

CS3101.3 Python: Lecture 3



This week

- Assignment 3
- Python 2 vs. Python 3
- Wrapping up functions
- Functional (and iterative) programming tools
- Regular expressions
- News:
 - Solutions to hw's 1,2 posted will be posted to courseworks on Thurs
 - Python 3.1 is now available on cunix.cc.columbia.edu

Assignment 3 (ex 1 of 2)

Sport Recommender

- Requirements:
 - Write a script which recommends a sport to play based on today's weather
 - Retrieve the current temperature using URLLIB, Regular Expressions, and a web service of your choice
 - See:
<http://developer.yahoo.com/weather/>
- Suggested output:
 - \$python sport.py
 - It's 36 degrees, you should ski!

Recommended web service:

<http://weather.yahooapis.com/forecastrss?w=12761356>

Yahoo! Weather – New York, NY

Yahoo! Weather for New York, NY

[Conditions for New York, NY at 7:51 pm EST](#)



Current Conditions:

Fair, 32 F

Forecast:

Mon – Partly Cloudy. High: 35 Low: 26

Tue – Cloudy. High: 36 Low: 31

Assignment 3 (ex 2 of 2)

News Parser

- Write a script which retrieves Columbia's webpage and prints only the titles of the news stories on the main page
- Use regular expressions and string operations

Suggested output:

- \$python news.py
- Today's stories are:
- 1. Alumnus Judd Gregg....



Python 2 vs. Python 3

Python 2 vs. Python 3

- *Intentionally backwards incompatible* (but mostly the same)
- Notable changes: dictionaries, strings, print
- Why? Removed many deprecated features, reorganized standard library, more modern approach
- Fairly easy to transition between them
- Python 3 is still in the oven
 - Most major libraries primarily support only 2.x
 - Most large scale new projects lean toward 3.x
 - Aims to be the emerging standard over a 5 year timeframe
- Reference:
 - <http://docs.python.org/3.1/whatsnew/3.0.html>

Print is now a function

Motivation: keyword arguments make the advanced functionality more accessible

Old: `print "The answer is", 2*2`

New: `print("The answer is", 2*2)`

Old: `print # Prints a newline`

New: `print() # You must call the print function!`

Old: `print >>sys.stderr, "fatal error"`

New: `print("fatal error", file=sys.stderr)`

`print("There are <", 2**32, "> possibilities!", sep="*")`

There are <4294967296> *possibilities!*

Views and Iterators instead of Lists

- **Motivation:** memory efficiency (avoid unnecessary copy), support dynamic refresh, allow arbitrary sizes
- [dict](#) methods [dict.keys\(\)](#), [dict.items\(\)](#) and [dict.values\(\)](#) return “views” instead of lists
 - Views are dynamic collections which provide a window into an object, and change with that object
- this no longer works: `k = d.keys(); k.sort()`. Use `k = sorted(d)` instead (this works in Python 2.x too and is just as efficient).
 - For a quick fix, use `list(d.keys())`
 - Also, the `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` methods are redundant and longer supported.
- [map\(\)](#) and [filter\(\)](#) return iterators. If you really need a list, a quick fix is e.g. `list(map(...))`
- [range\(\)](#) now behaves like `xrange()` used to behave, except it works with values of arbitrary size

Text vs. Data instead of Unicode vs. 8-bit

- Motivation: unicode is the future, language transparency “allows programs to consistently represent and manipulate text expressed in most of the world's writing systems”
 - UTF-8: *variable-length* encoding system for Unicode. That is, different characters take up a different number of bytes.
- Python 3.0 uses the concepts of **text** and (binary) **data** instead of Unicode strings and 8-bit strings
- An immutable sequence of numbers-between-0-and-255 is called a *bytes* object.
 - the type used to hold data is [bytes](#)
- An immutable sequence of Unicode characters is called a *string*.
 - The type used to hold text is [str](#)
- All text is Unicode; however *encoded* Unicode is represented as binary data.
- As the [str](#) and [bytes](#) types cannot be mixed, you must always explicitly convert between them.
- Use [str.encode\(\)](#) to go from [str](#) to [bytes](#)
- Use [bytes.decode\(\)](#) to go from [bytes](#) to [str](#).

Encoding and decoding bytes

- `>>> by = b'abcd\x65'`
- `>>> by`
`b'abcde'`
- `>>> type(by)`
`<class 'bytes'>`
- `>>> by.decode('utf-8') # (or 'ascii')`
`'abcde'`
- `>>> a_string = '深入 Python'`
- `>>> len(a_string)`
`9`
- `>>> by = a_string.encode('utf-8')`
- `>>> by`
`b'\xe6\xb7\xb1\xe5\x85\xa5 Python'`
- `>>> a_string.encode('ascii')`
`UnicodeEncodeError: 'ascii' codec can't encode`
`characters in position 0-1: ordinal not in range(128)`

Finishing up functions

Returning multiple values from functions

- Functions may return multiple values of arbitrary types, separated by commas

```
>>> def x():  
    ... return 1, 'a', [2, 3]  
>>> x  
<function x at 0x2b3348>  
>>> x()  
(1, 'a', [2, 3])  
>>> type(x())  
<class 'tuple'>
```

Review

Positional vs. Named arguments

Positional argument

- are just expressions
- supplies the value for the parameter that corresponds to it by the order in the function definition
- Are the usual scenario in C or JAVA
- Disadvantages:
 - Potential for typos, poor readability
 - Heaven help us deciphering functions which take long lists of arguments

Named arguments

- bind optional parameters to specific values, while letting other optional parameters take default ones
- may be specified in any order
- Great for readability / reliability purposes
 - Very hard to make a mistake

Examples

```
>>> def f(middle, begin='homer', end='donuts'):
...     return begin + ' ' + middle + ' ' + end
...
>>> f('likes')
'homer likes donuts'
>>> f(begin='lisa', middle='likes',
      end='veggies')
'lisa likes veggies'
>>> f('exercises', end='rarely')
'homer exercises rarely'
```

Optional arguments are everywhere

Python 3

```
>>> range(5)
range(0, 5)
>>> range(-5, 5)
range(-5, 5)
>>> range(-5,5,2)
range(-5, 5, 2)
>>> for val in range(-5,5,2):
...     print(val)
-5
-3
-1
1
3
```

Python 2

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(-5,5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
>>> range(-5,5,2)
[-5, -3, -1, 1, 3]
>>> xrange(5)
xrange(5)
```

`*`: Sequences of positional arguments

`*` collects unmatched positional arguments into a tuple

```
>>> def f(*args):  
...     print (args)  
...     print (type(args))  
...  
>>> f('homer', 'donuts', 'duffbeer')  
('homer', 'donuts', 'duffbeer')  
<class 'tuple'>
```


******: Sequences of Named Arguments

****** collects keyword arguments into a dictionary

```
>>> def f(**args):  
...     print (args)  
...     print (type(args))  
...  
>>> f(homer='donuts', lisa='veggies')  
{'homer': 'donuts', 'lisa': 'veggies'}  
<class 'dict'>
```

Argument matching rules

- General rule: more complicated to the right
- For both calling and defining functions
- All positional arguments must appear first
- Followed by all keyword arguments
- Followed by the * collections
- Followed by ** collections
- For maximally readable code, generally use named arguments when possible

File handling w/ the OS module

- Last time, we talked about files
- The os module provides some handy utility functions and attributes for file handling
 - Checking if a file exists
 - Getting the current directory
 - Checking if a path is a file or a directory
- Browsing the file system

File tests

- `import os`
- File tests
 - `os.path.exists('file')`, `os.path.isfile('file')`, `os.path.isdir('file')`
- Joining paths (proper formatting for the underlying OS as opposed to `'path1' + '/' + 'path2'`)
 - `os.path.join('path1', 'path2')`
- Path information
 - `os.getcwd()`, `os.path.walk(...)`

Exploring a directory structure

```
>>> i=0
>>> for (path, dirs, files) in os.walk(path):
...     i+=1
...     if i > 5: break
...     print (path)
...     print (dirs)
...     print (files)
...     print ('---')
...
/Developer
['About Xcode.app', 'Applications', 'Documentation',
 'Examples', 'Extras', 'Headers', 'Library',
 'Makefiles', 'Platforms', 'SDKs', 'Tools', 'usr']
['Icon\r']
---
/Developer/About Xcode.app
...
```

Generators, itertools, functional programming tools

Generators

- Generators are like normal functions in most respects but they automatically implement the iteration protocol to return a sequence of values over time
- Consider a generator when
 - you need to compute a series of values lazily
 - you need to work with an infinite series
- Generators use yield instead of return: that's it!
- When a yield statement is executed, the function execution is “frozen”
- Local variables, point of execution saved
- Expression after yield keyword returned
- If the function body ends, or a return statement is executed, an exception is raised to indicate the end of the iteration

Generators 101

Python 2

```
>>> def gen():
...     yield "first"
...     yield "second"
...
>>> f = gen()
>>> f.next()
'first'>>> f.next()
'second'
>>> f.next()
Traceback (most recent
call last):
  File "<stdin>", line 1,
    in <module>
StopIteration
```

Python 3

```
>>> def gen():
...     yield "first"
...     yield "second"
...
>>> f = gen()
>>> next(f)
'first'
>>> next(f)
'second'
>>> next(f)
Traceback (most recent call
last):
  File "<stdin>", line 1, in
    <module>
StopIteration
```


Infinite generators

```
# python 3.1
>>> def gen_squares():
...     i = 0
...     while True:
...         yield i ** 2
...         i += 1
...
>>> f = gen_squares()
>>> next(f)
0
>>> for i in range(4):
...     next(f)
...
1
4
9
```

Itertools

- Functions creating iterators for efficient looping
- Itertools implements a number of iterator building blocks inspired by APL, Haskell, and SML
- Produce sequences efficiently and elegantly
 - standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination.
 - Together, they form an “iterator algebra” making it possible to construct specialized tools
- See
 - <http://docs.python.org/library/itertools.html>

Itertools: cycle

```
>>> import itertools
>>> homer = ['donuts', 'more']
>>> itertools.cycle(homer)
<itertools.cycle object at 0x2b6df0>
>>> x = itertools.cycle(homer)
>>> next(x)
'donuts'
>>> next(x)
'more'
>>> next(x)
'donuts'
```

Itertools: permutations / combinations

- Iteratively returns r-length tuples in sorted order, no repeated elements

```
>>> import itertools
>>> itertools.permutations(p='abc', r=2)
>>> itertools.permutations('abc', 2)
<itertools.permutations object at
 0x2b18d0>
>>> list(itertools.permutations('abc', 2))
[('a', 'b'), ('a', 'c'), ('b', 'a'), ('b',
 'c'), ('c', 'a'), ('c', 'b')]
```

Many more..

- count
- cycle
- repeat
- chain
- dropwhile
- groupby
- ifilter
- islice
- imap
- starmap
-

Functools

- Advanced functionality
- Higher order functions and operations on callable objects
 - functions that act on or return other functions.
- See
 - <http://docs.python.org/library/functools.html>

Lambda Expressions

- In addition to the def statement, functions can also be defined using the lambda expression
- Since lambda is an expression, it can be in-lined similarly to Lisp
- Lambda functions are **anonymous** . No name is assigned: lambda returns the function itself
- Elegant when simple, unnecessarily obfuscated when complex

When to Use Lambda

- When being concise is reasonable
- Compared to `def()`:
 - Lambda's are expressions, `def` is statement
 - Lambda can be used in places `def` cannot
 - e.g., in the middle of a list declaration
 - or even inside a function call as a parameter
- Limitations:
 - Not as general as `def`: limited to a single expression
- Avoid sacrificing readability! A lot of programmers erroneously believe complex code is better – the opposite is true.

Lambda's 101

```
>>> lambda x: x + 5
<function <lambda> at 0x2b33d8>
>>> (lambda x: x + 5) (1)
6
>>> foo = (lambda x : x + ' ' + 'simpson')
>>> foo('lisa')
'lisa simpson'
>>> foo('homer')
'homer simpson'
```

Lambdas are Extremophiles

```
>>> # embedding in a list
>>> funcs = [(lambda x: x**2), (lambda x:
    x**3)]
>>> for f in funcs:
...     print(f(2), end = " ")
4 8
```

```
>>> # embedding in a dictionary
>>> ops = {'double' : (lambda x: x*2),
    'triple' : (lambda x: x * 3)}
>>> ops['double'](9)
18
```

Using Lambda (cont'd)

- Multiple arguments

- separate arguments by comma

```
(lambda x,y : x * y)(5,10) # 50
```

- State is maintained (like closures)

```
def rem(x):
```

```
    return (lambda y: x + y)
```

```
f = rem(10)
```

```
f(5) # 15
```

Map

- Common task: apply an operation to each element in a sequence
- Syntax:
 - `map(function, sequence)`
 - Calls `function(item)` for each item of sequence and returns a list of the return values

```
>>> donuts = [1,2,3]
>>> more = lambda x: x * 2
>>> map(more, donuts)
<map object at 0x2b79f0>
>>> list(map(more, donuts))
[2, 4, 6]
```

Remember: in Python 3.x `Map` and `Filter` return iterators, in Python 2.x, a list

Multiple map arguments

- Map is smart
- Map (func, **sequences**) will accept N sequences provided that the sequences correspond to the arguments expected by the function

```
>>> list(map(pow, [2, 4, 6], [1, 2, 3]))  
[2, 16, 216]
```

Filter

- Syntax:
 - filter(function, sequence)
 - Returns items of sequence iff function (item) evaluates to True
 - If the sequence is a string or a tuple, the returned value will be of the same type, otherwise, a list is returned

```
>>> def is_odd(x):  
...     return (x % 2) != 0  
>>> x = [1,2,3,4,5,6]  
>>> filter(is_odd, x)  
<filter object at 0x2b7af0>  
>>> list(filter(is_odd, x))  
[1, 3,5]
```

Remember: in Python 3.x Map and Filter return iterators, in Python 2.x, a list

Reduce

- Forewarning, toast in Python 3, but worth knowing about
- Applies a function to pairs of items in a sequence, producing a running result
- Syntax:
 - `reduce(function, sequence)`
 - function must take two arguments
 - First two items of sequence are used as the first two arguments to the function
 - Then, function is applied to the previous result and the next item in sequence
- Example:
 - `reduce(lambda x, y: x + y, range(10))`
 - Returns the sum (note: don't do this---there's already a `sum()` function)

Regular expressions

Regular Expressions (RE)

- A RE is a string that represents a pattern
- Their purpose is to test another string against the pattern
 - Discovering if any part of that string matches the pattern, and if so, where
- Very powerful – a bit of a bear syntactically
 - Can be used to match, search, replace, and split strings
- REs may be compiled or used on the fly
- Omnipresent in scripting languages (the ongoing joke: Perl REs are powerful enough to write just about any program).

Searching with REs

```
>>> import re
>>> pattern = re.compile('[a-z]+')
>>> m = pattern.search('5 donuts')
>>> print (m.group(), m.start(),
          m.end())
donuts 2 8
```

Matching with REs

```
>>> import re
>>> pattern = re.compile('[a-z]+')
>>> m = pattern.match('5 donuts')
>>> if m:
...     print ('Matched!')
... else:
...     print ('Failed')
...
Failed
```

Findall

```
>>> import re
>>> p = re.compile('\d\sdonut')
>>> hits = p.findall('homer has 4
    donuts, bart has 1 donut')
>>> print (hits)
['4 donut', '1 donut']
```

Findall (cont'd)

```
>>> import re
>>> p = re.compile('\d+\sdonuts')
>>> iter = p.finditer('99 donuts on the
    shelf, 98 donuts on the shelf...')
>>> for m in iter:
...     print (m.group(), m.span())
...
99 donuts (0, 9)
98 donuts (24, 33)
```

The syntax of patterns

- Alphanumeric characters match themselves
- A RE that is just a string of letters and digits will match the same string
- Punctuation is the opposite
 - Wildcard characters with special meaning
 - To return them to normal, they must be escaped (e.g., preceded by a backslash: \. or \{)
- Backslash character matched by repeated backslash (\\)

Common Patterns

Element	Meaning
.	Matches any character
^	Matches start of string
\$	Matches end of string
*	Matches zero or more cases of previous RE (greedy)
+	Matches one or more cases of previous RE (greedy)
?	Matches zero or one case of the previous RE (greedy)
*?, +?, ??	Nongreedy versions of *, +, ?

Common Patterns (cont'd)

Element	Meaning
{m, n}	Matches m to n cases of the previous RE (greedy)
[...]	Matches any one of a set of characters specified in brackets
	Matches either the preceding or following expression
(...)	Matches RE within group and indicates a group
\d, \D	Matches a digit, non-digit resp. (Like, [0-9] and [^0-9] resp.)
\s, \S	Matches whitespace (\t, \n, \r, \f, \v), non-whitespace resp.
\w, \W	Matches one alphanumeric character
\b, \B	Matches an empty string at the start or end of the word
\Z	Matches empty string at the end of a whole string

Character sets

- Sets of characters can be denoted by listing characters within brackets []
- You can denote a range of characters by giving the first and last characters separated by a hyphen
 - E.g. [a-z], first and last characters inclusive
 - Within a set, special characters stand for themselves
 - Except for \, [, and –

Alternatives: |s

- A vertical bar matches a pattern on either side

```
import re
p = re.compile('Homer|Simpson')
iterator=p.finditer("HomerJaySimpson")
for match in iterator:
    print (match.group(), match.span())
Homer (0, 5)
aco (8, 12)
```

Groups

- Groups are used to extract segments of a string that matched a pattern, or a segment of a pattern
- A RE can contain any number of groups
- Parentheses in a pattern indicate a group

```
import re
p = re.compile('(homer\s(jay))\ssimpson')
m = p.match('homer jay simpson')
print (m.group(0))
-- homer jay simpson
print (m.group(2))
-- jay
```

Optional Flags

- The compile function in the re module accepts optional flags
 - `re.compile(pattern, flags)`
- Some attributes:
 - `re.IGNORECASE`, `re.MULTILINE`, `re.VERBOSE`, `re.DOTALL`
- Example:
 - `re.compile('hello', re.IGNORECASE)`

RE Substitution

- Substitutions can be made based on regular expressions
- Syntax:
 - `r.sub(repl, s, count=0)`
 - Copy of `s` is returned where nonoverlapping matches with `r` is replaced by `repl`
 - When `count` is greater than 0, only the first count matches are replaced, otherwise, all are replaced
- Example:
 - `r = re.compile('world', re.IGNORECASE)`
 - `print (r.sub('Mars!', 'Hello world!', 1))`

Splitting

- Strings can also be split based on regular expressions
- Syntax:
 - `r.split(s, maxsplit=0)`
 - List of splits of `s` by `r` (i.e. substrings of `s` separated by nonoverlapping, nonempty matches with `r`) is returned
 - If `maxsplit` is greater than 0, then at most `maxsplit` splits are returned, otherwise, all splits are returned
- Example:
 - `r = re.compile('\d+')`
 - `print (r.split('lots 42 of random 12 digits 77'))`
 - `['lots', 'of', ...]`

Finding tags w/in HTML

```
>>> line = '<tag>my eyes! the goggles  
do nothing!</tag>'  
>>> r = re.compile('<tag>(.*?)</tag>',  
re.DOTALL)  
>>> m = r.search(line)  
>>> print (m.group(1))  
my eyes! the goggles do nothing!
```

References

- A great tutorial:
 - <http://www.amk.ca/python/howto/regex/>

A Word on Documentation

- Code is usually read far more than it is written
- It is worth it to document your code!
- Docstrings can be written for classes, modules and methods
- Usually consists of one sentence, followed by a blank, then a more detailed description
- Guideline for writing docstrings:
 - First line should be a concise and descriptive statement of purpose
 - Self-documentation is good, but do not simply repeat variable / method names
 - Next, describe the method and side effects
 - Describe arguments

Style Reference

- PEP 8
 - <http://www.python.org/dev/peps/pep-0008/>
- Google Python Style Guide
 - <http://code.google.com/p/soc/wiki/PythonStyleGuide>

Assignment 3

Due before class next week

Assignment 3 (ex 1 of 2)

Sport Recommender

- Requirements:
 - Write a script which recommends a sport to play based on today's weather
 - Retrieve the current temperature using URLLIB, Regular Expressions, and a web service of your choice
 - See:
<http://developer.yahoo.com/weather/>
- Suggested output:
 - \$python sport.py
 - It's 36 degrees, you should ski!

Recommended web service:

<http://weather.yahooapis.com/forecastrss?w=12761356>

Yahoo! Weather – New York, NY

Yahoo! Weather for New York, NY

[Conditions for New York, NY at 7:51 pm EST](#)



Current Conditions:

Fair, 32 F

Forecast:

Mon – Partly Cloudy. High: 35 Low: 26

Tue – Cloudy. High: 36 Low: 31

Assignment 3 (ex 2 of 2)

News Parser

- Write a script which retrieves Columbia's webpage and prints only the titles of the news stories on the main page
- Use regular expressions and string operations

Suggested output:

- \$python news.py
- Today's stories are:
- 1. Alumnus Judd Gregg....

