

# Introduction to Python

## Dictionaries, Sets, Exceptions

### Files and Text Processing

Christopher Barker

UW Continuing Education

October 22, 2013

# Table of Contents

- 1 Review/Questions
- 2 Dictionaries and Sets
- 3 Exceptions
- 4 File Reading and Writing
- 5 Paths and Directories

# Review of Previous Class

- Sequences
- Lists
- Tuples

Any questions?

# Lightning Talks

Lightning talks today:

( Jo-Anne Antoun )

Sako Eaton

Brandon Ivers

Gary Pei

Nathan Savage

# Notes on Workflow

For more than a few lines:

Write your code in a module

Have a way to re-run quickly

- Plain command line: `$ python my_script.py`
- iPython: `run my_script.py`
- The “run” button / keystroke in your IDE.

# Finish Last Class...

More on Looping

Strings!

# Lightning Talks

## Lightning Talks:

Jo-Anne Antoun

# Dictionary

Python calls it a dict

Other languages call it:

- dictionary
- associative array
- map
- hash table
- hash
- key-value pair



# Dictionary Constructors

```
>>> {'key1': 3, 'key2': 5}  
{'key1': 3, 'key2': 5}
```

```
>>> dict([('key1', 3), ('key2', 5)])  
{'key1': 3, 'key2': 5}
```

```
>>> dict(key1=3, key2= 5)  
{'key1': 3, 'key2': 5}
```

```
>>> d = {}  
>>> d['key1'] = 3  
>>> d['key2'] = 5  
>>> d  
{'key1': 3, 'key2': 5}
```

# Dictionary 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: 'non-existing key'
```

# Dictionary Indexing

Keys can be any immutable:

- numbers
- string
- tuples

```
In [325]: d[3] = 'string'
```

```
In [326]: d[3.14] = 'pi'
```

```
In [327]: d['pi'] = 3.14
```

```
In [328]: d[ (1,2,3) ] = 'a tuple key'
```

```
In [329]: d[ [1,2,3] ] = 'a list key'
```

```
TypeError: unhashable type: 'list'
```

Actually – any “hashable” type.

# Dictionary Indexing

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

always return the same proxy for the same input

MD5, SHA, etc

# Dictionary Indexing

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)

# Dictionary Indexing

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

Hashability requires immutability

# Dictionary Indexing

Key lookup is very efficient

Same average time regardless of size

also... Python name look-ups are implemented with dict:  
— its highly optimized

# Dictionary Indexing

key to value  
lookup is one way

value to 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)



## Dictionary Ordering (not)

dictionaries have no defined order

```
In [352]: d = {'one':1, 'two':2, 'three':3}
```

```
In [353]: d
```

```
Out[353]: {'one': 1, 'three': 3, 'two': 2}
```

```
In [354]: d.keys()
```

```
Out[354]: ['three', 'two', 'one']
```

# Dictionary Iterating

for iterates the keys

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

note the different order...

## dict keys and values

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

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

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

## dict keys and values

iterating on everything

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

# Dictionary Performance

- indexing is fast and constant time:  $O(1)$
- $x$  in  $s$  constant time:  $O(1)$
- visiting all is proportional to  $n$ :  $O(n)$
- inserting is constant time:  $O(1)$
- deleting is constant time:  $O(1)$

<http://wiki.python.org/moin/TimeComplexity>

# Sets

set is an unordered collection of distinct values

Essentially a dict with only keys

## Set Constructors

```
>>> 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])
```

## Set Properties

Set members must be hashable

Like dictionary keys – and for same reason (efficient lookup)

No indexing (unordered)

```
>>> s[1]
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'set' object does not support indexing
```



## Set Methods

```
>> s = set([1])
>>> s.pop() # an arbitrary member
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
```

# Set Methods

```
s.isdisjoint(other)
```

```
s.issubset(other)
```

```
s.union(other, ...)
```

```
s.intersection(other, ...)
```

```
s.difference(other, ...)
```

```
s.symmetric_difference( other, ...)
```

# Frozen Set

Also frozenset

immutable – for use as a key in a dict  
(or another set...)

```
>>> fs = frozenset((3,8,5))
```

```
>>> fs.add(9)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
AttributeError: 'frozenset' object has no attribute 'add'
```

# LAB

## Dictionary LAB:

`code/dict_lab.html` (rst)

# Lightning Talks

Lightning Talks:

Sako Eaton

Brandon Ivers

# Exceptions

Another Branching structure:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
except IOError:
    print "couldn't open missing.txt"
```

# Exceptions

Never Do this:

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
except:
    print "couldn't open missing.txt"
```

# Exceptions

Use Exceptions, rather than your own tests  
– Don't do this:

```
do_something()
if os.path.exists('missing.txt'):
    f = open('missing.txt')
    process(f)    # never called if file missing
```

it will almost always work – but the almost will drive you crazy



# Exceptions

"easier to ask forgiveness than permission"

– Grace Hopper

<http://www.youtube.com/watch?v=AZDWveIdqjY>  
(Pycon talk by Alex Martelli)

# Exceptions

For simple scripts, let exceptions happen

Only handle the exception if the code can and will do something about it

(much better debugging info when an error does occur)

## Exceptions – finally

```
try:
    do_something()
    f = open('missing.txt')
    process(f)    # never called if file missing
except IOError:
    print "couldn't open missing.txt"
finally:
    do_some_clean-up
```

the finally: clause will always run

## Exceptions – else

```
try:
    do_something()
    f = open('missing.txt')
except IOError:
    print "couldn't open missing.txt"
else:
    process(f) # only called if there was no exception
```

Advantage:  
you know where the Exception came from

## Exceptions – using them

```
try:
    do_something()
    f = open('missing.txt')
except IOError as the_error:
    print the_error
    the_error.extra_info = "some more information"
    raise
```

Particularly useful if you catch more than one exception:

```
except (IOError, BufferError, OSError) as the_error:
    do_something_with (the_error)
```

# Raising Exceptions

```
def divide(a,b):  
    if b == 0:  
        raise ZeroDivisionError("b can not be zero")  
    else:  
        return a / b
```

when you call it:

```
In [515]: divide (12,0)
```

```
ZeroDivisionError: b can not be zero
```

## Built in Exceptions

You can create your own custom exceptions  
But...

```
exp = \
    [name for name in dir(__builtin__) if "Error" in name]

len(exp)
32
```

For the most part, you can/should use a built in one

# LAB

## Exceptions Lab: Improving `raw_input`:

The `raw_input()` function can generate two exceptions: `EOFError` or `KeyboardInterrupt` on end-of-file (EOF) or canceled input.

Create a wrapper function, perhaps `safe_input()` that returns `None` rather than raising these exceptions, when the user enters `^C` for Keyboard Interrupt, or `^D` (`^Z` on Windows) for End Of File.



# Lightning Talks

## Lightning Talks:

Gary Pei

Nathan Savage

# Files

## Text Files

```
f = open('secrets.txt')  
secret_data = f.read()  
f.close()
```

secret\_data is a string

(can also use file() – open() is preferred)

# Files

## Binary Files

```
f = open('secrets.txt', 'rb')  
secret_data = f.read()  
f.close()
```

secret\_data is still a string  
(with arbitrary bytes in it)

(See the struct module to unpack binary data )

# Files

## File Opening Modes

```
f = open('secrets.txt', [mode])
```

'r', 'w', 'a'

'rb', 'wb', 'ab'

r+, w+, a+

r+b, w+b, a+b

U

U+

Gotcha – w mode always clears the file

# Text File Notes

## Text is default

- Newlines are translated: `\r\n` -> `\n`
- – reading and writing!
- Use `*nux`-style in your code: `\n`
- Open text files with `'U'` "Universal" flag

## Gotcha:

- no difference between text and binary on `*nix`
  - breaks on Windows

# File Reading

## Reading Part of a file

```
header_size = 4096
```

```
f = open('secrets.txt')  
secret_data = f.read(header_size)  
f.close()
```

# File Reading

## Common Idioms

```
for line in open('secrets.txt'):  
    print line
```

```
f = open('secrets.txt')  
while True:  
    line = f.readline()  
    if not line:  
        break  
    do_something_with_line()
```

# File Writing

```
outfile = open('output.txt', 'w')

for i in range(10):
    outfile.write("this is line: %i\n"%i)
```



# File Methods

## Commonly Used Methods

`f.read()` `f.readline()` `f.readlines()`

`f.write(str)` `f.writelines(seq)`

`f.seek(offset)` `f.tell()`

`f.flush()`

`f.close()`

# File Like Objects

## File-like objects

Many classes implement the file interface:

- `loggers`
- `sys.stdout`
- `urllib.open()`
- pipes, subprocesses
- `StringIO`

[http://docs.python.org/library/stdtypes.html#  
builtin-file-objects](http://docs.python.org/library/stdtypes.html#builtin-file-objects)

# StringIO

## StringIO

```
In [417]: import StringIO
```

```
In [420]: f = StringIO.StringIO()
```

```
In [421]: f.write("somestuff")
```

```
In [422]: f.seek(0)
```

```
In [423]: f.read()
```

```
Out[423]: 'somestuff'
```

handy for testing

# Paths

Relative paths:

```
secret.txt  
./secret.txt
```

Absolute paths:

```
/home/chris/secret.txt
```

Either work with `open()`, etc.

(working directory only makes sense with command-line programs...)

# os.path

```
os.getcwd() -- os.getcwdu()  
chdir(path)
```

```
os.path.abspath()  
os.path.relpath()
```

# os.path

```
os.path.split()  
os.path.splitext()  
os.path.basename()  
os.path.dirname()  
os.path.join()
```

(all platform independent)

# directories

```
os.listdir()
```

```
os.mkdir()
```

```
os.walk()
```

(higher level stuff in `shutil` module)

# LAB

## Paths and File Processing

- write a program which prints the full path to all files in the current directory, one per line
- write a program which copies a file from a source, to a destination (without using `shutil`, or the OS copy command)
- write a program that extracts all the programming languages that the students in this class used before (`code\students_languages.txt`)
- update mail-merge from the earlier lab to write output to individual files on disk



# Homework

## Recommended Reading

- Dive Into Python: Chapt. 13,14
- Unicode: [http:](http://www.joelonsoftware.com/articles/Unicode.html)

[//www.joelonsoftware.com/articles/Unicode.html](http://www.joelonsoftware.com/articles/Unicode.html)

## Do the Labs you didn't finish in class

- Coding Kata 14 - Dave Thomas  
[http://codekata.pragprog.com/2007/01/kata\\_fourteen\\_t.html](http://codekata.pragprog.com/2007/01/kata_fourteen_t.html)
- Use The Adventures of Sherlock Holmes as input:  
`code/sherlock.txt` (ascii)
- This is intentionally open-ended and underspecified. There are many interesting decisions to make.
- Experiment with different lengths for the lookup key. (3 words, 4 words, 3 letters, etc)