Programming with Python

Stefan Güttel, guettel.com (http://guettel.com)

Contents:

Files

Consider the following problem:

> Write a program that maintains students' scores in this course. The program has to allow adding students, editing their data (adding scores, for example), sorting and filtering the data, export to the web, etc.

This is a pretty standard request (obviously, written in a very abbreviated fashion). But inputting all the students' info each time we want to do anything with it is just not feasible: we need to save the data, using a more permanent medium than the working memory. This is usually a disk, and this is where the files come into play.

Where to save the data?

Traditionally, we save the data directly to files. We open them, write to them, and close them.

However, there are requests that go beyond just saving data to files. For example, Google cannot fit the list of all the web pages on the internet on a single computer, let alone in one file. Similarly, Facebook, Twitter, YouTube, and many other big services have simply too much data to fit them to a single file.

In such cases, *databases* are used. At the lowest level, these are still files, but we do not access them as such. Instead, specialized programs and modules -- so called *database engines* -- are used to access this data. They have their specific ways of use, which fall outside of the scope of this course.

More interested students are welcome to read more on the subject themselves at the various sources available on the internet. A good and fairly short introduction is Python Programming/Databases (http://en.wikibooks.org/wiki/Python_Programming/Databases), whith examples in some widely used database systems.

Note that SQLite (http://www.sqlite.org/) writes its data in a single file, as it is meant for smaller applications (for example, Chrome and Firefox are using it). Other systems usually work with several files per database and, more importantly, require a separate installation of the database engine. In other words, MySQL (http://www.mysql.com/)/MariaDB (https://mariadb.org/) (almost the same engine) will not work with just Python installed. In order to use them, you also need their respective database engines.

In the rest of this lecture, we don't work with databases. Instead, we focus on direct file access.

Caching

Compared to the working memory (RAM), disks are very slow. For that and some technical reasons, the data is rarely saved directly to a disk at the same moment that a program tells Python to save it. Instead, the operating system will store the data in a *buffer* (a part of the memory reserved for this purpose), postponing the actual disk writing until an opportune moment (usually until enough data is sent for saving).

This process is called *caching* (it is pronounced like *cashing*, not ~~*catching*~~) and it significantly improves computer's performance, while also saving on hardware wearing out, but it can also result in the loss of data.

The process of actually writing the data to a disk is called *flushing* the buffer.

Types of files

While all the files boil down to bytes and bits ("zeros and ones", just like the memory), the way we use them distinguishes them into two categories:

- *Text files* are ment for saving text and they are almost always human-readable. Examples include $LaTeX$ documents, source codes (for example, Python programs), HTML pages (including style and script files), and anything else made with text editors like Notepad (http://en.wikipedia.org/wiki/Notepad_%28software%29), Notepad++ (http://notepad-plus-plus.org/), Spyder (http://docs.continuum.io/anaconda/ide_integration.html#spyder), etc.
  Note that, generally, this does not include documents written with Microsoft Word and other text processors.
  For text files, the buffer is usually flushed whenever a new line is started.
- *Binary files* are closer to the way the data is stored in the computer's working memory. They are typically used for various media files (images, movies, music, etc), data collections (for example, ZIP, RAR, and similar archives), and some formated documents (for example, older versions of Word documents).

Generally speaking, a file created by some program written in some programming language and run on some computer system can be read by other programs, maybe written in some other programming language and/or run on a different computer system. However, some problems may occur between different programs on same or different systems.

Text files

Computers don't work with text, as everything inside a computer's memory is a number. When a computer has to display some text or write it to a file, it has to convert it to a proper format. The rules for these conversions are called *character encodings* (http://en.wikipedia.org/wiki/Character_encoding), as we have already mentioned in the first lecture.

Luckily, as long as the files are written and read in the same manner, there will be no problems. However, should your text get messed up, it is probably an encodings issue.

Another thing that differs between different operating systems is the marking of the new line. Each of the well established systems (Linux/Unix, Mac, Windows) has its own. However, many editors and all modern languages recognize these properly, as long as the files are opened as text files and not binary ones.

From a programmer's point of view, a text is written to a file, and it is read when needed. The text $\leftrightarrow$ bytes (numbers) conversion is done behind the scenes, and we need not worry about it.

Binary files

Binary files are more straightforward, but they may easily not be compatible between different systems.

Given that the text files are readable from text editors (including Spyder), we shall concentrate on them. The students who want to learn how to work with binary files can read about the `io.RawIOBase` (https://docs.python.org/3/library/io.html#io.RawIOBase) class which provides ways to directly manipulate files in a binary mode, or about bytes and `bytearray` operations (https://docs.python.org/3/library/stdtypes.html#bytes-methods) that make it possible to use file object's `read` and `write` methods. However, such a direct manipulation is rare in Python and it is far more likely that you'll use the binary mode with some specialized module.

Working with files

Whenever we need to work with files, we have three basic steps:

- **Opening a file** tells Python which file are we going to work with (file path and name) and how (read/write and are we using it as a text or as a binary file). If a file doesn't exist, it can be created or an exception may be raised, depending on how are we opening it.
- **Reading from or writing to a file** can be done only on an open, existing file.
- **Closing a file** tells Python that we are done with it, so it can "tidy up". Among other things, this includes flushing the buffer, which makes closing an important part of the process, especially when writing to files. Luckily, Python can do it automatically, as we shall soon see.

Let us see how a typical file read operation works in a traditional way (this is pretty much the same in many modern languages):

In [1]:

```
username = input("What's your name? ")
f = open("name.txt", mode="wt", encoding="utf8")
f.write(username)
f.close()
```

```
What's your name? Guido van Rossum
```

The next line is just a way to display the contents of the file `"name.txt"`. It is a feature of IPython Notebook interface (and Unix/Linux and Mac terminals), but it is not a part of Python itself (i.e., you cannot use this in your programs!).

In [2]:

```
cat name.txt
```

```
Guido van Rossum
```

Step by step

Opening a file

The first thing we need to do when working with a file is to open it:

```
f = open("name.txt", mode="wt", encoding="utf8")
```

The parameters are:

- The name of the file, here given as a string constant `"name.txt"`, but it can also be any string variable or expression. Apart from the file name, it can also contain an absolute or a relative path (http://en.wikipedia.org/wiki/Path_%28computing%29). For example,
  - a relative path: `"a/subdirectory/of/the/current/directory"`
  - a relative path: `"../a/subdirectory/of/the/current/directorys/parrent"`
  - an absolute path: `"/a/directory/in/the/root/of/the/filesystem"`
    On Windows, absolute paths begin with a drive letter (for example, `"C:"`).
- The `mode` argument defines how we access the file. More on this below.
- The `encoding` parameter defines the character encoding (http://en.wikipedia.org/wiki/Character_encoding) (hence, it should only be used with text files).
  It is not mandatory, but if left unspecified it will depend on the platform, which may cause incompatibilities when the files are created and read on different operating systems (even by the same program!).

Among those, only the file name is truly mandatory, but do use all three to prevent possible problems (for example, your coursework will not be marked on a Windows computer).

The variable `f` is called a *file object* and it contains everything that Python needs to work with the file.
Nowhere, except when opening the file, we refer to it by its name!

After the file is opened, variable `f` keeps track of the its current position in the file, which determines where the next read or write operation will occur.

File modes

These are the available file modes (taken from the documentation of the `open` function (https://docs.python.org/3/library/functions.html#open)):

Character	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing if the file already exists
'a'	open for writing, appending to the end of the file if it exists

Only one of these may be used. Additionally, we can add one of the following:

Character	Meaning
'b'	binary mode
't'	text mode (default)

So, the mode `"wt"` means:

> Open the file for **w**riting in **t**ext mode. If the file already exists, it will be truncated (i.e., its contents will be deleted). If it doesn't, it will be created.

Similarly, `"rb"` means:

> Open the file for **r**eading in **b**inary mode. If it doesn't exist, a `FileNotFoundError` exception is raised.

The mode `"a"` works like `"w"`, except that it doesn't truncate the file if it exists, and its starting position is at the end of the file.

One more character may be added: a plus character means "open for both reading and writing". For example, `"wt+"` means

> Create a new file or clear the existing one, in text mode, but open it for reading, as well as writing,

unlike `"wt"` which would mean the same, but without the ability to use the read operations.

Writing to a file

To write to the file `f` (both in text and in binary mode) that is open for writing, we use

```
f.write(data)
```

where `data` is the variable, expression, or a constant whose value we want to save. Unlike `print`, `f.write` will not add any extra characters.

Reading from a file

If the file is open for reading, we can fetch the data using `f.read`:

- `f.read()` reads the whole file from the current position to the end of the file (also called EOF),
- `f.read(b)` reads `b` bytes of the file.

For text files it is usually more convenient to use `f.readline()` which reads the data from the current position in the file to the end of that line (also called EOL). The function returns a string, with the newline character `"\n"` at the end (unless the read line is the last line in the file and it didn't end with a newline character).

To read the whole text file line by line, you can use `for` loop with the file object:

```
for line in f:
    print(line)
```

prints file text to the screen, line by line. This is a very readable piece of code, as well as a fast and memory efficient operation.

Neither `read` nor `readline` check that your computer actually has enough memory to perform the task.

Closing the file

Once we are done working with a file, we close it:

```
f.close()
```

To see why closing is especially important when writing, let us observe what happens with the file, line by line, as we execute the above code.

First, we load the data and open the file for writing:

In [3]:

```
username = input("What's your name? ")
f = open("name.txt", mode="wt", encoding="utf8")
```

```
What's your name? Guido van Rossum
```

The file is now empty (i.e., its old contents is lost):

In [4]:

```
cat name.txt
```

We now write our data to the file:

In [5]:

```
f.write(username)
```

Out[5]:

```
16
```

Note: the `write` function returns the number of characters written.

However, the file is still empty:

In [6]:

```
cat name.txt
```

But, after we close it:

In [7]:

```
f.close()
```

the cache is flushed and the file finally contains the data:

In [8]:

```
cat name.txt
```

```
Guido van Rossum
```

The Pythonic way

Python has a nice way of preventing the loss of data due to unclosed files. The following code does the same thing as the above one, but without a close statement (which is done automatically):

In [9]:

```
username = input("What's your name? ")
with open("name.txt", mode="wt", encoding="utf8") as f:
    f.write(username)
```

```
What's your name? Guido van Rossum
```

As before, we can easily check that the data was properly saved:

In [10]:

```
cat name.txt
```

```
Guido van Rossum
```

The difference between the traditional approach and the Pythonic one is that the latter uses a statement `with`. Once the flow of control exits the block belonging to `with`, the file is automatically closed.

This is done even if the `with` block is exited with a `break` statement (in which case, `with` itself needs to be in a loop), a `return` statement (in which case, `with` itself needs to be in a function), or an exception.

The downside is that all the work on the file needs to be done inside the block belonging to `with`. This is rarely a problem, but if there is a need for the file object to be used throughout the program (so, wider than a single block), one can always use the traditional approach.

Note that the following works fine because the function `save_name` is called inside the `with` block (i.e., before the flow of control exits it and closes the file).

In [11]:

```
def save_name(f, name):
    """
    Save name `name`, along with a decorative description, to a file
    opened for writing and accessed via a file object `f`.
    """
    f.write('Name: "{}"'.format(name))

username = input("What's your name? ")
with open("name.txt", mode="wt", encoding="utf8") as f:
    save_name(f, username)
```

```
What's your name? Guido van Rossum
```

In [12]:

```
cat name.txt
```

```
Name: "Guido van Rossum"
```

As we shall soon see, `with` can accept more than one file argument.

A more lively example

Problem. A text file `Average-Prices-SA.csv` (http://publicdata.landregistry.gov.uk/market-trend-data/house-price-index-data/Average-Prices-SA.csv) contains the average house prices throughout the UK, from 1995 until now, seasonally adjusted. The data is organized one record per line, each record having the following fields separated by commas:

1. "Date", in the format `yyyy-mm-01` (this is monthly data, so the day of the month is always the 1st),
2. "Name" of a category or a region, which is any string (obviously, not containing commas),
3. "Average_Price_SA", a real number containing the price.

The first line contains names of these fields and should not be treated as a part of the data.

Here are a first few lines from the file:

```
Date,Name,Average_Price_SA
1995-01-01,Barking And Dagenham,70837.428643366453
1995-01-01,Barnet,99572.784726549682
1995-01-01,Barnsley,49312.007625642422
1995-01-01,Bath And North East Somerset,68371.565466743603
```

Write two programs:

- One has to load two positive integers f (from) and t (to), and a string `oname`. It then copies the header (the first line) and lines $f, f+1, \ldots, t$ to a new file with the name `oname`.
- The other one has to input a name and print the average seasonally adjusted house price for all the records for that name, rounded to two digits.

The data was produced by Land Registry © Crown copyright 2014. and downloaded from this page (https://www.gov.uk/government/statistical-data-sets/house-price-index-background-tables).

Copy (part of) the file

The first problem has nothing to do with the data itself, apart from the fact that we start counting from the second line and the first one is copied regardless of the input.

In [13]:

```
oname = input("Output file name: ")
f = int(input("From line: "))
t = int(input("To line: "))
# Open both files
with open("Average-Prices-SA.csv", mode="rt", encoding="utf8") as ifile, \
     open(oname, mode="wt", encoding="utf8") as ofile:
    # Copy the first line
    ofile.write(ifile.readline())
    # Read the first `f-1` lines, as we don't want to copy these
    for _ in range(f-1):
        if ifile.readline() == "":
            break
    # Copy the following `t-f+1` lines:
    for k in range(t-f+1):
        line = ifile.readline()
        if line == "":
            break
        ofile.write(line)
```

```
Output file name: new.csv
From line: 7
To line: 17
```

In [14]:

```
cat new.csv
```

```
Date,Name,Average_Price_SA
1995-01-01,Birmingham,51373.387921890659
1995-01-01,Blackburn With Darwen,49393.909668908142
1995-01-01,Blackpool,48692.572354278418
1995-01-01,Blaenau Gwent,38708.137608856559
1995-01-01,Bolton,50753.829668958962
1995-01-01,Bournemouth,59917.997546378996
1995-01-01,Bracknell Forest,79517.508984866363
1995-01-01,Bradford,55868.08116983033
1995-01-01,Brent,73320.238836327509
1995-01-01,Bridgend,57239.059544701122
1995-01-01,Brighton And Hove,53047.456436018481
```

In the above code we check if `readline` calls have returned an empty string `""`. This happens only if we have reached the end of the file (as we stated before, the lines maintain their newline character, so even an empty line will be read as `"\n"` and not an empty string).

Notice the backslash character `\` at the end of the first `with` line. Normally, `with` statements cannot be broken at the comma, so a backslash is used to achieve that. It's meaning in Python (and some other languages) is "this statement continues in the next line".

Get the average price for a given name

To solve this problem, we need to read the file line by line (skipping the first one, i.e., the header), split each line into components, check the name, and -- if it is the same as the given one -- add the corresponding value to the sum of values and increase the counter (which will then be used for the division when computing the average value).

Note that we shall use a case insensitive string comparisons. To do that, we shall remember the lowercase version of the name we are searching for, which we shall then compare to the lowercase versions of the names in the file.

In [15]:

```
name = input("Category name: ")
name_lower = name.lower()
with open("Average-Prices-SA.csv", mode="rt", encoding="utf8") as f:
    # Skip the first line
    f.readline()
    # Initialize the sum and the counter
    price_sum = 0
    row_count = 0
    for line in f:
        # Try...except block will catch errors with splitting the lines
        # that have a wrong number of commas or convertin non-float prices
        # to floats
        try:
            (_, cat_name, price) = line.split(",")
            if cat_name.lower() == name_lower:
                # print(price.strip())  # uncomment this to check the program
                price_sum += float(price)
                row_count += 1
        except Exception:
            # Whatever the error it is, we ignore it
            pass
if row_count:
    print('There are {} records matching the name "{}", \
with the average price {:.2f} GBP.'.format(row_count, name, price_sum/row_count))
else:
    print('There are no records regarding the category "{}".'.format(name))
```

```
Category name: London
There are 240 records matching the name "London", with the average price 25
2982.58 GBP.
```

CSV

The format of a text file described above is called *Comma Separated Values*, or *CSV* in short. It is somewhat more complex than our data above (it can work with separators other than commas, and the separators themselves can be parts of the values if quotation marks are included), and it is used to save data tables in text files.

All spreadsheet programs (OpenOffice Calc, Google Sheets, Microsoft Excel,...) can export data to CSV files and import it from them. The export can be somewhat lacking, due to the limitations of the format itself (for example, the formats and the formulas are lost).

Luckily, we don't have to parse such files by hand, as we have done in the previous example. Instead, it is better to use the csv module (https://docs.python.org/3/library/csv.html), like this:

In [16]:

```
import csv

name = input("Category name: ")
name_lower = name.lower()
with open("Average-Prices-SA.csv", mode="rt", encoding="utf8", newline="") as f:
    # Create a CSV reader object
    prices_reader = csv.reader(f, delimiter=",")
    # Initialize the sum and the counter
    price_sum = 0
    row_count = 0
    for row in prices_reader:
        # Try...except block will catch errors like the improper
        # number of columns in the table or non-float prices
        try:
            (_, cat_name, price) = row
            if cat_name.lower() == name_lower:
                # print(price.strip())  # uncomment this to check the program
                price_sum += float(price)
                row_count += 1
        except Exception:
            # Whatever the error it is, we ignore it
            pass
if row_count:
    print('There are {} records matching the name "{}", \
with the average price {:.2f} GBP.'.format(row_count, name, price_sum/row_count))
else:
    print('There are no records regarding the category "{}".'.format(name))
```

```
Category name: London
There are 240 records matching the name "London", with the average price 25
2982.58 GBP.
```

This may not seem much simpler than the previous code, but this code is much more robust, it is easier to customize, and it handles the header automatically.

Note that the `csv` module also supports writing the data to CSV files, as well as various advanced customizations that allow the programmers to easily adapt their programs to CSV files created by different well know systems (so called *dialects*).

The supported dialects list is easy to find:

In [17]:

```
import csv
print(csv.list_dialects())
```

```
['excel', 'unix', 'excel-tab']
```

It is equally easy to define your own dialects. More details can be found in the documentation of the `csv` module (https://docs.python.org/3/library/csv.html).

Other text formats

While CSV is a very simple format for storing table data in text files, there are many other formats that store various other kinds of data. The two most common general purpose ones are XML and JSON, both of which are supported by standard Python modules (for XML, see here (https://docs.python.org/3/library/xml.html), and JSON is supported by the `json` module (https://docs.python.org/3.3/library/json.html)).

Pickles

Using binary files directly is beyond the scope of this course. However, there is a Python-specific approach: `pickle` module (https://docs.python.org/3/library/pickle.html) that allows us to save various values and read them later on. It is not supported by other languages, so it can only be used between the programs written in Python.

Let us save one integer and one float to a pickle file:

In [18]:

```
import pickle
with open("binary.dat", mode="wb") as f:
    pickle.dump(17, f)
    pickle.dump(17.19, f)
```

Now, let us read them from the same file and print them to the screen:

In [19]:

```
import pickle
with open("binary.dat", mode="rb") as f:
    x = pickle.load(f)
    y = pickle.load(f)
    print("x = {} (type: {})".format(x, type(x)))
    print("y = {} (type: {})".format(y, type(y)))
```

```
x = 17 (type: <class 'int'>)
y = 17.19 (type: <class 'float'>)
```

Let us also print the position of the file object in the file as we read the numbers:

In [20]:

```
import pickle
with open("binary.dat", mode="rb") as f:
    print("Position before the loads: ", f.tell())
    x = pickle.load(f)
    pos_between_x_and_y = f.tell()
    print("Position between the loads:", pos_between_x_and_y)
    y = pickle.load(f)
    print("Position after the loads:  ", f.tell())
    print("x = {} (type: {})".format(x, type(x)))
    print("y = {} (type: {})".format(y, type(y)))
```

```
Position before the loads:  0
Position between the loads: 5
Position after the loads:   17
x = 17 (type: <class 'int'>)
y = 17.19 (type: <class 'float'>)
```

The `ftell` (https://docs.python.org/3/library/io.html#io.IOBase.tell) returns the current position in the file (i.e., how much bytes have we read to get where we are). This can be used to move around the file with its counterpart, `seek` (https://docs.python.org/3/library/io.html#io.IOBase.seek). For example:

In [21]:

```
import pickle
with open("binary.dat", mode="rb") as f:
    f.seek(pos_between_x_and_y)
    val = pickle.load(f)
    print("Loaded value: {} (type: {})".format(val, type(val)))
```

```
Loaded value: 17.19 (type: <class 'float'>)
```

`seek` is rarely given an exact number as the position. Instead, it is customary to give it a position captured earlier with `tell`.

The exceptions are moving to the beginning and to the end of the file. For example,

In [22]:

```
import pickle
import io
with open("binary.dat", mode="rb+") as f:
    # Opened for reading and writing
    # Read x
    x = pickle.load(f)
    print("x:", x)
    # Compute new value
    val = 2*x+1
    # Move to the beginning of the file
    f.seek(0)
    # Overwrite `x` with the new value
    pickle.dump(val, f)
    # Move to the end
    f.seek(0, io.SEEK_END)
    print("Position at the end:", f.tell())
    # Write old `x` to the end of the file
    pickle.dump(x, f)
    print("Position after writing:", f.tell())
```

```
x: 17
Position at the end: 17
Position after writing: 22
```

If we execute the above code repeatedly, we will see that the positions are growing, as we are adding data to the file.

Notice that overwriting `x` with anything bigger than `int` will overwrite (part of) the value after it, so this kind of writing in the middle of the file has to be done with the utmost care.

Note: The functions from `pickle` module do quite a bit of work, preserving the types of the data, along with its values. This does not happen when binary files are used directly (for example, in more traditional programming languages).