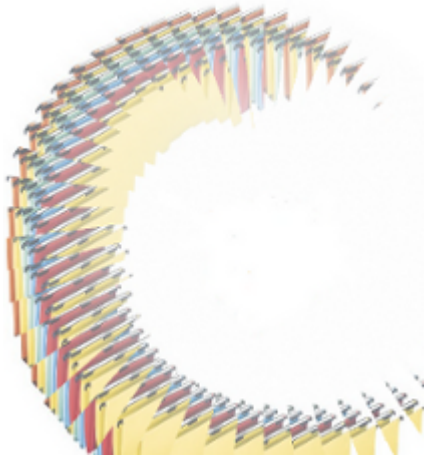
FILE MANAGEMENT

FILES IN GENERAL



It's hard to find anyone in the 21st Century, who doesn't know what a file is. If 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.

In our first example we want to show how to read data from a file. The way of telling Python that we want to read from a file is to use the open function. The first parameter is the name of the file we want

to read and with the second parameter, assigned to the value "r", we state that we want to read from the file:

```
fobj = open("ad_lesbiam.txt", "r")
```

The "r" is optional. An open() command with just a file name is opened for reading per default. The open() function returns a file object, which offers attributes and methods.

```
fobj = open("ad_lesbiam.txt")
```

After we have finished working with a file, we have to close it again by using the file object method close():

```
fobj.close()
```

Now we want to finally open and read a file. The method rstrip() in the following example is used to strip off whitespaces (newlines included) from the right side of the string "line":

```
fobj = open("ad_lesbiam.txt")
for line in fobj:
    print(line.rstrip())
fobj.close()
```

If we save this script and call it, we get the following output, provided that the text file "ad_lesbiam.txt" is available:

```
$ python file_read.py
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)
```

By the way, the poem above is a love poem of the ancient Roman poet Catull, who was hopelessly in love with a woman called Lesbia.

WRITE INTO A FILE

Writing to a file is as easy as reading from a file. To open a file for writing we set the second parameter to "w" instead of "r". To actually write the data into this file, we use the method write() of the file handle object.

Let's start with a very simple and straightforward example:

```
fh = open("example.txt", "w")
fh.write("To write or not to write\nthat is the question!\n")
fh.close()
```

Especially if you are writing to a file, you should never forget to close the file handle again. Otherwise you will risk to end up in a non consistent state of your data.

You will often find the with statement for reading and writing files. The advantage is that the file will be automatically closed after the indented block after the with has finished execution:

```
with open("example.txt", "w") as fh:
    fh.write("To write or not to write\nthat is the question!\n")
```

Our first example can also be rewritten like this with the with statement:

```
with open("ad_lesbiam.txt") as fobj:
    for line in fobj:
        print(line.rstrip())
```

Example for simultaneously reading and writing:

```
fobj_in = open("ad_lesbiam.txt")
fobj_out = open("ad_lesbiam2.txt","w")
i = 1
for line in fobj_in:
    print(line.rstrip())
    fobj_out.write(str(i) + ": " + line)
    i = i + 1
fobj_in.close()
fobj_out.close()
```

Every line of the input text file is prefixed by its line number. So the result looks like this:

```
$ more ad_lesbiam2.txt
1: V. ad Lesbiam
2:
3: VIVAMUS mea Lesbia, atque amemus,
4: rumoresque senum severiorum
5: omnes unius aestimemus assis!
6: soles occidere et redire possunt:
7: nobis cum semel occidit breuis lux,
8: nox est perpetua una dormienda.
9: da mi basia mille, deinde centum,
10: dein mille altera, dein secunda centum,
11: deinde usque altera mille, deinde centum.
12: dein, cum milia multa fecerimus,
13: conturbabimus illa, ne sciamus,
14: aut ne quis malus inuidere possit,
15: cum tantum sciat esse basiorum.
16: (GAIUS VALERIUS CATULLUS)
```

There is one possible problem, which we have to point out: What happens if we open a file for writing, and this file already exists. You can consider yourself fortunate, if the content of this file was of no importance, or if you have a backup of it. Otherwise you have a problem, because as soon as an open() with a "w" has been executed the file will be removed. This is often what you want, but sometimes you just want to append to the file, like it's the case with logfiles.

If you want to append something to an existing file, you have to use "a" instead of "w".

READING IN ONE GO

So far we worked on files line by line by using a for loop. Very often, especially if the file is not too large, it's more convenient to read the file into a complete data structure, e.g. a string or a list. The file can be closed after reading and the work is accomplished on this data structure:

```
>>> poem = open("ad_lesbiam.txt").readlines()
>>> print(poem)
['V. ad Lesbiam \n', '\n', 'VIVAMUS mea Lesbia, atque amemus,\n',
'rumoresque senum severiorum\n', 'omnes unius aestimemus
assis!\n', 'soles occidere et redire possunt:\n', 'nobis cum semel
occidit breuis lux,\n', 'nox est perpetua una dormienda.\n', 'da
mi basia mille, deinde centum,\n', 'dein mille altera, dein
secunda centum,\n', 'deinde usque altera mille, deinde centum.\n',
'dein, cum milia multa fecerimus,\n', 'conturbabimus illa, ne
sciamus,\n', 'aut ne quis malus inuidere possit,\n', 'cum tantum
sciat esse basiorum.\n', '(GAIUS VALERIUS CATULLUS)']
>>> print(poem[2])
VIVAMUS mea Lesbia, atque amemus,
```

In the above example, the complete poem is read into the list poem. We can acces e.g. the 3rd line with `poem[2]`.

Another convenient way to read in a file might be the method read() of open. With this method we can read the complete file into a string, as we can see in the next example:

```
>>> poem = open("ad_lesbiam.txt").read()
>>> print(poem[16:34])
VIVAMUS mea Lesbia
>>> type(poem)
<type 'str'>
>>>
```

This string contains the complete content of the file, which includes the carriage returns and line feeds.

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. It has only one parameter in Python3 (no "whence" is available as in Python2). 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. We will demonstrate the way of working with both seek and tell in the following example. You have to create a file called "buck_mulligan.txt" with the content "Stately, plump Buck Mulligan came from the stairhead, bearing a bowl of lather on which a mirror and a razor lay crossed.":

```
>>> fh = open("buck_mulligan.txt")
>>> fh.tell()
0
>>> fh.read(7)
'Stately'
>>> fh.tell()
7
>>> fh.read()
', plump Buck Mulligan came from the stairhead, bearing a bowl
of\nlather on which a mirror and a razor lay crossed.\n'
>>> fh.tell()
122
>>> fh.seek(9)
9
>>> fh.read(5)
'plump'
```

It's also possible to set the file position relative to the current position by using tell correspondingly:

```
>>> fh = open("buck_mulligan.txt")
>>> fh.read(15)
```

```
'Stately, plump '
>>> # set the current position 6 characters to the left:
...
>>> fh.seek(fh.tell() -6)
9
>>> fh.read(5)
'plump'
>>> # now, we will advance 29 characters to the
>>> # 'right' relative to the current position:
...
>>> fh.seek(fh.tell() + 29)
43
>>> fh.read(10)
'stairhead,'
>>>
```

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.

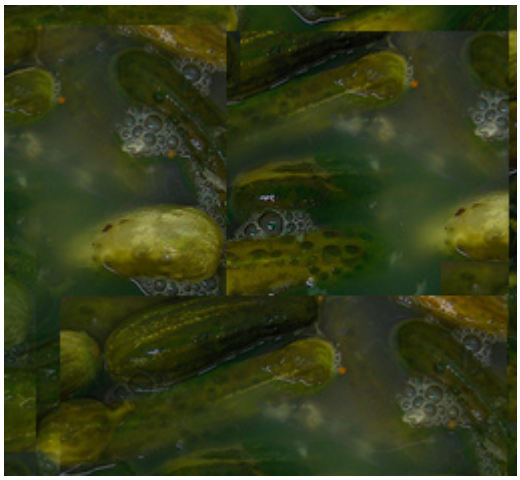
```
fh = open('colours.txt', 'w+')
fh.write('The colour brown')

# 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)
```

We get the following output:

```
brown
16
The colour green
```

"HOW TO GET INTO A PICKLE"



We don't mean what the heading says. On the contrary, we want to prevent any nasty situation, like loosing 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 for this purpose a module, 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 read back in again the data:

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

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:

```
>>> import shelve
>>> s = shelve.open("MyShelve")
```

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)
...
city
street
```

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
information.
>>> 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
information.
>>> import shelve
>>> tele = shelve.open("MyPhoneBook")
```

```
>>> tele["Steve"]["phone"]
'8745'
>>>
```

EXERCISES

1. The file cities_and_times.txt contains city names and times. 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)), ('Ankara', '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', 'Sun', (2, 52)), ('Winnipeg', 'Sun', (1,
52)), ('Zurich', 'Sun', (8, 52))]
```

   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 TO OUR EXERCISES

1.
```
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".