10: File operations

(and internet)

File operations

- File operations are necessary to
  - Handle larger input sets
  - Save results
  - Modify file and directory structures

File object in Python

- All file operations in Python are done by using a file object.

- File object contains several methods for different operations, including reading, writing and examining files

Creating a file object

- File object can be created by using the **open** function; note, that the object <u>must</u> be saved to a variable to use it later.

- Syntax:

```
fileVariable = open(filename, mode)
```

File names

- The file name parameter is given as a string.
- Naming of files depends on the OS, the common format however includes a name and an extension, separated with a dot (e.g. "example.txt" or "my_program.py").
- It's usually a good practice to avoid special characters and spaces in file names. Use the rules for naming variables in Python!

File modes

- The file mode parameter is a string. The most common options are:

File mode	Usage
r	Opens file for reading
w	Opens file for writing
a	Opens file for appending
rb	Opens file for reading in binary mode
wb	Opens file for writing in binary mode
r+	Opens the file for both reading and writing

Examples

- Open file readme.txt for reading:

```
myFile = open("readme.txt", "r")
```

- Open file results.dat for writing:

```
resultFile = open("results.dat", "w")
```

More about file objects

- File object is like any Python object
- Hence, it can be passed as an argument
- File objects can also be stored into data structures

Example

- Open a file, pass the file object as argument into a procedure:

```
def readFile(file):
    # file is used here…

f = open("my_file.dat", "r")
readFile(f)
```

File paths

- By default, the Python interpreter looks for the files in the current folder (which usually is the folder where your program is located in).

- However, it is possible to specify a path with the file name.

File separators

- In default, Python uses slash as a file separator (instead of Windows' backslash).

- Hence, a path is formed in syntax

```
directory/other_directory/other_directory…
```

File paths (cont.)

- To open a file "text.txt" located in "C:\texts\new\", use

```
myFile = open("C:\texts/new/text.txt", "r")
```

- To open a file "results.dat" from a folder "dat", located in the current folder, use:

```
myFile = open("dat/results.dat", "r")
```

Data file locations

- Usually the data files for programs should be located on the same folder than the actual program (or in the separate data folder under program folder).

- **Avoid using absolute paths!** This makes transferring programs very difficult.

Absolute paths

- So **do not do this:**

```
file = open("c:\\programs/python/data.txt")
```

- ..but instead something like

```
file = open("data/data.txt")
```

Closing a file

- After the file operations are all complete, the file needs to be closed. File is closed using the **close** method:

```
myFile.close()
resultFile.close()
```

- Note, that if the file is not closed, you may lose the information, system resources or the whole file!

Opening and closing several files

- Files can be opened and closed in any order
- Hence, it is a very good idea to close a file immediately when access is no longer needed
- Still, if a file is accessed several times in a short period, opening and closing it each time is not necessary (and not recommended).

Reading from a file

- There are several methods to read data from a file.

1. To read the whole file, use the **read** method.
2. To read a single line from a file, use the **readline()** method; all lines can be read using the **readlines()** method.
3. To iterate through a file, use the **for** statement.

Reading the whole file

- The **read** method reads the whole file. Note, that the file may be larger than available memory, which will lead to an error.

- Example:

```
myFile = open("testfile.txt", "r")
content = myFile.read()
print content
myFile.close()
```

Reading the whole file (cont.)

- The **read** method has an optional parameter, which specifies the number of bytes read from a file:

```
inputFile = open("data.dat", "r")
data = inputFile.read(1024) # read 1 kilobyte
inputFile.close()
```

Reading a single line

- The **readline** method reads a single line from the file. A newline character ("\n") is left at the end of the line.
- If there are no more lines left in the file, the method returns an empty string ("").
- The blank lines in file are represented by the newline character only ("\n")

Reading a single line (cont.)

- Example: read and output three lines from a file:

```
f = open("data.txt", "r")
for i in range(3):
    line = f.readline()
    if line != "":
        print line
f.close()
```

Reading all lines

- To read all lines from the file, use the **readlines** method.
- The method splits the lines into a list by using a newline character ("\n") as a delimiter.
- Again, if the file is too large to fit into the memory, an error is thrown.

Reading all lines (cont.)

- Example: read all lines into a list, then iterate through a list and remove newline characters.

```
f = open("testfile.txt", "r")
allLines = f.readlines()

# enumerate() iterates through index and value
for index, value in enumerate(allLines):
   allLines[index] = value.replace("\n","")
```

Iterating through a file

- Often the easiest solution is to iterate through a file using the **for** statement:

```
f = open("testfile.txt","r")
for line in f:
   print line
f.close()
```

Reading data other than strings

- Since all methods for reading data from a file return a string, you need to convert the data yourself.

- Example: Read all numbers from a file and count their average:

```
f = open("numbers.txt","r")
sum, count = 0,0
for line in f:
   sum = sum + int(line)
   count = count + 1
f.close()
avg = float(sum) / count
```

Writing to a file

- There are two alternative modes when writing files:
- In write mode ("w") all previous content of the file are overwritten.
- In append mode ("a") new content is written after the existing content.

Writing to a file (cont.)

- If you're using the write mode ("w") and the file with given name is not found, Python creates a file with that name:

```
# create a new empty file newfile.txt
f = open("newfile.txt", "w")
f.close()
```

Writing to a file (cont.)

- Data is written to a file using the **write** method. The method gets a string containing the data as a parameter. For example:

```
s = "Hello, everyone!"
f = open("newfile.txt","w")
f.write(s)
f.close()
```

Writing to a file (cont.)

- Program that queries the user for a string and appends it into end of an existing file.

```
diary = open("my_diary.txt", "a")
s = raw_input("How are you today? ")
diary.write(s + "\n")
diary.close()
```

Writing data other than strings

- Since the write method only accepts strings as parameter, all other types need to be converted to strings before writing:

```
a = 240
f = open("myfile.dat","w")
f.write(str(a))
f.close()
```

Writing data other... (cont.)

- Note, that you can't concatenate other objects with strings inside a function call. This:

```
n = 100
f.write("This is a number:", n)
```

- ...would throw an error since Python interprets is as a function call with two arguments (i.e. a tuple with two items). Instead, use

```
f.write ("This is a number:" + str(n))
```

Slicing data

- Often there are several values in one row of the file. To split the data into separate values, you can use the **split** method of a string object.

- The method returns a list with all items from a string, split by given *delimiter.* If no delimiter is specified, any white space character is used.

Slicing data (cont.)

- Example: split a string using comma "," as a delimiter and then output all items in a resulting list:

```
tst = "aaa,bbb,ccc,ddd,eee,fff,ggg"
myList = tst.split(",")
for item in myList:
   print item
```

Moving the pointer in file

- For each file, Python creates a "pointer" which holds the current position in file.

- When the file is opened, the pointer is at location zero. Each read or write operation moves the pointer ahead.

Moving the pointer (cont.)

- The current position in file can be returned by the **tell** method, which returns the offset in bytes:

```
f = open("myfile.txt", "r")
print f.tell() # outputs 0
f.close()
```

Moving the pointer (cont.)

- The location can be changed with the **seek()** method, which gets the offset in bytes as a parameter:

```
myFile = open("file.dat", "r")
line = myFile.readline()
myFile.seek(0) # pointer back to start of file
line2 = myFile.readline() # reads the same line
myFile.close()
```

Example 1

- Read a matrix containing a number of integers in each row, separated with commas, from a file and save it into Python matrix:

```
mf = open("matrix.txt", "r")
matrix = []
for row in mf:
   matrix.append(row.split(","))
mf.close()
```

Example 2

- Read names from file "names.txt" one by one, capitalize them and write them to file "cap_names.txt":

```
fileIn = open("names.txt","r")
fileOut = open("cap_names.txt", "w")
for name in fileIn:
   fileOut.write(name.capitalize())
fileOut.close()
fileIn.close()
```

Example 3

- Function that returns the next line from the file received as an argument, or None, if no such line:

```
def getNextLine(file):
    s = file.readline()
    if s != "":
        return s
    return None
```

Reading data from internet

- Python can read web site contents quite similarly to reading file contents
- For this, the urllib library needs to be used

Example

```
# open the urllib library
import urllib

# open an access to a web page
wpage = urllib.urlopen("http://learninganalytics.fi/en")

# read the HTML source
content = wpage.read()
print content

# close the stream
wpage.close()
```

Note

- By default, only read operations are permitted when using the urllib
- If you read a web page, it will usually contain the HTML tags as well
- The images and other files need to be read (and handled) separately