# Python  Working with files

May 4, 2017

# Working with files

So far, everything we have done in Python was using *in-memory* operations.

After closing the Python interpreter or after the script was done, all our input and output was gone.

In real life applications we want to work with files stored on the computers hard drive.

# Working with files

Example situations were files are necessary:

- A collaborator sends you raw data via email.
- A program you are using reads input data from a file. When running the program multiple times, you don't want to recreate the data every time it is needed, but store it in a file and reuse it.
- Another program produces lots of output that you want to analyze using your own program.
- You want to keep intermediate calculations for further processing.

# Working with files

Scientific data is typically stored in `plain text` files.
This means human readable files, with no internal format information.
Although there are sometimes special binary based formats.

In Python to save or load data you need to use a special `file handle` object. The built-in `open()` function will return a file handle object. It takes a path to the file you want to open as an argument.
So let’s say we want to open a file called `data.txt` and get a file handle `f` to operate on the file. This can be accomplished using the following code:

```python
f = open(’data.txt’)
```

# Working with files

The `open()` function implicitly performed the following actions:

- checks if `data.txt` exists, throws an exception otherwise
- creates the file handle
- sets the *cursor* position to the start of the file, `pos = 0`

It did not read anything from the file, nor did it write to it. The file is also not closed after this function call returns. All these actions must be done separately.

# Working with files

## File methods

Here is a list of the most important file methods.

```
f.read(n=-1) # Reads n byte from the file. If n not set or is -1
             # the entire rest of the file is read.

f.readline() # Reads in the next full line and returns a string
             # which includes the newline character and the end.

f.readlines() # Reads the remaining lines into a list of strings.

f.seek(pos) # Moves the file cursor to a specific position.

f.tell() # Returns the current position of the cursor.

f.write(s) # Inserts the string s at the current position.

f.flush() # Performs all pending write operations, making
          # sure that they are really on disk.

f.close() # Closes the file. No more reading or writing possible
          # After this call.
```

# Working with files

## Example

Let say we have a file `matrix.txt` with the following content:

```
1,4,15,9
0,11,7,3
2,8,12,13
14,5,10,6
```

To read this file into Python we could use the following code snippet:

```
f = open(’matrix.txt’)
matrix = []
for line in f.readlines():
    row = [int(x) for x in line.split(’,’)]
    matrix.append(row)
f.close()
```

Another version:

```
matrix = []
with open(’matrix.txt’) as f:
    for line in f.readlines():
        row = [int(x) for x in line.split(’,’)]
        matrix.append(row)
```

# Working with files

## File modes

Files are opened in one of multiple *modes*.

The mode determines the methods that can be used on the file handle. Invalid methods are still callable, but will throw an exception.

So far we only used the default *read-only* mode, passing no additional argument to the `open()` function. If we wanted to open a file for writing, we would use the following code:

```
f = open('data.txt', 'w')
```

# Working with files

## File modes

Here is a list modes available:

```
'r' # read only mode, starting pos = 0

'w' # write mode, creates file if it does not exist
    # overwrites existing files (CAUTION!), starting pos = 0

'a' # append mode, like write mode, but pos is at the
    # end of the file

'+' # update mode, opens for both reading and writing,
    # but does not delete current content, starting pos = 0
```

These modes are the same for other programming languages (especially C) as well.

# Working with files

## File mode example

Let’s look at the `matrix.txt` example again.

```
# old matrix.txt
# 1,4,15,9
# 0,11,7,3
# 2,8,12,13
# 14,5,10,6

f = open('matrix.txt', 'r+') # open in read and write mode
orig = f.read() # read entire file into a single string
f.seek(0) # rewind file (go back to the start)
f.write('0,0,0,0\n') # write new line, overwriting existing one
f.write(orig) # write original contents back after added line
f.write('1,1,1,1\n') # add another line to the end
f.close() # close file, we're done here

# new matrix.txt
# 0,0,0,0
# 1,4,15,9
# 0,11,7,3
# 2,8,12,13
# 14,5,10,6
# 1,1,1,1
```

# Working with files

## numpy

The `numpy` module comes with special functions to read and write data text files. To read a data file, where every row has the same amount of columns you could use numpy's `loadtxt()` function.

```
from numpy import loadtxt

matrix = loadtxt(’matrix.txt’)
```

The full list of options for the `loadtxt()` function looks like this:

```
loadtxt(fname, dtype=<type ’float’>, comments=’#’,
        delimiter=None, converters=None, skiprows=0,
        usecols=None, unpack=False, ndmin=0)
```

# Working with files

## loadtxt options

The most important `loadtxt()` options are the following:

```
fname # filename or str

dtype # data type of resulting array. default value: float

comments # str, optional, character used for comments in the file
         # default value: '#'

delimiter # str, optional, String used to separate values
          # default value is a whitespace

skiprows # int, optional, skips the number of rows given

usecols # sequence, optional read only given columns
        # (1, 4, 5) reads second, 5th and 6th column/line

unpack # bool, optional, default: False
       # if True, the returned array is transposed
```

# Working with files

## numpy savetxt

Saving data to a file works just as easy as reading them. numpy provides the savetxt() function to accomplish this.

```
from numpy import savetxt

matrix = [ [1, 4, 15, 9],
           [0, 11, 7, 3],
           [2, 8, 12, 13],
           [14, 5, 10, 6] ]

savetxt(’matrix.txt’, matrix)
```

Again you can customize how the data is saved using the optional parameters of savetxt():

```
savetxt(fname, X, fmt=’%.18e’, delimiter=’ ’, newline=’\n’,
        header=’’, footer=’’, comments=’# ’)
```

# Working with files

## savetxt options

The most important `savetxt()` options are the following:

```
fname # filename or str

X # array, data to be saved
  # use transpose([x, y]) to save two arrays in two columns

fmt # str or sequence of strs, optional
    # defines the output format for the data

comments # str, optional, character used for comments in the file
         # default value: ’# ’

delimiter # str, optional, String used to separate values
          # default value is a whitespace

header # str, optional
       # inserted before data

footer # str, optional
       # inserted at the end of the file
```

# Working with files

## savetxt - format parameter

Here I want to explain the `fmt` parameter in more detail.
The full string looks something like this:

```
(%[flag]width[.precision]specifier)
```

possible flags:

```
- # left justify
+ # Forces to preceed result with the sign (+ or -)
0 # Left pad the number with zeroes instead of space
```

`width` is the minimum number of characters to be printed, longer values will not be truncated to `width`.
Precision:

- for integer specifiers (eg. `d, i, o, x`), the minimum number of digits
- for `e, E` and `f`, number of digits after decimal points
- for `g` and `G`, number of significant digits
- for `s`, maximum number of characters

# Working with files

## savetxt - format parameter

Specifiers:

```
c # character

d or i # signed decimal integer

e or E # scientific notation (1e6 or 10E-9)

f # decimal floating point

g or G # use the shorter e, E or f

o # signed octal

s # string of characters

u # unsigned decimal integer

x, X # unsigned hexadecimal integer
```