Some material adapted from Upenn cis391 slides and other sources

## Python 2

### Dictionaries: A Mapping type

- Dictionaries store a *mapping* between a set of keys and a set of values
- Keys can be any *immutable* type.
- Values can be any type
- A single dictionary can store values of different types
- You can define, modify, view, lookup or delete the key-value pairs in the dictionary
- Python’s dictionaries are also known as *hash tables* and *associative arrays*

### Creating & accessing dictionaries

```python
>>> d = {‘user’: ‘bozo’, ‘pswd’: 1234}
>>> d[‘user’]
‘bozo’
>>> d[‘pswd’]
123
>>> d[‘bozo’]
Traceback (innermost last):
  File ‘<interactive input>’ line 1, in ?
KeyError: bozo
```
Updating Dictionaries

```python
>>> d = {'user': 'bozo', 'pswd': 1234}
>>> d['user'] = 'clown'
>>> d
{'user': 'clown', 'pswd': 1234}
• Keys must be unique
• Assigning to an existing key replaces its value

>>> d['id'] = 45
>>> d
{'user': 'clown', 'id': 45, 'pswd': 1234}

• Dictionaries are unordered
• New entries can appear anywhere in output
• Dictionaries work by hashing
```

Removing dictionary entries

```python
>>> d = {'user': 'bozo', 'p': 1234, 'i': 34}
>>> del d['user']  # Remove one.
>>> d
{'p': 1234, 'i': 34}

>>> d.clear()  # Remove all.
>>> d
{}
```

```
>>> a = [1, 2]
>>> del a[1]  # del works on lists, too
>>> a
[1]
```
#!/usr/bin/python
import sys
freq = {}   # frequency of words in text
for line in sys.stdin:
    for word in line.split():
        if word in freq:
            freq[word] = 1 + freq[word]
        else:
            freq[word] = 1
print freq

This is a common pattern

#!/usr/bin/python
import sys
freq = {}   # frequency of words in text
for line in sys.stdin:
    for word in line.split():
        freq[word] = freq.get(word,0)
for w in sorted(freq.keys()):
    print w, freq[w]
#!/usr/bin/python
import sys
from operator import itemgetter

punctuation = """!""""#$%&'()*+,-./:;<=>? @[\]^_`{|}~"""

freq = {}    # frequency of words in text
stop_words = {}    
for line in open("stop_words.txt"):
    stop_words[line.strip()] = True

for line in sys.stdin:
    for word in line.split():
        word = word.strip(punctuation).lower()
        if not word in stop_words:
            freq[word] = freq.get(word,0) + 1

words = sorted(freq.iteritems(),
    key=itemgetter(1), reverse=True)
for w,f in words:
    print w, f

from optparse import OptionParser
# read command line arguments and process
parser = OptionParser()
parser.add_option("-n", "--number", type="int",
    default=-1, help='number of words to report')
parser.add_option("-t", "--threshold", type="int",
    default=0, help='print if frequency > threshold")
(options, args) = parser.parse_args()
...
# print the top option.number words but only those
# with freq>=option.threshold
for (word, freq) in words[:options.number]:
    if freq > options.threshold:
        print freq, word

---

<table>
<thead>
<tr>
<th>Dictionary example wf4.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>#!/usr/bin/python</td>
</tr>
<tr>
<td>import sys</td>
</tr>
<tr>
<td>from operator import itemgetter</td>
</tr>
<tr>
<td>punctuation = &quot;&quot;&quot;!&quot;&quot;&quot;&quot;#$%&amp;'()*+,-./:;&lt;=&gt;? @[]^_`{</td>
</tr>
<tr>
<td>freq = {}    # frequency of words in text</td>
</tr>
<tr>
<td>stop_words = {}</td>
</tr>
</tbody>
</table>
| for line in open("stop_words.txt"):
    stop_words[line.strip()] = True |

<table>
<thead>
<tr>
<th>Dictionary example wf5.py</th>
</tr>
</thead>
</table>
| for line in sys.stdin:
    for word in line.split():
        word = word.strip(punctuation).lower()
        if not word in stop_words:
            freq[word] = freq.get(word,0) + 1 |
| words = sorted(freq.iteritems(),
    key=itemgetter(1), reverse=True) |
| for w,f in words:
    print w, f |

<table>
<thead>
<tr>
<th>Dictionary example wf6.py</th>
</tr>
</thead>
<tbody>
<tr>
<td>from optparse import OptionParser</td>
</tr>
<tr>
<td># read command line arguments and process</td>
</tr>
<tr>
<td>parser = OptionParser()</td>
</tr>
</tbody>
</table>
| parser.add_option("-n", "--number", type="int",
    default=-1, help='number of words to report') |
| parser.add_option("-t", "--threshold", type="int",
    default=0, help='print if frequency > threshold") |
| (options, args) = parser.parse_args() |
| ... |
| # print the top option.number words but only those |
| # with freq>=option.threshold |
| for (word, freq) in words[:options.number]:
    if freq > options.threshold:
        print freq, word |

<table>
<thead>
<tr>
<th>Why must keys be immutable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The keys used in a dictionary must be immutable objects?</td>
</tr>
<tr>
<td>&gt;&gt;&gt; name1, name2 = 'john', ['bob', 'marley']</td>
</tr>
<tr>
<td>&gt;&gt;&gt; fav = name2</td>
</tr>
<tr>
<td>&gt;&gt;&gt; d = {name1: 'alive', name2: 'dead'}</td>
</tr>
<tr>
<td>Traceback (most recent call last):</td>
</tr>
<tr>
<td>File &quot;&lt;stdin&gt;&quot;, line 1, in &lt;module&gt;</td>
</tr>
<tr>
<td>TypeError: list objects are unhashable</td>
</tr>
<tr>
<td>• Why is this?</td>
</tr>
<tr>
<td>• Suppose we could index a value for name2</td>
</tr>
<tr>
<td>• and then did fav[0] = “Bobby”</td>
</tr>
<tr>
<td>• Could we find d[name2] or d[fav] or …?</td>
</tr>
</tbody>
</table>
Functions in Python

Python and Types

- **Dynamic typing**: Python determines the data types of variable bindings in a program automatically.
- **Strong typing**: But Python’s not casual about types, it enforces the types of objects.
- For example, you can’t just append an integer to a string, but must first convert it to a string.

```python
x = "the answer is " # x bound to a string
y = 23 # y bound to an integer.
print x + y # Python will complain!
```

Defining Functions

Function definition begins with "def.” Function name and its arguments.

```python
def get_final_answer(filename):
    """Documentation String""
    line1
    line2
    return total_counter
```

No header file or declaration of types of function or arguments

Calling a Function

- The syntax for a function call is:

  ```python
  >>> def myfun(x, y):
  ...    return x * y
  >>> myfun(3, 4)
  12
  ```

- Parameters in Python are Call by Assignment
- Old values for the variables that are parameter names are hidden, and these variables are simply made to refer to the new values.
- All assignment in Python, including binding function parameters, uses reference semantics.
**Functions without returns**
- All functions in Python have a return value, even if no `return` line inside the code
- Functions without a `return` return the special value `None`
- `None` is a special constant in the language
- `None` is used like `NULL`, `void`, or `nil` in other languages
- `None` is also logically equivalent to False
- The interpreter doesn’t print `None`

**Function overloading? No.**
- There is no function overloading in Python
- Unlike C++, a Python function is specified by its name alone
  - The number, order, names, or types of its arguments cannot be used to distinguish between two functions with the same name
- Two different functions can’t have the same name, even if they have different arguments
- But: see operator overloading in later slides
  (Note: van Rossum playing with function overloading for the future)

**Default Values for Arguments**
- You can provide default values for a function’s arguments
- These arguments are optional when the function is called

```python
>>> def myfun(b, c=3, d="hello"): return b + c
>>> myfun(5,3,"hello")
>>> myfun(5,3)
>>> myfun(5)
```

All of the above function calls return 8

**Keyword Arguments**
- You can call a function with some or all of its arguments out of order as long as you specify their names
- You can also just use keywords for a final subset of the arguments.

```python
>>> def myfun(a, b, c):
    return a-b
>>> myfun(2, 1, 43)
>>> myfun(2, i, 43)
>>> myfun(c=43, b=1, a=2)
>>> myfun(2, c=43, b=1)
```
Functions are first-class objects

Functions can be used as any other datatype, eg:
• Arguments to function
• Return values of functions
• Assigned to variables
• Parts of tuples, lists, etc

```python
>>> def square(x):
    return x*x
>>> def applier(q, x):
    return q(x)
>>> applier(square, 7)
49
```

Lambda Notation

• Python uses a lambda notation to create anonymous functions
  ```python
  >>> applier(lambda z: z * 4, 7)
  28
  ```
• Python supports functional programming idioms, including closures and continuations

```python
>>> def square(x):
    return x*x
>>> def twice(f):
    return lambda x: f(f(x))
>>> quad = twice(square)
>>> quad(5)
625
```

Lambda Notation

Be careful with the syntax

```python
>>> f = lambda x,y : 2 * x + y
>>> f
<function <lambda> at 0x87d30>
>>> f(3, 4)
10
>>> v = lambda x: x*x(100)
>>> v
<function <lambda> at 0x87df0>
>>> v = (lambda x: x*x)(100)
>>> v
10000
```
Example: closure

```python
>>> def counter(start=0, step=1):
    x = [start]
    def _inc():
        x[0] += step
        return x[0]
    return _inc

>>> c1 = counter()
>>> c2 = counter(100, -10)
>>> c1()
1
>>> c2()
90
```

Logical Expressions

True and False

- `True` and `False` are constants in Python.
- Other values equivalent to `True` and `False`:
  - `False`: zero, `None`, empty container or object
  - `True`: non-zero numbers, non-empty objects
- Comparison operators: `==`, `!=`, `<`, `<=`, etc.
- X and Y have same value: `x == y`
- Compare with `x is y`:
  - X and Y are two variables that refer to the identical same object.

Boolean Logic Expressions

- You can also combine Boolean expressions.
- `True` if a is True and b is True: `a and b`
- `True` if a is True or b is True: `a or b`
- `True` if a is False: `not a`
- Use parentheses as needed to disambiguate complex Boolean expressions.
Special Properties of and & or

- Actually and and or don’t return True or False but value of one of their sub-expressions, which may be a non-Boolean value
- X and Y and Z
  - If all are true, returns value of Z
  - Otherwise, returns value of first false sub-expression
- X or Y or Z
  - If all are false, returns value of Z
  - Otherwise, returns value of first true sub-expression
- And and or use lazy evaluation, so no further expressions are evaluated

The “and-or” Trick

- An old deprecated trick to implement a simple conditional
  ```python
  result = test and expr1 or expr2
  ```
- When test is True, result is assigned expr1
- When test is False, result is assigned expr2
- Works almost like C++’s `(test ? expr1 : expr2)`
- But if the value of expr1 is ever False, the trick doesn’t work
- Don’t use it; made unnecessary by conditional expressions in Python 2.5 (see next slide)

Conditional Expressions in Python 2.5

- `x = true_value if condition else false_value`
- Uses lazy evaluation:
  - First, condition is evaluated
  - If True, true_value is evaluated and returned
  - If False, false_value is evaluated and returned
- Standard use:
  ```python
  x = (true_value if condition else false_value)
  ```

Control of Flow
**if Statements**

```python
if x == 3:
    print "X equals 3."
elif x == 2:
    print "X equals 2."
else:
    print "X equals something else."
    print "This is outside the 'if'."  
```

Be careful! The keyword *if* is also used in the syntax of filtered *list comprehensions*. Note:

- Use of indentation for blocks
- Colon (:) after boolean expression

**while Loops**

```python
>>> x = 3
>>> while x < 5:
    print x, "still in the loop"
    x = x + 1
3 still in the loop
4 still in the loop
>>> x = 6
>>> while x < 5:
    print x, "still in the loop"

>>>  
```

**break and continue**

- You can use the keyword *break* inside a loop to leave the *while* loop entirely.
- You can use the keyword *continue* inside a loop to stop processing the current iteration of the loop and to immediately go on to the next one.

**assert**

- An *assert* statement will check to make sure that something is true during the course of a program.
- If the condition if false, the program stops — (more accurately: the program throws an exception)

```python
assert(number_of_players < 5)
```
Python supports higher-order functions that operate on lists similar to Scheme's

```python
>>> def square(x):
    return x*x
>>> def even(x):
    return 0 == x % 2
>>> map(square, range(10,20))
[100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
>>> filter(even, range(10,20))
[10, 12, 14, 16, 18]
>>> map(square, filter(even, range(10,20)))
[100, 144, 196, 256, 324]
```

But many Python programmers prefer to use list comprehensions, instead

A list comprehension is a programming language construct for creating a list based on existing lists

- Haskell, Erlang, Scala and Python have them
- Why “comprehension”? The term is borrowed from math’s set comprehension notation for defining sets in terms of other sets
- A powerful and popular feature in Python
  - Generate a new list by applying a function to every member of an original list
- Python’s notation:
  ```python
  [ expression for name in list ]
  ```

The syntax of a list comprehension is somewhat tricky

```python
[x-10 for x in grades if x>0]
```

- Syntax suggests that of a for-loop, an in operation, or an if statement
- All three of these keywords (‘for’, ‘in’, and ‘if’) are also used in the syntax of forms of list comprehensions

```python
[ expression for name in list ]
```
**List Comprehensions**

Note: Non-standard colors on next few slides clarify the list comprehension syntax.

```
>>> li = [3, 6, 2, 7]
>>> [elem*2 for elem in li]
[6, 12, 4, 14]
```

```python
[ expression for name in list ]
```
- Where `expression` is some calculation or operation acting upon the variable `name`.
- For each member of the `list`, the list comprehension
  1. sets `name` equal to that member,
  2. calculates a new value using `expression`,
- It then collects these new values into a list which is the return value of the list comprehension.

```
>>> li = [('a', 1), ('b', 2), ('c', 7)]
>>> [n * 3 for (x, n) in li]
[3, 6, 21]
```

• `expression` for `name` in `list`

**List Comprehensions**

```
>>> def subtract(a, b):
...     return a - b

>>> oplist = [(6, 3), (1, 7), (5, 5)]
>>> [subtract(y, x) for (x, y) in oplist]
[-3, 6, 0]
```

**Syntactic sugar**

List comprehensions can be viewed as syntactic sugar for a typical higher-order functions

```
[ expression for name in list ]
map( lambda name: expression, list )
```

```
[ 2*x+1 for x in [10, 20, 30] ]
map( lambda x: 2*x+1, [10, 20, 30] )
```
**Filtered List Comprehension**

- **Filter** determines whether expression is performed on each member of the list.
- For each element of list, checks if it satisfies the filter condition.
- If the filter condition returns False, that element is omitted from the list before the list comprehension is evaluated.

```python
>>> li = [3, 6, 2, 7, 1, 9]
>>> [elem*2 for elem in li if elem > 4]
[12, 14, 18]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produce.

**More syntactic sugar**

Including an if clause begins to show the benefits of the sweetened form

```python
[ expression for name in list if filter ]
```

```python
map( lambda name . expression, filter( filt, list ) )
```

```python
[ 2*x+1 for x in [10, 20, 30] if x > 0 ]
map( lambda x: 2*x+1, filter( lambda x: x > 0 , [10, 20, 30] )
```

**Nested List Comprehensions**

- Since list comprehensions take a list as input and produce a list as output, they are easily nested.

```python
>>> li = [3, 2, 4, 1]
>>> [elem*2 for elem in li if elem > 0]
[6, 8, 8]
```

- Only 6, 7, and 9 satisfy the filter condition.
- So, only 12, 14, and 18 are produce.
Python's list comprehensions provide a natural idiom that usually requires a for-loop in other programming languages.

As a result, Python code uses many fewer for-loops

Nevertheless, it’s important to learn about for-loops.

Take care! The keywords for and in are also used in the syntax of list comprehensions, but this is a totally different construction.
For Loops 2

for <item> in <collection>:
  <statements>
• <item> can be more than a single variable name
• When the <collection> elements are themselves sequences, then <item> can match the structure of the elements.
• This multiple assignment can make it easier to access the individual parts of each element
  for (x,y) in [(a,1),(b,2),(c,3),(d,4)]:
    print x

For loops & the range() function

• Since a variable often ranges over some sequence of numbers, the range() function returns a list of numbers from 0 up to but not including the number we pass to it.
• range(5) returns [0,1,2,3,4]
• So we could say:
  for x in range(5):
    print x
• (There are more complex forms of range() that provide richer functionality…)

For Loops and Dictionaries

>>> ages = { "Sam" : 4, "Mary" : 3, "Bill" : 2 }
>>> ages
{'Bill': 2, 'Mary': 3, 'Sam': 4}
>>> for name in ages.keys():
    print name, ages[name]
Bill 2
Mary 3
Sam 4
>>>
Multiple Assignment with Sequences

• We’ve seen multiple assignment before:
  >>> x, y = 2, 3

• But you can also do it with sequences.
  • The type and “shape” just has to match.
  >>> (x, y, (w, z)) = (2, 3, (4, 5))
  >>> [x, y] = [4, 5]

Empty Containers 1

• Assignment creates a name, if it didn’t exist already.
  x = 3  Creates name x of type integer.
• Assignment is also what creates named references to containers.
  >>> d = {'a':3, 'b':4}
• We can also create empty containers:
  >>> li = []
  >>> tu = ()
  >>> di = {}

  Note: an empty container is logically equivalent to False. (Just like None.)

• These three are empty, but of different types

Empty Containers 2

Why create a named reference to empty container?
  • To initialize an empty list, e.g., before using append
  • This would cause an unknown name error if a named reference to the right data type wasn’t created first

  >>> g.append(3)
  Python complains here about the unknown name ‘g’!

  >>> g = []
  >>> g.append(3)
  >>> g
  [3]

String Operations
String Operations

- A number of methods for the string class perform useful formatting operations:
  
  ```
  >>> "hello".upper()
  'HELLO'
  ```

- Check the Python documentation for many other handy string operations.

- Helpful hint: use `string`.strip() to strip off final newlines from lines read from files

String Formatting Operator: %

- The operator `%` allows strings to be built out of many data items a la “fill in the blanks”
  - Allows control of how the final output appears
  - For example, we could force a number to display with a specific number of digits after the decimal point
  - Very similar to the sprintf command of C.
    ```
    >>> x = "abc"
    >>> y = 34
    >>> "%s xyz %d" % (x, y)
    'abc xyz 34'
    ```

- The tuple following the `%` operator used to fill in blanks in original string marked with `%s` or `%d`
- Check Python documentation for codes

Printing with Python

- You can print a string to the screen using `print`
- Using the `%` operator in combination with print, we can format our output text
  ```
  >>> print "%s xyz %d" % ("abc", 34)
  abc xyz 34
  ```

- `Print` adds a newline to the end of the string. If you include a list of strings, it will concatenate them with a space between them
  ```
  >>> print "abc"  >>> print "abc", "def"
  abc                abc def
  ```

- Useful trick: `>>> print "abc"`, doesn’t add newline just a single space

String Conversions
Join and Split

• Join turns a list of strings into one string
  <separator_string>.join( <some_list> )

  >>> ";".join( ["abc", "def", "ghi"] )
  "abc;def;ghi"

• Split turns one string into a list of strings
  <some_string>.split( <separator_string> )

  >>> "abc;def;ghi".split( ";" )
  ["abc", "def", "ghi"]

• Note the inversion in the syntax

Convert Anything to a String

• The builtin str() function can convert an instance of any data type into a string.
• You define how this function behaves for user-created data types
• You can also redefine the behavior of this function for many types.

  >>> "Hello " + str(2)
  "Hello 2"

Split & Join with List Comprehensions

• Split and join can be used in a list comprehension in the following Python idiom:
  " ",join( [s.capitalize() for s in "this is a test ".split( )] )
  'This Is A Test'

  >>> # For clarification:
  >>> "this is a test".split( )
  ['this', 'is', 'a', 'test']

  >>> [s.capitalize() for s in "this is a test ".split()]
  ['This', 'Is', 'A', 'Test']