Dictionaries and strings (part 1)

Ole Christian Lingjærde, Dept of Informatics, UiO

18 October 2017

Today's agenda

- Exercise 5.16 and 5.18
- Dictionaries - a tool for storing value pairs
- Reading files into dictionaries

Exercise 5.16

Plot data from a file

- The files `density_water.dat` and `density_air.dat` files in the folder `src/plot13` contain data about the density of water and air (respectively) for different temperatures. The data files have some comment lines starting with # and some lines are blank. The rest of the lines contain density data: the temperature in the first column and the corresponding density in the second column.
- The goal of this exercise is to read the data in such a file and plot the density versus the temperature as distinct (small) circles for each data point. Let the program take the name of the data file as command-line argument. Apply the program to both files.

Filename: `read_density_data`

Planning the work

- We obtain `density_water.dat` and `density_air.dat` from the site `https://github.com/hplgit/scipro-primer/` and save them in a local directory where we also put the program file to be made.
- In the program we obtain the file name using `sys.argv[1]`.
- We read each file one line at a time and then split into words to obtain the two numbers on each line using `split()`.
- For every line we read, we first check if it is a comment/blank line and in that case we do nothing (except continue reading).
- We plot using the package `matplotlib.pyplot`.

Implementation

```
### Exercise 5.16

import sys
import matplotlib.pyplot as plt

fname = sys.argv[1]
temp = []; dens = []
infile = open(fname, 'r')
for line in infile:
    words = line.split();
    if len(words) >= 2 and words[0] != '#':
        temp.append(float(words[0]))
        dens.append(float(words[1]))
infile.close()
plt.plot(temp, dens, 'bo')
plt.show()
```

Exercise 5.18

Fit a polynomial to data points

The purpose of this exercise is to find a simple mathematical formula for how the density of water or air depends on the temperature.

The idea is to load density and temperature data from file as explained in Exercise 5.16 and then apply some NumPy utilities that can find a polynomial that approximates the density as a function of the temperature.

Exercise 5.18 (cont'd)

NumPy has a function `polyfit(x, y, deg)` for finding a best fit of a polynomial of degree deg to a set of data points given by the array arguments `x` and `y`. The polyfit function returns a list of the coefficients in the fitted polynomial, where the first element is the coefficient for the term with the highest degree, and the last element corresponds to the constant term.

For example, given points in `x` and `y`, `polyfit(x, y, 1)` returns the coefficients a, b in a polynomial $a * x + b$ that fits the data in the best way.

Exercise 5.18 (cont'd)

NumPy also has a utility `poly1d`, which can take the tuple or list of coefficients calculated by, e.g., `polyfit` and return the polynomial as a Python function that can be evaluated. The following code snippet demonstrates the use of `polyfit` and `poly1d`:

```
coeff = polyfit(x, y, deg)
p = poly1d(coeff)
print(p)            # Prints the polynomial expression
y_fitted = p(x)     # Computes the polynomial at the x points

# Use red circles for data points and a blue line for the polyn.
plot(x, y, 'ro', x, y_fitted, 'b-',
legend=('data', 'fitted polynomial of degree %d' % deg))
```

Exercise 5.18 (cont'd)

Questions:

- Write a function `fit(x, y, deg)` that creates a plot of data in `x` and `y` arrays along with polynomial approximations of degrees collected in the list `deg` as explained above.
- We want to call fit to make a plot of the density of water versus temperature and another plot of the density of air versus temperature. In both calls, use `deg=[1,2]` such that we can compare linear and quadratic approximations to the data.
- From a visual inspection of the plots, can you suggest simple mathematical formulas that relate the density of air to temperature and the density of water to temperature?

Filename: `fit_density_data`.

Answer to 5.18 a)

```
def fit(x, y, deg):
    import matplotlib.pyplot as plt
    import numpy as np
    plt.plot(x, y, 'bo')
    x1 = np.linspace(np.min(x), np.max(x), 100)
    for d in deg:
        coef = np.polyfit(x, y, d)  # Polynomial coefficients
        p = np.poly1d(coef)         # Polynomial function
        y1 = p(x1)                  # Polynomial value at x1 points
        plt.plot(x1, y1, 'r-')
    plt.show()
```

Answer to 5.18 b)

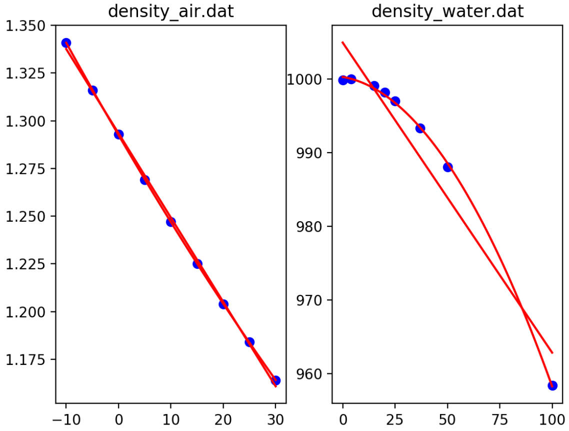
```
import matplotlib.pyplot as plt
import numpy as np

def fit(x, y, deg):
    plt.plot(x, y, 'bo')
    x1 = np.linspace(np.min(x), np.max(x), 100)
    for d in deg:
        coef = np.polyfit(x, y, d)  # Polynomial coefficients
        p = np.poly1d(coef)         # Polynomial function
        y1 = p(x1)                  # Polynomial value at x1 points
        plt.plot(x1, y1, 'r-')

fnames = ['density_air.dat', 'density_water.dat']

for i in range(len(fnames)):
    plt.subplot(1,2,i+1)
    temp = []; dens = []
    infile = open(fnames[i], 'r')
    for line in infile:
        words = line.split();
        if len(words) >= 2 and words[0] != '#':
            temp.append(float(words[0]))
            dens.append(float(words[1]))
    infile.close()
    fit(temp, dens, deg=[1,2])
    plt.title(fnames[i])
plt.show()
```

Result


density_air.dat
1.350
1.325
1.300
1.275
1.250
1.225
1.200
1.175
−10
0
10
20
30
density_water.dat
1000
990
980
970
960
0
25
50
75
100

Answer to 5.18 c)

Conclusion

- In one plot the linear and quadratic fits are almost identical, and both fit the data well.
- In the other plot the linear and quadratic fits are quite different, and the quadratic fit follows the data very well.
- We conclude that in both cases, the data are consistent with a quadratic relationship between x = temperature and y = density.

Implementing mappings in Python

A mathematical function (or "mapping") f is a rule that assigns a unique value $f(x)$ to a given x. We can implement this in Python with *functions* or with *dictionaries*.

Task: A mathematical function f has the rules $f(5) = 10$ and $f(11) = 15$. Implement this mapping in Python.

Using functions

```
def f(x):
    if x == 5:
        return 10
    elif x==11:
        return 15
print(f(5))         # Result: prints out 10
```

Using dictionaries

```
d = {5: 10, 11: 15}
print(d[5])         # Result: prints out 10
```

Another example

Task: Implement a mapping with the rules:

```
'Norway' --> 'Oslo'
'Sweden' --> 'Stockholm'
'France' --> 'Paris'
```

Using functions

```python
def f(x):
    if x == 'Norway':
        return 'Oslo'
    elif x == 'Sweden':
        return 'Stockholm'
    elif x == 'France':
        return 'Paris'
```

Using dictionaries

```python
d = {'Norway': 'Oslo', 'Sweden': 'Stockholm', 'France': 'Paris'}
```

Extending an existing mapping

Task: Extend the previous mapping with an additional pair:

```
'Norway' --> 'Oslo'
'Sweden' --> 'Stockholm'
'France' --> 'Paris'
'Nepal'  --> 'Kathmandu'
```

Using functions

We basically have to write the function all over again, this time with the four pairs above.

Using dictionaries

```
# We can easily extend the previous dictionary:
d['Nepal'] = 'Kathmandu'
```

Lists vs. dictionaries

Lists

We can think of a list in Python as a collection of pairs:

```
a = [6, 3, 7] # The pairs (0,6), (1,3), (2,7)
print(a[2])   # Access the value with index 2
```

The index is implicit when we define the list, but it can used later to access values in the list.

Dictionaries

A dictionary is also a collection of pairs:

```
d = {0:6, 1:3, 2:7} # The pairs (0,6), (1,3), (2,7)
print(d[2])         # Access the value with index 2
```

The index (=key) is explicitly given when we define the dictionary, and can also be used later to access values in the dictionary.

Dictionaries give index freedom

For a list, indexes are *always* `0, 1, ...., N-1` (where `N` is the length of the list). For a dictionary, indexes (= keys) can be whatever we like.

Examples:

```
# The pairs (-2,6), (6,3), (3,7)
d = {-2:6, 6:3, 3:7}

# The pairs ('Hamar','Norway'), ('Uppsala','Sweden')
d = {'Hamar':'Norway', 'Uppsala':'Sweden'}

# The pairs ('pi', 3.14), ('e', 2.718), ('g', 9.81)
d = {'pi':3.14, 'e':2.718, 'g':9.81}

# The pairs ((0,0), 'lowerleft') and ((1,1), 'upperright')
d = {(0,0):'lowerleft', (1,1):'upperright'}
```

There are some restrictions on keys: They should be unique and of an immutable data type (int, float, complex, string, tuple, ...).

Any immutable object can be used as key

Some data objects in Python can be modified after creation - such objects are called *mutable*. Other data objects are not modifiable - these are called *immutable*. Examples:

- A list object can be modified. For example, we can replace the *i*th value by a new value (`a[i] = new_value`) without creating an entirely new list.
- A tuple object can not be modified. Attempting to change the *i*th value by a new value results in an error message. If we need to make changes, we have to create a whole new tuple from scratch.

Keys in dictionaries can be any immutable object. Examples:

```
d = {1: 34, 2: 67, 3: 0}  # Key is int
d = {1:6424, 'X':64345}   # Key is int or string
d = {(0,0): 4, (1,-1): 5} # Key is a tuple
d = {[0,0]: 4, [-1,1]: 5} # WRONG: key cannot be a list
```

A dictionary is an unordered collection of key-value pairs

Keys and values

We have seen that a dictionary is a collection of pairs `(a,b)`. The first element in each pair is called a *key* and the second element a *value*. In the dictionary `d = {'Oslo': 3}`, 'Oslo' is a key and 3 the corresponding value. Note that the keys must be unique, i.e. two pairs can not have the same key.

Dictionaries are unordered

A dictionary does not store pairs in a particular sequence. One may envisage a dictionary as a bag of pairs `(a,b)` in no particular order.

Creating dictionaries

To create a dictionary, we initialize it and then insert new pairs as needed. A typical application is to store data that we read from file where one of the columns constitutes a unique identifier.

Initializing a dictionary

```
# Create an empty dictionary:
d = {}

# Create a dictionary with two pairs:
d = {'Oslo': 13, 'London': 15.4}

# Alternative to create a dictionary with two pairs:
d = dict(Oslo=13, London=15.4)
```

Extending a dictionary

```
# Suppose d is a dictionary

# Add a pair to the dictionary d:
d['Madrid'] = 26.0

# Add a dictionary d2 to d:
d.update(d2)
```

Removing an entry from a dictionary

We can remove a pair `(a,b)` from a dictionary using `del d[a]`.

Examples:

```
In [1]: d = {'pi':3.14, 'e':2.718, 'g':9.81}
In [2]: del d['e']  # Remove the pair ('e', 2.718)
In [3]: del d['pi'] # Remove the pair ('pi', 3.14)
In [4]: d = dict(Yes=1, No=0)
In [5]: del d['Maybe']
----------------------------------------------------------------------
KeyError                                  Traceback (most recent call
<ipython-input-37-759d96d71ff6> in <module>()
----> 1 del d['Maybe']

KeyError: 'Maybe'
```

Is a key present in a dictionary?

We can test if a key is present in a dictionary using 'key in d'.

Examples:

```
d = {'Bergen': 'Norway', 'Cambridge': 'UK'}

if 'Oxford' in d:
    print('Oxford is a key in the dictionary')
else:
    print('Oxford is not a key in the dictionary')

value = d.get('Bergen', '')     # Now value == 'Norway'
value = d.get('Oxford', '')     # Now value = ''
```

Looping over the elements in a dictionary

Looping over elements in arbitrary order

```
d = {-2:6, 6:3, 3:7}
for key in d:
    print('Key = %g and value = %g' % (key, d[key]))
```

Looping over elements in sorted key order

```
d = {-2:6, 6:3, 3:7}
for key in sorted(d):
    print('Key = %g and value = %g' % (key, d[key]))
```

Obtaining a list of all keys/values

Suppose we have defined a list

```
d = {'Paris': 17.5, 'London': 15.4, 'Madrid': 26.0}
```

Python 2

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

Python 3

```
>>> d.keys()
<listiterator at 0x111b2a4d0>
>>> list(d.keys())
['Paris', 'London', 'Madrid']
>>> d.values()
<listiterator at 0x111b2a710>
>>> list(d.values())
[17.5, 15.4, 26.0]
```

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`):

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

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

```python
def f(p,x):
    val = 0.0
    for power in p:
        val += p[power] * x ** power
    return val
```

Shorter version:

```python
def f2(p,x):
    return sum(p[power] * x ** power for power in p)
```

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, 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 f3(p, x):
    val = 0
    for power in range(len(p)):
        val += p[power] * x ** power
    return val
```

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 f(p, x=4)
```

Example: reading a file with two columns

Many data files consist of columns of data, such as this one where the first column is a unique patient identifier and the second is a measure of DNA damage:

```
MB.0000       0.00096
MB.0002       0.24787
MB.0005       0.2779
MB.0006       0.29428
MB.0010       0.61225
...      ...
```

We can read this file into a dictionary as follows:

```python
damage = {}
infile = open('DNAdamage.txt', 'r')
for line in infile:
    words = line.split()
    damage[words[0]] = float(words[1])
infile.close()

# To print the DNA damage for patient MB.0005:
print(damage['MB.0005'])   # 0.00096
```

Example: reading a file with three columns

Data file:

```
Oslo          21.8     'Norway'
Bergen        17.6     'Norway'
London        18.1     'UK'
Berlin        19       'Germany'
Paris         23       'France'
Rome          26       'Italy'
Helsinki      17.8     'Finland'
```

Program:

```python
infile = open('cityinfo.txt', 'r')
data = {}
for line in infile:
    words = line.split()
    data[words[0]] = [float(words[1]), words[2]]
infile.close()

# To print the information about Paris:
print(data['Paris'])     # [23.0, "'France'"]
print(data['Paris'][0])  # 23.0
print(data['Paris'][1])  # 'France'
```

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!