11_Files

December 11, 2014

Figure 1:

Authors : *Sonia Estradé*
José M. Gómez
Ricardo Graciani
Manuel López
Xavier Luri
Josep Sabater

In many circumstances, it is necessary to store and retrieve data in order to avoid losing it. This is what we do with our documents. Python also allows to save information in a file and load it when required.

0.1 Magic functions in Notebook

iPython Notebook has a set of predefined 'magic functions' that you can call within a notebook **cell of code**. There are two kinds of magics: line-oriented (%) and cell-oriented (%%).

The %lsmagic function is used to list all available magics, and it will show both line and cell magics currently defined:

```
In [1]: %lsmagic
```

```
Out[1]: Available line magics:
        %alias  %alias_magic  %autocall  %automagic  %autosave  %bookmark  %cat  %cd  %clear  %colors  %

        Available cell magics:
        %%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%latex  %%perl  %

        Automagic is ON, % prefix IS NOT needed for line magics.
```

For example, we can use the Unix/Linux command `pwd` to show the current work directory:

```
In [2]: %pwd
```

```
Out[2]: '/home/joseps/Projects/Pgm/svn_infor/Python/ipython'
```

In the following sections, we will use more magic functions, specially those related with files. The reader can find more examples about magic functions here.

0.2 Working with basic text data

The first step is to read or write unstructured data in a file.

0.2.1 Reading a simple text file

First we create a simple file, using the cell-oriented magic function %%file:

```
In [3]: %%file test.txt
        This is a text file created to check
        Python file read and write.
```

```
Overwriting test.txt
```

To read the file, first of all it is necessary to open it:

```
In [4]: inputFile = open('test.txt', 'rU')

        print(inputFile)
```

```
<_io.TextIOWrapper name='test.txt' mode='rU' encoding='UTF-8'>
```

The parameters are:

- name of the file.
- read mode (`r`), we can also use binary mode (`b`) to avoid problems with text conversion.
- the optional universal line-end mode (`U`) to be able to interchange documents between operating systems.

Next we read the lines from the file:

```
In [5]: file_in = inputFile.readlines()
        print(file_in)

        for line in file_in:
            print(line, end='')
```

```
['This is a text file created to check\n', 'Python file read and write.']
This is a text file created to check
Python file read and write.
```

Next the file must be closed:

```
In [6]: inputFile.close()
```

0.2.2 Writing a simple text file

Now, following a similar procedure to write an unstructured file.

```
In [7]: outputFile = open('secondTest.txt', 'w')
```

```
In [8]: outputFile.write('Another file\n')
        outputFile.write('A second line\n')
```

```
Out[8]: 14
```

```
In [9]: outputFile.close()
```

```
In [10]: %less secondTest.txt
```

0.2.3 The `with` statement

Sometimes, the files cannot be read or written properly and errors arise. And if an error appears, the file must be always closed. To avoid problems, python provides the `with` statement that is in charge of all this aspects.

```
In [11]: with open('thirdTest.txt', 'w') as outputFile:
             for i in range(10):
                 outputFile.write('New test. ' +
                                  str(i) + '\n')
```

```
In [12]: %less thirdTest.txt
```

```
In [13]: outputFile.closed
```

```
Out[13]: True
```

The file has been written and closed.

```
In [14]: with open('thirdTest.txt', 'rU') as inputFile:
             print(inputFile.read(), end = '')
```

```
New test. 0
New test. 1
New test. 2
New test. 3
New test. 4
New test. 5
New test. 6
New test. 7
New test. 8
New test. 9
```

0.3 Working with helper modules

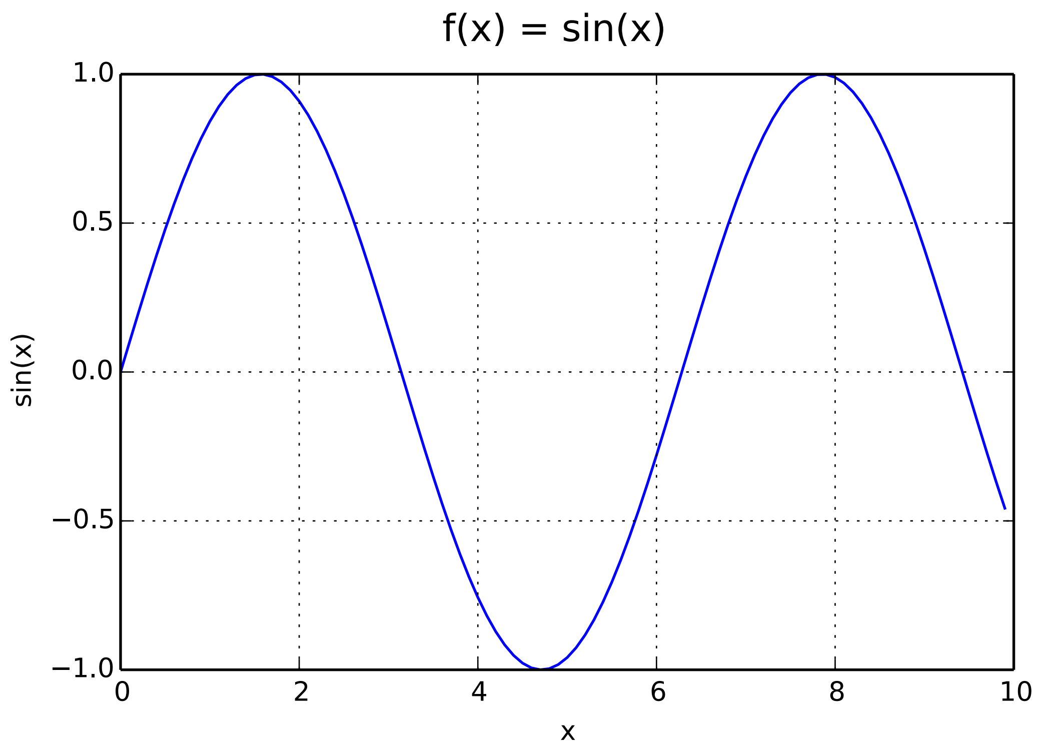
Now we will try to write data. For example, we can create two arrays of data:

```
In [15]: import math

         x = [point/10. for point in range(0, 100)]
         y = [math.sin(x_point) for x_point in x]
```

```
In [16]: %pylab inline
         %config InlineBackend.figure_format = 'svg'

         import matplotlib.pyplot as plt

         str_title = 'f(x) = sin(x)'
         fig1 = plt.figure()
         fig1.suptitle(str_title, fontsize=14)
         fig1_ax = fig1.add_subplot(1,1,1)
         fig1_ax.set_xlabel('x')
         fig1_ax.set_ylabel('sin(x)')
         fig1_ax.grid(True, which='both')

         fig1_ax.plot(x, y)
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
Out[16]: [<matplotlib.lines.Line2D at 0x7fd6834b3a58>]
```



0.3.1 Writting a file with `json`

The simplest way to store data from Python is using the `json` module. In this case, it is only necessary to use the function `dump` with the variable to store as first argument and the file as second:

```
In [17]: import json

         with open('sin.json', 'w') as outputFile:
             json.dump({'x': x, 'y': y}, outputFile, sort_keys=True)
```

With this, a file has been created that contains the information from x an y.

```
In [18]: %less sin.json
```

But the file is not easy to understand.

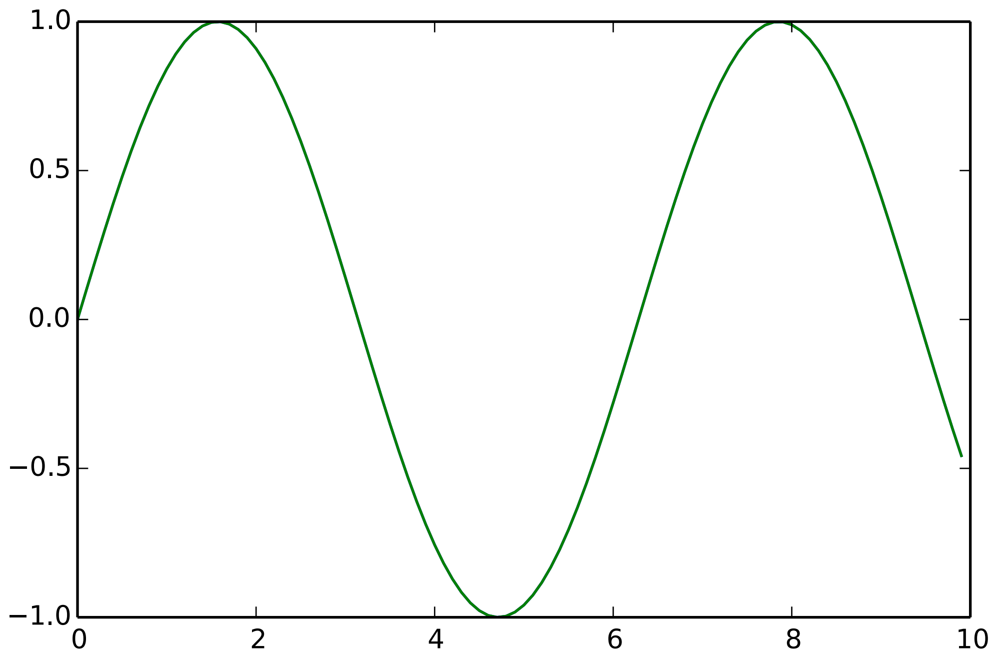
0.3.2 Reading the `json` file

Now it can be retrieved with the function `load`, which return the contents of the variable. It is important to notice that the order to read the data must be exactly the same as the one it was stored:

```
In [19]: with open('sin.json', 'r') as inputFile:
             data = json.load(inputFile)
             xf = data['x']
             yf = data['y']
```

```
In [20]: import matplotlib.pyplot as plt

         plt.plot(xf, yf, x, y)
```

```
Out[20]: [<matplotlib.lines.Line2D at 0x7fd6833aea58>,
          <matplotlib.lines.Line2D at 0x7fd6833aedd8>]
```



The major difficulty with this format is that only Python understands it. For this reason, if it is desired to interchange data, for example, with an spreadsheet, it would be necessary to use an structured format.

0.4 Structured data

The basic way would be working with `txt` files:

0.4.1 Working with structured `txt`

First it is necessary to define the format. To simplify how to read it, we will write it in a similar way as we will do with a table. The columns will be separated by tabulators:

```
In [21]: with open('sin.txt', 'w') as outputFile:
             for dataX, dataY in zip(x, y):
                 outputFile.write(str(dataX) +
                                  "\t" + str(dataY) +
                                  "\n")
```

The data is stored interleaved, the x in the first column and the y in the second, and separated by a tabulator.

```
In [22]: %less sin.txt
```

The major difficulty is reading back the data, as it has to be decoded. First the file is open and all the text lines are retrieved:

```
In [23]: with open('sin.txt', 'r') as inputFile:
             lines = []

             # The lines are read from the file
             for line in inputFile.readlines():

                 # The line is stored in the list
                 lines.append(line)
```

Every line has the two values, the x and the y, although mixed with the tabulation and the new line:

```
In [24]: lines[1]

Out[24]: '0.1\t0.09983341664682815\n'
```

First it is necessary to separate the different values. For this, two steps are needed, first the unnecesary space characters must be taken out (`strip`):

```
In [25]: lineStripped = lines[0].strip()

         lineStripped

Out[25]: '0.0\t0.0'
```

The second step would be to `split` the line in two cells using the `\t`.

```
In [26]: lineSplitted = lineStripped.split('\t')

         lineSplitted

Out[26]: ['0.0', '0.0']
```

Finally, it is necessary to convert the strings in float values, using the `float` function:

```
In [27]: float(lineSplitted[0])

Out[27]: 0.0
```

Using this procedure, the data can be retrieved:

```
In [28]: xf = []
         yf = []

         for line in lines:
             # The line is stripped taking out all the unnecesary characters
             lineStripped = line.strip()

             # The line is splitted using the tab character
             lineSplitted = lineStripped.split('\t')

             xf.append(float(lineSplitted[0]))
             yf.append(float(lineSplitted[1]))
```

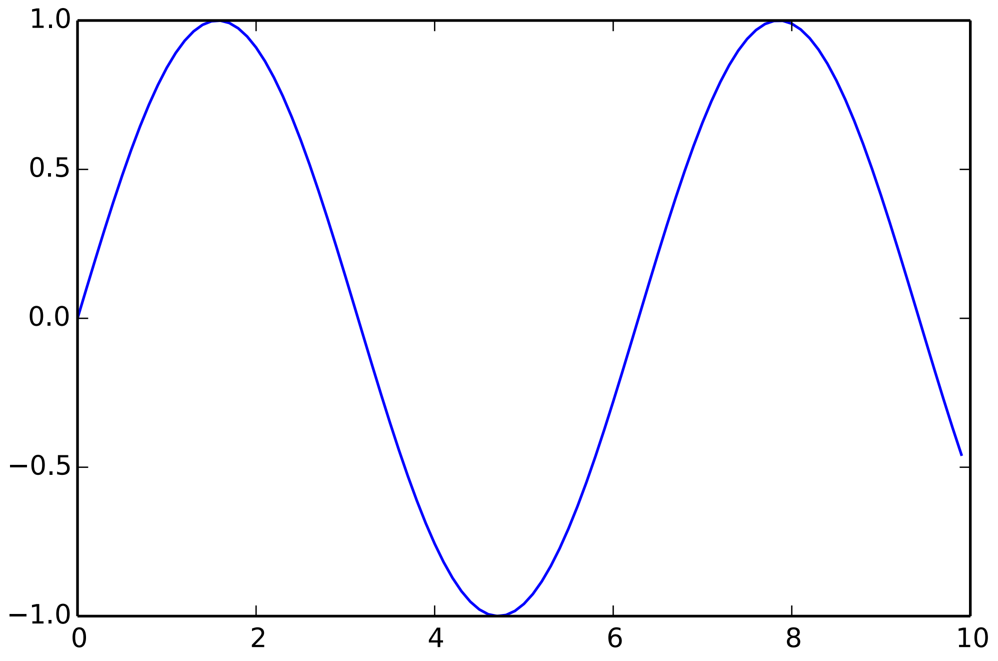
The whole procedure can be grouped:

```
In [29]: xf = []
         yf = []

         with open('sin.txt', 'r') as inputFile:
             for line in inputFile.readlines():
                 lineStripped = line.strip()
                 lineSplitted = lineStripped.split('\t')

                 xf.append(float(lineSplitted[0]))
                 yf.append(float(lineSplitted[1]))
```

Now the data can be shown:

```
In [30]: import matplotlib.pyplot as plt

         plt.plot(xf, yf)
```

```
Out[30]: [<matplotlib.lines.Line2D at 0x7fd683329860>]
```



This implies that for every file, it is necessary to write the strip and split code and process the information. Python provides another alternative.

0.4.2 Using the `csv` module

The same operations can be followed, but with the csv, that makes files compatible with Excel or Calc. In this case, we need to use the locale of the computer, to be sure that both programs correctly understand the files contents. Let's see an example:

```
In [31]: import locale

         locale.setlocale(locale.LC_ALL, '')
```

```
        with open('sinCSV.csv', 'w') as outputFile:
            for dataX, dataY in zip(x, y):
                outputFile.write('{:n};{:n}\n'.format(dataX, dataY))
```

```
In [32]: %less sinCSV.csv
```

The file separates the column using semi-colons. Now we can read it with Excel or Calc.

We can do the same process, but using the csv module. In this case, to be sure that we have the desired native format, we must modify a little the procedure, usign list comprenhension:

```
In [33]: import locale
         import csv

         locale.setlocale(locale.LC_ALL, '')

         with open('sinCSV.csv', 'w') as outputFile:
             writer = csv.writer(outputFile, delimiter=';')
             dataX = [locale.str(value) for value in x]
             dataY = [locale.str(value) for value in y]
             writer.writerows(zip(dataX, dataY))
```

The result is equivalent. We can also read the file:

```
In [34]: %less sinCSV.csv
```

We can also read the file. This can be easily done with the `csv` module:

```
In [35]: import csv

         sinList = []

         with open('sinCSV.csv', 'rU') as csvfile:
             sinReader = csv.reader(csvfile,
                                    delimiter=';',
                                    quotechar='"')
             for row in sinReader:
                 sinList.append(row)
```

```
In [36]: len(sinList[0])
```

```
Out[36]: 2
```

In this case, the cells are fully separated:

```
In [37]: type(sinList[0][0])
```

```
Out[37]: str
```

Now the values can be converted to floats as in the previous case. To simplify things, a comprenhension list can be used:

```
In [38]: data = [[float(cell) for cell in row] for row in sinList]
```

A new problem appears, the use of the comma instead of a dot, does not allow to convert the string to a float. This can be solved using the `locale` module another time. We convert the strings to floats using the `atof` function from locale:

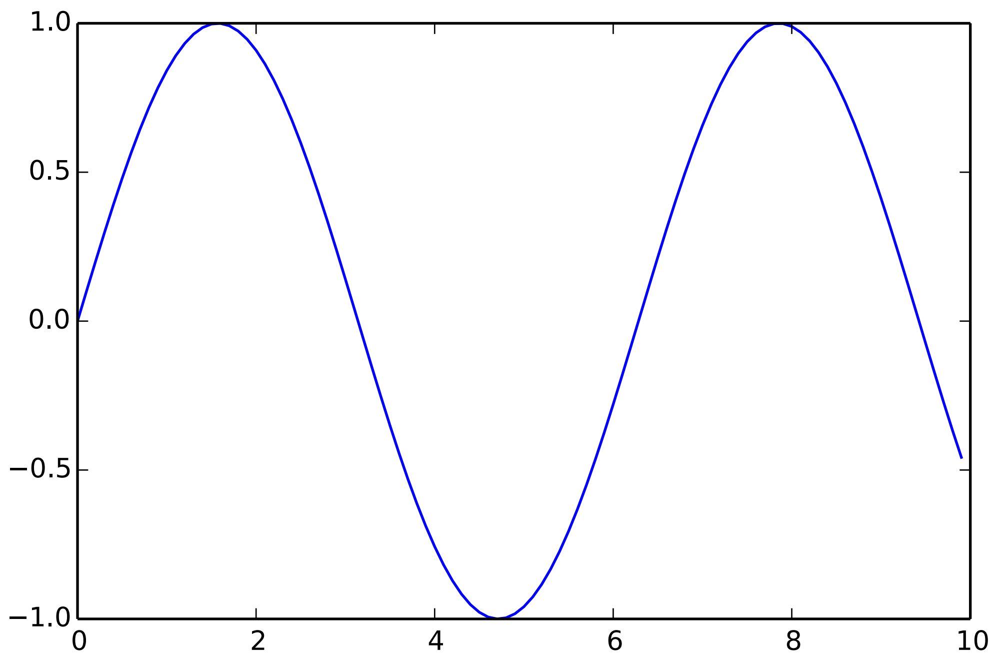
```
In [39]: data = [[locale.atof(cell) for cell in row] for row in sinList]
```

Now this data can be transferred to two lists to draw them:

```
In [40]: xf = []
         yf = []

         with open('sinCSV.csv', 'rU') as inputFile:
             sinReader = csv.reader(inputFile, delimiter=';', quotechar='"')
             for row in sinReader:
                 xf.append(locale.atof(row[0]))
                 yf.append(locale.atof(row[1]))
```

```
In [41]: %config InlineBackend.figure_format = 'svg'

         import matplotlib.pyplot as plt

         plt.plot(xf, yf)
```

```
Out[41]: [<matplotlib.lines.Line2D at 0x7fd683221be0>]
```



Now it is possible to read and write files.