Script language: Python

Advanced data structures

Cédric Saule

Technische Fakultät
Universität Bielefeld

13. Februar 2016

Python

–

List description.

range, xrange, Generators, List description

range() vs. xrange()

range([[start, [end]], step]) Create a list from `start`, to `end` with an increment of `step`.

- Returned value: `list` with `int`.
- List is persistent in memory before.

xrange([[start, [end]], step])) Generator with the same properties as `range` (lazy evaluation).

- Returned value: an object `xrange`, not a list.
- Is *iterable*.
- Used in loops and is optimized for the memory → Has the same property as a list generated with `range`.
- Particularly efficiently if all elements are not called.

range() vs. xrange

```
> range(3)
[0, 1, 2]
> g = xrange(3)
> g
xrange(3)
> for x, y in enumerate(range(2,4)):
>     print x, y/2.
0 1.0
1 1.5
> for x, y in enumerate(g):
>     print x, y/2.
0 1.0
1 1.5
```

Generators

How to create a list with exactly this (y) content?

```
> retL = []
> for y in xrange(2,4):
>     retL.append(y/2.)
> retL
[1.0, 1.5]
```

More efficient?

```
> retLfast = []
> g = ( y/2 for y in xrange(2, 4))
> for y in g:
>    retLfast.append(y)
> retLfast
[1.0, 1.5]
```

Generators

The term `(y/2 for y in xrange(2, 4))` is called *generator* and can be used in loops as generating function similarly to `xrange`.

- A defined arithmetic operation is executed only if we have another pass through the loop.
- Memory efficiency, because the entire list is not generated.
- If a `break` occurs, we can continue working elsewhere with the next element.

In our case, the list generation is still not fast enough and elegant.

List description

The generator expression can be used almost 1 : 1 to directly generate a list:

```
> [y/2 for y in xrange(2,4)]
[1.0, 1.5]
```

The same is true for the production of `dict` and `set`:

```
> #dict
> {y:y/2 for y in xrange(2,4)}
{2: 1.0, 3: 1.5}

> #set
> {y/2 for y in xrange(2,4)}
{1.0, 1.5}
```

List description

Which list is created here?

```
> [ letter*N for N in range(1,5), for letter in 'abcd' ]
```

List description

```
> [ letter*N for N in range(1,5), for letter in 'abcd' ]
['a',
 'b',
 'c',
 'd',
 'aa',
 'bb',
 'cc',
 'dd',
 'aaa',
 'bbb',
 'ccc',
 'ddd',
 'aaaa',
 'bbbb',
 'cccc',
 'dddd']
```

Exercise – FASTA list description

Consider a FASTA format to describe a DNA sequence. Create a sequence and return a list that looks like
`["A", "A", "C", "T", "A", "G"]`

Create a function that expects two parameters: `n`, the length of the sequence and `s`, the sequence name. This function has to generate a DNA sequence from the previous list and to return the name and sequence in FASTA format. The output has to be in upper case:
`'AACTAG'`

Logical generator

The following little script uses a sophisticated List generator that returns truth values.

```
> [ '%s␣is␣%s' % (item, bool(item)) for item in
   [0, 1, 0.0, 0.1, 0j, 1j, [], [0], (), (0,), {}, {'a':'A'}, None]]

['0␣is␣False',
 '1␣is␣True',
 '0.0␣is␣False',
 '0.1␣is␣True',
 '0j␣is␣False',
 '1j␣is␣True',
 '[]␣is␣False',
 '[0]␣is␣True',
 '()␣is␣False',
 '(0,)␣is␣True',
 '{}␣is␣False',
 "{'a':␣'A'}␣is␣True",
 'None␣is␣False']
```

Python

–

High order functions.

map(), filter()

High order functions

High order functions are functions that can be given as arguments and can have functions as returned values.

Example in Haskell:

```
> map :: (a -> b) -> [a] -> [b]
> map f []     = []
> map f (x:xs) = (f x):map f xs
```

`map (\x -> x^2) [1,2,3,4]` is evaluated to `[1,4,9,16]`

But then, how can we do that in Python? Quick reminder: In Python, a function is an object!

map()

`map(<FUNC>, <SEQ>[, <SEQ>, ...])` expects exactly one function and at least one sequence as arguments. If more sequences are specified as a sequence, they will be considered as additional arguments to the function. If the size does not fit, the corresponding elements will be considered as `None`.

```
> def pow_2(n, e=2):
>  return n**e

> map(pow_2, range(1,5))
[1, 4, 9, 16]
> map(pow_2, range(1,5), range(2,6))
[1, 8, 81, 1024]
> map(lambda n:n**2, range(1,5))
[1, 4, 9, 16]
```

Exercise – `map()` with `lambda`

Create the `lambda` expression in the `map()` of the previous slide in order to make the functionality of the `pow_2(n,e=2)` corresponds to the function.

Proper high order functions

We can also write our proper high order functions in Python. Here is a small example for $f(x) = x + 3$ and $g(x) = f(f(x))$:

```
> def f(x):
>  return x + 3

> def twice(function, x):
>  return function(function(x))

> print(twice(f, 7))
```

What the result looks like ?

filter()

`filter(<FUNC> | None, <SEQ>)` creates a list in which the use of `FUNC` return `True`. If `FUNC = None`, the only elements that are copied into the new sequence are `True`. If `SEQ` is a tuple or a string, the returned type depends on the input type, otherwise a list is always returned.

```
> def is_odd(n):
>  return n%2

> filter(is_odd, range(4))
[1, 3]
> filter(None, range(4))
[1, 2, 3]
```

Python

–

More about sort()

cmp(), key, reverse

sort()

Hitherto we have only used `sort()` with the default values, even with the dual functions. Let us remind quickly
`sort([cmp[, key[, reverse]]])`:

- **cmp** A new function of comparison which must return -1, 0 or 1.
- **key** Function that defines which property has to be sorted in the sequence of elements (`key=str.lower` or `lambda (k,v) : v`)
- **reverse** If `True`, the sort order is reversed.

sort()

```
> mydict = { 'b1': (1, 6),  'b3': (1, 3),
 'a4': (1, 2), 'b5': (1, 7), 'a6': (3, 7) }

 > sort(mydict.iteritems())
[('a4', (1, 2)),
 ('a6', (3, 7)),
 ('b1', (1, 6)),
 ('b3', (1, 3)),
 ('b5', (1, 7))]
```

The function `sort()` sorts all *iterable* data types and returns a list with the sorted elements. We see here that they are sorted by the *keys*.

sort()

to reverse the order → `reverse=True`

```
> sorted(mydict.iteritems(), reverse=True)
[('b5', (1, 7)),
 ('b3', (1, 3)),
 ('b1', (1, 6)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort()

Suppose that the tuple value defines a fraction. How could we sort them by ?

```
> def cmpN(x, y):
>  return cmp(float(x[1][0]) / x[1][1],
     float(y[1][0]) / y[1][1])

> sorted(mydict.iteritems(), cmp=cmpN)
[('b5', (1, 7)),
 ('b1', (1, 6)),
 ('b3', (1, 3)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort()

That was a very ugly notation. Why not directly access the tutelage? We have to mess around ...

```
> def cmpNew(x, y):
>  return cmp(float(x[0]) / x[1],
    float(y[0]) / y[1])

> sorted(mydict.iteritems(), cmp=cmpNew,
    key=lambda (k,v):v )
[('b5', (1, 7)),
 ('b1', (1, 6)),
 ('b3', (1, 3)),
 ('a6', (3, 7)),
 ('a4', (1, 2))]
```

sort() – key() and cmp()

When using `cmp` and `key` there is several things to consider:

- `cmp()` is carried out at each comparison.
- The function `key` is executed once per element, before the `cmp()` sorting step.
- Optimal: Precomputation, `cmp()` is performed before the actual comparison, already running in `key`!

Exercise – sort()

Superimposed functionality of the function `cmpNew`, so that the precomputing is already conducted once in the term `lambda` and no additional `cmp()` must be defined.