STATS 701 Data Analysis using Python

Lecture 9: Functional Programming I: itertools

Functional Programming

In the last few lectures, we saw ideas from object oriented programming "Everything is an object" Every operation is the responsibility of some class/object

Use side effects to our advantage (e.g., modifying attributes)

In **functional programming**, functions are the central concept, not objects "Everything is a function", "data is immutable" Avoid side effects at all costs Use pure functions (and "meta-functions") as much as possible

Iterators (or their equivalents) become hugely important

An iterator is an object that represents a "data stream"

Supports method ___next__():

returns next element of the stream/sequence raises StopIteration error when there are no more elements left

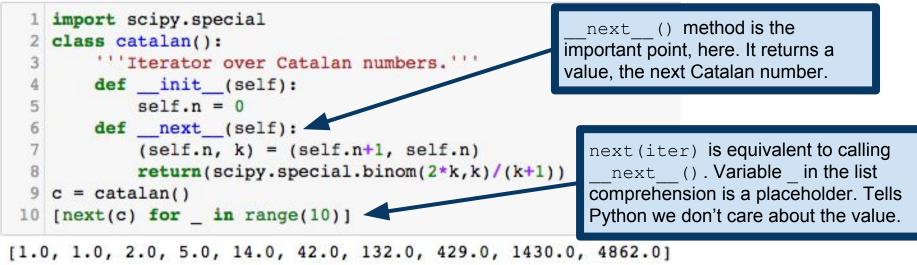
Catalan numbers show up a lot in counting problems. <u>https://en.wikipedia.org/wiki/Catalan_number</u>

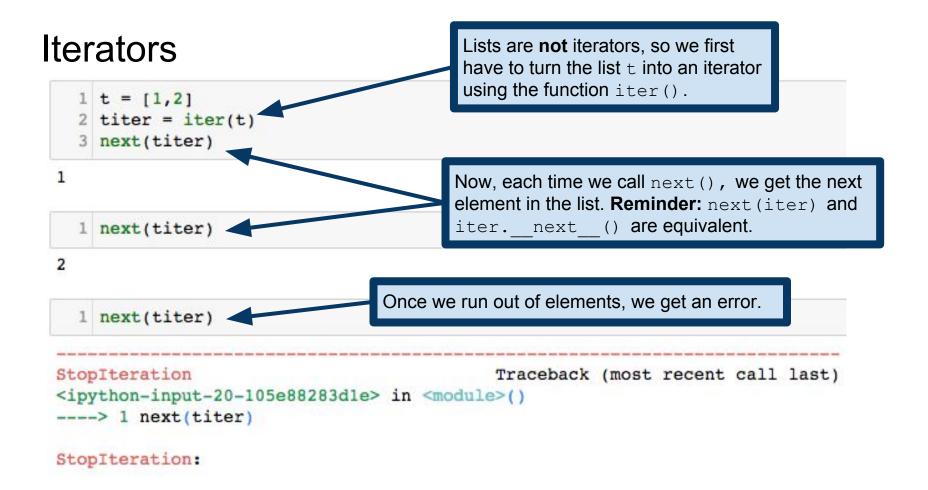
An iterator is an object that represents a "data stream"

```
Supports method ___next__():
```

returns next element of the stream/sequence

raises StopIteration error when there are no more elements left





| Iterators | Lists are not iterators, but we can turn a list into an iterator by calling iter() on it. Thus, lists are iterable , meaning | |
|--|--|--|
| <pre>1 t = [1,2] 2 titer = iter(t) 3 next(titer)</pre> | that it is possible to obtain an iterator over their elements. https://docs.python.org/3/glossary.html#term-iterable | |
| 1 | From the documentation: "When an iterable object is passed as an argument to the built-in function <u>iter()</u> , it returns an iterator for | |
| 1 next(titer) | the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call | |
| 2 | <u>iter()</u> or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop." | |
| <pre>1 next(titer)</pre> | | |

StopIteration

```
Traceback (most recent call last)
```

```
<ipython-input-20-105e88283dle> in <module>()
```

----> 1 next(titer)

StopIteration:

You are already familiar with iterators from previous lectures. When you ask Python to traverse an object obj with a for-loop, Python calls iter(obj) to obtain an iterator over the elements of obj.

1 t = [1,2,3]
2 for x in t:
3 print(x)
4 print()
5 for x in iter(t):
6 print(x)

Iterators

3

23

These two for-loops are equivalent. The first one hides the call to iter() from you, whereas in the second, we are doing the work that Python would otherwise do for us by casting t to an iterator.

You are already familiar with iterators from previous lectures. When you ask Python to traverse an object obj with a for-loop, Python calls iter(obj) to obtain an iterator over the elements of obj.

1 t = [1,2,3]
2 for x in t:
3 print(x)
4 print()
5 for x in iter(t):
6 print(x)

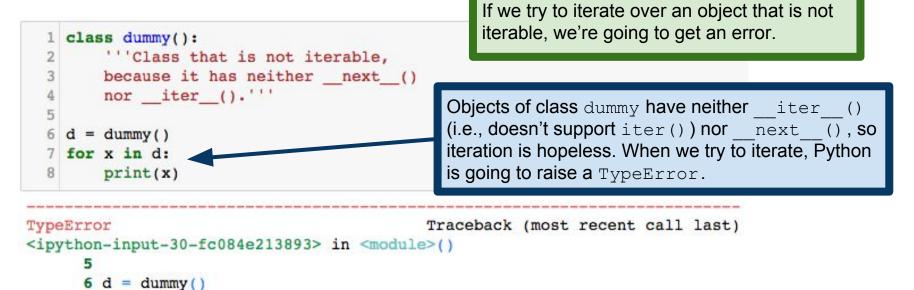
These two for-loops are equivalent. The first one hides the call to iter() from you, whereas in the second, we are doing the work that Python would otherwise do for us by casting t to an iterator.

Apropos a question from Jarvis a few weeks ago: "There is a subtlety when the sequence is being modified by the loop (this can only occur for mutable sequences, i.e. lists). An internal counter is used to keep track of which item is used next, and this is incremented on each iteration. When this counter has reached the length of the sequence the loop terminates. This means that if the suite deletes the current (or a previous) item from the sequence, the next item will be skipped (since it gets the index of the current item which has already been treated). Likewise, if the suite inserts an item in the sequence before the current item, the current item will be treated again the next time through the loop."

Iterators

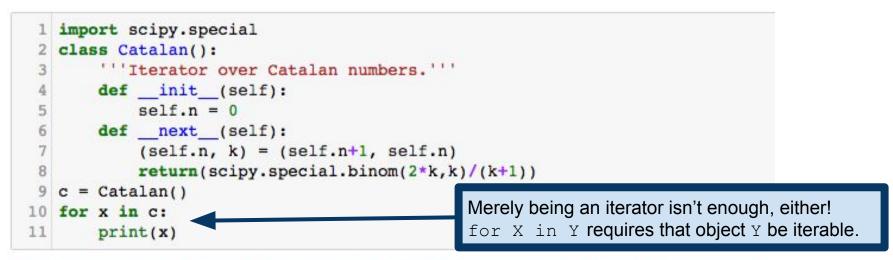
----> 7 for x in d:

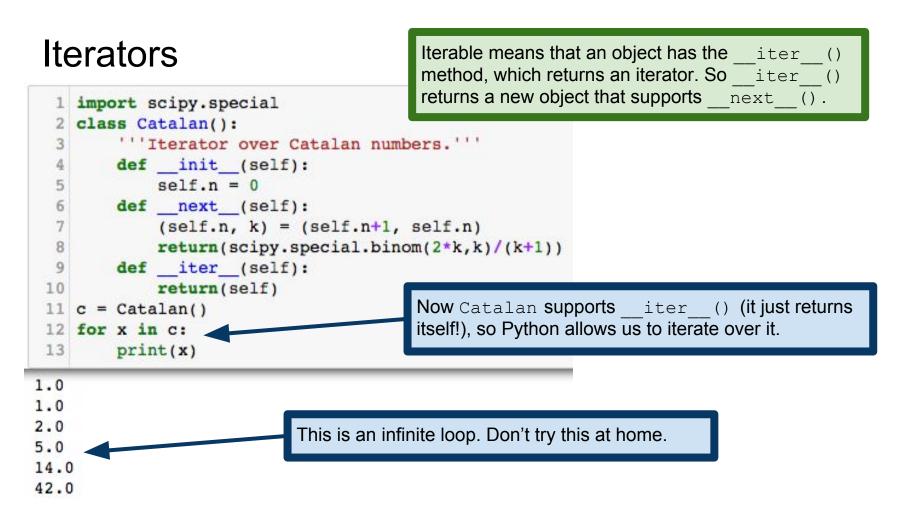
8

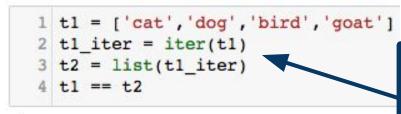


TypeError: 'dummy' object is not iterable

print(x)







We can turn an iterator *back* into a list, tuple, etc. **Caution:** if you have an iterator like our Catalan example earlier, this list is infinite and you'll just run out of memory.

True

```
1 t1 is t2
```

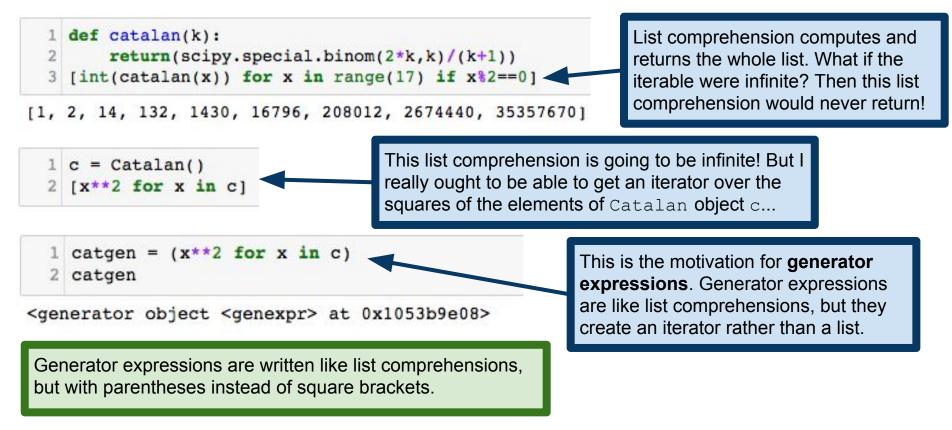
False

Many built-in functions work on iterators. e.g., max, min, sum, work on any iterator (provided elements support the operation); in operator will also work on any iterator

Warning: Once again, care must be taken if the iterator is infinite.

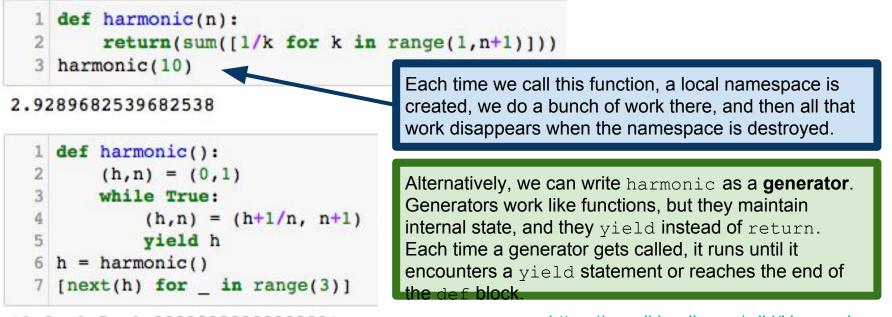
List Comprehensions and Generator Expressions

Recall that a list comprehension creates a list from an iterable



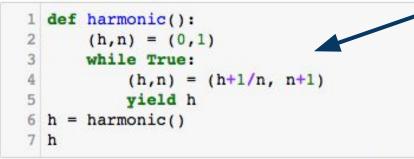
Related to generator expressions are generators

Provide a simple way to write iterators (avoids having to create a new class)



[1.0, 1.5, 1.833333333333333333]

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



<generator object harmonic at 0x1053b9fc0>

1 next(h)
1 0

1.0

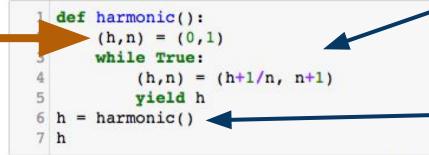
| 1 | next(h) | | |
|---|---------|--|--|
|---|---------|--|--|

1.5

1 next(h)

1.83333333333333333333

Python sees the yield keyword and determines that this should be a generator definition rather than a function definition.



<generator object harmonic at 0x1053b9fc0>

1 next(h)

1 next(h)

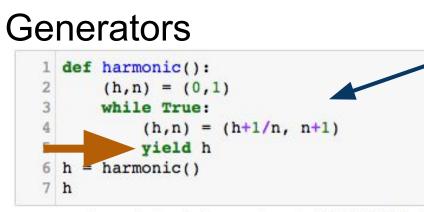
1.5

1 next(h)

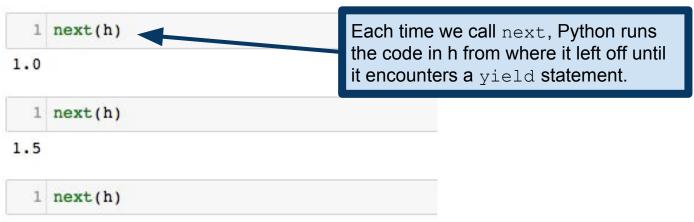
1.833333333333333333333

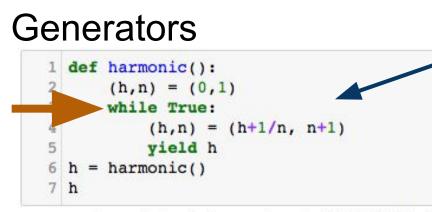
Python sees the yield keyword and determines that this should be a generator definition rather than a function definition.

Create a new harmonic generator. Inside this object, Python keeps track of where in the def code we are. So far, no code has been run.



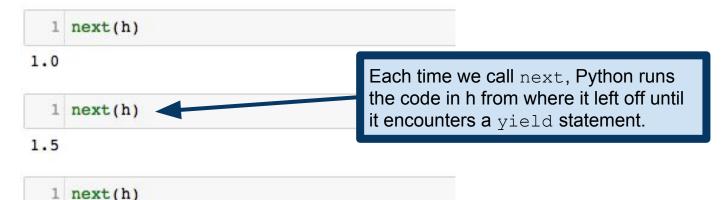
<generator object harmonic at 0x1053b9fc0>

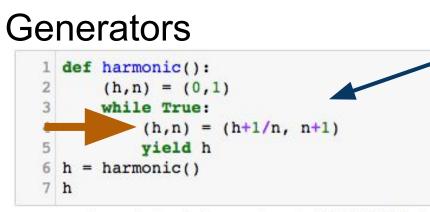




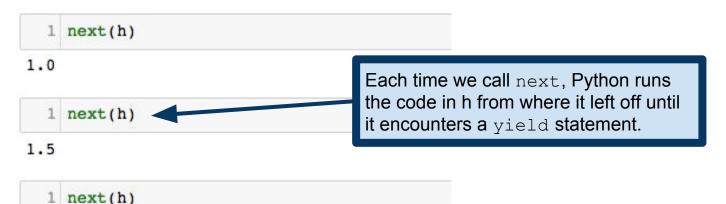
<generator object harmonic at 0x1053b9fc0>

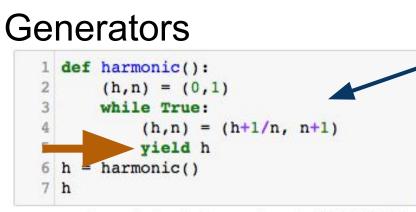
Python sees the yield keyword and determines that this should be a generator definition rather than a function definition.



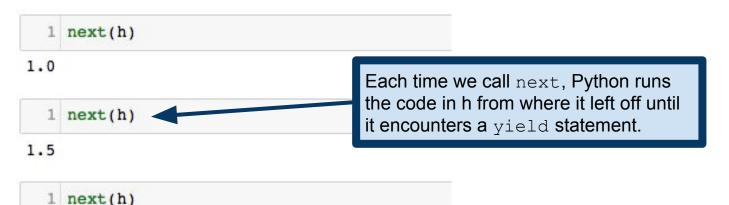


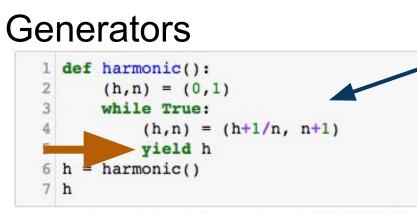
<generator object harmonic at 0x1053b9fc0>





<generator object harmonic at 0x1053b9fc0>





<generator object harmonic at 0x1053b9fc0>

| 1 | next(h) | |
|-----|---------|--|
| 1.0 | | If/when we run out of yield statements (i.e., because we reach the end of the definition block), |
| 1 | next(h) | the generator returns a StopIteration error, |
| 1.5 | | as required of an iterator (not shown here). |
| 1 | next(h) | |

Generators supply a few more bells and whistles Ability to pass values *into* the generator to modify behavior Can make generators both produce and consume information **Coroutines** as opposed to **subroutines**

See generator documentation for more:

https://docs.python.org/3/reference/expressions.html#generator-iterator-methods

Map and Filter

Recall:

map operation applies a function to every element of a sequence Yields a new, transformed sequence

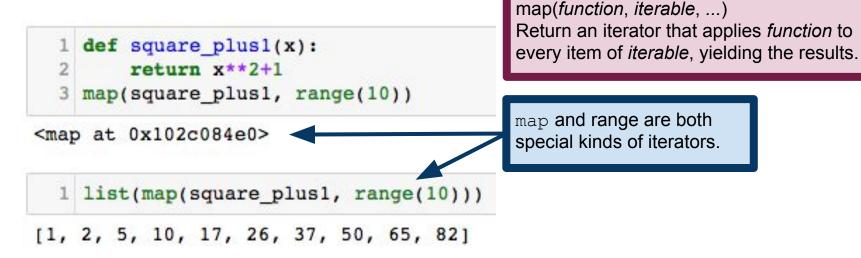
filter operation removes from a sequence all elements failing some condition Again, yields a new, filtered sequence

Мар

We saw how to achieve a map operation using list comprehensions

From the documentation:

But there's also the Python **map** function:



Мар

```
1 def poly(x,y):
2    return(x*y - 3*x - y)
3 list(map(poly, range(1,11), range(10,0,-1)))
```

The first argument to map is a function; remaining arguments are one or more iterables.

```
[-3, 3, 7, 9, 9, 7, 3, -3, -11, -21]
```

1 list(map(max, [1, 1, 2, 3, 5, 8, 13], range(1,8), 7*[2]))

```
[2, 2, 3, 4, 5, 8, 13]
```

1 list(map(poly, range(10), range(10), range(10)))

Number of iterables and number of function arguments must agree!

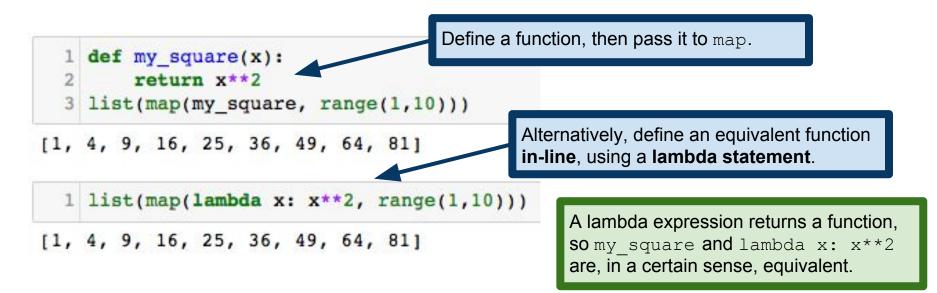
```
TypeError Traceback (most recent call last)
<ipython-input-247-150a3296c401> in <module>()
----> 1 list(map(poly, range(10), range(10), range(10)))
```

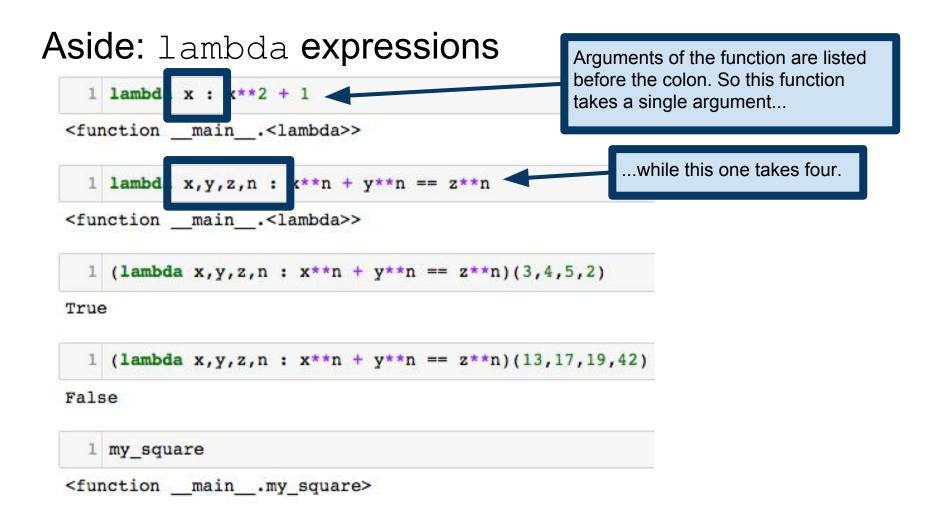
TypeError: poly() takes 2 positional arguments but 3 were given

Aside: lambda expressions

Lambda expressions let you define functions without using a def statement Called an **in-line function** or **anonymous function**

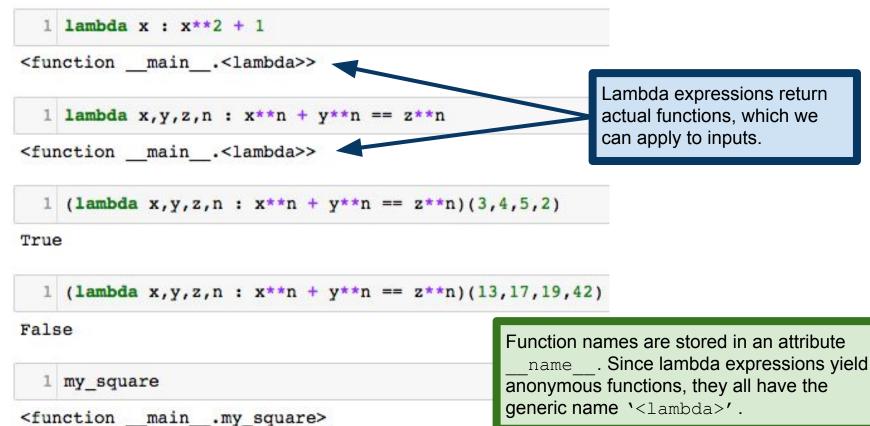
Name is a reference to lambda calculus, a concept from symbolic logic





| Aside: lambda expressions | Return value of the function is listed on the right of the colon. So this function returns the square of its input plus 1 |
|---|---|
| <pre>1 lambda x,y,z,r : x**n + y**n == z**n <functionmain<lambda>></functionmain<lambda></pre> | and this one returns a Boolean stating whether or not the four numbers satisfy Fermat's last theorem. |
| <pre>1 (lambda x,y,z,n : x**n + y**n == z**n)(3, True</pre> | 4,5,2) |
| <pre>1 (lambda x,y,z,n : x**n + y**n == z**n)(13 False</pre> | ,17,19,42) |
| 1 my_square <functionmain .my_square=""> <u>https://e</u></functionmain> | en.wikipedia.org/wiki/Fermat's_Last_Theorem |

Aside: lambda expressions



Aside: lambda expressions

1 f = lambda x : x+'goat' 2 f('cat')

'catgoat'

```
1 (lambda x : 2*x)(21)
```

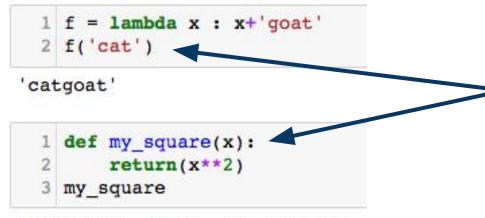
Lambda expressions can be used anywhere you would use a function. Note that the term **anonymous function** makes sense: the lambda expression defines a function, but it never gets a variable name (unless we assign it to something, like in the `goat' example to the left).

42

1 list(map(lambda x: x**2, range(1,10)))

[1, 4, 9, 16, 25, 36, 49, 64, 81]

First-class functions

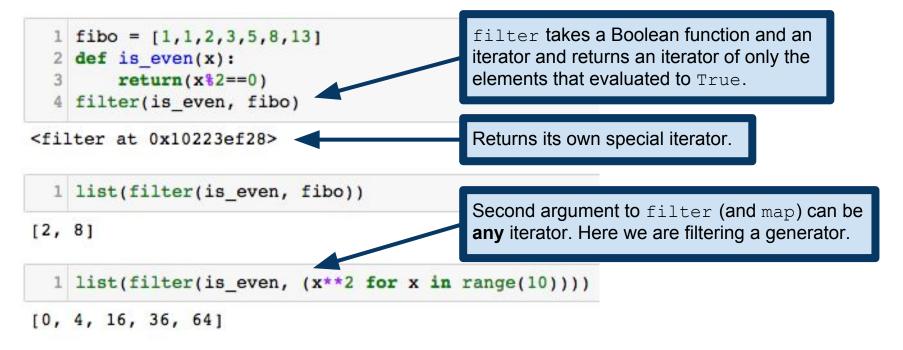


The fact that we can get variables whose values are functions is actually quite special. We say that Python has **first-class functions**. That is, functions are perfectly reasonable values for a variable to have.

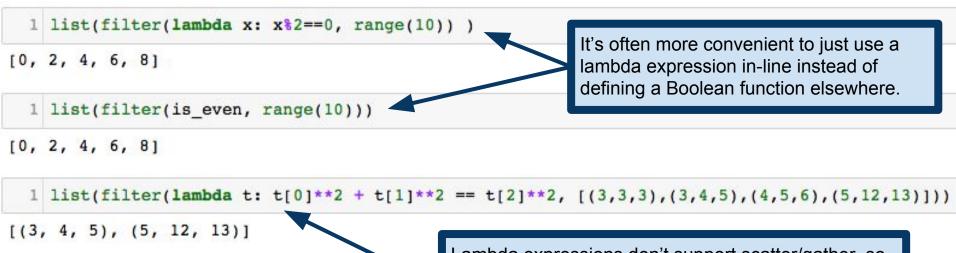
You've seen these ideas before if you've used R's tapply (or similar), MATLAB's function handles, C/C++ function pointers, etc.

Filter

The list filter expression also has an analogous function, filter.

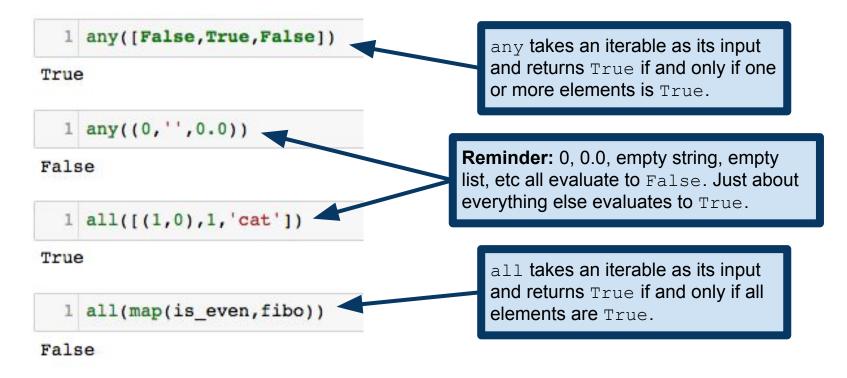


Filter



Lambda expressions don't support scatter/gather, so you have to use this kind of pattern to process tuples. Worry not! Another Python module does support this, and we'll see it in the next lecture.

Quantifiers over iterables: any() and all()



${\tt zip},$ revisited

1 h = harmonic()
2 c = Catalan()
3 z = zip(h,c)
4 z

<zip at 0x101c11a08>

1 [next(z) for x in range(10)]

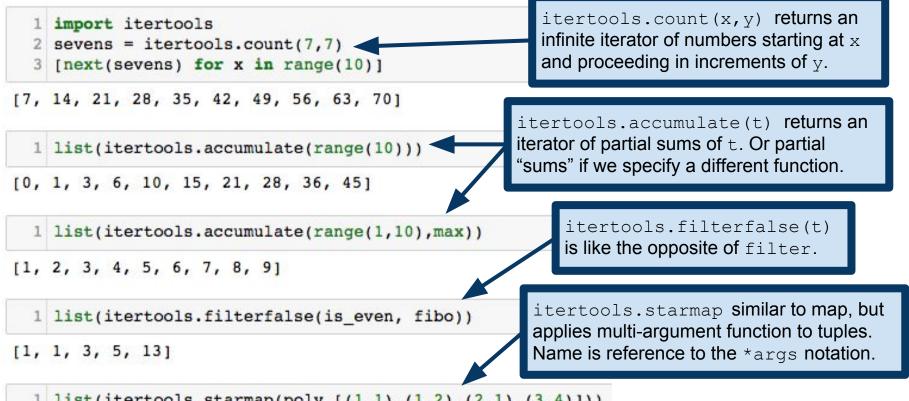
```
[(1.0, 1.0),
(1.5, 1.0),
(1.83333333333333333, 2.0),
(2.083333333333333, 5.0),
(2.28333333333333, 5.0),
(2.2833333333333, 14.0),
(2.4499999999999997, 42.0),
(2.5928571428571425, 132.0),
(2.5928571428571425, 132.0),
(2.7178571428571425, 429.0),
(2.8289682539682537, 1430.0),
(2.9289682539682538, 4862.0)]
```

Recall that zip takes two or more iterables and returns an iterator over tuples

Here are two infinite iterators, and we zip them. So z should also be an infinite iterator. But this expression doesn't result in an infinite evaluation...

The trick is that zip uses **lazy evaluation**. Rather than trying to build all the tuples right when we call zip, Python is lazy. It only builds tuples as we ask for them! We'll see this plenty more in this course. https://en.wikipedia.org/wiki/Lazy_evaluation

Working with iterators: itertools



1 list(itertools.starmap(poly,[(1,1),(1,2),(2,1),(3,4)]))

[-3, -3, -5, -1]

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

More itertools: combinations

1 list(itertools.combinations([1,2,3,4], 2))

[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]

1 list(itertools.permutations([1,2,3], 2))

[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]

1 list(itertools.combinations_with_replacement([1,2,3,4], 2))
[(1, 1),
(1, 2),
(1, 3),
(1, 4),
(1, 4),
(2, 2),
(2, 3),
(2, 4),
(2, 3),
(2, 4),
(3, 3),
(3, 4),
(4, 4)]

Readings (this lecture)

Required:

Python itertools documentation

https://docs.python.org/3/library/itertools.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:

Python functools documentation

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

A. M. Kuchling. *Functional Programming HOWTO* <u>https://docs.python.org/3/howto/functional.html</u>

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