PRG – PROGRAMMING ESSENTIALS

Lecture 7 – Files, I/O

Milan Nemy

Czech Technical University in Prague,
Faculty of Electrical Engineering, Dept. of Cybernetics

https://beat.ciirc.cvut.cz/people/milan-nemy/

milan.nemy@cvut.cz

RECAP: MODULES – SCOPE

- A **scope** is a *textual region of a Python program where a namespace is directly accessible*

What types of scopes can be defined?

- **Local scope** refers to identifiers declared within a function / class *(these identifiers are kept in the namespace that belongs to the function, and each function has its own namespace)*
- **Global scope** refers to all the identifiers declared within the current module (file)
- **Built-in scope** refers to all the identifiers built into Python *(those like range and min that can be used without having to import anything*)

source http://openbookproject.net/thinkcs/python/english3e/modules.html

RECAP: MODULES – SCOPE

```
1  def range(n):
2      return 123*n
3
4  print(range(10))
```

```
1  n = 10
2  m = 3
3  def f(n):
4      m = 7
5      return 2*n+m
6
7  print(f(5), n, m)
```

What are the scope precedence rules?

- The same name can occur in more than one of these scopes, but the **innermost, or local scope, will always take precedence over the global scope**, and the global scope always gets used in preference to the built-in scope

- Names can be "**hidden**" from use if own variables or functions reuse those names (shadowing)
- EXAMPLE: *variables **n** and **m** are created just for the duration of the execution of **f** since they are created in the local namespace of function **f** (precedence rules apply)*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

RECAP: MODULES – THE DOT OPERATOR

```
1  import math
2  x = math.sqrt(10)
```

```
1  from math import cos, sin, sqrt
2  x = sqrt(10)
```

```
1  def area(radius):
2      import math
3      return math.pi * radius * radius
4
5  x = math.sqrt(10)      # This gives an error
```

```
1  from math import *    # Import all the identifiers from math,
2                        #   adding them to the current namespace.
3  x = sqrt(10)          # Use them without qualification.
```

```
1  >>> import math as m
2  >>> m.pi
3  3.141592653589793
```

- Variables defined **inside a module** are called **attributes** of the module
- Attributes are accessed using the **dot** operator (**.**)
- When a dotted name is used it is often referred to it as a *fully qualified name*

source http://openbookproject.net/thinkcs/python/english3e/modules.html

FILES

- During program execution, its data are stored in random access memory (**RAM**)
- RAM is fast and inexpensive but **volatile**
- To preserve data when the system is not powered the data has to be written to a **non-volatile storage medium**
- Data on **non-volatile** storage media are stored in **named locations** on the media called **files**
- By **reading** and **writing** files, programs can save information between program runs
- To open a file, we specify its **name (path)** and indicate whether we want to **read** or **write**.

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  myfile = open("test.txt", "w")
2  myfile.write("My first file written from Python\n")
3  myfile.write("---------------------------------\n")
4  myfile.write("Hello, world!\n")
5  myfile.close()
```

- EXAMPLE: *program writes three lines of text into a file*

- **Line 1**: the **open** function takes two arguments:
  the first is the **name** of the file, and the second is the **mode**
- **Mode "w"** means that we are opening the file for **writing**:
  - **If there is no file on the disk, it will be created**
  - **If the file exists it will be replaced**

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  myfile = open("test.txt", "w")
2  myfile.write("My first file written from Python\n")
3  myfile.write("-------------------------------\n")
4  myfile.write("Hello, world!\n")
5  myfile.close()
```

- EXAMPLE: *program writes three lines of text into a file*

- Opening a file creates a file **handle**
- Variable *myfile* refers to the new handle object

- Program calls **methods on the handle** (dot notation) changing the actual file which is usually located on our disk

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  myfile = open("test.txt", "w")
2  myfile.write("My first file written from Python\n")
3  myfile.write("-------------------------------\n")
4  myfile.write("Hello, world!\n")
5  myfile.close()
```

- To **store data** into the file we invoke the **write method** on the handle (lines 2, 3 and 4)
- **Lines 2 – 4:** should usually be replaced by a loop that writes more lines into the file, i.e. the *content we want to store*
- **Line 5: closing** the file handle tells the system that writing the content is finished and makes the disk file available for reading by other programs

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – HANDLE

A handle is somewhat like a TV remote control

We're all familiar with a remote control for a TV. We perform operations on the remote control — switch channels, change the volume, etc. But the real action happens on the TV. So, by simple analogy, we'd call the remote control our `handle` to the underlying TV.

Sometimes we want to emphasize the difference — the file handle is not the same as the file, and the remote control is not the same as the TV. But at other times we prefer to treat them as a single mental chunk, or abstraction, and we'll just say "close the file", or "flip the TV channel".

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  mynewhandle = open("test.txt", "r")
2  while True:                              # Keep reading forever
3      theline = mynewhandle.readline()     # Try to read next line
4      if len(theline) == 0:                # If there are no more lines
5          break                            #     leave the loop
6
7      # Now process the line we've just read
8      print(theline, end="")
9
10 mynewhandle.close()
```

- Reading a file one **line-at-a-time** using the mode argument is **"r"** for reading
- More extensive logic into the body of the loop at line 8

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  mynewhandle = open("test.txt", "r")
2  while True:                              # Keep reading forever
3      theline = mynewhandle.readline()     # Try to read next line
4      if len(theline) == 0:                # If there are no more lines
5          break                            #     leave the loop
6
7      # Now process the line we've just read
8      print(theline, end="")
9
10 mynewhandle.close()
```

- **Line 8:** the newline character that print usually appends to our strings is suppressed
- The string already has its own newline: the **readline** method in line 3 returns everything up to and **including the newline**
- The end-of-file detection logic: when there are no more lines to be read from the file, **readline** returns an **empty string**
  (*no newline at the end, hence its length is 0*)

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – END OF FILE

Fail first ...

In our sample case here, we have three lines in the file, yet we enter the loop *four* times. In Python, you only learn that the file has no more lines by failure to read another line. In some other programming languages (e.g. Pascal), things are different: there you read three lines, but you have what is called *look ahead* — after reading the third line you already know that there are no more lines in the file. You're not even allowed to try to read the fourth line.

So the templates for working line-at-a-time in Pascal and Python are subtly different!

When you transfer your Python skills to your next computer language, be sure to ask how you'll know when the file has ended: is the style in the language "try, and after you fail you'll know", or is it "look ahead"?

If we try to open a file that doesn't exist, we get an error:

```
>>> mynewhandle = open("wharrah.txt", "r")
IOError: [Errno 2] No such file or directory: "wharrah.txt"
```

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – READLINES vs. READ

```
1  f = open("friends.txt", "r")
2  xs = f.readlines()
3  f.close()
4
5  xs.sort()
6
7  g = open("sortedfriends.txt", "w")
8  for v in xs:
9      g.write(v)
10 g.close()
```

- EXAMPLE: *fetch data from a disk file, perform processing (**sorting**) and turn it into a list of lines written back into the file*
- The **readlines** method in **line 2** reads all the lines and returns a list of the strings

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – READLINES vs. READ

```
1  f = open("somefile.txt")
2  content = f.read()
3  f.close()
4
5  words = content.split()
6  print("There are {0} words in the file.".format(len(words)))
```

- EXAMPLE: *reading the **whole file at once***
- Read the complete contents of the file into a single string, and then to use string-processing skills to work with the contents
- Not interested in the line structure of the file

- EXAMPLE: *use the **split** method on strings which can break a string into words (e.g. counting the number of words in a file)*
- The **"r"** mode in **line 1** is omitted since **by default** Python opens the file for reading

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES

```
1  f = open("somefile.txt")
2  content = f.read()
3  f.close()
4
5  words = content.split()
6  print("There are {0} words in the file.".format(len(words)))
```

Your file paths may need to be explicitly named.

In the above example, we're assuming that the file `somefile.txt` is in the same directory as your Python source code. If this is not the case, you may need to provide a full or a relative path to the file. On Windows, a full path could look like `"C:\\temp\\somefile.txt"`, while on a Unix system the full path could be `"/home/jimmy/somefile.txt"`.

We'll return to this later in this chapter.

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – BINARY

```
1  f = open("somefile.zip", "rb")
2  g = open("thecopy.zip", "wb")
3
4  while True:
5      buf = f.read(1024)
6      if len(buf) == 0:
7          break
8      g.write(buf)
9
10 f.close()
11 g.close()
```

- Working with **binary files**
- Binary files usually hold *photographs*, *videos*, *zip* files, *executable* programs
- Binary files are **not organized into lines** and cannot be opened with a normal text editor
- Reading binary files gets **bytes** back rather than a string

source http://openbookproject.net/thinkcs/python/english3e/files.html

FILES – BINARY

```
1  f = open("somefile.zip", "rb")
2  g = open("thecopy.zip", "wb")
3
4  while True:
5      buf = f.read(1024)
6      if len(buf) == 0:
7          break
8      g.write(buf)
9
10 f.close()
11 g.close()
```

- Mode **"b"** to tell Python that the files are binary
- **Line 5:** read takes an argument telling how many bytes to attempt to read from the file
  (*read and write up to 1024 bytes on each iteration of the loop*)
- When an **empty buffer** is returned from the attempt to read, break out of the loop and close both the files
- The type of **buf** is bytes

source http://openbookproject.net/thinkcs/python/english3e/files.html

EXAMPLE – FILE CONTENT FILTER

```
1   def filter(oldfile, newfile):
2       infile = open(oldfile, "r")
3       outfile = open(newfile, "w")
4       while True:
5           text = infile.readline()
6           if len(text) == 0:
7               break
8           if text[0] == "#":
9               continue
10
11          # Put any more processing logic here
12          outfile.write(text)
13
14      infile.close()
15      outfile.close()
```

source http://openbookproject.net/thinkcs/python/english3e/files.html

EXAMPLE – FILE CONTENT FILTER

```
1  def filter(oldfile, newfile):
2      infile = open(oldfile, "r")
3      outfile = open(newfile, "w")
4      while True:
5          text = infile.readline()
6          if len(text) == 0:
7              break
8          if text[0] == "#":
9              continue
10
11         # Put any more processing logic here
12         outfile.write(text)
13
14     infile.close()
15     outfile.close()
```

- EXAMPLE: *filter that copies one file to another, omitting any lines that begin with #, i.e. comments*
- **Line 9:** the **continue** statement skips over remaining lines in the current iteration of the loop, but the loop will still iterate

source http://openbookproject.net/thinkcs/python/english3e/files.html

EXAMPLE – FILE CONTENT FILTER

```
1   def filter(oldfile, newfile):
2       infile = open(oldfile, "r")
3       outfile = open(newfile, "w")
4       while True:
5           text = infile.readline()
6           if len(text) == 0:
7               break
8           if text[0] == "#":
9               continue
10
11          # Put any more processing logic here
12          outfile.write(text)
13
14      infile.close()
15      outfile.close()
```

- If text is the **empty string**, the loop exits
- If the first character of text is a **hash mark**, the flow of execution goes to the top of the loop, ready to start processing the next line
- Only if *both conditions fail, writing the line into the new file*

source http://openbookproject.net/thinkcs/python/english3e/files.html

DIRECTORIES

Directories

A **directory** is an organizational unit of the file system that allows for the hierarchichal structure. It can contain files and other directories. Each directory is itself contained in some other directory.

The **root directory** is special. It is always present in the filesystem and designates the start of the hierarchy. It is denoted with forward slash (/) on Linux-types of systems. On Windows, there is a separate root directory on each disk, and they are denoted by backslashes (\).

Each directory contains 2 special entries:

- `.` is the current directory itself
- `..` is the parent directory

source courtesy of Petr Posik BE5b33PR 2016/2017

DIRECTORIES

```
>>> wordsfile = open("/usr/share/dict/words", "r")
>>> wordlist = wordsfile.readlines()
>>> print(wordlist[:6])
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

- Files on non-volatile storage media are organized by a set of rules known as a **file system**
- File systems are made of **files** and **directories** (and *symbolic links*), which are **containers** for files and other directories.
- When we create a new file by opening it and writing, the new file goes into the **current directory**
- When we want to open a file somewhere else, we have to specify the **path** to the file, which is the **name of the directory** (or folder) where the file is located

source http://openbookproject.net/thinkcs/python/english3e/files.html

PATHS

Current working directory

Python keeps track of the **current working directory** (CWD), in which it looks for files. The default setting of CWD is platform dependent, but usually it is the directory from which the Python interpreter was run (not the directory where the interpreter is stored).

You can find out the CWD using function `getcwd` from module `os`. Let's see what the CWD is for the current notebook:

```
import os
print(os.getcwd())
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

source courtesy of Petr Posik BE5b33PR 2016/2017

PATHS

Paths

Paths are sequences of directory names possibly ended with a filename which unambiguosly resolve to a directory or file name in the filesystem.

Absolute paths always start from the root directory (`/`, forward slash), on Windows often preceded by drive letter (`C:\\`). An example:

```
/home/posik/teaching/prg/lectures/files.pdf
```

Relative paths start from the current working directory. Examples, assuming the CWD is `/home/posik/teaching`:

```
prg/lectures/files.pdf
../../svoboda/presentations/upload_system.pdf
```

source courtesy of Petr Posik BE5b33PR 2016/2017

PATHS

```
>>> wordsfile = open("/usr/share/dict/words", "r")
>>> wordlist = wordsfile.readlines()
>>> print(wordlist[:6])
['\n', 'A\n', "A's\n", 'AOL\n', "AOL's\n", 'Aachen\n']
```

- A Windows path might be:
  "C:/temp/words.txt" or **"C:\temp\words.txt"**
- Backslashes are used to escape things like newlines and tabs, we need to **write two backslashes in a literal string** to get one! (*the length of these two strings is the same*)

- We cannot use / or \ as part of a filename
  (*reserved as a delimiter between directory and filenames*)
- *The file **/usr/share/dict/words** should exist on Unix-based systems, and contains a list of words in alphabetical order*

source http://openbookproject.net/thinkcs/python/english3e/files.html

PATHS

Navigating in the file system

In the OS shell, you would use command `cd` or `chdir` to change the working directory. Similarly, you can use the Python function `os.chdir()`:

```
import os
orig_wd = os.getcwd()
os.chdir('/P/0Teaching')
print(os.getcwd())
```

```
C:\P\0Teaching
```

Now we are in a different directory. And we can change it back:

```
os.chdir(orig_wd)
print(os.getcwd())
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES

Working with file paths

Module `os.path` contains functions for working with file paths:

```
fpath = os.path.abspath('files.pdf')
print(fpath)
```

```
C:\P\0Teaching\programming essentials\prg-notes\files.pdf
```

```
print(os.path.dirname(fpath))
```

```
C:\P\0Teaching\programming essentials\prg-notes
```

```
print(os.path.basename(fpath))
```

```
files.pdf
```

```
print(os.path.splitext(os.path.basename(fpath)))
```

```
('files', '.pdf')
```

How to correctly create a path from fragments?

```
fpath2 = os.path.join('\\', 'P', '0Teaching')
print(fpath2)
```

```
\P\0Teaching
```

How to get a path to a directory or a file relative to CWD?

```
print(os.path.relpath(fpath2))
```

```
..\..
```

source courtesy of Petr Posik BE5b33PR 2016/2017

ENCODING

Encoding of strings and files

Strings are actually an abstraction. They are just sequences of bytes, but these bytes (or their groups) are interpreted as indices into a table of symbols containing upper- and lowercase letters, numbers, special characters and other symbols. The table with these symbols is called an **encoding**. The same string may look as complete gibberish if it is read with different encoding than the one used when creating it.

- ASCII: contains 127 characters, english upper- and lowercase letters, numbers, and some symbols. No characters from national alphabets.
- ...
- **UTF-8**: Unicode encoding that supports virtually any national alphabet, contains ASCII as its subset. **USE IT!**

This holds also for text files!

source courtesy of Petr Posik BE5b33PR 2016/2017

ENCODING

Opening text file with encoding

Function `open()` accepts several other parameters, among them the `encoding`. If you will **explicitly use UTF-8 each time** you call that function, you will save yourself a lot of trouble:

```
f = open('file_to_open.txt', 'r', encoding='utf-8')
...
f.close()
```

or

```
with open('file_to_open.txt', 'r', encoding='utf-8') as f:
    ...
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES – „WITH“ STATEMENT

The `with` statement

Because every call to `open()` should have a corresponding call to the `close()` method, Python provides a `with` statement that automatically closes a file when the end of the block is reached. The code

```
f = open('text.txt', 'r', encoding='utf-8')
contents = f.read()
f.close()
print(contents)
```

is equivalent to the following code using the `with` statement:

```
with open('text.txt', 'r', encoding='utf-8') as f:
    contents = f.read()
print(contents)
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES – „WITH“ STATEMENT

File reading: `file.readlines()`

Use this technique if you want to get a Python list of strings containing the individual lines from a file.

```
with open('text.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()
print(lines)
```

```
['Hello, world!\n', 'How are you?']
```

Note that the strings representing the individual lines contain also the newline character, `\n`. The last line may or may not end with a newline char. You can get rid of them using the `str.strip()` method.

```
for line in lines:
    print(line.strip())
```

```
Hello, world!
How are you?
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES – „WITH“ STATEMENT

File reading: `file.read()`

Use this technique when you want to read the file contents into a single (possibly huge) string, or when you want to specify, how many character shall be read.

```
with open('text.txt', 'r', encoding='utf-8') as f:
    contents = f.read()
print(contents)
```

```
Hello, world!
How are you?
```

When called with no arguments, it reads everything from the current file cursor all the way to the end of the file. When called with an integer argument, it reads that many characters and moves the cursor right after the characters that were just read.

```
with open('text.txt', 'r', encoding='utf-8') as f:
    first_10_chars = f.read(10)
    the_rest = f.read()
print("The first 10 chars:", first_10_chars)
print("The rest:", the_rest)
```

```
The first 10 chars: Hello, wor
The rest: ld!
How are you?
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES

File reading: `for <line> in <file>`

Use this technique when you want to do the same thing to every line from the file cursor to the end of a file. While the previous techniques read all the content of a file at once (which may not fit to memory), this technique reads one line at a time allowing to process large files.

```
with open('text.txt', 'r', encoding='utf-8') as f:
    for line in f:
        s = line.strip()
        print("The line '" + s + "' contains " + str(len(s)) + " charac
ters.")
```

```
The line 'Hello, world!' contains 13 characters.
The line 'How are you?' contains 12 characters.
```

source courtesy of Petr Posik BE5b33PR 2016/2017

EXAMPLE – COLLATZ SEQUENCE

File reading: `file.readline()`

This technique allows you to read a single line from a file, which is useful when you want to read only a part of the file.

Assume that we want to read the following text file which contains several different parts. The first line is the short description of the data. The next lines starting with # are comments. The following part contains the data.

```
%%writefile data_collatz_5.txt
Collatz 3n+1 sequence, starting from 5.
# The next number in a Collatz sequence is either 3n+1 if n is odd,
# or n/2 if n is even.
  5
 16
  8
  4
  2
  1
```

```
Overwriting data_collatz_5.txt
```

source courtesy of Petr Posik BE5b33PR 2016/2017

EXAMPLE – COLLATZ SEQUENCE

```
with open('data_collatz_5.txt', 'r', encoding='utf-8') as f:
    # Read the description line
    description = f.readline().strip()
    # Read all the comment lines
    comments = []
    line = f.readline().strip()
    while line.startswith('#'):
        comments.append(line)
        line = f.readline().strip()
    data = []
    data.append(int(line))
    for line in f:
        data.append(int(line))

print("Description:", description)
print("Comments:", comments)
print("Data:", data)
```

```
Description: Collatz 3n+1 sequence, starting from 5.
Comments: ['# The next number in a Collatz sequence is either 3n+1 if
n is odd,', '# or n/2 if n is even.']
Data: [5, 16, 8, 4, 2, 1]
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES – WRITE vs. APPEND

Writing files

Writing some text into a text file is very similar to reading it. Also, when reading, Python did not strip the newline characters; when writing, you have to put the newlines there manually as well.

```
with open('topics.txt', 'w', encoding='utf-8') as f:
    f.write('Computer Science\n')
    f.write('Programming\n')
    f.write('Clean code\n')
```

```
!cat topics.txt
```

```
Computer Science
Programming
Clean code
```

source courtesy of Petr Posik BE5b33PR 2016/2017

FILES – WRITE vs. APPEND

Appending to a file

When `'w'` is specified as the file mode, a new file is created if it does not exist; if it exists the file is overwritten. We can specify the `'a'` as a file mode: then we open an existing file for appending, i.e. the new information is added to its end.

```
with open('topics.txt', 'a', encoding='utf-8') as f:
    f.write('Software Engineering\n')
```

```
!cat topics.txt
```

```
Computer Science
Programming
Clean code
Software Engineering
```

source courtesy of Petr Posik BE5b33PR 2016/2017

EXAMPLE – READ and WRITE

Example: Reading and writing

Suppose we have a file with 2 numbers on each line, like this:

```
%%writefile number_pairs.txt
1 1
10 20
1.3 2.7
```

```
Overwriting number_pairs.txt
```

Let us write a function that takes 2 filenames as arguments, reads the number pairs from the first file and writes them together with its sum into the second file.

source courtesy of Petr Posik BE5b33PR 2016/2017

EXAMPLE – READ and WRITE

```
def sum_number_pairs(infname, outfname):
    """Read data from input file, sum each row, write results to output
 file.

    (str, str) -> None

    infname: the name of the input file containing a pair of numbers
             separated by whitespace on each line
    outfname: the name of the output file
    """
    with open(infname, 'r', encoding='utf-8') as infile, \
         open(outfname, 'w', encoding='utf-8') as outfile:
        for pair in infile:
            pair = pair.strip()
            operands = pair.split()
            total = float(operands[0]) + float(operands[1])
            new_line = '{} {}\n'.format(pair, total)
            outfile.write(new_line)
```

When called, this function creates the required output file containing the sums.

```
sum_number_pairs('number_pairs.txt', 'number_pairs_with_totals.txt')
!cat number_pairs_with_totals.txt
```

```
1 1 2.0
10 20 30.0
1.3 2.7 4.0
```

source courtesy of Petr Posik BE5b33PR 2016/2017

EXAMPLE – DATA FROM WEB

```
1  import urllib.request
2
3  url = "http://xml.resource.org/public/rfc/txt/rfc793.txt"
4  destination_filename = "rfc793.txt"
5
6  urllib.request.urlretrieve(url, destination_filename)
```

- EXAMPLE: *copy contents at some **web URL** to a local file*

- The **urlretrieve** function can be used to download any kind of content from the Internet (*resources to fetch must exist*)
- Need of **permissions** to write to the **destination filename**, and the file will be created in the **"current directory"** (*i.e. the same folder that the Python program is saved in*)

- Authorization necessary if behind a proxy server

source http://openbookproject.net/thinkcs/python/english3e/files.html

EXAMPLE – DATA FROM WEB

```
1  import urllib.request
2
3  def retrieve_page(url):
4      """ Retrieve the contents of a web page.
5          The contents is converted to a string before returning it.
6      """
7      my_socket = urllib.request.urlopen(url)
8      dta = str(my_socket.readall())
9      my_socket.close()
10     return dta
11
12 the_text = retrieve_page("http://xml.resource.org/public/rfc/txt/rfc793.txt")
13 print(the_text)
```

- Rather than saving the web resource to local disk, we read it directly into a string, and return it

- Opening the **remote url** returns a **socket** (***handle*** *to end of the connection between the program and the remote web server*)
- Call **read**, **write**, and **close** methods on the socket object

source http://openbookproject.net/thinkcs/python/english3e/files.html

SUMMARY

Summary

- Working with paths using `os.path` module.
- Before reading from a file or writing to it, you must first `open()` it.
  - Always specify encoding: `open(filename, mode, encoding='utf-8')`.
- When you are done, you must `f.close()` the file.
- By using `with`, the file is closed automatically:

```
with open('text.txt', 'r', encoding='utf-8') as f:
    contents = f.read()
    # ... and do other things to the opened file
# When you get here, the file is not opened anymore.
```

source http://openbookproject.net/thinkcs/python/english3e/files.html

REFERENCES

This lecture re-uses selected parts of the OPEN BOOK PROJECT
Learning with Python 3 (RLE)
http://openbookproject.net/thinkcs/python/english3e/index.html
available under GNU Free Documentation License Version 1.3)

- Version date: October 2012
- by Peter Wentworth, Jeffrey Elkner, Allen B. Downey, and Chris Meyers (based on 2nd edition by Jeffrey Elkner, Allen B. Downey, and Chris Meyers)
- Source repository is at https://code.launchpad.net/~thinkcspy-rle-team/thinkcspy/thinkcspy3-rle
- For offline use, download a zip file of the html or a pdf version from http://www.ict.ru.ac.za/Resources/cspw/thinkcspy3/