Introduction to Python Programming

17 – List Comprehensions (and a few other bits and pieces)

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23

Recap

- What are iterators?
- What is an iterable object?
- How do iterators differ from collection types such as lists? What can be done with a list but not with an iterator?
- Any other differences?
- Why are iterators useful in the first place?

An Example

- Let‘s implement an iterator PI (“pair iterator”) that implements the following functionality:

```
>>> lis = ["a", 1, "b", 2, "c", 3]
>>>
>>> list(PI(lis))
[('a', 1), ('b', 2), ('c', 3)]
>>>
>>> dict(PI(lis))
{'a': 1, 'c': 3, 'b': 2}
>>>
```

Solution

```
class PI:
    def __init__(self, iterable):
        self.iterator = iter(iterable)
    def __iter__(self):
        return self
    def __next__(self):
        x = next(self.iterator)
        y = next(self.iterator)
        return (x, y)
```

Another example: An “infinite” Iterator

```
 1 class Numbers:
 2     def __init__(self):
 3         self.i = -1
 4     def __iter__(self):
 5         return self
 6     def __next__(self):
 7         self.i += 1
 8         return self.i
 9
10 for x in Numbers():
11     print(x)
```

File Iterators are like …

```
 1 class FileIterator:
 2     def __init__(self, file):
 3         self.file = file
 4     def __iter__(self):
 5         return self
 6     def __next__(self):
 7         line = self.file.readline()
 8         if line == '':
 9             raise StopIteration
10         return line
11
12 with open('example.txt') as f:
13     for line in FileIterator(f):
14         print(line)
```

Exercise

Let‘s implement an iterator that iterates over a file by word

```
class ByWord:
    ...

with open("example.txt") as f:
    for word in ByWord(f):
        print(word)
```

Solution

```
 1  class ByWord:
 2      def __init__(self, file):
 3          self.file = file
 4          self.buffer = []
 5      def __iter__(self):
 6          return self
 7      def __next__(self):
 8          if self.buffer:
 9              return self.buffer.pop(0)
10          else:
11              line = self.file.readline()
12              if line == '':
13                  raise StopIteration
14              self.buffer = line.split()
15              return next(self)
```

Mapping Elements in Sequences

Mapping Elements in a Sequence

A typical task in CL/NLP/LT: suppose we have a list of “word/POS” pairs and want to compute a list of words (without POS)

One option:

```
1 inlist = ["A/DT", "student/NN", "read/VBD", ...]
2
3 outlist = []
4
5 for token in inlist:
6     (word, pos) = token.rsplit('/', 1)
7     outlist.append(word)
```

Mapping Elements in a Sequence

A typical task in CL/NLP/LT: Suppose we have a list of “word/POS” pairs and want to compute a list of words (without POS)

Another option:

```
1 inlist = ["A/DT", "student/NN", "read/VBD", ...]
2
3 def strip_pos(token):
4     return token.rsplit('/', 1)[0]
5
6 outlist = map(strip_pos, inlist)
```

Note: outlist is now an iterator (not a list as before)

map

- `map(function, iterable)`
  - calls `function(item)` for each item in `iterable`
  - returns an iterator over the list of return values
- Roughly equivalent to:

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

Functions …

- Functions are “first class citizens” in Python
  - functions can be passed as arguments to other functions
  - functions can return other functions
- Functions that take other functions as arguments are called “higher order functions”

filter

- `filter(function, iterable)`
  - call `function` for evey item in `iterable`
  - returns an iterator over the list of items for which the function call returns true

```
1 inlist = ["A/DT", "student/NN", "read/VBD", ...]
2
3 def is_noun(token):
4     return token.rsplit('/', 1)[1] in ('NN', 'NNS')
5
6 nouns = filter(is_noun, inlist)
```

lambda expressions

- Functions are usually defined using “def name(...): ...”
  - ⇒ Functions have a name
- We can define anonymous functions as follows:
  - lambda ⟨parameters⟩: ⟨expression⟩
- Lambda expressions:
  - evaluate to a function
  - the return value of the function is the value of ⟨expression⟩
  - are useful when used in combination with higher order functions such as map or filter
- Note: exactly one ⟨expression⟩, statements are not allowed

lambda expressions

```
>>> lambda x: x + 1
<function <lambda> at 0xb736ddac>
>>> (lambda x: x + 1)(2)
3
>>> add1 = lambda x: x + 1
>>> add1(2)
3
>>> list(map(lambda x: x + 1, [1, 2, 3]))
[2, 3, 4]
>>> list(filter(lambda x: x%2!=0, [1, 2, 3]))
[1, 3]
```

An aside: if expressions

- Standard way of expressing conditionals:
  - if ⟨expression⟩: ⟨block⟩ else: ⟨block⟩
  - (this is a statement)
- Alternatively: conditional expressions
  - ⟨expression$_1$⟩ if ⟨condition⟩ else ⟨expression$_2$⟩
- Conditional expressions …
  - evaluate to the value of ⟨expression$_1$⟩ if ⟨condition⟩ evaluates to True
  - otherwise the conditional expression evaluates to the value of ⟨expression$_2$⟩

List Comprehensions

List comprehensions

- Set comprehension in ordinary maths …
- A way of writing sets
- $\{0, 2\}$

- $\{0, 2, 4, 6, 8, \ldots\}$
- $\{x | x \text{ is even number incl. } 0\}$

List comprehensions

- List comprehensions = a concise way to create lists
- Syntax (simplified)
  - [⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩]
- This expression evaluates to a list of items obtained by evaluating ⟨expr⟩ for every ⟨var⟩ in ⟨iterable⟩
- Simple example:
  - [x + 1 for x in [1,2,3]] ⇒ [2,3,4]

List comprehensions

- Equivalent (almost):

```
result = [⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩]
```

```
result = []
for ⟨var⟩ in ⟨iterable⟩:
    result.append(⟨expr⟩)
```

- Difference:
  - the variables introduced in a list comprehension are introduced in their own namespace
  - i.e., are local to the list comprehension.

List comprehensions

- Equivalent (almost):

```
result = [x + 1 for x in [1,2,3]]
```

```
result = []
for x in [1,2,3]:
    result.append(x + 1)
```

- Difference:
  - the variables introduced in a list comprehension are introduced in their own namespace
  - i.e., are local to the list comprehension.

Another Example

```
>>> doc = ["A/DT", "student/NN", "read/VBD", ...]
>>>
>>> [w.split('/')[0] for w in doc]
['A', 'student', 'read', ...]
>>>
>>> [w.split('/')[0].upper() for w in doc]
['A', 'STUDENT', 'READ', ...]
>>>
>>>
>>>
```

List comprehensions

- List comprehensions can contain several for loops
  - [x + y for x in [1, 2, 3] for y in [4, 5, 6]]
  - ⇒ [5, 6, 7, 6, 7, 8, 7, 8, 9]
- This expression is (almost) equivalent to the following:

```
result = []
for x in [1, 2, 3]:
    for y in [4, 5, 6]:
        result.append(x + y)
```

List comprehensions

- List comprehensions can contain conditions:

```
>>> [x for x in [2,3] if x % 2 == 0]
[2]
>>>
>>> [x + y for x in [2,3] if x % 2 == 0 for y in [5,6]]
[7, 8]
>>>
>>> [x + y for x in [2,3] for y in [5,6] if x % 2 == 0]
[7, 8]
>>>
```

Set and dict comprehensions

- Like for lists, there is a syntax for set and a dict comprehensions:

```
>>> { x for x in [1, 2, 2, 3, 1] }
{1, 2, 3}
>>> { x : 2 ** x for x in [1, 2, 3] }
{1: 2, 2: 4, 3: 8}
```

Generator Expressions

- Variant of list comprehensions:
  - (⟨expr⟩ for ⟨var⟩ in ⟨iterable⟩)
- This expression evaluates to a “generator” (iterator)
- Special case: Brackets can be ommitted if a generator expression is used as a single argument in a function call
  - sum((x * x for x in [1, 2, 3]))
  - sum(x * x for x in [1, 2, 3])

Exercise #1

- Write a list comprehension that creates a list of tuples.
  - Each tuple gives temperatures in Celsius and Fahrenheit.
  - Create one list for Celsius values from 0 to 100 in steps of 5
  - Fahrenheit = ((Celsius × 9) / 5) + 32

Exercise #2

Write a function "count" that takes a list of words as argument and returns a list of word-frequency pairs (possibly containing duplicates). The function should use a list comprehension.

For instance:

```
>>> count("Wenn Fliegen hinter Fliegen fliegen fliegen
Fliegen Fliegen nach".split())
[('Wenn', 1), ('Fliegen', 4), ('hinter', 1),
('Fliegen', 4), ('fliegen', 2), ('fliegen', 2),
('Fliegen', 4), ('Fliegen', 4), ('nach', 1)]
```

Note: result may contain duplicate pairs

Exercise #3

Rewrite the function f1 as function f2 using a single list comprehension.

```
def f1(words):
    result = []
    for word in words:
        wordlenpair = (word, len(word))
        result.append(wordlenpair)
    return result
```