Files

Petr Pošík

Department of Cybernetics, FEE CTU in Prague

EECS, BE5B33PRG: Programming Essentials, 2015

Requirements:

- Loops

Intro

Information on a computer is stored in named chunks of data called **files**. We will learn how to read info from files, and how to write info into files.

Files on disk are organized hierarchically in directories (folders). We will first review some basics about working with them.

Directories and paths

Directories

A **directory** is an organizational unit of the file system that allows for the hierarchichal structure. It can contain files and other directories. Each directory is itself contained in some other directory.

The **root directory** is special. It is always present in the filesystem and designates the start of the hierarchy. It is denoted with forward slash (`/`) on Linux-types of systems. On Windows, there is a separate root directory on each disk, and they are denoted by backslashes (`\`).

Each directory contains 2 special entries:

- `.` is the current directory itself
- `..` is the parent directory

Current working directory

Python keeps track of the **current working directory** (CWD), in which it looks for files. The default setting of CWD is platform dependent, but usually it is the directory from which the Python interpreter was run (not the directory where the interpreter is stored).

You can find out the CWD using function `getcwd` from module `os`. Let's see what the CWD is for the current notebook:

In [1]:
```
import os
print(os.getcwd())
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

Paths

Paths (https://en.wikipedia.org/wiki/Path_%28computing%29) are sequences of directory names possibly ended with a filename which unambiguously resolve to a directory or file name in the filesystem.

Absolute paths always start from the root directory (`/`, forward slash), on Windows often preceded by drive letter (`C:\\`). An example:

```
/home/posik/teaching/prg/lectures/files.pdf
```

Relative paths start from the current working directory. Examples, assuming the CWD is `/home/posik/teaching`:

```
prg/lectures/files.pdf
../../svoboda/presentations/upload_system.pdf
```

Navigating in the file system

In the OS shell, you would use command `cd` or `chdir` to change the working directory. Similarly, you can use the Python function `os.chdir()`:

In [2]:
```
import os
orig_wd = os.getcwd()
os.chdir('/P/0Teaching')
print(os.getcwd())
```

```
C:\P\0Teaching
```

Now we are in a different directory. And we can change it back:

In [3]:
```
os.chdir(orig_wd)
print(os.getcwd())
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

Working with file paths

Module `os.path` contains functions for working with file paths:

In [4]:
```
fpath = os.path.abspath('files.pdf')
print(fpath)
```

```
C:\P\0Teaching\programming essentials\prg-notes\files.pdf
```

In [5]:
```
print(os.path.dirname(fpath))
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

In [6]:
```
print(os.path.basename(fpath))
```

```
files.pdf
```

In [7]:
```
print(os.path.splitext(os.path.basename(fpath)))
```

```
('files', '.pdf')
```

How to correctly create a path from fragments?

In [8]:
```
fpath2 = os.path.join('\\', 'P', '0Teaching')
print(fpath2)
```

```
\P\0Teaching
```

How to get a path to a directory or a file relative to CWD?

In [9]:
```
print(os.path.relpath(fpath2))
```

```
..\..
```

Files

File types

- **Plain text files**:
  - contain only characters
  - readable in any text editor
- **Binary files**:
  - music files, videos, word processor documents, presentations, ...
  - contain various formatting information specific to the particular file format
  - require special program that understands that format

In this lecture, we shall concentrate on plain text files.

Text files

- Take up little disk space (an empty text file is truly empty, it has size of 0).
- They may still obey certain structure:
  - Source code
  - Comma separated values (CSV)
  - HTML files
  - ...

Open and close

If you want to look at the contents of a drawer, or if you want to put something in, you have to open the drawer first. When you are done, you have to close it. The same holds for files.

When you open a drawer, you hold its **handle** which allows you to close the drawer again. The **file handle** allows you to do all sorts of things with an open file. You can read the file, seek some position in the file, etc.

Example: read file contents

Let's create a simple text file `text.txt` in the current directory (remember, the following is an IPython way of creating a text file):

In [10]:
```
%%writefile text.txt
Hello, world!
How are you?
```

```
Overwriting text.txt
```

Now, we can read that file in Python and display its contents:

In [11]:
```
file = open('text.txt', 'r')
contents = file.read()
file.close()
print(contents)
```

```
Hello, world!
How are you?
```

1. The first line instructs Python (and operating system) to **open** a file named `text.txt` (argument 1), and return a file handle to this file. Argument 2, `'r'` (also called *file mode*), indicates that the file shall be opened for reading. There are several **file modes** that can be used to open a file for reading (`'r'`), writing (`'w'`), appending (`'a'`), and which also specifies whether we want to open the file in text mode or in binary mode (`'wb'`, `'rb'`, ...).
2. The second line calls the `read()` method of the file handle object which reads all the contents of the file as one huge string, which is then assigned to variable `contents`.
3. The third line uses the file handle to close the file.
4. Finally, we print the contents of the file, i.e. it releases all the resources associated with the open file object.

Encoding of strings and files

Strings are actually an abstraction. They are just sequences of bytes, but these bytes (or their groups) are interpreted as indices into a table of symbols containing upper- and lowercase letters, numbers, special characters and other symbols. The table with these symbols is called an **encoding**. The same string may look as complete gibberish if it is read with different encoding than the one used when creating it.

- ASCII: contains 127 characters, english upper- and lowercase letters, numbers, and some symbols. No characters from national alphabets.
- ...
- **UTF-8**: Unicode encoding that supports virtually any national alphabet, contains ASCII as its subset. **USE IT!**

This holds also for text files!

Opening text file with encoding

Function `open()` accepts several other parameters, among them the `encoding`. If you will **explicitly use UTF-8 each time** you call that function, you will save yourself a lot of trouble:

```
f = open('file_to_open.txt', 'r', encoding='utf-8')
...
f.close()
```

or

```
with open('file_to_open.txt', 'r', encoding='utf-8') as f:
    ...
```

The `with` statement

Because every call to `open()` should have a corresponding call to the `close()` method, Python provides a `with` statement that automatically closes a file when the end of the block is reached. The code

```
f = open('text.txt', 'r', encoding='utf-8')
contents = f.read()
f.close()
print(contents)
```

is equivalent to the following code using the `with` statement:

```
with open('text.txt', 'r', encoding='utf-8') as f:
     contents = f.read()
print(contents)
```

File reading: `file.read()`

Use this technique when you want to read the file contents into a single (possibly huge) string, or when you want to specify, how many character shall be read.

In [12]:
```
with open('text.txt', 'r', encoding='utf-8') as f:
    contents = f.read()
print(contents)
```

```
Hello, world!
How are you?
```

When called with no arguments, it reads everything from the current file cursor all the way to the end of the file. When called with an integer argument, it reads that many characters and moves the cursor right after the characters that were just read.

In [13]:
```
with open('text.txt', 'r', encoding='utf-8') as f:
    first_10_chars = f.read(10)
    the_rest = f.read()
print("The first 10 chars:", first_10_chars)
print("The rest:", the_rest)
```

```
The first 10 chars: Hello, wor
The rest: ld!
How are you?
```

File reading: `file.readlines()`

Use this technique if you want to get a Python list of strings containing the individual lines from a file.

In [14]:
```
with open('text.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
print(lines)
```

```
['Hello, world!\n', 'How are you?']
```

Note that the strings representing the individual lines contain also the newline character, `\n`. The last line may or may not end with a newline char. You can get rid of them using the `str.strip()` method.

In [15]:
```
for line in lines:
    print(line.strip())
```

```
Hello, world!
How are you?
```

File reading: `for <line> in <file>`

Use this technique when you want to do the same thing to every line from the file cursor to the end of a file. While the previous techniques read all the content of a file at once (which may not fit to memory), this technique reads one line at a time allowing to process large files.

In [16]:
```
with open('text.txt', 'r', encoding='utf-8') as f:
    for line in f:
        s = line.strip()
        print("The line '" + s + "' contains " + str(len(s)) + " characters.")
```

```
The line 'Hello, world!' contains 13 characters.
The line 'How are you?' contains 12 characters.
```

File reading: `file.readline()`

This technique allows you to read a single line from a file, which is useful when you want to read only a part of the file.

Assume that we want to read the following text file which contains several different parts. The first line is the short description of the data. The next lines starting with # are comments. The following part contains the data.

In [17]:
```
%%writefile data_collatz_5.txt
Collatz 3n+1 sequence, starting from 5.
# The next number in a Collatz sequence is either 3n+1 if n is odd,
# or n/2 if n is even.
  5
 16
  8
  4
  2
  1
```

```
Overwriting data_collatz_5.txt
```

We will use `readline()` to read the header lines, then we will use `for line in file` to read the data.

In [18]:
```
with open('data_collatz_5.txt', 'r', encoding='utf-8') as f:
    # Read the description line
    description = f.readline().strip()
    # Read all the comment lines
    comments = []
    line = f.readline().strip()
    while line.startswith('#'):
        comments.append(line)
        line = f.readline().strip()
    data = []
    data.append(int(line))
    for line in f:
        data.append(int(line))

print("Description:", description)
print("Comments:", comments)
print("Data:", data)
```

```
Description: Collatz 3n+1 sequence, starting from 5.
Comments: ['# The next number in a Collatz sequence is either 3n+1 if n is odd,', '#
or n/2 if n is even.']
Data: [5, 16, 8, 4, 2, 1]
```

Reading file from Internet

If a file is accessible on the Internet, we can read it just as we do a local file, using function `urllib.request.urlopen()`. (Of course, you have to be connected to the Internet.)

There is only one difference: since `urlopen` cannot know what type of file you want to read, the methods `read`, `readline`, etc. return a type called `bytes`. To get a string from it, we need to decode it, i.e. assign the the symbol lookup table, preferably UTF-8.

In [19]:
```
url = r'http://www.gutenberg.org/cache/epub/1661/pg1661.txt'
import urllib.request
with urllib.request.urlopen(url) as text:
    intro = text.read()
intro = intro.decode('utf-8')
print(intro[:300])
```

```
Project Gutenberg's The Adventures of Sherlock Holmes, by Arthur Conan Doyle

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
re-use it under the terms of the Project Gutenberg License included
with this eBoo
```

Writing files

Writing some text into a text file is very similar to reading it. Also, when reading, Python did not strip the newline characters; when writing, you have to put the newlines there manually as well.

In [20]:
```
with open('topics.txt', 'w', encoding='utf-8') as f:
    f.write('Computer Science\n')
    f.write('Programming\n')
    f.write('Clean code\n')
```

In [21]:
```
!cat topics.txt
```

```
Computer Science
Programming
Clean code
```

Appending to a file

When `'w'` is specified as the file mode, a new file is created if it does not exist; if it exists the file is overwritten. We can specify the `'a'` as a file mode: then we open an existing file for appending, i.e. the new information is added to its end.

In [22]:
```
with open('topics.txt', 'a', encoding='utf-8') as f:
    f.write('Software Engineering\n')
```

In [23]:
```
!cat topics.txt
```

```
Computer Science
Programming
Clean code
Software Engineering
```

Example: Reading and writing

Suppose we have a file with 2 numbers on each line, like this:

In [24]:
```
%%writefile number_pairs.txt
1 1
10 20
1.3 2.7
```

```
Overwriting number_pairs.txt
```

Let us write a function that takes 2 filenames as arguments, reads the number pairs from the first file and writes them together with its sum into the second file.

In [25]:
```
def sum_number_pairs(infname, outfname):
    """Read data from input file, sum each row, write results to output file.

    (str, str) -> None

    infname: the name of the input file containing a pair of numbers
             separated by whitespace on each line
    outfname: the name of the output file
    """
    with open(infname, 'r', encoding='utf-8') as infile, \
         open(outfname, 'w', encoding='utf-8') as outfile:
        for pair in infile:
            pair = pair.strip()
            operands = pair.split()
            total = float(operands[0]) + float(operands[1])
            new_line = '{} {}\n'.format(pair, total)
            outfile.write(new_line)
```

When called, this function creates the required output file containing the sums.

In [26]:
```
sum_number_pairs('number_pairs.txt', 'number_pairs_with_totals.txt')
!cat number_pairs_with_totals.txt
```

```
1 1 2.0
10 20 30.0
1.3 2.7 4.0
```

Summary

- Working with paths using `os.path` module.
- Before reading from a file or writing to it, you must first `open()` it.
  - Always specify encoding: `open(filename, mode, encoding='utf-8')`.
- When you are done, you must `f.close()` the file.
- By using `with`, the file is closed automatically:

```
with open('text.txt', 'r', encoding='utf-8') as f:
      contents = f.read()
      # ... and do other things to the opened file
# When you get here, the file is not opened anymore.
```

Notebook config

Some setup follows. Ignore it.

In [27]:
```
from notebook.services.config import ConfigManager
cm = ConfigManager()
cm.update('livereveal', {
              'theme': 'Simple',
              'transition': 'slide',
              'start_slideshow_at': 'selected',
              'width': 1268,
              'height': 768,
              'minScale': 1.0
})
```

Out[27]:
```
{'height': 768,
 'minScale': 1.0,
 'start_slideshow_at': 'selected',
 'theme': 'Simple',
 'transition': 'slide',
 'width': 1268}
```

In [28]:
```
%%HTML
<style>
.reveal #notebook-container { width: 90% !important; }
.CodeMirror { max-width: 100% !important; }
pre, code, .CodeMirror-code, .reveal pre, .reveal code {
    font-family: "Consolas", "Source Code Pro", "Courier New", Courier, monospace;
}
pre, code, .CodeMirror-code {
    font-size: inherit !important;
}
.reveal .code_cell {
    font-size: 130% !important;
    line-height: 130% !important;
}
</style>
```