Python: Functions and Generators

Giuseppe Attardi

Functional Programming

Slides by Felix Hernandez-Campos

List Comprehensions

```
>>> freshfruit = ['  banana', '
  loganberry ', 'passion fruit  ']
>>> [x.strip() for x in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

List Comprehensions

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

List Comprehensions

```
>>> [{x: x**2} for x in vec]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

List Comprehensions

```
>>> [(x, x**2) for x in vec]
[(2, 4), (4, 16), (6, 36)]
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

List Comprehensions

```
def quicksort(list):

  if not list:
    return []
  else:
    pivot = list[0]
    l1 = quicksort([x for x in list[1:] if x < pivot])
    l2 = quicksort([x for x in list[1:] if x >= pivot])
    return l1 + [pivot] + l2
```

Higher-Order Functions

- Higher-order functions are functions that take other functions as arguments
- They can be use to implement algorithmic *skeletons*
  - Generic algorithmic techniques
- Three predefined higher-order functions are specially useful for working with list
  - `map`
  - `fold`
  - `filter`

Map

- "map(function, sequence)" calls function(item) for each of the sequence's items and returns a list of the return values.
- For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729,
  1000]
```

Map

- More than one sequence may be passed
- The function is called with the corresponding item from each sequence (or None if some sequence is shorter than another).
- If None is passed for the function, a function returning its argument(s) is substituted.

Map

- Combining these two special cases, we see that "map(None, list1, list2)" is a convenient way of turning a pair of lists into a list of pairs.
- For example

```
>>> seq = range(8)
>>> def square(x): return x*x
...
>>> map(None, seq, map(square, seq))
[(0, 0), (1, 1), (2, 4), (3, 9), (4,
  16), (5, 25), (6, 36), (7, 49)]
```

Zip

- Zip combines two lists into a list of pairs

```
>>> zip([1, 2, 3], ['a', 'b', 'c'])
[[1, 'a'], [2, 'b'], [3, 'c']]
```

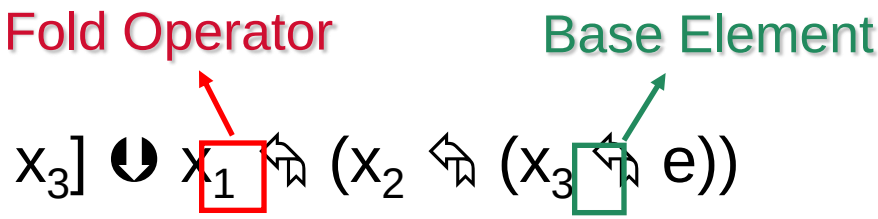
Filter

- filter(function, sequence)" returns a sequence (of the same type, if possible) consisting of those items from the sequence for which function(item) is true.
- For example, to compute some primes:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

Fold

- Takes in a function and *folds* it in between the elements of a list
- Two flavors:
  - *Right-wise* fold: $[x_1, x_2, x_3]$ 🡅 x_1 🡄 (x_2 🡄 (x_3 🡄 e))
  - *Left-wise* fold: $[x_1, x_2, x_3]$ 🡇 ((e 🡄 x_1) 🡄 x_2) 🡄 x_3



Folding in Python: Reduce

- "reduce(*func*, sequence)" returns a single value constructed by calling the binary function *func* on the first two items of the sequence, then on the result and the next item, and so on.
- For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x,y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

- If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

Reduce

- A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on.
- For example,

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Lambda Abstractions

- Anonymous functions can be defined through lambda abstraction

```
>>> car = lambda lst: lst[0]
>>> cdr = lambda lst: lst[1:]
>>> sum2 = lambda lst: car(lst)+car(cdr(lst))
>>> sum2(range(10))
1
```

More on Python Functional Programming

- Articles by David Mertz
- http://www-106.ibm.com/developerworks/linux/library/l-prog.html
- http://www-106.ibm.com/developerworks/library/l-prog2.html

Iterators and Generators

Slides by Thomas Wouters

Iteration

- The act of going over a collection
- Explicit in the form of ‘for’
- Implicit in many forms:
  - list(), tuple(), dict(), …
  - map(), reduce(), zip(), …
  - ‘in’ in absence of __contains__()
  - ‘extended call’ syntax: func(*…)
    - but not apply()

Iterators

- Protocol of 2 methods (no special class):
  - __iter__(): get iterator
  - next(): get next value
    - raises StopIteration when done
- Explicit iterator creation with iter()
- Turn iteration(-state) into objects
- Interchangeable with iterable for iteration

iter()

- Creates a new iterator for an object
  - Calls __iter__() for creation
  - Falls back to __getitem__()
  - Called implicitly for iteration
- Wraps a function in an iterator
  - iter(f, o) calls f until o is returned:

```
for line in iter(file.readline, ""):
    handle(line)
```

Examples

```
>>> l = [1, 2, 3, 4, 5, 6]
>>> it = iter(l)
>>> for num in it:
...     if num > 1: break
>>> for num in it:
...     print num; break
3
>>> print list(it)
[4, 5, 6]
```

Writing Iterators

```
class IRange:
    def __init__(self, end):
        self.end = end
        self.cur = 0
    def next(self):
        cur = self.cur
        if cur >= self.end:
            raise StopIteration
        self.cur += 1
        return cur
    def __iter__(self):
        return self
```

Generators

- Use new keyword 'yield'
  - any function with 'yield' is special
- Turn function-state into objects
- Use the iterator protocol
- Not unavoidable
  - just very very convenient

IRange generator

```
>>> def irange(end):
...     cur = 0
...     while cur < end:
...         yield cur
...         cur += 1
>>> print irange(10)
<generator object at 0x4701c0>
>>> print list(irange(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Many useful generators in the 'itertools' module

Example

```
def map(func, iterable):
    result = []
    for item in iterable:
        result.append(func(item))
    return result

def imap(func, iterable):
    for item in iterable:
        yield func(item)
```

'yield' and 'try'/'finally'

- Python does not allow 'yield' inside a 'try' block with a 'finally' clause:

```
try:
    yield x
    yield x
finally:
    print x
```

- 'yield' inside 'finally' or in 'try'/'except' is allowed