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

Iterators

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

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

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

```
import scipy.special
class catalan():
    '''Iterator over Catalan numbers.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(scipy.special.binom(2*k,k)/(k+1))
c = catalan()
[next(c) for _ in range(10)]
```

```
[1.0, 1.0, 2.0, 5.0, 14.0, 42.0, 132.0, 429.0, 1430.0, 4862.0]
```

`__next__()` method is the important point, here. It returns a value, the next Catalan number.

`next(iter)` is equivalent to calling `__next__()`. Variable _ in the list comprehension is a placeholder. Tells Python we don’t care about the value.

Iterators

Lists are **not** iterators, so we first have to turn the list `t` into an iterator using the function `iter()`.

```
t = [1,2]
titer = iter(t)
next(titer)
```

```
1
```

Now, each time we call `next()`, we get the next element in the list. **Reminder:** `next(iter)` and `iter.__next__()` are equivalent.

```
next(titer)
```

```
2
```

Once we run out of elements, we get an error.

```
next(titer)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-105e88283d1e> in <module>()
----> 1 next(titer)

StopIteration:
```

Iterators

Lists are **not** iterators, but we can turn a list **into** an iterator by calling `iter()` on it. Thus, lists are **iterable**, meaning that it is possible to obtain an iterator over their elements.
https://docs.python.org/3/glossary.html#term-iterable

```
t = [1,2]
titer = iter(t)
next(titer)
```

```
1
```

```
next(titer)
```

```
2
```

```
next(titer)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-20-105e88283d1e> in <module>()
----> 1 next(titer)

StopIteration:
```

From the documentation: “When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() 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.”

Iterators

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`.

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

```
1
2
3

1
2
3
```

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.

Iterators

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`.

```
t = [1,2,3]
for x in t:
    print(x)
print()
for x in iter(t):
    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.

```
1
2
3

1
2
3
```

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

If we try to iterate over an object that is not iterable, we're going to get an error.

```
class dummy():
    '''Class that is not iterable,
    because it has neither __next__()
    nor __iter__().'''

d = dummy()
for x in d:
    print(x)
```

Objects of class `dummy` have neither `__iter__()` (i.e., doesn't support `iter()`) nor `__next__()`, so iteration is hopeless. When we try to iterate, Python is going to raise a `TypeError`.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-30-fc084e213893> in <module>()
      5
      6 d = dummy()
----> 7 for x in d:
      8     print(x)

TypeError: 'dummy' object is not iterable
```

Iterators

```
import scipy.special
class Catalan():
    '''Iterator over Catalan numbers.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(scipy.special.binom(2*k,k)/(k+1))
c = Catalan()
for x in c:
    print(x)
```

Merely being an iterator isn't enough, either!
`for X in Y` requires that object `Y` be iterable.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-48-7d73260f5272> in <module>()
      8         return(scipy.special.binom(2*k,k)/(k+1))
      9 c = Catalan()
---> 10 for x in c:
     11     print(x)

TypeError: 'Catalan' object is not iterable
```

Iterators

Iterable means that an object has the `__iter__()` method, which returns an iterator. So `__iter__()` returns a new object that supports `__next__()`.

```
import scipy.special
class Catalan():
    '''Iterator over Catalan numbers.'''
    def __init__(self):
        self.n = 0
    def __next__(self):
        (self.n, k) = (self.n+1, self.n)
        return(scipy.special.binom(2*k,k)/(k+1))
    def __iter__(self):
        return(self)
c = Catalan()
for x in c:
    print(x)
```

Now `Catalan` supports `__iter__()` (it just returns itself!), so Python allows us to iterate over it.

```
1.0
1.0
2.0
5.0
14.0
42.0
```

This is an infinite loop. Don’t try this at home.

Iterators

```
t1 = ['cat','dog','bird','goat']
t1_iter = iter(t1)
t2 = list(t1_iter)
t1 == t2
```

```
True
```

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.

```
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

```
def catalan(k):
    return(scipy.special.binom(2*k,k)/(k+1))
[int(catalan(x)) for x in range(17) if x%2==0]
```

```
[1, 2, 14, 132, 1430, 16796, 208012, 2674440, 35357670]
```

List comprehension computes and returns the whole list. What if the iterable were infinite? Then this list comprehension would never return!

```
c = Catalan()
[x**2 for x in c]
```

This list comprehension is going to be infinite! But I really ought to be able to get an iterator over the squares of the elements of `Catalan` object `c`...

```
catgen = (x**2 for x in c)
catgen
```

```
<generator object <genexpr> at 0x1053b9e08>
```

This is the motivation for **generator expressions**. Generator expressions are like list comprehensions, but they create an iterator rather than a list.

Generator expressions are written like list comprehensions, but with parentheses instead of square brackets.

Generators

Related to generator expressions are **generators**

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

```
def harmonic(n):
    return(sum([1/k for k in range(1,n+1)]))
harmonic(10)
```

```
2.9289682539682538
```

Each time we call this function, a local namespace is created, we do a bunch of work there, and then all that work disappears when the namespace is destroyed.

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
[next(h) for _ in range(3)]
```

```
[1.0, 1.5, 1.8333333333333333]
```

Alternatively, we can write `harmonic` as a **generator**. Generators work like functions, but they maintain internal state, and they `yield` instead of `return`. Each time a generator gets called, it runs until it encounters a `yield` statement or reaches the end of the `def` block.

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

Generators

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

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

```
next(h)
```

```
1.8333333333333333
```

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.

Generators

```python
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

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

```python
next(h)
```

```
1.0
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```python
next(h)
```

```
1.5
```

```python
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

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

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```
next(h)
```

```
1.8333333333333333
```

Generators

```python
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

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

```python
next(h)
```

```
1.0
```

```python
next(h)
```

```
1.5
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```python
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

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

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

Each time we call `next`, Python runs the code in h from where it left off until it encounters a `yield` statement.

```
next(h)
```

```
1.8333333333333333
```

Generators

```
def harmonic():
    (h,n) = (0,1)
    while True:
        (h,n) = (h+1/n, n+1)
        yield h
h = harmonic()
h
```

```
<generator object harmonic at 0x1053b9fc0>
```

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

```
next(h)
```

```
1.0
```

```
next(h)
```

```
1.5
```

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

```
next(h)
```

```
1.8333333333333333
```

Generators

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

Map

We saw how to achieve a map operation using list comprehensions

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

```
def square_plus1(x):
    return x**2+1
map(square_plus1, range(10))
```

```
<map at 0x102c084e0>
```

```
list(map(square_plus1, range(10)))
```

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

From the documentation:
map(*function*, *iterable*, *...*)
Return an iterator that applies *function* to every item of *iterable*, yielding the results.

`map` and range are both special kinds of iterators.

Map

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

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

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

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

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

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

```
---------------------------------------------------------------------------
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
```

Number of iterables and number of function arguments must agree!

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

```
def my_square(x):
    return x**2
list(map(my_square, range(1,10)))
```

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

Define a function, then pass it to `map`.

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

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

Alternatively, define an equivalent function **in-line**, using a **lambda statement**.

A lambda expression returns a function, so `my_square` and `lambda x: x**2` are, in a certain sense, equivalent.

Aside: `lambda` expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Arguments of the function are listed before the colon. So this function takes a single argument...

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...while this one takes four.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Aside: `lambda` expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

Return value of the function is listed on the right of the colon. So this function returns the square of its input plus 1....

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

...and this one returns a Boolean stating whether or not the four numbers satisfy Fermat's last theorem.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

https://en.wikipedia.org/wiki/Fermat's_Last_Theorem

Aside: lambda expressions

```
lambda x : x**2 + 1
```

```
<function __main__.<lambda>>
```

```
lambda x,y,z,n : x**n + y**n == z**n
```

```
<function __main__.<lambda>>
```

Lambda expressions return actual functions, which we can apply to inputs.

```
(lambda x,y,z,n : x**n + y**n == z**n)(3,4,5,2)
```

```
True
```

```
(lambda x,y,z,n : x**n + y**n == z**n)(13,17,19,42)
```

```
False
```

```
my_square
```

```
<function __main__.my_square>
```

Function names are stored in an attribute __name__. Since lambda expressions yield anonymous functions, they all have the generic name '<lambda>'.

Aside: `lambda` expressions

```
f = lambda x : x+'goat'
f('cat')
```

```
'catgoat'
```

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

```
42
```

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

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

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).

First-class functions

```
f = lambda x : x+'goat'
f('cat')
```

```
'catgoat'
```

```
def my_square(x):
    return(x**2)
my_square
```

```
<function __main__.my_square>
```

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`.

```
fibo = [1,1,2,3,5,8,13]
def is_even(x):
    return(x%2==0)
filter(is_even, fibo)
```

```
<filter at 0x10223ef28>
```

`filter` takes a Boolean function and an iterator and returns an iterator of only the elements that evaluated to `True`.

Returns its own special iterator.

```
list(filter(is_even, fibo))
```

```
[2, 8]
```

Second argument to `filter` (and `map`) can be **any** iterator. Here we are filtering a generator.

```
list(filter(is_even, (x**2 for x in range(10))))
```

```
[0, 4, 16, 36, 64]
```

Filter

```
list(filter(lambda x: x%2==0, range(10)) )
```

```
[0, 2, 4, 6, 8]
```

It's often more convenient to just use a lambda expression in-line instead of defining a Boolean function elsewhere.

```
list(filter(is_even, range(10)))
```

```
[0, 2, 4, 6, 8]
```

```
list(filter(lambda t: t[0]**2 + t[1]**2 == t[2]**2, [(3,3,3),(3,4,5),(4,5,6),(5,12,13)]))
```

```
[(3, 4, 5), (5, 12, 13)]
```

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()`

```
1 any([False,True,False])
```

True

```
1 any((0,'',0.0))
```

False

```
1 all([(1,0),1,'cat'])
```

True

```
1 all(map(is_even,fibo))
```

False

`any` takes an iterable as its input and returns `True` if and only if one or more elements is `True`.

Reminder: 0, 0.0, empty string, empty list, etc all evaluate to `False`. Just about everything else evaluates to `True`.

`all` takes an iterable as its input and returns `True` if and only if all elements are `True`.

`zip`, revisited

```
h = harmonic()
c = Catalan()
z = zip(h,c)
z
```

```
<zip at 0x101c11a08>
```

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...

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

```
[(1.0, 1.0),
 (1.5, 1.0),
 (1.8333333333333333, 2.0),
 (2.083333333333333, 5.0),
 (2.283333333333333, 14.0),
 (2.4499999999999997, 42.0),
 (2.5928571428571425, 132.0),
 (2.7178571428571425, 429.0),
 (2.8289682539682537, 1430.0),
 (2.9289682539682538, 4862.0)]
```

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`

```
import itertools
sevens = itertools.count(7,7)
[next(sevens) for x in range(10)]
```

```
[7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

`itertools.count(x,y)` returns an infinite iterator of numbers starting at `x` and proceeding in increments of `y`.

```
list(itertools.accumulate(range(10)))
```

```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

```
list(itertools.accumulate(range(1,10),max))
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`itertools.accumulate(t)` returns an iterator of partial sums of `t`. Or partial "sums" if we specify a different function.

```
list(itertools.filterfalse(is_even, fibo))
```

```
[1, 1, 3, 5, 13]
```

`itertools.filterfalse(t)` is like the opposite of `filter`.

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

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

`itertools.starmap` similar to map, but applies multi-argument function to tuples. Name is reference to the `*args` notation.

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

More `itertools`: combinations

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

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

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

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

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

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

`itertools` also includes some combinatorial functions that can be useful on occasion.

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*

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