Python for Data Analysis

Session 3. Functional programming in Python

Session agenda

- Functional programming concepts
- Lambdas
- Iterators, generators and generator expressions
- Functional programming tools
- Itertools module
- Using functional programming to perform data analysis tasks

Functional programming concepts

- Decompose a problem into a set of functions
- Ideally, functions only take inputs and produce outputs, and don’t have any internal state that affects the output produced for a given input
- Functional style discourages functions with side effects that modify internal state or make other changes that aren’t visible in the function’s return value
- Functional program execution consists of sequence of functions application on a given input

Functional programming concepts

- There are theoretical and practical advantages to the functional style:
  - Formal provability.
  - Modularity.
  - Composability.
  - Ease of debugging and testing.
- Functional programming can be considered the opposite of object-oriented programming
- In Python both approaches can be combined

Lambda functions

- Last time, we covered function concepts in depth. We also mentioned that Python allows for the use of a special kind of function, a *lambda* function.
- Lambda functions are small, anonymous functions based on the lambda abstractions that appear in many functional languages.
- Lambda abstractions and lambda calculus was introduced by mathematician Alonzo Church in the 1930s
- Lambda calculus is Turing complete

Lambda functions

- Lambda functions within Python.
  - Use the keyword *lambda* instead of *def.*
  - Can be used wherever function objects are used.
  - Restricted to one expression.
  - Typically used with functional programming tools.

```
>>> def f(x):
...     return x**2
...
>>> print(f(8))
64
>>> g = lambda x: x**2
>>> print(g(8))
64
```

Iterables and iterators

- An ***iterable*** is any Python object with the following properties:
  - It can be looped over (e.g. lists, strings, files, etc).
  - Can be used as an argument to `iter()`, which returns an iterator.
  - Must define `__iter__()` (or `__getitem__()`).
- An ***iterator*** is a Python object with the following properties:
  - Must define `__iter__()` to return itself.
  - Must define the `next()` method to return the next value every time it is invoked.
  - Must track the "position" over the container of which it is an iterator.

Iterables and iterators

- A common iterable is the list. Lists, however, are not iterators. They are simply Python objects for which iterators may be created.

```
>>> a = [1, 2, 3, 4]
>>> # a list is iterable - it has the __iter__ method
>>> a.__iter__
<method-wrapper '__iter__' of list object at 0x014E5D78>
>>> # a list doesn't have the next method, so it's not an iterator
>>> a.next
AttributeError: 'list' object has no attribute 'next'
>>> # a list is not its own iterator
>>> iter(a) is a
False
```

Iterables and iterators

- The listiterator object is the iterator object associated with a list. The iterator version of a listiterator object is itself, since it is already an iterator.

```
>>> # iterator for a list is actually a 'listi_terator' object
>>> ia = iter(a)
>>> ia
<list_iterator object at 0x014DF2F0>
>>> # a list_iterator object is its own iterator
>>> iter(ia) is ia
True
```

Iterables and iterators

- Behind-the-scenes actions taken when we use a for-loop.

```
>>> mylist = [1, 2, 3, 4]
>>> for item in mylist:
... print(item)
```

Is equivalent to

```
>>> mylist = [1, 2, 3, 4]
>>> i = iter(mylist)
>>> #i = mylist.__iter__()
>>> print(i.next())
1
>>> print(i.next())
2
>>> print(i.next())
3
>>> print(i.next())
4
>>> print(i.next())
# StopIteration Exception Raised
```

Iterator pattern

- Iterator pattern is a design pattern in which an iterator is used to traverse a container and access the container's elements
- The iterator pattern decouples traversal algorithms from containers
- Provide a way to access the elements of an aggregate object according to traversal algorithm without exposing its underlying representation
- For example breadth-first search algorithm, when implemented as an iterator can be effectively used to implement different graph’s analytic tasks

Generators

- Generators are a way of defining iterators using a simple function notation.
- Generators use the `yield` statement to return results when they are ready, but Python will remember the context of the generator when this happens.
- Even though generators are not technically iterator objects, they can be used wherever iterators are used.
- Generators are desirable because they are *lazy*: they do no work until the first value is requested, and they only do enough work to produce that value. As a result, they use fewer resources, and are usable on more kinds of iterables.

Generators - example

```
>>>def count_generator():
...     n = 0
...     while True:
...         yield n
...         n = n + 1

>>> counter = count_generator()
>>> counter
<generator object count_generator at 0x…>
>>> next(counter)
0
>>> next(counter)
1
>>> iter(counter)
<generator object count_generator at 0x…>
>>> iter(counter) is counter
True
>>> type(counter)
<type 'generator'>
```

Generator expressions

- There are also generator expressions, which are very similar to list comprehensions.

```
>>> l1 = [x**2 for x in range(10)] # list
>>> g1 = (x**2 for x in range(10)) # gen
```

- Equivalent to:

```
def __gen(exp):
    for x in exp:
        yield x**2

g1 = __gen(iter(range(10)))
```

Functional programming tools

- Functional decomposition of the computational task can be simplified with the following functions:
  - Filter – filters input data
  - Map – applies specified function to every provided data element
  - Reduce – performs data convolution (reduction) using specified convolution function
- Map/Reduce is a popular technique in parallel computing since function application with map can be done in parallel.
- More on parallel Map/Reduce in Session 5

Functional programming tools

- Built-in filter(*function*, *iterable*) filters items from iterable sequence for which function(*item*) is true.
- Returns an object of type filter, which is an iterator

```
def even(x):
    if x % 2 == 0:
        return True
    else:
        return False

print(list(filter(even, range(0,30))))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]

#Equivalent to following generator expression
print(list(x for x in range(0,30) if even(x)))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Functional programming tools

- Built-in map(*function, iterable*) applies function to each item in iterable sequence
- Returns the object of type map, which is an iterator

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

print(list(map(square, range(0,11))))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

#Equivalent to following generator expression
print(list(square(x) for x in range(0,11)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Functional programming tools

- For functions with multiple arguments map(*function,iterable,…*) can be called with multiple iterables
- The order of iterables corresponds to the order of function’s variables
- The function is applied to the current values returned by all iterables

```
def power(x, y):
    return x**y

print(list(map(power, range(1,6), range(0,5))))
[1, 2, 9, 64, 625]
```

Functional programming tools

- functools.reduce(*function, iterable*) returns a single value computed as the result of performing *function* on the first two items, then on the result with the next item, etc.
- There's an optional third argument which is the starting value.

```
import functools
def multiply(x, y):
    return x*y

print(functools.reduce(multiply, range(1,5)))
24
```

Functional programming tools

- We can combine lambda abstractions with functional programming tools. This is especially useful when our function is small – we can avoid the overhead of creating a function definition for it by essentially defining it in-line.

```
>>> print(list(map(lambda x: x**2, range(0,11))))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

itertools

- The itertools module is inspired by functional programming languages such as APL, Haskell and SML. The methods provided are fast and memory-efficient and, together, form an “iterator algebra” for constructing specialized iterators.
- Itertools provide three groups of iterators:
  - Inifinite iterators
  - Iterators terminating on the shortest input sequence
  - Combinatoric iterators

itertools – infinite iterators

- Implemented as generators
- `itertools.count(`*`start=0`*`, step=1)` – creates an iterator that returns evenly-spaced values starting with *start*.
- `itertools.cycle(`*`iterable`*`)` – creates an iterator returning elements from the *iterable* and saving a copy of each. When the iterable is exhausted, return elements from the saved copy, repeating indefinitely.
- `itertools.repeat(`*`object`*`[,`*`times`*`])` -- creates an iterator that returns *object* over and over again. Runs indefinitely unless the times argument is specified.

Infinite iterators - examples

```
>>> import itertools
>>> for i in itertools.count(10, 2)
...     print(i)
...     if i > 19:
...         break
...
10
12
14
16
18
20
```

Infinite iterators - examples

```
>>> import itertools
>>> counter = 0
>>> for i in itertools.cycle([1, 2, 3])
...     print(i),
...     counter = counter + 1
...     if counter > 12:
...         break
...
1 2 3 1 2 3 1 2 3 1 2 3 1
```

Infinite iterators - examples

```
>>> import itertools
>>> for i in itertools.repeat("hi", 5)
...     print(i),
...
hi hi hi hi hi
```

Itertools – filters

- Itertools provide additional filtering iterators, which compliment built-in filter function

Iterator	Arguments	Results
`filterfalse`()	pred, seq	elements of seq where pred(elem) is false
`takewhile`()	pred, seq	seq[0], seq[1], until pred fails
`dropwhile`()	pred, seq	seq[n], seq[n+1], starting when pred fails
`islice`()	seq, [start,] stop [, step]	elements from seq[start:stop:step]
`compress`()	data, selectors	(d[0] if s[0]), (d[1] if s[1]), …

Itertools – filters

```
it = itertools.filterfalse(lambda x: x % 2, range(20))
print(list(it))
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

it2 = itertools.takewhile(lambda x: x < 10, range(20))
print(list(it2))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

it3 = itertools.dropwhile(lambda x: x < 10, range(20))
print(list(it3))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

it4 = itertools.islice(range(20), 10, 15)
print(list(it4))
[10, 11, 12, 13, 14]

it5 = itertools.compress("ABCDEF", [1,1,0,1,1,0,])
print(''.join(it5))
ABDE
```

Itertools – aggregators

- Itertools provide iterators, which combine other iterators, groups iterated data or duplicates iterators

Iterator	Arguments	Results
`chain()`	p, q, …	p0, p1, … plast, q0, q1, …
`chain.from_iterable()`	iterable	p0, p1, … plast, q0, q1, …
`zip_longest()`	p, q, …	(p[0], q[0]), (p[1], q[1]), …
`groupby()`	iterable[, keyfunc]	sub-iterators grouped by value of keyfunc(v)
`tee()`	it, n	it1, it2, … itn splits one iterator into n

Itertools – aggregators

```
it6 = itertools.chain(range(5),range(5,10))
print(list(it6))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

it7 = itertools.chain.from_iterable(['ABC','DEF'])
print(list(it7))
['A', 'B', 'C', 'D', 'E', 'F']

it8 = itertools.zip_longest('GACTGAA','CTGACTT')
print(list(it8))
[('G', 'C'), ('A', 'T'), ('C', 'G'), ('T', 'A'), ('G', 'C')]
```

Itertools – aggregators

```
it9 = itertools.groupby(range(12), lambda x: x // 5)
for key, group in it9:
    print( key, list(group))
0 [0, 1, 2, 3, 4]
1 [5, 6, 7, 8, 9]
2 [10, 11]

it10 = itertools.tee(range(10),2)
next(it10[1])
print(list(map(lambda x,y:x+y,it10[0],it10[1])))
[1, 3, 5, 7, 9, 11, 13, 15, 17]
```

Itertools – modifiers

- Itertools provide iterators, which modify provided iterator

Iterator	Arguments	Results
`starmap()`	func, seq	func(*seq[0]), func(*seq[1]), …
`accumulate()`	p [,func]	p0, p0+p1, p0+p1+p2, …

```
it8 = itertools.zip_longest('GACTG','CTGAC')
it11 = itertools.starmap(lambda x,y: x+y,it8)
print(list(it11))
['GC', 'AT', 'CG', 'TA', 'GC']

it12 = itertools.accumulate(range(1,6),lambda x,y:x*y)
print(list(it12))
[1, 2, 6, 24, 120]
```

Itertools – combinatorics iterators

- Implemented as generators
- `product(*iterables [, r])` – returns cartesian product of input iterables repeated *r* times
- `permutations(iterable[, r])` – returns successive *r* length permutations of elements in the *iterable*.
- `combinations(iterable, r)` – returns *r* length subsequences of elements from the input *iterable*.

Combinatorics iterators - examples

```
>>> from itertools import *
>>> for i in permutations('ABC', 2):
...     print(i)
...
('A', 'B') ('A', 'C') ('B', 'A') ('B', 'C') ('C', 'A') ('C', 'B')
>>> for i in combinations([1,2,3,4], 2):
...     print(i),
...
(1, 2) (1, 3) (1, 4) (2, 3) (2, 4) (3, 4)
```

Using combinatorics iterators

- Combinatoric iterators should be used when you really need to iterate over all or some combinatoric objects
- When you just need to find a specific combinatoric object you can use math or other algorithms to avoid unnecessary iteration

Lexicographic permutations

- A permutation is an ordered arrangement of objects. For example, 3124 is one possible permutation of the digits 1, 2, 3 and 4. If all of the permutations are listed numerically or alphabetically, we call it lexicographic order. The lexicographic permutations of 0, 1 and 2 are:

  012   021   102   120   201   210

- What is the millionth lexicographic permutation of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9?

Solution with iterators

```
from itertools import permutations
import string

def find_perm(num):
    for i, p in enumerate(permutations(string.digits),
start=1):
        if i == num:
            return ''.join(p)

%time find_perm(100000)
Wall time: 9 ms
'0358926714'

%time find_perm(1000000)
Wall time: 81.7 ms
'2783915604'
```

Solution without iterators

```
from math import factorial, floor
def kthperm(S, k):
    P = []
    while S != []:
        f = factorial(len(S)-1)
        i = int(floor(k/f))
        x = S[i]
        k = k%f
        P.append(x)
        S = S[:i] + S[i+1:]
    return P

%time kthperm(list(string.digits), 100000)
Wall time: 0 ns
['0', '3', '5', '8', '9', '2', '6', '7', '1', '4']

%time kthperm(list(string.digits), 1000000)
Wall time: 0 ns
['2', '7', '8', '3', '9', '1', '5', '6', '0', '4']
```

Using functional programming for data analysis

- A number of data analysis tasks can be effectively implemented using functional programming concepts
- Single instruction multiple data(SIMD) paradigm – one operation must be performed for every data element.
- Usually raw data preprocessing and mangling is just a sequence of SIMD operations.
- SIMD operations can be done in parallel

Using functional programming for data analysis

- Iterators can help with the following tasks:
  - Provide more efficient way to manipulate and transform data
  - Abstract data manipulation process from the way data is stored
  - Control memory usage, when working with big data
  - Implement serial and on-line data analysis tasks
  - Implement different sampling procedures