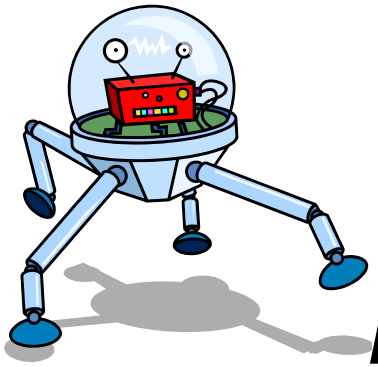
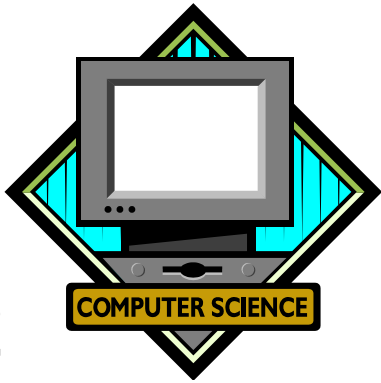
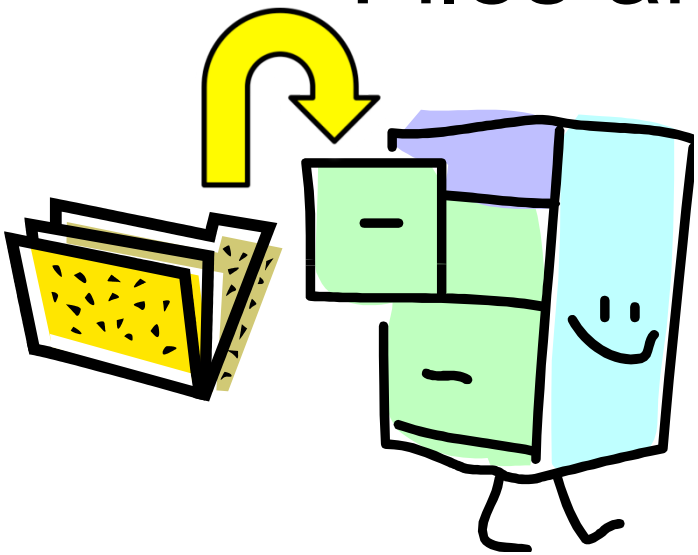
Caltech/LEAD Summer 2012 Computer Science





Lecture 10: July 18, 2012

Files and dictionaries



This lecture

- Files
- Dictionaries

Files

- Data that programs act on can be either temporary or permanent
- Values of program variables are temporary
  - they disappear when program exits
- Often want to work with more permanent data

Files

- Data that is stored on computer's hard drive is generally in the form of files
- Files are said to be "persistent"
  - Still around after the program exits
  - Still around after the computer turned off!
- Files are massively useful
  - Most programs need to store some data

Binary files and text files

- We can distinguish two kinds of files based on the way the data is formatted
- Files that contain only textual (character) data we call text files
- Files that contain "raw data" we call binary files (raw → just 0s and 1s)
- (This is an oversimplification)
- We will mostly work with text files

Example

- Let's assume there is a file containing temperature data taken once/day at noon
  - file called `temps.txt`
- Can be very large (> 1000 entries)
- Assume one number per line
- Want to read this data and compute values from it

Example

- The `temps.txt` file contains textual (character) data which can be interpreted as numeric:

```
78.2
68.3
59.0
88.1
49.5
99.0
```

(etc.)

Opening a file

- Files in Python are represented as file objects
  - *i.e.* objects which represent a file on the hard drive
  - These are not the same thing as the file, just a way to interact with the file from Python
- Before we work with a file, we have to create a file object in Python that is linked to the real file
- After that, file operations are just methods of the file object
- File objects are created using the `open` function

Opening a file

```
>>> temps = open('temps.txt', 'r')
>>> temps
<open file 'temps.txt', mode 'r' at
  0x559f8>
```

- Now the file `temps.txt` has a corresponding Python object called `temps`
- Method calls on `temps` will do something to the file `temps.txt`

Ways of opening a file

- The `open` function looks like this:

```
open(<name of file>, <mode>)
```

- `open` returns a value: a Python file object
- `<name of file>` is just the file's name, as a string
- `<mode>` determines how you can interact with the file object that `open` returns

Ways of opening a file

- Three typical values of **<mode>**:
- **'r'** – means that the file has been opened "**r**ead-only"
- **'w'** – means that the file has been opened for "**w**riting"
- **'a'** – means that the file has been opened for "**a**ppending"
- Some other modes exist (won't discuss now)

Ways of opening a file

- Read-only mode: use on an existing file you don't want to change
  - file must already exist
- Write mode: use when creating a new file from scratch
  - if file exists, it will be wiped out and overwritten!
- Append mode: use when adding to the end of an existing file
  - file must already exist, will be changed

Ways of opening a file

- Here, we will be reading from an existing file called `temps.txt`, but not writing to it
  - so we need to use the `'r'` (read-only) mode
- If we try to write to a read-only file, an error will occur
- Similarly, trying to read from a write-only file will result in an error

Closing a file

- Once we're done working with a file object, we should close it
  - prevents further actions from occurring to the file
- If we forget, file object will be closed anyway when program exits
  - but this is sloppy and bad programming practice

Closing a file

- Assume we created the `temps` file object before, corresponding to the file `temps.txt`
- To close it, we do:

```
temps.close()
```

- This causes the `close` method of the file object temps to be called
  - which closes the file `temps.txt`

Closing a file

- After you close a file, you can't do anything to it!
  - can't read from it or write to it
- So make sure you are truly finished with the file object before you `close` it!

Pattern for files

- Code that interacts with files will typically have this pattern:

```
f = open('temps.txt', 'r')
# do something with file object f
f.close()
```

- (May use a different file name or mode, of course)

Reading from text files

- When handling text files, can think of the file as a bunch of lines (strings ending in newline (`'\n'`) character)
- Python methods for reading from text files:
  - **`readline`** – read a single line from a text file
  - **`readlines`** – read *all* the lines of the text file

readlines

- Simplest pattern:

```
temps = open('temps.txt', 'r')
# Read all lines in file:
lines = temps.readlines()
temps.close()
```

- Now `lines` is a list of strings, one for every line in the file

readlines

- The `lines` list will look like this:

```
['78.2\n', '68.3\n', '59.0\n',
 '88.1\n', '49.5\n', '99.0\n', ...]
```

- Notes:
  1. Each element of the list is a string; need to convert it to a number before using it
  2. Each string ends in a newline character (`'\n'`)
  3. Changing the elements of the list will *not* cause the contents of the file to change (the list and the file are independent once `readlines` completes)

readlines

- The `lines` list will look like this:

```
['78.2\n', '68.3\n', '59.0\n',
 '88.1\n', '49.5\n', '99.0\n', ...]
```

- Problem 1:
  - To do anything useful with this list, need to convert all the strings into floats
- Problem 2:
  - If the file is extremely large, you now have a very large list in memory (may be more than computer can handle)

readlines

- It would be nice if we could read in lines from a file one at a time instead of all at once
- Then, could convert to numbers right after reading
- Could also process right away instead of storing into a single list
  - (if that's possible)
- Python has another useful method: `readline` (without the `'s'` at the end)

`readline` (without the `'s'`)

- The **`readline`** method reads a single line (ending in a newline (**`'\n'`**) character) and returns the line read (with the newline)
- If there are no more lines (at end of file), **`readline`** returns the empty string
  - but does *not* report an error!
- Note: an empty line in the file will return a line which consists only of the newline character
  - only way to return the empty string is at end of file

Sample problem

- Read all the lines of the file
- Assume each line contains a floating-point number
  - (won't do any error checking)
- Compute the sum of all the numbers

readline

- `readline` used with our `temps.txt` file:

```
>>> sum_nums = 0.0
>>> temps = open('temps.txt', 'r')
>>> sum_nums += float(temps.readline())
>>> sum_nums += float(temps.readline())
>>> sum_nums += float(temps.readline())
```

(etc.)

- Problem: how do we know when to stop?

`readline`

- Idea:
  - read a line
  - if the line is not empty (not at end of file), convert to `float`, add to `sum_nums` and keep reading
- Let's translate that into code

readline

- Could use a **while** loop:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
line = temps.readline()
while line != '':  # '' is empty string
    sum_nums += float(line)
    line = temps.readline()
```

- This works, but something is still not great

D.R.Y. again

- Repeated code:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
line = temps.readline()
while line != '':
    sum_nums += float(line)
    line = temps.readline()
```

- There should be a better (DRYer) way

D.R.Y. again

- Previously we used an infinite loop and a **`break`** statement to DRY up our code
- Can the same approach work this time?
- Let's give it a try...
  - (give it a *D.R.Y.*...)

D.R.Y. again

- With an infinite loop:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
while True:
    line = temps.readline()
    if line == '':
        break
    sum_nums += float(line)
```

- No repetition, very DRY

The code, in words

1. Open the file
2. Initialize `sum_nums` to zero
3. Repeat:
   a) read a line from the file
   b) if the line is empty, we're at the end of the file, so the loop is done
   c) otherwise, convert the line to a float and add to `sum_nums`

Another advantage

- Using `readline()` instead of `readlines()` to get the values in the file one-by-one means that you don't have to create a very large file in the computer's memory
- Can use this code on arbitrarily large files without having your computer run out of memory
  - important for large data sets

Using `for` with files

- Python allows an amazing shortcut using the `for` statement:

```
temps = open('temps.txt', 'r')
sum_nums = 0.0
for line in temps:
    sum_nums += float(line)
```

- This is the preferred way to write this

Using `for` with files

- Previously, we had:

```
for <name> in <something>:
    # block of code
```

- The **`<something>`** after the **`in`** had to be a list, a tuple or a string
- Python actually allows more than just lists or strings after the **`in`**
  - files being one example

Using `for` with files

- Conceptually, we have

```
for <name> in <sequence>:
    # block of code
```

- Lists are sequences
- Strings are sequences
- And files can be considered to be "sequences of lines"

Using `for` with files

- However, just because files work here:

```
for <name> in <sequence>:
    # block of code
```

- Doesn't mean you can do *all* sequence operations on files!

Using `for` with files

- For instance, this won't work:

```
# assume that the file 'foo.txt' exists
f = open('foo.txt', 'r')
print f[0]  # print first line in file?
```

- Result:

```
TypeError: 'file' object is unsubscriptable
```

- *Moral*: files are *not* full-fledged Python sequences
  - but `for` does work with files

Dictionaries

- A *dictionary* is a new kind of Python data type
- Before we describe what it is, let's describe a problem it could solve

Phone numbers

- You want to keep track of your friends' phone numbers
- But you (naturally) have so many friends, this is a difficult job
- How can the computer help?

Phone numbers

- For each friend, need to store:
  - the *name* of the friend
  - the *phone number* of the friend
- Also, want to be able to retrieve the phone number for a given friend
- Given what you know now, how can you do this?

List?

- You could have a list of names and phone numbers:

```
phone_numbers = ['Joe', '567-8901',
                 'Jane', '123-4567',
                 ...]
```

- But it would not be easy to find the number corresponding to a different name
- It would be better if a name and the corresponding phone number were connected in some way

List of tuples?

- You could have a list of (*name*, *phone number*) tuples:

```
phone_numbers = [('Joe', '567-8901'),
                 ('Jane', '123-4567'),
                 ...]
```

- Let's see what we would need to do in order to find the phone number corresponding to a particular name
  - *e.g.* `'Donnie'`

List of tuples?

- We could write code like this:

```
for (name, phone) in phone_numbers:
    if name == 'Donnie':
        print 'Phone number: %s' % phone
```

- This is not too bad, but
  - can't modify the phone number!
    - (tuples are immutable)
  - have to look through entire list in worst case to find one number
  - cumbersome!

Dictionaries

- A dictionary is a data structure that stores associations between *keys* and *values*
- In the previous example:
  - *key*: the name of the friend
  - *value*: the phone number
- Dictionaries make it easy to:
  - find the value given the key
  - change the value given the key
  - add more key/value associations
- And they're fast!

Keys and values

- The *values* stored in a dictionary can be any Python value
- *Keys* can only be *immutable* (unchangeable) Python values, *e.g.*
  - strings
  - tuples
  - numbers (rare)
- There is a technical reason for this (which we won't go into)
  - Usually use strings as keys

Dictionary syntax

- The contents of a dictionary are written between curly braces ({ and })
- The empty dictionary (no key/value pairs) is written like this:

```
{ }
```

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

key

```
{'Joe'  : '567-8910',
 'Jane' : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

value

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

colon

```
{'Joe' : '567-8910',
 'Jane' : '123-4567' }
```

- Colon (:) separates a key from its value

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

comma

- Comma (,) separates different key/value pairs

Dictionary syntax

- A typical dictionary might look like this:

key/value pair #1

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

Dictionary syntax

- A typical dictionary might look like this:

```
{'Joe'   : '567-8910',
 'Jane'  : '123-4567' }
```

key/value pair #2

Dictionary syntax

- Dictionary values can be expressions:

```
{'Joe'  : 2 + 3,
 'Jane' : '123-' + '4567' }
```

Dictionary syntax

- Dictionary keys can also be expressions:

```
{'Joe' + ' Blow' : '567-8901',
 'Jane' + ' Doe' : '123-4567' }
```

- (This is very rare, though)
- Expressions are always evaluated while creating the dictionary

Getting a value given a key

- We have:

```
phone_numbers = {'Joe'  : '567-8901',
                 'Jane' : '123-4567' }
```

- To get Joe's phone number:

```
Joes_phone_number = phone_numbers['Joe']
```

Getting a value given a key

- Notice that:

```
phone_numbers['Joe']
```

looks like accessing a list with a value of `'Joe'`

- Python is *overloading* the meaning of the square brackets
- Before, the value inside the brackets could only be an integer
- With a dictionary, it's any key value

Changing a value at a key

- Let's say that Joe's phone number changes
- Can change the dictionary value too:

```
phone_numbers['Joe'] = '314-1592'
```

- Like the syntax for changing a list value
  - except that the "index" is a string, not a number

Adding a new key/value pair

- Add a new key/value pair by "changing" a key that wasn't there before:

```
phone_numbers['Donnie'] = '111-1111'
phone_numbers['Mike'] = '000-0000'
```

Accessing a nonexistent key

- Here's what happens if you try to access a key that isn't in the dictionary:

```
>>> phone_numbers['Quentin']
KeyError: 'Quentin'
```

Deleting a key/value pair

- To remove a key/value pair from a dictionary:

```
>>> del phone_numbers['Joe']
>>> phone_numbers
{ 'Jane' : '123-4567',
  'Mike' : '000-0000',
  'Donnie' : '111-1111' }
```

`del`

- **`del`** is actually a special Python statement, like **`print`**
  - it's not a function, so no parentheses around its argument
- **`del`** can remove elements from things other than dictionaries (*e.g.* lists)
  - but more useful with dictionaries than lists

Back to the example

- Let's improve the example by using a tuples of first and last names as keys:

```
phone_numbers = \
   { ('Joe', 'Smith') : 567-8910,
     ('Jane', 'Doe') : 123-4567,
     ('Mike', 'Vanier') : 000-0000,
     ('Donnie', 'Pinkston') : 111-1111 }
```

- This is OK, because both tuples and strings are immutable
  - so tuple of strings is immutable too, hence OK as key

Back to the example

- Can access phone numbers using tuple as key:

```
>>> phone_numbers[('Joe', 'Smith')]
'567-8910'
>>> phone_numbers['Joe']
KeyError: 'Joe'
>>> phone_numbers['Smith']
KeyError: 'Smith'
```

- You have to use the correct type of key for the dictionary, or it's an error!

Dictionaries and `for` loops

- We've seen many things that can be looped over using `for` loops:
  - lists
  - strings
  - files
- Should it surprise you to learn that dictionaries can also be looped over in a `for` loop?
  - We hope not!

Dictionaries and `for` loops

- Looping over a dictionary looks like this:

```
for key in phone_numbers:
    print key
```

- Looping over a dictionary loops over the *keys* in the dictionary (not the values)

Dictionaries and `for` loops

```
for key in phone_numbers:
    print key
```

- This will print:

```
('Mike', 'Vanier')
('Joe', 'Smith')
('Donnie', 'Pinkston')
('Jane', 'Doe')
```

Dictionaries and `for` loops

- Note:

```
('Mike', 'Vanier')
('Joe', 'Smith')
('Donnie', 'Pinkston')
('Jane', 'Doe')
```

- is *not* the order in which keys were originally entered in dictionary
- Dictionaries are *unordered* (not a sequence)
  - the "location" of any key/value pair is unimportant

Dictionaries and `for` loops

- We usually want the values, not the keys:

```
for key in phone_numbers:
    print phone_numbers[key]
```

- gives:

```
'000-0000'
'567-8910'
'111-1111'
'123-4567'
```

Searching

- Problem: print out the phone number of every person whose first name is `'Joe'`
  - using `(<first name>, <last name>)` tuples as keys in the dictionary

```
for key in phone_numbers:
    (first_name, last_name) = key
    if first_name == 'Joe':
        print phone_numbers[key]
```

- Easy peasy!

Dictionary methods

- Dictionaries are objects in Python
  - like lists, and strings, and files
- Therefore, they have methods
- We will discuss these methods:
  - `clear`
  - `keys`
  - `has_key`
  - `values`
  - `update`
- though there are many more

clear

- The `clear` method just empties out the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.clear()
>>> d
{}
```

keys

- The **keys** method returns a list of all the keys in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.keys()
['baz', 'foo', 'bar']
```

has_key

- The `has_key` method returns `True` if its argument is a key in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.has_key('foo')
True
>>> d.has_key('fnord')
False
```

values

- The `values` method returns a list of all the values in the dictionary:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.values()
[3, 1, 2]
```

update

- The `update` method adds the key/value pairs from another dictionary into this one
  - overwriting old values if other dictionary has same keys with different values

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.update({'xxx' : 4, 'yyy' : 5})
>>> d
{'baz' : 3, 'xxx' : 4, 'foo' : 1,
 'bar' : 2, 'yyy' : 5}
```

update

- Another example:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d.update({'foo' : 4, 'yyy' : 5})
>>> d
{'baz' : 3, 'foo' : 4, 'bar' : 2,
 'yyy' : 5}
```

- New value of key `'foo'` overwrites the old one

What about append?

- There is no `append` method for dictionaries
  - not needed!
- To add a new key/value pair, just use normal assignment syntax:

```
>>> d = {'foo' : 1, 'bar' : 2, 'baz' : 3}
>>> d
{'baz' : 3, 'foo' : 1, 'bar' : 2}
>>> d['fnord'] = 4   # add key/value pair
>>> d
{'fnord' : 4, 'baz' : 3, 'foo' : 1,
 'bar' : 2}
```

New example

- We have a list of words
- Want to create a frequency table
  - for each word, how many times does it occur in list?
- Solve by creating a dictionary
  - *key*: a word in the list
  - *value*: the count of that word
- Let's write the code...

New example

```
words = [ ... ]  # whatever
freqs = {}
for word in words:
    if freqs.has_key(word):
        freqs[word] += 1
    else:
        freqs[word] = 1
```

- And we're done!
- But wait! We want to print out the results...

New example

```
for key in freqs:
    print "Word %s occurs %d times" % \
        (key, freqs[key])
```

- Now we're done
- However, there is a short cut we can use
- Instead of `freq.has_key(word)` we can write `word in freq`
- This makes the code more readable

New example

```
words = [ ... ]   # whatever
freqs = {}
for word in words:
    if word in freqs:
        freqs[word] += 1
    else:
        freqs[word] = 1
for key in freqs:
    # same as before...
```

New example

- Now we're really done
- Dictionaries are awesome!
  - used in most Python programs
  - makes it much easier to write programs to solve a wide variety of tasks

Next lectures

- Graphics!