Ch.6: Dictionaries and strings

Hans Petter Langtangen[1,2]

Simula Research Laboratory[1]

University of Oslo, Dept. of Informatics[2]

Aug 15, 2015

Goals

- Learn more about file reading
- Store file data in a new object type: *dictionary*
- Interpret content in files via string manipulation

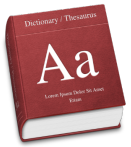
The main focus in the course is on working with files, dictionaries and strings. The book has additional material on how to utilize data from the Internet.

Dictionaries

```
figfiles = {'fig1.pdf': 81761, 'fig2.png': 8754}

figfiles['fig3.png'] = os.path.getsize(filename)

for name in figfiles:
    print 'File size of %s is %d:' % (name, figfiles[name])
```



A dictionary is a generalization of a list

- Features of lists:
  - store a *sequence* of elements in a single object (`[1,3,-1]`)
  - each element is a Python object
  - the elements are indexed by integers 0, 1, ...
- Dictionaries can index objects in a collection via text (= "lists with text index")
- Dictionary in Python is called hash, HashMap and associative array in other languages

The list index is sometimes unnatural for locating an element of a collection of objects

Suppose we need to store the temperatures in Oslo, London and Paris.
List solution:

```
temps = [13, 15.4, 17.5]
# temps[0]: Oslo
# temps[1]: London
# temps[2]: Paris
print 'The temperature in Oslo is', temps[0]
```

Can look up a temperature by mapping city to index to float
But it would be more natural to write `temps[Oslo]`!

Dictionaries map strings to objects

```
# Initialize dictionary
temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}

# Applications
print 'The temperature in London is', temps['London']
print 'The temperature in Oslo is',   temps['Oslo']
```

Important:

- The string index, like `Oslo`, is called *key*, while `temps['Oslo']` is the associated *value*
- A dictionary is an *unordered* collection of key-value pairs

Initializing dictionaries

Two ways of initializing a collection of key-value pairs:

```
mydict = {'key1': value1, 'key2': value2, ...}

temps = {'Oslo': 13, 'London': 15.4, 'Paris': 17.5}

# or
mydict = dict(key1=value1, key2=value2, ...)

temps = dict(Oslo=13, London=15.4, Paris=17.5)
```

Add a new element to a dict (dict = dictionary):

```
>>> temps['Madrid'] = 26.0
>>> print temps
{'Oslo': 13, 'London': 15.4, 'Paris': 17.5,
 'Madrid': 26.0}
```

Looping (iterating) over a dict means looping over the keys

```
for key in dictionary:
    value = dictionary[key]
    print value
```

Example:

```
>>> for city in temps:
...     print 'The %s temperature is %g' % (city, temps[city])
...
The Paris temperature is 17.5
The Oslo temperature is 13
The London temperature is 15.4
The Madrid temperature is 26
```

Note: the sequence of keys is arbitrary! Use sort if you need a particular sequence:

```
for city in sorted(temps):   # alphabetic sort of keys
    value = temps[city]
    print value
```

Can test for particular keys, delete elements, etc

Does the dict have a particular key?

```
>>> if 'Berlin' in temps:
...     print 'Berlin:', temps['Berlin']
... else:
...     print 'No temperature data for Berlin'
...
No temperature data for Berlin
>>> 'Oslo' in temps     # standard boolean expression
True
```

Delete an element of a dict:

```
>>> del temps['Oslo']   # remove Oslo key w/value
>>> temps
{'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
>>> len(temps)          # no of key-value pairs in dict.
3
```

The keys and values can be reached as lists

Python version 2:

```
>>> temps.keys()
['Paris', 'London', 'Madrid']
>>> temps.values()
[17.5, 15.4, 26.0]
```

Python version 3: `temps.keys()` and `temps.values()` are *iterators*, not lists!

```
>>> for city in temps.keys():  # works in Py 2 and 3
>>>     print city
...
Paris
Madrid
London
>>> keys_list = list(temps.keys())  # Py 3: iterator -> list
```

Caution: two variables can alter the same dictionary

```
>>> t1 = temps
>>> t1['Stockholm'] = 10.0    # change t1
>>> temps                     # temps is also changed!
{'Stockholm': 10.0, 'Paris': 17.5, 'London': 15.4,
  'Madrid': 26.0}
>>> t2 = temps.copy()         # take a copy
>>> t2['Paris'] = 16
>>> t1['Paris']               # t1 was not changed
17.5
```

Recall the same for lists:

```
>>> L = [1, 2, 3]
>>> M = L
>>> M[1] = 8
>>> L[1]
8
>>> M = L[:]  # take copy of L
>>> M[2] = 0
>>> L[2]
3
```

Any constant object can be used as key

- So far: key is text (string object)
- Keys can be any *immutable* (constant) object (!)

```
>>> d = {1: 34, 2: 67, 3: 0}   # key is int
>>> d = {13: 'Oslo', 15.4: 'London'} # possible
>>> d = {(0,0): 4, (1,-1): 5}  # key is tuple
>>> d = {[0,0]: 4, [-1,1]: 5}  # list is mutable/changeable
...
TypeError: unhashable type: 'list'
```

Example: Polynomials represented by dictionaries

The information in the polynomial

$$p(x) = -1 + x^2 + 3x^7$$

can be represented by a dict with power as key (`int`) and coefficient as value (`float`):

```
p = {0: -1, 2: 1, 7: 3.5}
```

Evaluate such a polynomial $\sum_{i \in I} c_i x^i$ for some x:

```
def eval_poly_dict(poly, x):
    sum = 0.0
    for power in poly:
        sum += poly[power]*x**power
    return sum
```

Short pro version:

```
def eval_poly_dict2(poly, x):
    # Python's sum can add elements of an iterator
    return sum(poly[power]*x**power for power in poly)
```

Polynomials can also be represented by lists

The list index corresponds to the power, e.g., the data in $-1 + x^2 + 3x^7$ is represented as

```
p = [-1, 0, 1, 0, 0, 0, 3]
```

The general polynomial $\sum_{i=0}^{N} c_i x^i$ is stored as `[c0, c1, c2, ..., cN]`.

Evaluate such a polynomial $\sum_{i=0}^{N} c_i x^i$ for some x:

```
def eval_poly_list(poly, x):
    sum = 0
    for power in range(len(poly)):
        sum += poly[power]*x**power
    return sum
```

What is best for polynomials: lists or dictionaries?

Dictionaries need only store the nonzero terms. Compare dict vs list for the polynomial $1 - x^{200}$:

```
p = {0: 1, 200: -1}          # len(p) is 2
p = [1, 0, 0, 0, ..., 200]   # len(p) is 201
```

Dictionaries can easily handle negative powers, e.g., $\frac{1}{2}x^{-3} + 2x^4$

```
p = {-3: 0.5, 4: 2}
print eval_poly_dict(p, x=4)
```

Quick recap of file reading

```
infile  = open(filename, 'r') # open file for reading

line    = infile.readline()   # read the next line
filestr = infile.read()       # read rest of file into string
lines   = infile.readlines()  # read rest of file into list
for line in infile:           # read rest of file line by line

infile.close()                # recall to close!
```

Example: Read file data into a dictionary

Data file:

```
Oslo:       21.8
London:     18.1
Berlin:     19
Paris:      23
Rome:       26
Helsinki:   17.8
```

Store in dict, with city names as keys and temperatures as values

Program:

```
infile = open('deg2.dat', 'r')
temps = {}                    # start with empty dict
for line in infile.readlines():
    city, temp = line.split()
    city = city[:-1]          # remove last char (:)
    temps[city]  = float(temp)
```

A tabular file can be read into a nested dictionary

Data file `table.dat`:

	A	B	C	D
1	11.7	0.035	2017	99.1
2	9.2	0.037	2019	101.2
3	12.2	no	no	105.2
4	10.1	0.031	no	102.1
5	9.1	0.033	2009	103.3
6	8.7	0.036	2015	101.9

Create a dict `data[p][i]` (dict of dict) to hold measurement no. `i` (1, 2, etc.) of property `p` (`'A'`, `'B'`, etc.)

We must first develop the plan (algorithm) for doing this

1. Examine the first line:
   1. split it into words
   2. initialize a dictionary with the property names as keys and empty dictionaries {} as values
2. For each of the remaining lines:
   1. split line into words
   2. for each word after the first: if word is not no, convert to float and store

Good exercise: do this now!
(See the book for a complete implementation.)

Example: Download data from the web and visualize

Problem:

- Compare the stock prices of Microsoft, Apple, and Google over decades
- http://finance.yahoo.com/ offers such data in files with tabular form

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-02-03,502.61,551.19,499.30,545.99,12244400,545.99
2014-01-02,555.68,560.20,493.55,500.60,15698500,497.62
2013-12-02,558.00,575.14,538.80,561.02,12382100,557.68
2013-11-01,524.02,558.33,512.38,556.07,9898700,552.76
2013-10-01,478.45,539.25,478.28,522.70,12598400,516.57
...
1984-10-01,25.00,27.37,22.50,24.87,5654600,2.73
1984-09-07,26.50,29.00,24.62,25.12,5328800,2.76
```

We need to analyze the file format to find the algorithm for interpreting the content

```
Date,Open,High,Low,Close,Volume,Adj Close
2014-02-03,502.61,551.19,499.30,545.99,12244400,545.99
2014-01-02,555.68,560.20,493.55,500.60,15698500,497.62
2013-12-02,558.00,575.14,538.80,561.02,12382100,557.68
2013-11-01,524.02,558.33,512.38,556.07,9898700,552.76
2013-10-01,478.45,539.25,478.28,522.70,12598400,516.57
...
1984-10-01,25.00,27.37,22.50,24.87,5654600,2.73
1984-09-07,26.50,29.00,24.62,25.12,5328800,2.76
```

File format:

- Columns are separated by comma
- First column is the date, the final is the price of interest
- The prizes start at different dates

We need algorithms before we can write code

Algorithm for reading data:

1. skip first line
2. read line by line
3. split each line wrt. comma
4. store first word (date) in a list of dates
5. store final word (prize) in a list of prices
6. collect date and price list in a dictionary (key is company)
7. make a function for reading one company's file

Plotting:

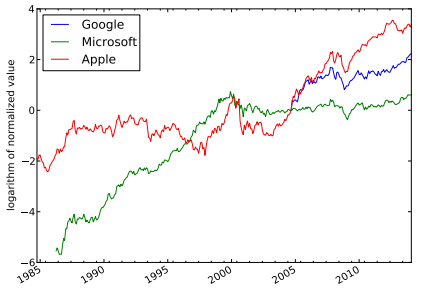
1. Convert year-month-day time specifications in strings into year coordinates along the x axis
2. Note that the companies' price history starts at different years

No code is presented here...

See the book for all details. If you understand this quite comprehensive example, you know and understand a lot!

Plot of normalized stock prices in logarithmic scale

Much computer history in this plot:



String manipulation

```
>>> s = 'This is a string'
>>> s.split()
['This', 'is', 'a', 'string']
>>> 'This' in s
True
>>> s.find('is')
4
>>> ', '.join(s.split())
'This, is, a, string'
```

String manipulation is key to interpret the content of files

- Text in Python is represented as strings
- Inspecting and manipulating strings is the way we can understand the contents of files
- Plan: first show basic operations, then address real examples

Sample string used for illustrations:

```
>>> s = 'Berlin: 18.4 C at 4 pm'
```

Strings behave much like lists/tuples - they are a sequence of characters:

```
>>> s[0]
'B'
>>> s[1]
'e'
>>> s[-1]
'm'
```

Extracting substrings

Substrings are just as slices of lists and arrays:

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s[8:]      # from index 8 to the end of the string
'18.4 C at 4 pm'
>>> s[8:12]    # index 8, 9, 10 and 11 (not 12!)
'18.4'
>>> s[8:-1]
'18.4 C at 4 p'
>>> s[8:-8]
'18.4 C'
```

Find start of substring:

```
>>> s.find('Berlin')  # where does 'Berlin' start?
0                     # at index 0
>>> s.find('pm')
20
>>> s.find('Oslo')    # not found
-1
```

Checking if a substring is contained in a string

```
>>> 'Berlin' in s:
True
>>> 'Oslo' in s:
False

>>> if 'C' in s:
...     print 'C found'
... else:
...     print 'no C'
...
C found
```

Substituting a substring by another string

`s.replace(s1, s2)`: replace `s1` by `s2`

```
>>> s.replace(' ', '__')
'Berlin:__18.4__C__at__4__pm'
>>> s.replace('Berlin', 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

Example: replace the text before the first colon by 'Bonn'

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s.replace(s[:s.find(':')], 'Bonn')
'Bonn: 18.4 C at 4 pm'
```

1) `s.find(':')` returns 6, 2) `s[:6]` is 'Berlin', 3) Berlin is replaced by 'Bonn'

Splitting a string into a list of substrings

`s.split(sep)`: split `s` into a list of substrings separated by `sep` (no separator implies split wrt whitespace):

```
>>> s
'Berlin: 18.4 C at 4 pm'
>>> s.split(':')
['Berlin', ' 18.4 C at 4 pm']
>>> s.split()
['Berlin:', '18.4', 'C', 'at', '4', 'pm']
```

Try to understand this one:

```
>>> s.split(':')[1].split()[0]
'18.4'
>>> deg = float(_)  # _ represents the last result
>>> deg
18.4
```

Splitting a string into lines

- Very often, a string contains lots of text and we want to split the text into separate lines
- Lines may be separated by different control characters on different platforms: \n on Unix/Linux/Mac, \r\n on Windows

```
>>> t = '1st line\n2nd line\n3rd line'      # Unix-line
>>> print t
1st line
2nd line
3rd line
>>> t.split('\n')
['1st line', '2nd line', '3rd line']
>>> t.splitlines()
['1st line', '2nd line', '3rd line']
>>> t = '1st line\r\n2nd line\r\n3rd line' # Windows
>>> t.split('\n')
['1st line\r', '2nd line\r', '3rd line']    # not what we want
>>> t.splitlines()                          # cross platform!
['1st line', '2nd line', '3rd line']
```

Strings are constant - immutable - objects

You cannot change a string in-place (as you can with lists and arrays) - all changes of a strings results in a new string

```
>>> s[18] = 5
...
TypeError: 'str' object does not support item assignment

>>> # build a new string by adding pieces of s:
>>> s2 = s[:18] + '5' + s[19:]
>>> s2
'Berlin: 18.4 C at 5 pm'
```

Stripping off leading/trailing whitespace

```
>>> s = '   text with leading/trailing space   \n'
>>> s.strip()
'text with leading/trailing space'
>>> s.lstrip()   # left strip
'text with leading/trailing space   \n'
>>> s.rstrip()   # right strip
'   text with leading/trailing space'
```

Some convenient string functions

```
>>> '214'.isdigit()
True
>>> '  214 '.isdigit()
False
>>> '2.14'.isdigit()
False

>>> s.lower()
'berlin: 18.4 c at 4 pm'
>>> s.upper()
'BERLIN: 18.4 C AT 4 PM'

>>> s.startswith('Berlin')
True
>>> s.endswith('am')
False

>>> '    '.isspace()   # blanks
True
>>> '  \n'.isspace()   # newline
True
>>> '  \t '.isspace()  # TAB
True
>>> ''.isspace()       # empty string
False
```

Joining a list of substrings to a new string

We can put strings together with a delimiter in between:

```
>>> strings = ['Newton', 'Secant', 'Bisection']
>>> ', '.join(strings)
'Newton, Secant, Bisection'
```

These are inverse operations:

```
t = delimiter.join(stringlist)
stringlist = t.split(delimiter)
```

Split off the first two words on a line:

```
>>> line = 'This is a line of words separated by space'
>>> words = line.split()
>>> line2 = ' '.join(words[2:])
>>> line2
'a line of words separated by space'
```

Example: Read pairs of numbers (x,y) from a file

Sample file:

```
(1.3,0)     (-1,2)     (3,-1.5)
(0,1)       (1,0)      (1,1)
(0,-0.01)   (10.5,-1)  (2.5,-2.5)
```

Algorithm:

1. Read line by line
2. For each line, split line into words
3. For each word, strip off the parethesis and split the rest wrt comma

The code for reading pairs

```
lines = open('read_pairs.dat', 'r').readlines()

pairs = []   # list of (n1, n2) pairs of numbers
for line in lines:
    words = line.split()
    for word in words:
        word = word[1:-1]  # strip off parenthesis
        n1, n2 = word.split(',')
        n1 = float(n1);  n2 = float(n2)
        pair = (n1, n2)
        pairs.append(pair)
```

Output of a pretty print of the pairs list

```
[(1.3, 0.0),
 (-1.0, 2.0),
 (3.0, -1.5),
 (0.0, 1.0),
 (1.0, 0.0),
 (1.0, 1.0),
 (0.0, -0.01),
 (10.5, -1.0),
 (2.5, -2.5)]
```

Alternative solution: Python syntax in file format

Suppose the file format

```
(1.3, 0)    (-1, 2)    (3, -1.5)
...
```

was slightly different:

```
[(1.3, 0),    (-1, 2),    (3, -1.5),
...
]
```

Running `eval` on the perturbed format produces the desired list!

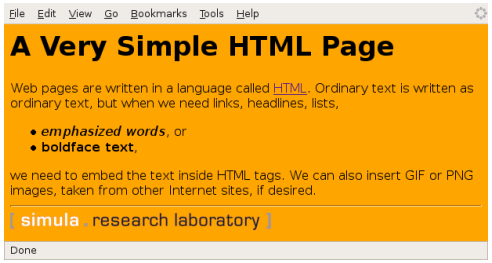
```
text = open('read_pairs2.dat', 'r').read()
text = '[' + text.replace(')', '),') + ']'
pairs = eval(text)
```

Web pages are nothing but text files

The text is a mix of HTML commands and the text displayed in the browser:

```
<html>
<body bgcolor="orange">
<h1>A Very Simple Web Page</h1> <!-- headline -->
Ordinary text is written as ordinary text, but when we
need headlines, lists,
<ul>
<li><em>emphasized words</em>, or
<li> <b>boldfaced words</b>,
</ul>
we need to embed the text inside HTML tags. We can also
insert GIF or PNG images, taken from other Internet sites,
if desired.
<hr> <!-- horizontal line -->
<img src="http://www.simula.no/simula_logo.gif">
</body>
</html>
```

The web page generated by HTML code from the previous slide



Programs can extract data from web pages

- A program can download a web page, as an HTML file, and extract data by interpreting the text in the file (using string operations).
- Example: climate data from the UK

Download `oxforddata.txt` to a local file `Oxford.txt`:

```
import urllib
baseurl = 'http://www.metoffice.gov.uk/climate/uk/stationdata'
filename = 'oxforddata.txt'
url = baseurl + '/' + filename
urllib.urlretrieve(url, filename='Oxford.txt')
```

The structure of the Oxfort.txt weather data file

```
Oxford
Location: 4509E 2072N, 63 metres amsl
Estimated data is marked with a * after the value.
Missing data (more than 2 days missing in month) is marked by  ---.
Sunshine data taken from an automatic ...
    yyyy  mm   tmax    tmin      af    rain     sun
                degC    degC    days      mm   hours
    1853   1    8.4     2.7       4    62.8     ---
    1853   2    3.2    -1.8      19    29.3     ---
    1853   3    7.7    -0.6      20    25.9     ---
    1853   4   12.6     4.5       0    60.1     ---
    1853   5   16.8     6.1       0    59.5     ---

...

    2010   5   17.6     7.3       0    28.6   207.4
    2010   6   23.0    11.1       0    34.5   230.5
    2010   7   23.3*   14.1*      0*   24.4*  184.4*  Provisional
    2010  10   14.6     7.4       2    43.5   128.8   Provisional
```

Reading the climate data

Algorithm:

1. Read the place and location in the file header
2. Skip the next 5 (for us uninteresting) lines
3. Read the column data and store in dictionary
4. Test for numbers with special annotation, "provisional" column, etc.

Program, part 1:

```
local_file = 'Oxford.txt'
infile = open(local_file, 'r')
data = {}
data['place'] = infile.readline().strip()
data['location'] = infile.readline().strip()
# Skip the next 5 lines
for i in range(5):
    infile.readline()
```

Reading the climate data - program, part 2

Program, part 2:

```
data['data'] ={}
for line in infile:
    columns = line.split()

    year = int(columns[0])
    month = int(columns[1])

    if columns[-1] == 'Provisional':
        del columns[-1]
    for i in range(2, len(columns)):
        if columns[i] == '---':
            columns[i] = None
        elif columns[i][-1] == '*' or columns[i][-1] == '#':
            # Strip off trasling character
            columns[i] = float(columns[i][:-1])
        else:
            columns[i] = float(columns[i])
```

Reading the climate data - program, part 3

Program, part 3

```
for line in infile:
    ...
    tmax, tmin, air_frost, rain, sun = columns[2:]

    if not year in data['data']:
        data['data'][year] = {}
    data['data'][year][month] = {'tmax': tmax,
                                 'tmin': tmin,
                                 'air frost': air_frost,
                                 'sun': sun}
```

Summary of dictionary functionality

Construction	Meaning
`a = {}`	initialize an empty dictionary
`a = {'point': [0,0.1], 'value': 7}`	initialize a dictionary
`a = dict(point=[2,7], value=3)`	initialize a dictionary w/string keys
`a.update(b)`	add/update key-value pairs from b in a
`a.update(key1=value1, key2=value2)`	add/update key-value pairs in a
`a['hide'] = True`	add new key-value pair to a
`a['point']`	get value corresponding to key point
`for key in a:`	loop over keys in unknown order
`for key in sorted(a):`	loop over keys in alphabetic order
`'value' in a`	True if string value is a key in a
`del a['point']`	delete a key-value pair from a
`list(a.keys())`	list of keys
`list(a.values())`	list of values
`len(a)`	number of key-value pairs in a
`isinstance(a, dict)`	is True if a is a dictionary

Summary of some string operations

```
s = 'Berlin: 18.4 C at 4 pm'
s[8:17]          # extract substring
s.find(':')      # index where first ':' is found
s.split(':')     # split into substrings
s.split()        # split wrt whitespace
'Berlin' in s    # test if substring is in s
s.replace('18.4', '20')
s.lower()        # lower case letters only
s.upper()        # upper case letters only
s.split()[4].isdigit()
s.strip()        # remove leading/trailing blanks
', '.join(list_of_words)
```