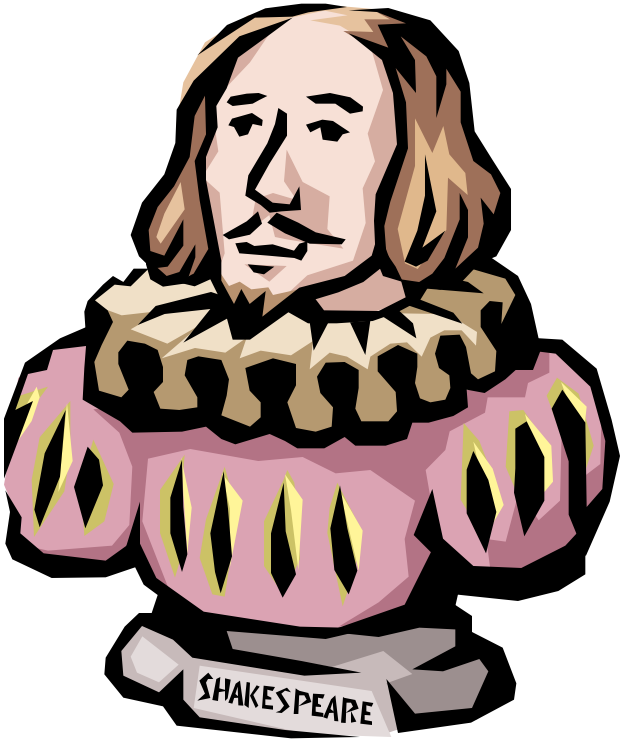
Monkeys, Typewriters, and The Complete Works of Shakespeare



A Brief Introduction To List Comprehensions

MarginHound
hound@marginhound.com
October13th 2011

Introduction

“Arthur looked up. "Ford!" he said, "there's an infinite number of monkeys outside who want to talk to us about this script for Hamlet they've worked out.“

- Hitchhiker’s Guide To The Galaxy

- List comprehensions evolved out of notation used by mathematicians to describe sets of numbers
- Generates a new list object using a single statement.
  - Concise syntax performs the work of many other statements, with less code
- Can be used for many things – not just list generation
  - Obvious application is mathematics and data slinging
  - But can be used for more – templates/markup, file reads,…
  - Excellent replacement for messy / repetitive for loops

Map, Filter, Reduce

Terminology which describes “doing stuff” to lists of values

- Map (function, list)

  Create a new list by applying that function to each element of the original list (one-for-one mapping)

- Filter (function, list)

  Create a new list by applying a function to every element of the original list; return all elements where the function evaluates True

- Reduce (function, list)

  Returns a single value from calling a binary function f(a, b) on the first two elements of the list; then on the result and the next item…

First Example – Filtering & Transforming Data

The Request:

- We have a list of primates (by name)
  - Specifically a list of dicts (key: value pairs)
  - Could generate this from a database, file, web service, etc.

```
[   {   'name': 'larry', 'type': 'monkey'},
    {   'name': 'curly', 'type': 'monkey'},
    {   'name': 'moe', 'type': 'monkey'},
    {   'name': 'king kong', 'type': 'gorilla'}]
```

- Find the monkeys – and generate a list of their names…

The Code…

```
def for_loops_example(source_list):
    result = []

    for elem in source_list:
        if elem['type']=='monkey':
            result.append(elem['name'])

    return result
```

5 lines > 1 line

Could actually do this without the function call

```
def list_comps_example(source_list):
    return [elem['name'] for elem in source_list if elem['type']=='monkey']
```

Both Functions Generate The Same Result

```
for loop solution:  ['larry', 'curly', 'moe']
list comp solution: ['larry', 'curly', 'moe']
```

Basic Construction

new list = [**<output expression>** for **<item-reference>** in **<source_list>** if **<filter-expression>]**

<u>The Elements:</u>

- **Source List** An iterable object (list, dict, generator, tuple)
  - Can transform using functions; can be a list comp, may have multiple sources
- **Item Reference** How to unpack the source list; becomes a local variable
- **Output Expression** How to create the elements of the new list
  - Can apply functions, build lists / tuples, nest list comprehensions
- **Filter Expression** Optional condition (boolean) to filter out records

```
>>> test_dict = primates[0]
>>> [(v,k) for k,v in test_dict.items()]
[('monkey', 'type'), ('larry', 'name')]
>>> [item for item in primates if item['type'] != 'monkey' and
item['name'] not in test_dict]
[{'type': 'gorilla', 'name': 'king kong'}]
```

Multiple Lists

Can iterate across multiple lists, like a nested for loop

- List comp cycles through lists from left-to-right (left = top for loop)
- Second “loop” can be a second list or “unpack” items from the first list

The Code… (for loop vs. list comp)

```
typists = ['larry','curly','moe']

machines = ['IBM Selectric',
            'Underwood Universal Portable',
            'Remington SL3']

def double_for_loop(list_a, list_b):
    result = []

    for item_a in list_a:
        for item_b in list_b:
            result.append("%s : %s" % (item_a, item_b))

    return result

def two_list_lc(list_a, list_b):

    return ["%s : %s" % (item_a, item_b)
            for item_a in list_a for item_b in list_b]
```

Results (Same For Both)

```
[   'larry : IBM Selectric',
    'larry : Underwood Universal Portable',
    'larry : Remington SL3',
    'curly : IBM Selectric',
    'curly : Underwood Universal Portable',
    'curly : Remington SL3',
    'moe : IBM Selectric',
    'moe : Underwood Universal Portable',
    'moe : Remington SL3']
```

Another Example – Templates

- Can use functions & list comps to wrap a “template” around lists of data
- This example builds (and serves) a simple HTML select box

```
import SimpleHTTPServer, SocketServer
page_template = "<html><head></head><body> %s </body></html>"

def wrap_tag(inner_content, symbol = "li"):
    return "<%s>%s</%s>" % (symbol, inner_content, symbol)

class handler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):
            self.send_response(200)
            self.send_header("Content-type", "text/html")
            self.end_headers()

            option_list = [wrap_tag(animal['name'],"option")
                for animal in primates if animal['type']=='monkey']
            dynamic_content = wrap_tag("".join(option_list),"select")

            self.wfile.write(page_template % dynamic_content)
            return

server = SocketServer.TCPServer(("localhost",80), handler)
server.serve_forever()
```

- Filter records
- Extract name element
- Wrap HTML Tag around content
- Return new list of HTML Tags
- Join into string, wrap <select> around string

Advanced Construction

Can Combine List Comprehensions With Other Tools

- Transform source list using functions:
  - Sorted () Function
  - Itertools – Groupby, Chain, etc.
  - List Slicing
- Nested List Comprehensions / Generators
  - Use another list comprehension as a source
  - Use another list comprehension as part of the output statement
- Can reduce sets of grouped data to a single value using len, sum, etc.

An Example – Count Monkeys By Type

```
>>> print [(k, len(list(g))) for k, g
           in itertools.groupby(
               sorted(primates, key=operator.itemgetter('type')
               key = operator.itemgetter('type'))]
[('gorilla', 1), ('monkey', 3)]
```

Other Tricks…

Some things to think about…

- Can iterate across ranges of the original list
  - Note: used an inline if statement in output expression to handle default values (zero if items has insufficient history to calculate)

```
>>> def moving_average (source_list, period):
    return [float(sum(source_list[i+1 - period:i+1]))/period
                if i >= period -1 else 0
                for i in range(0,len(source_list))]

>>> range(0,20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> moving_average(range(0,20),7)
[0, 0, 0, 0, 0, 0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0]
```

- It is possible use a list comprehension purely for a side-effect of the function being applied to a list (printing, accumulating values)
  - This is generally considered bad style (readability, efficiency)

Generators / Generator Comprehensions

- Not the focus of this talk, but worth keeping in mind for large series…
  - Python object which generates a series of values, returning one at a time
  - Internal syntax similar to a function but:
    - Uses a yield statement instead of a return statement
    - Internally remembers position within the series
    - Can get the next value by calling next(<object ref>)
- Key Benefit – don't need to create / retain entire list in memory
- However…
  - Single Pass – need to re-create if you want to rewind
  - This can be an issue if the data source is "expensive" (database, web service)
- Generator comprehensions – formed by using () instead of []

```
def generator_example(source_list):
    return (elem['name'] for elem in source_list if elem['type']=='monkey')
```

The Complete Works Of Shakespeare

- Read A File
- Create Dictionary with Word Counts
- Sort & Identify Top 25 Words

- Note: nested list comprehension (upgraded to generator comprehension), sort function modifies initial list

```
import sys, pprint
file = open('shakespeare.txt', 'r')

# Code To Build Dictionary of Word Counts
word_counts = {}

# List Comp -> Generator Expression for Efficiency
get_words = (word.upper() for line in file for word in line.split())

# For Loop To Build Dictionary (avoiding side effects)
for word in get_words:
    word_counts[word] = word_counts.get(word, 0) + 1

print "Number of Words", len(word_counts)

# Code To Print Most Common Words
results = [item for item in
     sorted(word_counts.items(),
            key=lambda(k,v):(v,k), reverse=True)[:25]]

print "Most Common Words"
for elem in results:
    print "%s %s" %(str(elem[0]).ljust(15),str(elem[1]).rjust(5))
```

```
>>>
Number of Words 59739
Most Common Words
THE              27729
AND              26099
I                19540
TO               18763
OF               18126
A                14436
MY               12455
IN               10730
YOU              10696
THAT             10501
IS                9168
FOR               8000
WITH              7981
NOT               7663
YOUR              6878
HIS               6749
BE                6717
THIS              5930
AS                5893
BUT               5891
HE                5886
IT                5879
HAVE              5683
THOU              5138
ME                4851
```