FILE MANAGEMENT

FILES IN GENERAL

It's hard to find anyone in the 21st century, who doesn't know what a file is. When we say file, we mean of course, a file on a computer. There may be people who don't know anymore the "container", like a cabinet or a folder, for keeping papers archived in a convenient order. A file on a computer is the modern counterpart of this. It is a collection of information, which can be accessed and used by a computer program. Usually, a file resides on a durable storage. Durable means that the data is persistent, i.e. it can be used by other programs after the program which has created or manipulated it, has terminated.

The term file management in the context of computers refers to the manipulation of data in a file or files and documents on a computer. Though everybody has an understanding of the term file, we present a formal definition anyway:

A file or a computer file is a chunk of logically related data or information which can be used by computer programs. Usually a file is kept on a permanent storage media, e.g. a hard drive disk. A unique name and path is used by human users or in programs or scripts to access a file for reading and modification purposes.

The term "file" - as we have described it in the previous paragraph - appeared in the history of computers very early. Usage can be tracked down to the year 1952, when punch cards where used.

A programming language without the capability to store and retrieve previously stored information would be hardly useful.

The most basic tasks involved in file manipulation are reading data from files and writing or appending data to files.

READING AND WRITING FILES IN PYTHON

The syntax for reading and writing files in Python is similar to programming languages like C, C++, Java, Perl, and others but a lot easier to handle.

We will start with writing a file. We have a string which contains part of the definition of a general file from Wikipedia:

```
definition = """
A computer file is a computer resource for recording data discretely
in a
computer storage device. Just as words can be written
to paper, so can information be written to a computer
file. Files can be edited and transferred through the
internet on that particular computer system."""
```

We will write this into a file with the name `file_definition.txt`:

```
open("file_definition.txt", "w").write(definition)
```

Output: `283`

If you check in your file browser, you will see a file with this name. The file will look like this: file_definition.txt

We successfully created and have written to a text file. Now, we want to see how to read this file from Python. We can read the whole text file into one string, as you can see in the following code:

```
text = open("file_definition.txt").read()
```

If you call `print(text)`, you will see the text from above again.

Reading in a text file in one string object is okay, as long as the file is not too large. If a file is large, wwe can read in the file line by line. We demonstrate how this can be achieved in the following example with a small file:

```
with open("ad_lesbiam.txt", "r") as fh:
    for line in fh:
        print(line.strip())
```

```
V. ad Lesbiam

VIVAMUS mea Lesbia, atque amemus,
rumoresque senum severiorum
omnes unius aestimemus assis!
soles occidere et redire possunt:
nobis cum semel occidit breuis lux,
nox est perpetua una dormienda.
da mi basia mille, deinde centum,
dein mille altera, dein secunda centum,
deinde usque altera mille, deinde centum.
dein, cum milia multa fecerimus,
conturbabimus illa, ne sciamus,
aut ne quis malus inuidere possit,
cum tantum sciat esse basiorum.
(GAIUS VALERIUS CATULLUS)
```

Some people don't use the with statement to read or write files. This is not a good idea. The code above without with looks like this:

```
fh = open("ad_lesbiam.txt")
for line in fh:
        print(line.strip())
fh.close()
```

A striking difference between both implementation consists in the usage of `close`. If we use `with`, we do not have to explicitly close the file. The file will be closed automatically, when the with blocks ends. Without `with`, we have to explicitly close the file, like in our second example with `fh.close()`. There is a more important difference between them: If an exception occurs inside of the `with` block, the file will be closed. If an exception occurs in the variant without `with` before the `close`, the file will not be closed. This means, you should alwawys use the `with` statement.

We saw already how to write into a file with "write". The following code is an example, in which we show how to read in from one file line by line, change the lines and write the changed content into another file. The file can be downloaded: pythonista_and_python.txt:

```
with open("pythonista_and_python.txt") as infile:
    with open("python_newbie_and_the_guru.txt", "w") as outfile:
        for line in infile:
            line = line.replace("Pythonista", "Python newbie")
            line = line.replace("Python snake", "Python guru")
            print(line.rstrip())
            # write the line into the file:
            outfile.write(line)
```

```
A blue Python newbie, green behind the ears, went to Python
ia.
She wanted to visit the famous wise green Python guru.
She wanted to ask her about the white way to avoid the blac
k.
The bright path to program in a yellow, green, or blue styl
e.
The green Python turned red, when she addressed her.
The Python newbie turned yellow in turn.
After a long but not endless loop  the wise Python uttered:
"The rainbow!"
```

As we have already mentioned: If a file is not to large and if we have to do replacements like we did in the previous example, we wouldn't read in and write out the file line by line. It is much better to use the `read` method, which returns a string containing the complete content of the file, including the carriage returns and line feeds. We can apply the changes to this string and save it into the new file. Working like this, there is no need for a `with` construct, because there will be no reference to the file, i.e. it will be immediately deleted afeter reading and writing:

```
txt = open("pythonista_and_python.txt").read()
txt = txt.replace("Pythonista", "Python newbie")
txt = txt.replace("Python snake", "Python guru")
open("python_newbie_and_the_guru.txt", "w").write(txt)
;
```

Output: ,,

RESETTING THE FILES CURRENT POSITION

It's possible to set - or reset - a file's position to a certain position, also called the offset. To do this, we use the method `seek`. The parameter of seek determines the offset which we want to set the current position to. To work with `seek`, we will often need the method `tell` which "tells" us the current position. When we have just opened a file, it will be zero. Before we demonstrate the way of working of both `seek` and `tell`, we create a simple file on which we will perform our commands:

```
open("small_text.txt", "w").write("brown is her favorite colour")
;
```

Output: ,,

The method `tell` returns the current stream position, i.e. the position where we will continue, when we use a "read", "readline" or so on:

```
fh = open("small_text.txt")
fh.tell()
```

Output: 0

Zero tells us that we are positioned at the first character of the file.

We will read now the next five characters of the file:

```
fh.read(5)
```

Output: 'brown'

Using `tell` again, shows that we are located at position 5:

```
fh.tell()
```

Output: 5

Using `read` without parameters will read the remainder of the file starting from this position:

```
fh.read()
```

Output: ' is her favorite colour'

Using `tell` again, tells us about the position after the last character of the file. This number corresponds to the number of characters of the file!

```
fh.tell()
```

Output:
```
28
```

With `seek` we can move the position to an arbitrary place in the file. The method `seek` takes two parameters:

```
fh.seek(offset, startpoint_for_offset)
```

where fh is the file pointer, we are working with. The parameter `offset` specifies how many positions the pointer will be moved. The question is from which position should the pointer be moved. This position is specified by the second parameter `startpoint_for_offset`. It can have the follwoing values:

```
0: reference point is the beginning of the file
1: reference point is the current file position
2: reference point is the end of the file
```

if the `startpoint_for_offset` parameter is not given, it defaults to 0.

WARNING: The values 1 and 2 for the second parameter work only, if the file has been opened for binary reading. We will cover this later!

The following examples, use the default behaviour:

```
fh.seek(13)
print(fh.tell())    # just to show you, what seek did!
fh.read()           # reading the remainder of the file
```

```
13
```

Output:
```
'favorite colour'
```

It is also possible to move the position relative to the current position. If we want to move `k` characters to the right, we can just set the argument of `seek` to `fh.tell() + k`

```
k = 6
fh.seek(5)     # setting the position to 5
fh.seek(fh.tell() + k)    #  moving k positions to the right
print("We are now at position: ", fh.tell())
```

```
We are now at position:  11
```

`seek` doesn't like negative arguments for the position. On the other hand it doesn't matter, if the value for the position is larger than the length of the file. We define a function in the following, which will set the position to zero, if a negative value is applied. As there is no efficient way to check the length of a file and because it doesn't matter, if the position is greater than the length of the file, we will keep possible values greater than the length of a file.

```
def relative_seek(fp, k):
    """ rel_seek moves the position of the file pointer k characters
to
    the left (k<0) or right (k>0)
    """
    position = fp.tell() + k
    if position < 0:
        position = 0
    fh.seek(position)


with open("small_text.txt") as fh:
    print(fh.tell())
    relative_seek(fh, 7)
    print(fh.tell())
    relative_seek(fh, -5)
    print(fh.tell())
    relative_seek(fh, -10)
    print(fh.tell())
```

```
0
7
2
0
```

You might have thought, when we wrote the function `relative_seek` why do we not use the second parameter of `seek`. After all the help file says "1 -- current stream position;". What the help file doesn't say is the fact that `seek` needs a file pointer opened with "br" (binary read), if the second parameter is set to 1 or 2. We show this in the next subchapter.

BINARY READ

So far we have only used the first parameter of `open`, i.e. the filename. The second parameter is optional and is set to "r" (read) by default. "r" means that the file is read in text mode. In text mode, if encoding (another parameter of open) is not specified the encoding used is platform dependent: locale.getpreferredencoding(False) is called to get the current locale encoding.

The second parameter specifies the mode of access to the file or in other words the mode in which the file is opened. Files opened in binary mode (appending 'b' to the mode argument) return contents as bytes objects without any decoding.

We will demonstrate this in the following example. To demonstrate the different effects we need a string which uses characters which are not included in standard ASCII. This is why we use a Turkish text, because it uses many special characters and Umlaute. the English translation means "See you, I'll come tomorrow.".

We will write a file with the Turkish text "Görüşürüz, yarın geleceğim.":

```
txt = "Görüşürüz, yarın geleceğim."
number_of_chars_written = open("see_you_tomorrow.txt", "w").write(tx
t)
```

We will read in this files in text mode and binary mode to demonstrate the differences:

```
text = open("see_you_tomorrow.txt", "r").read()
print("text mode: ", text)
text_binary = open("see_you_tomorrow.txt", "rb").read()
print("binary mode: ", text_binary)
```

```
text mode:  Görüşürüz, yarın geleceğim.
binary mode:  b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbc
z, yar\xc4\xb1n gelece\xc4\x9fim.'
```

In binary mode, the characters which are not plain ASCII like "ö", "ü", "ş", "ğ" and "ı" are represented by more than one byte. In our case by two characters. 14 bytes are needed for "görüşürüz":

```
text_binary[:14]
```

Output: `b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'`

"ö" for example consists of the two bytes "\xc3" and "\xb6".

```
text[:9]
```

Output: `'Görüşürüz'`

There are two ways to turn a byte string into a string again:

```
t = text_binary.decode("utf-8")
print(t)
t2 = str(text_binary, "utf-8")
print(t2)
```

```
Görüşürüz, yarın geleceğim.
Görüşürüz, yarın geleceğim.
```

It is possible to use the values "1" and "2" for the second parameter of `seek`, if we open a file in binary format:

```
with open("see_you_tomorrow.txt", "rb") as fh:
    x = fh.read(14)
    print(x)
    # move 5 bytes to the right from the current position:
    fh.seek(5, 1)
    x = fh.read(3)
    print(x)
    print(str(x, "utf-8"))
    # let's move to the 8th byte from the right side of the byte str
ing:
    fh.seek(-8, 2)
    x = fh.read(5)
    print(x)
    print(str(x, "utf-8"))
```

```
b'G\xc3\xb6r\xc3\xbc\xc5\x9f\xc3\xbcr\xc3\xbcz'
b'\xc4\xb1n'
ın
b'ece\xc4\x9f'
eceğ
```

READ AND WRITE TO THE SAME FILE

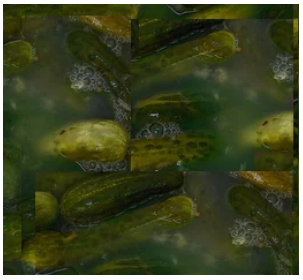
In the following example we will open a file for reading and writing at the same time. If the file doesn't exist, it will be created. If you want to open an existing file for read and write, you should better use "r+", because this will not delete the content of the file.

```
fh = open('colours.txt', 'w+')
fh.write('The colour brown')

#Go to the 12th byte in the file, counting starts with 0
fh.seek(11)
print(fh.read(5))
print(fh.tell())
fh.seek(11)
fh.write('green')
fh.seek(0)
content = fh.read()
print(content)
```

```
brown
16
The colour green
```

"HOW TO GET INTO A PICKLE"



We don't really mean what the header says. On the contrary, we want to prevent any nasty situation, like losing the data, which your Python program has calculated. So, we will show you, how you can save your data in an easy way that you or better your program can reread them at a later date again. We are "pickling" the data, so that nothing gets lost.

Python offers a module for this purpose, which is called "pickle". With the algorithms of the pickle module we can serialize and de-serialize Python object structures. "Pickling" denotes the process which converts a Python object hierarchy into a byte stream, and "unpickling" on the other hand is the inverse operation, i.e. the byte stream is converted back into an object hierarchy. What we call pickling (and unpickling) is also known as "serialization" or "flattening" a data structure.

An object can be dumped with the dump method of the pickle module:

```
pickle.dump(obj, file[,protocol, *, fix_imports=True])
```

dump() writes a pickled representation of obj to the open file object file. The optional protocol argument tells the pickler to use the given protocol:

- Protocol version 0 is the original (before Python3) human-readable (ascii) protocol and is backwards compatible with previous versions of Python.
- Protocol version 1 is the old binary format which is also compatible with previous versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.
- Protocol version 3 was introduced with Python 3.0. It has explicit support for bytes and cannot be unpickled by Python 2.x pickle modules. It's the recommended protocol of Python 3.x.

The default protocol of Python3 is 3.

If fix_imports is True and protocol is less than 3, pickle will try to map the new Python3 names to the old module names used in Python2, so that the pickle data stream is readable with Python 2.

Objects which have been dumped to a file with pickle.dump can be reread into a program by using the method pickle.load(file). pickle.load recognizes automatically, which format had been used for writing the data. A simple example:

```
import pickle

cities = ["Paris", "Dijon", "Lyon", "Strasbourg"]
fh = open("data.pkl", "bw")
pickle.dump(cities, fh)
fh.close()
```

The file data.pkl can be read in again by Python in the same or another session or by a different program:

```
import pickle
f = open("data.pkl", "rb")
villes = pickle.load(f)
print(villes)
['Paris', 'Dijon', 'Lyon', 'Strasbourg']
```

Only the objects and not their names are saved. That's why we use the assignment to villes in the previous

example, i.e. data = pickle.load(f).

In our previous example, we had pickled only one object, i.e. a list of French cities. But what about pickling multiple objects? The solution is easy: We pack the objects into another object, so we will only have to pickle one object again. We will pack two lists "programming_languages" and "python_dialects" into a list pickle_objects in the following example:

```
import pickle
fh = open("data.pkl","bw")
programming_languages = ["Python", "Perl", "C++", "Java", "Lisp"]
python_dialects = ["Jython", "IronPython", "CPython"]
pickle_object = (programming_languages, python_dialects)
pickle.dump(pickle_object,fh)
fh.close()
```

The pickled data from the previous example, - i.e. the data which we have written to the file data.pkl, - can be separated into two lists again, when we reread the data:

</pre> import pickle f = open("data.pkl","rb") languages, dialects) = pickle.load(f) print(languages, dialects) ['Python', 'Perl', 'C++', 'Java', 'Lisp'] ['Jython', 'IronPython', 'CPython'] </pre>

SHELVE MODULE

One drawback of the pickle module is that it is only capable of pickling one object at the time, which has to be unpickled in one go. Let's imagine this data object is a dictionary. It may be desirable that we don't have to save and load every time the whole dictionary, but save and load just a single value corresponding to just one key. The shelve module is the solution to this request. A "shelf" - as used in the shelve module - is a persistent, dictionary-like object. The difference with dbm databases is that the values (not the keys!) in a shelf can be essentially arbitrary Python objects -- anything that the "pickle" module can handle. This includes most class instances, recursive data types, and objects containing lots of shared sub-objects. The keys have to be strings.

The shelve module can be easily used. Actually, it is as easy as using a dictionary in Python. Before we can use a shelf object, we have to import the module. After this, we have to open a shelve object with the shelve method open. The open method opens a special shelf file for reading and writing:

</pre> import shelve s = shelve.open("MyShelve")</pre>

If the file "MyShelve" already exists, the open method will try to open it. If it isn't a shelf file, - i.e. a file which has been created with the shelve module, - we will get an error message. If the file doesn't exist, it will be created.

We can use s like an ordinary dictionary, if we use strings as keys:

```
s["street"] = "Fleet Str"
s["city"] = "London"
for key in s:
    print(key)
```

A shelf object has to be closed with the close method:

```
s.close()
```

We can use the previously created shelf file in another program or in an interactive Python session:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informatio
n.
import shelve
s = shelve.open("MyShelve")
s["street"]
'Fleet Str'
s["city"]
'London'
```

It is also possible to cast a shelf object into an "ordinary" dictionary with the dict function:

```
s
≤shelve.DbfilenameShelf object at 0xb7133dcc>
>>> dict(s)
{'city': 'London', 'street': 'Fleet Str'}
```

The following example uses more complex values for our shelf object:

```
import shelve
tele = shelve.open("MyPhoneBook")
tele["Mike"] = {"first":"Mike", "last":"Miller", "phone":"4689"}
tele["Steve"] = {"first":"Stephan", "last":"Burns", "phone":"8745"}
tele["Eve"] = {"first":"Eve", "last":"Naomi", "phone":"9069"}
tele["Eve"]["phone"]
'9069'
```

The data is persistent!

To demonstrate this once more, we reopen our MyPhoneBook:

```
$ python3
Python 3.2.3 (default, Feb 28 2014, 00:22:33)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more informatio
n.
    import shelve
    tele = shelve.open("MyPhoneBook")
    tele["Steve"]["phone"]
'8745'
```

EXERCISES

EXERCISE 1

Write a function which reads in a text from file and returns a list of the paragraphs. You may use one of the following books:

- Virginia Woolf: To the Lighthouse
- Samuel Butler: The Way of all Flash
- Herman Melville: Moby Dick
- David Herbert Lawrence: Sons and Lovers
- Daniel Defoe: The Life and Adventures of Robinson Crusoe
- James Joyce: Ulysses

EXERCISE 2

Save the following text containing city names and times as "cities_and_times.txt".

```
Chicago Sun 01:52
Columbus Sun 02:52
Riyadh  Sun 10:52
Copenhagen  Sun 08:52
Kuwait City Sun 10:52
Rome    Sun 08:52
Dallas  Sun 01:52
Salt Lake City  Sun 01:52
San Francisco Sun 00:52
Amsterdam Sun 08:52
Denver Sun 01:52
San Salvador Sun 01:52
Detroit Sun 02:52
Las Vegas Sun 00:52
Santiago    Sun 04:52
Anchorage Sat 23:52
Ankara Sun 10:52
Lisbon  Sun 07:52
São Paulo   Sun 05:52
Dubai   Sun 11:52
London  Sun 07:52
Seattle Sun 00:52
Dublin  Sun 07:52
Los Angeles Sun 00:52
Athens  Sun 09:52
Edmonton Sun 01:52
Madrid  Sun 08:52
Shanghai Sun 15:52
Atlanta Sun 02:52
Frankfurt   Sun 08:52
Singapore Sun 15:52
Auckland Sun 20:52
Halifax Sun 03:52
```

```
Melbourne Sun 18:52
Stockholm   Sun 08:52
Barcelona   Sun 08:52
Miami   Sun 02:52
Minneapolis Sun 01:52
Sydney Sun 18:52
Beirut  Sun 09:52
Helsinki    Sun 09:52
Montreal    Sun 02:52
Berlin  Sun 08:52
Houston Sun 01:52
Moscow  Sun 10:52
Indianapolis    Sun 02:52
Boston  Sun 02:52
Tokyo   Sun 16:52
Brasilia Sun 05:52
Istanbul Sun 10:52
Toronto Sun 02:52
Vancouver   Sun 00:52
Brussels    Sun 08:52
Jerusalem   Sun 09:52
New Orleans Sun 01:52
Vienna  Sun 08:52
Bucharest   Sun 09:52
Johannesburg    Sun 09:52
New York    Sun 02:52
Warsaw  Sun 08:52
Budapest    Sun 08:52
Oslo    Sun 08:52
Washington DC   Sun 02:52
Ottawa  Sun 02:52
Winnipeg    Sun 01:52
Cairo   Sun 09:52
Paris   Sun 08:52
Calgary Sun 01:52
Kathmandu   Sun 13:37
Philadelphia    Sun 02:52
Zurich  Sun 08:52
Cape Town   Sun 09:52
Phoenix Sun 00:52
Prague  Sun 08:52
Casablanca  Sun 07:52
Reykjavik   Sun 07:52
```

Each line contains the name of the city, followed by the name of the day ("Sun") and the time in the form hh:mm. Read in the file and create an alphabetically ordered list of the form

```
[('Amsterdam', 'Sun', (8, 52)), ('Anchorage', 'Sat', (23, 52)), ('Ank
ara', 'Sun', (10, 52)), ('Athens', 'Sun', (9, 52)), ('Atlanta', 'Sun
', (2, 52)), ('Auckland', 'Sun', (20, 52)), ('Barcelona', 'Sun', (8,
52)), ('Beirut', 'Sun', (9, 52)),

...

 ('Toronto', 'Sun', (2, 52)), ('Vancouver', 'Sun', (0, 52)), ('Vienna
', 'Sun', (8, 52)), ('Warsaw', 'Sun', (8, 52)), ('Washington DC', 'Su
```

```
n', (2, 52)), ('Winnipeg', 'Sun', (1, 52)), ('Zurich', 'Sun', (8, 5
2))]
```

Finally, the list should be dumped for later usage with the pickle module. We will use this list in our chapter on Numpy dtype.

SOLUTIONS

SOLUTION 1

```
def text2paragraphs(filename, min_size=1):
    """ A text contained in the file 'filename' will be read
    and chopped into paragraphs.
    Paragraphs with a string length less than min_size will be ignor
ed.
    A list of paragraph strings will be returned"""

    txt = open(filename).read()
    paragraphs = [para for para in txt.split("\n\n") if len(para) >
min_size]
    return paragraphs

paragraphs = text2paragraphs("books/to_the_lighthouse_woolf.txt", mi
n_size=100)

for i in range(10, 14):
    print(paragraphs[i])
```

```
“I should think there would be no one to talk to in Manches
ter,” she
replied at random. Mr. Fortescue had been observing her for
a moment or
two, as novelists are inclined to observe, and at this rema
rk he smiled,
and made it the text for a little further speculation.
“In spite of a slight tendency to exaggeration, Katharine d
ecidedly
hits the mark,” he said, and lying back in his chair, with
his opaque
contemplative eyes fixed on the ceiling, and the tips of hi
s fingers
pressed together, he depicted, first the horrors of the str
eets of
Manchester, and then the bare, immense moors on the outskir
ts of the
town, and then the scrubby little house in which the girl w
ould live,
and then the professors and the miserable young students de
voted to the
more strenuous works of our younger dramatists, who would v
isit her,
and how her appearance would change by degrees, and how she
would fly to
London, and how Katharine would have to lead her about, as
one leads an
eager dog on a chain, past rows of clamorous butchers’ shop
s, poor dear
creature.
“Oh, Mr. Fortescue,” exclaimed Mrs. Hilbery, as he finishe
d, “I had just
written to say how I envied her! I was thinking of the big
gardens and
the dear old ladies in mittens, who read nothing but the “S
pectator,”
 and snuff the candles. Have they ALL disappeared? I told h
er she would
find the nice things of London without the horrid streets t
hat depress
one so.”
“There is the University,” said the thin gentleman, who had
previously
insisted upon the existence of people knowing Persian.
```

SOLUTION 2

```
import pickle

lines = open("cities_and_times.txt").readlines()
lines.sort()

cities = []
for line  in lines:
    *city, day, time = line.split()
     hours, minutes = time.split(":")
     cities.append((" ".join(city), day, (int(hours), int(minutes))
))

fh = open("cities_and_times.pkl", "bw")
pickle.dump(cities, fh)
```

City names can consist of multiple words like "Salt Lake City". That is why we have to use the asterisk in the line, in which we split a line. So city will be a list with the words of the city, e.g. ["Salt", "Lake", "City"]. " ".join(city) turns such a list into a "proper" string with the city name, i.e. in our example "Salt Lake City".