `z`, returns `2*3*z` (reminder: partial fills positional arguments in order).

```
par = functools.partial(f,2,3)
par(4)
```

```
24
```

Equivalently, just pass both arguments in one call to partial.

Currying is named after logician Haskell Curry
https://en.wikipedia.org/wiki/Currying

Pure functions, again

Recall that a **pure function** was a function that did not have any side effects

Pure functions are especially important in functional programming

- A pure function is really a function (in the mathematical sense)
- Given the same input, it always produces the same output
  - (And doesn't change the state of our program!)

```python
a = 0
def increment_mod():
    global a
    a += 1
def increment_pure(x):
    return x+1
```

This function is a modifier.
It has side effects.

This is a pure function.

Pure functions, again

Recall that a **pure function** was a function that did not have any side effects

Pure functions are especially important in functional programming

- A pure function is really a function (in the mathematical sense)
- Given the same input, it always produces the same output
  - (And doesn't change the state of our program!)

```
a = 0
def increment_mod():
    global a
    a += 1
def increment_pure(x):
    return x+1
```

Pure functions are also crucial to having **immutable data**. Think about processing the observations in a data set. We don't want to change the original data file in the process of our analysis! We want to be able to write a pipeline, in which we pass data from one function to another, producing a transformed version of the data at each step.

Pure functions and higher-order functions

Pure functions are useful because they are very naturally composed and arise naturally in map/reduce frameworks

```python
def compose(*funcs):
    '''Return a new function that is the
    composition of the argument functions.'''
    def inner(data, funcs=funcs):
        result = data
        for f in reversed(funcs):
            result = f(result)
        return result
    return inner
f = lambda x: x**2
g = lambda x: x+1
# compose(g,f) == g(f(x)) == x**2 + 1
list(map(compose(g,f), range(10)))
```

```
[1, 2, 5, 10, 17, 26, 37, 50, 65, 82]
```

Here's a good example of a higher-order function. `compose` takes functions and produces a new function.

Returning a function is okay, because Python has first-class functions.

You can see why we prefer pure functions for these kinds of tricks. If `f` and/or `g` had side effects, this would be a big mess!

Example credit: D. Mertz, *Functional Programming in Python*

Functional vs Object-oriented Programming

```
class LetterCounter():
    '''Counts letters in a text stream'''
    def __init__(self, letter):
        self.letter=letter
        self.count=0
    def increment(self):
        self.count+=1
    def process_file(self, filename):
        with open(filename, 'r') as f:
            for line in f:
                self.process_line(line)
    def process_line(self, line):
        for x in line:
            if x==self.letter:
                self.increment()
    def get_count(self):
        return self.count
lc = LetterCounter('e')
fname = '/Users/keith/Downloads/mobydick.txt'
lc.process_file(fname)
lc.get_count()
```

```
118501
```

Of course, I'm exaggerating the complexity of this object here, but this really is what object-oriented code ends up looking like in the wild.

Contrast that with the simplicity of this functional version of the same letter-counting operation.

```
def count_letter(fname, letter):
    with open(fname, 'r') as f:
        return(sum([c==letter
                    for line in f
                    for w in line
                    for c in w]))
count_letter(fname, 'e')
```

```
118501
```

Why use functional programming?

Some problems are especially well-suited to this paradigm

Example: quicksort

```
def quicksort(t):
    if len(t) <= 1:
        return t # list is already sorted.
    else:
        pivot = t[0]
        return(quicksort([x for x in t if x < pivot]) +
                     [x for x in t if x==pivot] +
                    quicksort([x for x in t if x > pivot]))
quicksort([3,5,4,2,1,6,5,7,4,0,0,2,4])
```

```
[0, 0, 1, 2, 2, 3, 4, 4, 4, 5, 5, 6, 7]
```

See the quicksort Wikipedia page for examples of what this looks like when written in a non-functional style.

https://en.wikipedia.org/wiki/Quicksort

A note on recursion in Python: tail call optimization

M. R. Cook, *A Practical Introduction to Functional Programming*:
“Tail call optimisation is a programming language feature. Each time a function recurses, a new stack frame is created. A stack frame is used to store the arguments and local values for the current function invocation. If a function recurses a large number of times, it is possible for the interpreter or compiler to run out of memory. Languages with tail call optimisation reuse the same stack frame for their entire sequence of recursive calls. Languages like Python that do not have tail call optimisation generally limit the number of times a function may recurse to some number in the thousands.”

Python doesn’t have tail call recursion, so some functional programing patterns simply aren’t well-suited if we may encounter many thousands of layers of recursion. Recall our memoized function for computing the Fibonacci numbers.

Declarative Programming

Describe what the program **should do**, rather than how it does it

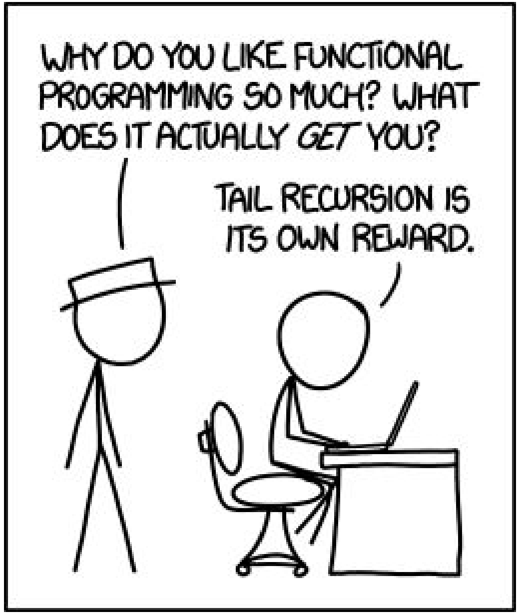
- Implementation details are left up to the language as much as possible

In contrast to **imperative/procedural programming**

- Sequence of statements describes **how** program should proceed
- Most programming you have done in the past is procedural
- Program consists of subroutines that get called, change state of program

> Don’t worry too much about these distinctions. Most languages are a mix of them, and no single approach is a silver bullet.
> **Different applications call for different programming paradigms.**

Congratulations! You know enough functional programming to get the joke in this xkcd comic!



Alt-text: Functional programming combines the flexibility and power of abstract mathematics with the intuitive clarity of abstract mathematics.

Readings (this lecture)

Required:

Python `functools` documentation

https://docs.python.org/3/library/functools.html

A. M. Kuchling. *Functional Programming HOWTO*

https://docs.python.org/3/howto/functional.html

Recommended:

M. R. Cook. *A Practical Introduction to Functional Programming*

https://maryrosecook.com/blog/post/a-practical-introduction-to-functional-programming

D. Mertz. *Functional Programming in Python*.

http://www.oreilly.com/programming/free/functional-programming-python.csp

Readings (next lecture)

Required:

Numpy quickstart tutorial:

https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

Pyplot tutorial:

http://matplotlib.org/tutorials/introductory/pyplot.html#sphx-glr-tutorials-introductory-pyplot-py

Recommended:

SciPy tutorial: https://docs.scipy.org/doc/scipy/reference/tutorial/index.html

Pyplot API: http://matplotlib.org/api/pyplot_summary.html

The Visual Display of Quantitative Information by Edward Tufte

Visual and Statistical Thinking: Displays of Evidence for Making Decisions by Edward Tufte This is essentially a reprint of Chapter 2 of the book above.