Programming in Python

Week 4

Syllabus:
`http://briandorsey.github.com/uwpython_onsite/`

Brian Dorsey
brian@dorseys.org
http://briandorsey.info

Introductions, comments (...)

readings this week

questions?

Unit A
(25)

list
tuple

these topics are used in nearly all Python programs

this week and last are the most important in the whole class

this is where it gets really, really good

list

list literals

```
>>> []
[]

>>> list()
[]

>>> [1, 2, 3]
[1, 2, 3]

>>> [1, 3.14, "abc"]
[1, 3.14, 'abc']
```

indexing

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[2]
'ham'
>>> food[0]
'spam'
>>> food[42]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

```
>>> food = ['spam', 'eggs', 'ham']
>>> food[1] = 'rasberries'
>>> food
['spam', 'rasberries', 'ham']
```

each element is a value, and can be in multiple lists and have multiple names

names

```
>>> name = 'Brian'
>>> a = [1, 2, name]
>>> b = [3, 4, name]
>>> name
'Brian'
>>> a
[1, 2, 'Brian']
>>> b
[3, 4, 'Brian']
>>> a[2]
'Brian'
>>> b[2]
'Brian'
```

append(), insert()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.append('sushi')
>>> food
['spam', 'eggs', 'ham', 'sushi']
>>> food.insert(0, 'carrots')
>>> food
['carrots', 'spam', 'eggs', 'ham', 'sushi']
```

extend

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.extend(['fish', 'chips'])
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

pop(), remove()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.pop()
'ham'
```

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.pop(0)
'spam'
```

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.remove('ham')
>>>
```

list() accepts any sequence and returns a list of that sequence

```
>>> word = 'Python '
>>> chars = []
>>> for char in word:
...     chars.append(char)
...
>>> chars
['P', 'y', 't', 'h', 'o', 'n', ' ']
```

```
>>> list(word)
['P', 'y', 't', 'h', 'o', 'n', ' ']
```

if you need to change individual letters... you can do this, but usually somestring.replace() will be enough

```
>>> name = 'Bryan'
>>> lname = list(name)
>>> lname[2] = 'i'
>>> name = ''.join(lname)
>>> name
'Brian'
```

[:] makes a shallow copy of a sequence

slices

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food[1:3]
['eggs', 'ham']
```

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food_copy = food[:]
>>> food
['spam', 'eggs', 'ham', 'sushi']
>>> food_copy
['spam', 'eggs', 'ham', 'sushi']
>>> food is food_copy
False
```

this is important

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food_again = food
>>> food_copy = food[:]
>>> food.remove('sushi')
>>> food
['spam', 'eggs', 'ham']
>>> food_again
['spam', 'eggs', 'ham']
>>> food_copy
['spam', 'eggs', 'ham', 'sushi']
```

iteration

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food:
...     print x
...
spam
eggs
ham
sushi
```

common pattern

```
filtered = []
for x in somelist:
    if should_be_included(x):
        filtered.append(x)
del(somelist)  # maybe
```

```
# works, but...
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2


# use this form
for i, num in enumerate(numbers):
    numbers[i] = num * 2


# or list comprehensions
numbers = [n * 2 for n in numbers]
```

slice trick

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food[:2]:
...     print x
...
spam
eggs
```

if you're going to change the list, iterate over a copy for safety - very insidious bugs otherwise

mutating a list

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> for x in food[:]:
...     # change the list somehow
...
```

choose based on clarity, also... follow the names and values

operators vs methods

```
>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food = food + more
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

```
>>> food = ['spam', 'eggs', 'ham']
>>> more = ['fish', 'chips']
>>> food.extend(more)
>>> food
['spam', 'eggs', 'ham', 'fish', 'chips']
```

http://docs.python.org/library/operator.html#mapping-operators-to-functions

in

```
>>> food = ['spam', 'eggs', 'ham']
>>> 'eggs' in food
True
>>> 'chicken feet' in food
False
```

reverse()

```
>>> food = ['spam', 'eggs', 'ham']
>>> food.reverse()
>>> food
['ham', 'eggs', 'spam']
```

sort()

```
>>> food = ['spam', 'eggs', 'ham', 'sushi']
>>> food.sort()
>>> food
['eggs', 'ham', 'spam', 'sushi']
```

how should this sort?

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]
```

```
>>> s
[[2, 'a'], [1, 'b'], [1, 'c'], [1, 'a'], [2, 'c']]
>>> s.sort()
>>> s
[[1, 'a'], [1, 'b'], [1, 'c'], [2, 'a'], [2, 'c']]
```

mixed types can be sorted

"objects of different types always compare unequal, and are ordered consistently but arbitrarily."

http://docs.python.org/reference/expressions.html#not-in

usually a list will contain values of the same type

simplifies for loops, sorting

most of the list methods return None

s.append('a').sort() fails

AttributeError: 'NoneType' object has no attribute 'sort'

- indexing is fast and constant time: O(1)
- x in s proportional to n: O(n)
- visiting all is proportional to n: O(n)
- operating on the end of list is fast and constant time: O(1)
  `append(), pop()`
- operating on the front of the list depends on n: O(n)
  `pop(0), insert(0, v)`
  But, reversing is fast. Also, collections.deque
- http://wiki.python.org/moin/TimeComplexity

tuple

also a sequence of values

work mostly like strings and lists

immutable, like strings

basically, any bare comma separated values are actually tuples

```
>>> (1, 2, 3, 4)
(1, 2, 3, 4)

>>> 1, 2, 3, 4
(1, 2, 3, 4)

>>> (1,)
(1,)

>>> 1,  # ick!
(1,)

>>> tuple([1, 2, 3, 4])  # any sequence
(1, 2, 3, 4)

>>> ()  # kinda ick
()

>>> tuple()
()
```

Don't work:

```
>>> (1)
1
>>> (1 + 1)
2
```

indexing works

immutable like strings

slices return tuples

tuple unpacking

```
>>> a, b = b, a
```

```
>>> first, last, score = ('Brian', 'Dorsey', 42)
>>> first
'Brian'
>>> last
'Dorsey'
>>> score
42
```

how to choose between tuple and list?

- base your decision on clarity and performance of your code
- tuples - when you want to group multiple values into one logical thing
- lists - collections of similar items
- lists are collections, tuples are philosophically closer to structs or classes

- do the same operation to each element?
- small collection of values which make a single logical item?
- to document that these values won't change?
- build it iteratively?
- transform, filter, etc?

- do the same operation to each element? **list**

- small collection of values which make a single logical item? **tuple**

- to document that these values won't change? **tuple**

- build it iteratively? **list**

- transform, filter, etc? **list**

collections.namedtuple()

```
>>> Rating = collections.namedtuple('Rating',
['name', 'score'])
>>> r = Rating('Brian', 42)
>>> r[0]
'Brian'
>>> r.name
'Brian'
>>> r
Rating(name='Brian', score=42)
>>> name, score = r
>>> name
'Brian'
>>> score
42
```

Lightning Talks!

(15)

Unit B
(25)

dict
set

dict

dictionary
associative array
map
hash table
hash
key/value pair

which to use depends what shape your data is in

```
>>> {'key' : 'value'}
{'key': 'value'}

>>> dict([('key', 'value')])
{'key': 'value'}

>>> dict(key='value')
{'key': 'value'}

>>> d = {}
>>> d['key'] = 'value'
>>> d
{'key': 'value'}
```

somelist[0] is offset, somedict[0] is a lookup

indexing

```
>>> d = {'name': 'Brian', 'score': 42}
>>> d['score']
42

>>> d = {1: 'one', 0: 'zero'}
>>> d[0]
'zero'

>>> d['non-existing key']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'fish'
```

```
>>> d = {'name': 'Brian', 'score': 42}
>>> d.get('name')
'Brian'

>>> d.get('name', 'unknown')
'Brian'

>>> d.get('non-existing key', 'unknown')
'unknown'
```

```
>>> d = {'name': 'Brian', 'score': 42}
>>> 'score' in d
True
```

~~dictionaries are unordered~~

dictionaries have no defined order

also... Python classes are implemented with dict, it's highly optimized

key lookup is very efficient

same average time regardless of size

http://wikipedia.org/wiki/Hash_table

dictionary keys must be ‘hashable’

hash functions
convert a large data
to a small proxy
(usually int)

always return the same
proxy for the same input

MD5, SHA, etc

dictionaries hash the key to an integer proxy and use it to find the key and value

key lookup is efficient because the hash function leads directly to a bucket with a very few keys (often just one)

what would happen if the proxy changed after storing a key?

hashability requires immutability

dictionary keys must be immutable

key → value
lookup is one way

value → key
requires visiting the whole dict

if you need to check dict values often, create another dict or set (up to you to keep them in sync)

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for x in d:
...     print x
...
score
name
```

```
>>> for x in d:
...     print repr(x)
...
'score'
'name'
```

```
>>> d.keys()
['score', 'name']

>>> d.values()
[42, 'Brian']

>>> d.items()
[('score', 42), ('name', 'Brian')]
```

```
>>> d = {'name': 'Brian', 'score': 42}
>>> for k, v in d.items():
...     print "%s: %s" % (k, v)
...
score: 42
name: Brian
```

- indexing is fast: constant time O(1)
- x in s constant time typically O(1)
- visiting all is proportional to n O(n)
- http://wiki.python.org/moin/TimeComplexity

set

unordered collection of distinct values

sets are like dicts with only keys

```
>>> set()
set([])

>>> set([1, 2, 3])
set([1, 2, 3])

# as of 2.7
>>> {1, 2, 3}
set([1, 2, 3])

>>> s = set()
>>> s.update([1, 2, 3])
>>> s
set([1, 2, 3])
```

```
>>> s = set()
>>> s.update([1, 2, 3])
>>> s
set([1, 2, 3])
>>> s.update([2, 3, 4, 5])
>>> s
set([1, 2, 3, 4, 5])
```

```
>>> s = set([1, 2, 3, 4, 5])
>>> 3 in s
True
>>> 42 in s
False
```

set members
must be hashable
like dictionary keys
and for same reason
(efficient lookup)

but, no index operator

```
>>> s[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

```
>>> s = set([1, 2, 3])
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
>>> s.add(4)
>>> s
set([1, 2, 3, 4])
>>> s.update([2, 4, 6, 8, 'who do we appreciate?'])
>>> s
set([1, 2, 3, 4, 6, 8, 'who do we appreciate?'])
```

ooh! KeyError!

```
>> s = set([1])
>>> s.pop()
1
>>> s.pop()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pop from an empty set'
```

```
>>> s = set([1, 2, 3])
>>> s.remove(2)
>>> s.remove(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
```

- `s.isdisjoint(other)`
- `s.issubset(other)`
- `s.union(other, ...)`
- `s.intersection(other, ...)`
- `s.difference(other, ...)`
- `s.symmetric_difference(other, ...)`

also: frozenset

when you want an
immutable set
(to use as a key, etc.)

Lab B (20)

```
git pull

svn update
```

Lab B

- Coding Kata 14 - Dave Thomas
  http://codekata.pragprog.com/2007/01/kata_fourteen_t.html
- See how far you can get on this task using "The Adventures of Sherlock Holmes" as input: sherlock.txt in the week04 directory (UTF-8)
- This is intentionally open-ended and underspecified. There are many interesting decisions to make.

Unit C
(25)

data structures and advice

aliasing / references

names, list items, tuple items, dict values, function parameters... later instance variables, etc

demo references to mutable values (compared to immutable)

sounds obvious when you say it this way... can be surprising in practice

function parameters are references to the values

when mutable, they can be changed!

```
>>> def some_function(x, z):
...     x = x * 2
...     z.append(x)
...     return z
...
>>> s = []
>>> some_function(2, s)
[4]
>>> s
[4]
>>> some_function(8, s)
[4, 16]
>>> s
[4, 16]
```

this is a shallow copy, only protects the list not mutable values in the list - see copy.deepcopy()

sending in a copy

```
>>> def some_function(x, z):
...     x = x * 2
...     z.append(x)
...     return z
...
>>> s = []
>>> some_function(2, s[:])
[4]
>>> s
[]
>>> some_function(8, s[:])
[16]
>>> s
[]
```

what about default values?

```
>>> def some_function(x=1, y='abc', z=[]):
...     x = x * 2
...     z.append(x)
...     return z
...
>>> a = 2
>>> some_function(a)
[4]
>>> a
2
>>> some_function(12)
[4, 24]
>>> some_function(42)
[4, 24, 84]
```

*args **kwargs

```
>>> def anything(*args, **kwargs):
...     print args
...     print kwargs
...
>>> anything()
()
{}
>>> anything(1)
(1,)
{}
>>> anything(1, 'hi', [1], key='value')
(1, 'hi', [1])
{'key': 'value'}
```

```
>>> def notekeeper(function, *args, **kwargs):
...     print 'Calling %s with %s and %s' % (
                function.__name__, args, kwargs)
...     ret = function(*args, **kwargs)
...     print 'Returned: %s' % ret
...     return ret
...
>>> notekeeper(ord, 'a')
Calling ord with ('a',) and {}
Returned: 97
97
>>> notekeeper(range, 3, 5)
Calling range with (3, 5) and {}
Returned: [3, 4]
[3, 4]
```

```
>>> d = {'name': 'brian'}
>>> d
{'name': 'brian'}
>>> dict(score=42, **d)
{'score': 42, 'name': 'brian'}
```

```
# remember?
>>> a, b = b, a

# it's just tuples
>>> a, b = (1, 2)
>>> a, b = [1, 2]
>>> a, b = 1, 2
```

returning multiple values

```
>>> import os.path
>>> file, ext = os.path.splitext('secrets.txt')
>>> file
'secrets'
>>> ext
'.txt'
>>> ret = os.path.splitext('secrets.txt')
```

Does this work? If so, what is ret?

awesome feature doesn't actually exist?!?!

```
>>> import os.path
>>> file, ext = os.path.splitext('secrets.txt')
>>> file
'secrets'
>>> ext
'.txt'
>>> ret = os.path.splitext('secrets.txt')
>>> ret
('secrets', '.txt')
>>> type(ret)
<type 'tuple'>
```

choosing datatypes

when to use what?

list

tuple

dict

set

list of lists

sometimes namedtuple or custom object (csv readers, SQL drivers, for example)

list of dict

or collections.Counter

dict of int

see collections.defaultdict for a clean way of dealing with this common pattern

dict of lists

pprint()

common patterns

building a string iteratively

```
>>> output = []
>>> output.append(header)
>>> for x in s:
...     output.append(x)
...
>>> output.append(footer)
>>> data = ''.join(output)

# or...
>>> data = '\n'.join(output)
```

use dictionary as switch

tuples as dictionary keys

sorting

but what if you want to sort by some other property of the values?

`list.sort()`

DSU

Decorate
Sort
Undecorate

```
[a, b, c, d]
```

↓

```
[(2,a), (1,b), (42,c), (8,d)]
```

↓

```
[(1,b), (2,a), (8,d), (42,c)]
```

↓

```
[b, a, d, c]
```

`list.sort(key=function)`

`function(i)` is passed each item, returns what should be sorted

only lists have `.sort()`

for everything else:
`sorted()`

sort is stable

does minimal rework

allows multiple sorts

See:

http://wiki.python.org/moin/HowTo/Sorting

collections module

namedtuple

deque

Counter

OrderedDict

defaultdict

Lab C

(20)

```
git pull

svn update
```

Lab C

- section 11.1 of the text shows a histogram function:
  >>> h = histogram('brontosaurus')
  >>> print h
  {'a': 1, 'b': 1, 'o': 2, 'n': 1, 's': 2, 'r': 2, 'u': 2, 't': 1}
- do exercise 11.2
  rewrite histogram(), simplifying using .get()
- rewrite histogram(), simplifying using .setdefault()
- rewrite histogram(), simplifying using collections.defaultdict
- rewrite histogram(), simplifying using collections.Counter
- (or, play with the trigrams more)

wrapup

Assignment

- Spend more time with the Coding Kata from lab B. Get it basically working.
- Experiment with different lengths for the lookup key. (3 words, 4 words, 3 letters, etc)
- Send me a few paragraphs-ish of output & your code as it was at some point. ;)
- This assignment is about playing around with the algorithm and data.