File operations, data parsing and batch files

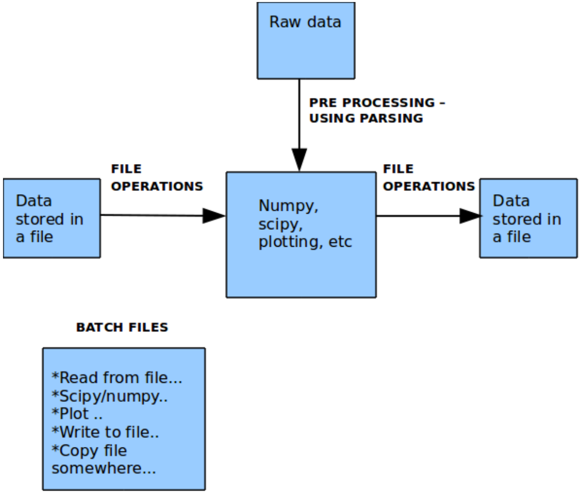
Felix Hoffmann

felix.hoffmann@jupiter.uni-freiburg.de

Bernstein Center Freiburg

October 30, 2014

File operations and what ?



File operations: Reading

Opening an existing file

```
>>> f = open("test.txt","rb")
>>> print f
<open file 'test.txt', mode 'rb' at 0x...>
```

Reading it:

```
>>> f.read()
'hello world'
```

Closing it:

```
>>> f.close()
>>> print f
<closed file 'test.txt', mode 'rb' at 0x...>
```

File operations: Writing

Opening a (new) file

```
>>> f = open("new_test.txt","wb")
>>> print f
<open file 'test.txt', mode 'wb' at 0x...>
```

Writing to it:

```
>>> f.write("hello world, again")
>>> f.write("... and again")
>>> f.close()
```

⇒ Only after calling close() the changes appear in the file for editing elsewhere!

File operations: Appending

Opening an existing file

```
>>> f = open("test.txt","ab")
>>> print f
<open file 'test.txt', mode 'ab' at 0x...>
```

Appending to it:

```
>>> f.write("hello world, again")
>>> f.write("... and again")
>>> f.close()
```

⇒ In append mode the **file pointer** is set to the end of the opened file.

File operations: More about file pointers

```
f = open("lines_test.txt", "wb")
for i in range(10):
    f.write("this is line %d \n" %(i+1))
f.close()
```

Reading from the file:

```
>>> f = open("lines_test.txt", "rb")
>>> f.readline()
'this is line 1 \n'
>>> f.readline()
'this is line 2 \n'
>>> f.read(14)
'this is line 3'
>>> f.read(2)
' \n'
```

File operations: More about file pointers

`f.tell()`	gives current position within file **`f`**	
`f.seek(x[, from])`	change file pointer position within file **`f`**, where	
	from = 0	from beginning of file
	from = 1	from current position
	from = 2	from end of file

```
>>> f = open("lines_test.txt", "rb")
>>> f.tell()
0
>>> f.read(10)
'this is li'
>>> f.tell()
10
```

File operations: More about file pointers

```
>>> f.seek(5)
>>> f.tell()
5
>>> f.seek(10,1)
>>> f.tell()
15
>>> f.seek(-10,2)
>>> f.tell()
151
>>> f.read()
' line 10 \n'
```

File operations: Other Modes

rb+	Opens a file for both reading and writing. The file pointer will be at the beginning of the file.
wb+	Opens a file for both writing and reading. Overwrites the existing file if the file exists. If the file does not exist, creates a new file for reading and writing.
ab+	Opens a file for both appending and reading in binary format. The file pointer is at the end of the file if the file exists. The file opens in the append mode. If the file does not exist, it creates a new file for reading and writing.

tutorialspoint.com/python/python_files_io.htm

Sav

```
>>> import pickle
>>> f = open('save_file.p', 'wb')
>>> ex_dict = {'hello': 'world'}
>>> pickle.dump(ex_dict, f)
>>> f.close()
```

```
>>> import pickle
>>> f = open('save_file.p', 'rb')
>>> loadobj = pickle.load(f)
>>> print loadobj['hello']
world
```

Best practice: With Statement

```
import pickle

ex_dict = {'hello': 'world'}

with open('save_file.p', 'wb') as f:
    pickle.dump(ex_dict, f)
```

```
import pickle

with open('save_file.p', 'rb') as f:
    loadobj = pickle.load(f)

print loadobj['hello']
```

⇒ Use this!

Need for parsing

Imagine that

Data files are generated by a third party (no control over the format)

& the data files need pre-processing

⇒ Regular expressions provide a powerful and concise way to perform pattern match/search/replace over the data

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Regular expressions - A case study

Formatting street names

```
>>> s = '100 NORTH MAIN ROAD'
>>> s.replace('ROAD', 'RD.')
'100 NORTH MAIN RD.'
>>> s = '100 NORTH BROAD ROAD'
>>> s.replace('ROAD', 'RD.')
'100 NORTH BRD. RD.'
>>> s[:-4] + s[-4:].replace('ROAD', 'RD.')
'100 NORTH BROAD RD.'
```

Better use regular expressions!

```
>>> import re
>>> re.sub(r'ROAD$', 'RD.', s)
'100 NORTH BROAD RD.'
```

Pattern matching with regular expressions

^	Matches beginning of line/pattern
$	Matches end of line/pattern
.	Matches any character except newline
[..]	Matches any single character in brackets
[^.]	Matches any single character not in brackets
re*	Matches 0 or more occurrences of the preceding expression
re+	Matches 1 or more occurrences of the preceding expression
re?	Matches 0 or 1 occurrence
re{**n**}	Match exactly n occurrences
re{**n,**}	Match n or more occurrences
re{**n,m**}	Match at least n and at most m

$\Rightarrow$ Use cheatsheets, trainers, tutorials, builders, etc..

re.search() & matches

```
>>> import re
>>> data = "I like python"
>>> m = re.search(r'python',data)
>>> print m
<_sre.SRE_Match object at 0x...>
```

Important properties of the match object:

group()	Return the string matched by the RE
start()	Return the starting position of the match
end()	Return the ending position of the match
span()	Return a tuple containing the (start, end) positions of the match

re.search() & matches

For example:

```
>>> import re
>>> data = "I like python"
>>> m = re.search(r'python',data)
>>> m.group()
'python'
>>> m.start()
7
>>> m.span()
(7,13)
```

For a complete list of match object properties see for example the Python Documentation:

https://docs.python.org/2/library/re.html#match-objects

re.findall()

```
>>> import re
>>> data = "Python is great. I like python"
>>> m = re.search(r'[pP]ython',data)
>>> m.group()
'Python'
```

⇒ **re.search()** returns only the first match, use **re.findall()** instead:

```
>>> import re
>>> data = "Python is great. I like python"
>>> l = re.findall(r'[pP]ython',data)
>>> print l
['Python', 'python']
```

⇒ Returns list instead of match object!

re.findall() - Example

```python
import re

with open("history.txt", "rb") as f:
    text = f.read()

year_dates = re.findall(r'19[0-9]{2}', text)
```

re.split()

Suppose the data stream has well-defined delimiter

```
>>> data = "x = 20"
>>> re.split(r'=',data)
['x ', ' 20']

>>> data = 'ftp://python.about.com'
>>> re.split(r':/{1,3}', data)
['ftp', 'python.about.com']

>>> data = '25.657'
>>> re.split(r'\.',data)
['25', '657']
```

re.sub()

Replace patterns by other patterns.

```
>>> data = "2004-959-559 # my phone number"
>>> re.sub(r'#.*','',data)
'2004-959-559 '
```

A more interesting example:

```
>>> data = "2004-959-559"
>>> re.sub(r'([0-9]*)-([0-9]*)-([0-9]*)',
>>>        r'\3-\2-\1', data)
'559-959-2004'
```

⇒ Groups are captured in parenthesis and referenced in the replacement string by \1, \2, ...

os module

Provides a way of using os dependent functionality:

os.mkdir()	Creates a directory (like mkdir)
os.chmod()	Change the permissions (like chmod)
os.rename()	Rename the old file name with the new file name.
os.listdir()	List the contents of the directory
os.getcwd()	Get the current working directory path
os.path	Submodule for useful functions on pathnames

For example, list all files in the current directory:

```
>>> from os import listdir
>>>
>>> for f in listdir("."):
>>>     print f
```