Lab 7

S2 2017

Lab 7

Note: If you do not have time to finish all exercises (in particular, the programming problems) during the lab time, you should continue working on them later. You can do this at home (if you have a computer with python set up), in the CSIT lab rooms outside teaching hours, or on one of the computers available in the university libraries or other teaching spaces.

If you have any questions about or difficulties with any of the material covered in the course so far, ask your tutor for help during the lab.

Objectives

The purpose of this week's lab is to:

- Understand more about variable scope, in the context of function calls and recursion.
- Practice reading and writing files.

Functions: namespaces and scope

Recall that:

- A namespace is a mapping that associates (variable) names with references to objects. Whenever an expression involving a variable is evaluated, a namespace is used to find the current value of a variable. (The namespace is also used to find the function associated with a function name. Functions are objects too, and function names are no different from variables.)

- In python there can be multiple namespaces. Variables with the same name can appear in several namespaces, but those are different variables. A variable name in one namespace can reference (have as value) any object, and the same name in a different namespace (which is a different variable) can reference a different object.

- When a variable is evaluated, the python interpreter follows a process to search for the name in the current namespaces.
- Whenever a function is called, a new local namespace for the function is created.
- The function's parameters are assigned references to the argument values in the local namespace. Any other assignments made to variables during execution of the function are also done in the local namespace.
- The local namespace disappears when the function finishes.

The name-resolution rule in python is Local - Enclosing - Global - Built-in (abbreviated LEGB). This rule defines the order in which namespaces are searched when looking for the value associated with a name (variable, function, etc). The built-in namespace stores python's built-in functions. Here we will only focus on the two that are most important for understanding variable scope in function calls: the local and global namespaces.

The *local* namespace is the namespace that is created when a function is called, and stops existing when the function ends. The *global* namespace is created when the program (or the python interpreter) starts executing, and persists until it finishes. (More technically, the global namespace is associated with the module `__main__`.)

Python provides two built-in functions, called `locals()` and `globals()`, that allow us to examine the contents of the current local and global namespace, respectively. (These functions return the contents of the respective namespace in the form of a "dictionary" - a data type which we will say more about later in the course. However, for the moment, the only thing we need to know about a dictionary is how to print its contents, which can be done with a for loop as shown in the examples below.)

Exercise 0(a)

The following code is adapted from an example in Punch & Enbody's book (page 417). Save it to a file (say, `local.py`) and execute it:

```
global_X = 27

def my_function(param1=123, param2="hi mom"):
    local_X = 654.321
    print("\n=== local namespace ===")  # line 1
    for key,val in list(locals().items()):
        print("name:", key, "value:", val)
    print("=======================")
    print("local_X:", local_X)
    # print("global_X:", global_X)  # line 2

my_function()
```

It should execute without error.

- How many entries are there in the local namespace?
- Does the variable `global_X` appear in the local namespace? If not, can the function still find the value of `global_X`? To test, uncomment the print call marked "# line 2" and see if you are able to print its value.
- Add the statement `global_X = 5` to the function, before printing the contents of the local namespace (i.e., just before line marked "# line 1"). What value then is printed on # line 2? Why?
- Print the value of `global_X` after the call to `my_function`. Does its value change after reassignment in the function?

Exercise 0(b)

The following program is from the next example in Punch & Enbody's book (page 418). Save it to a file (say, `global.py`) and run it:

```
import math
global_X = 27

def my_function(param1=123, param2="hi mom"):
    local_X = 654.321
    print("\n=== local namespace ===")
    for key,val in list(locals().items()):
        print("name:", key, "value:", val)
    print("=====================")
    print("local_X:", local_X)
    print("global_X:", global_X)

my_function()
print("\n--- global namespace ---")   # line 1
for key,val in list(globals().items()):
    print("name:", key, "value:", val)
print('------------------------')     # line 3
# print('local_X: ',local_X)          # line 4
print('global_X:',global_X)
print('math.pi: ',math.pi)
# print('pi:',pi)                     # line 5
```

The program should execute without error.

- What entries are in the global namespace but not the `my_function` local namespace?
- What entries are in the local namespace of `my_function` but not in the global namespace?
- Move the loop that prints the global namespace (from the line marked "# line 2" to "# line 3") into `my_function` (just after the loop that prints the local namespace) and run it again. Does the output change?
- Is the value of `local_X` printed if you uncomment the line marked "# line 4"? If not, why not?

- Is the value of pi printed if you uncomment “# line 5”? If not, why not?

The Local Assignment Rule

Python’s local assignment rule states that if an assignment to a (variable) name is made anywhere in a function, then that name is local: This means that the assignment creates an association in the local namespace, and that any lookup of the name will be made only in the local namespace. The following program will work:

```
x = 27

def increment_x():
    print("x before increment:", x)
    y = x + 1
    print("x after increment:", y)

increment_x()
```

but the following will not:

```
x = 27

def increment_x():
    print("x before increment:", x)
    x = x + 1
    print("x after increment:", x)

increment_x()
```

Note where the program fails: The first attempt to get the value of `x` causes the error, because `x` has not been assigned a value in the local namespace.

The `global` keyword can be used inside a function to escape from the local assignment rule. If a `global name` statement appears anywhere in the function, *name* will be searched for, and assigned, in the global namespace (and only in the global namespace). Therefore, the following also works:

```
x = 27

def increment_x():
    global x
    print("x before increment:", x)
    x = x + 1
    print("x after increment:", x)

increment_x()
```

and the function now references, and changes the global variable `x`.

The file system

Files and directories are an abstraction provided by the operating system (OS) to make it easier for programmers and users to interact with the underlying storage device (hard drive, USB key, network file server, etc).

In a unix system (like the one in the CSIT labs), the file system is organised into a single directory tree. That means directories can contain several (sub-)directories (or none), but each directory has only one directory directly above it, the one that it is contained in. (This is often called the "parent directory".) The top-most directory in the tree, which is called `/`, has no parent directory. Every file is located in some directory.

Exercise 1: Navigating the directory structure

For this exercise, you will need some directories and files to work with. As we recommended in Lab 1, create a directory called `comp1730` in your home directory (if you haven't already), and within that directory create one called `lab7`. You can do this using whichever tool you're familiar with (the commandline terminal or the graphical folders tool).

Next, create a file with a few lines of text (including some empty lines). You can use the editor in the IDE, or any other text editor, to write the file. (If you don't know what to write, just copy some text from this page.) Save your file with the name `sample.txt` in the `lab7` directory.

When you are done, you should have something like Figure 1.

In the python3 shell, import the `os` module:

```
In [1]: import os
```

This module provides several useful functions for interacting with the file system; in particular, it can let you know what the current working directory is, and change that. Try the following:

```
In [2]: os.getcwd()

Out [2]: '/students/uNNNNNNN'
```

The value returned will be a string like the above, where `NNNNNNN` is your uni ID number. (For the rest of the lab, wherever it is written `uNNNNNNN` you should substitute your uID.) Depending on how you started the shell, you may see a different output. Now try

```
In [3]: os.chdir('/students/uNNNNNNN/comp1730')

In [4]: os.getcwd()

Out [4]: ...
```

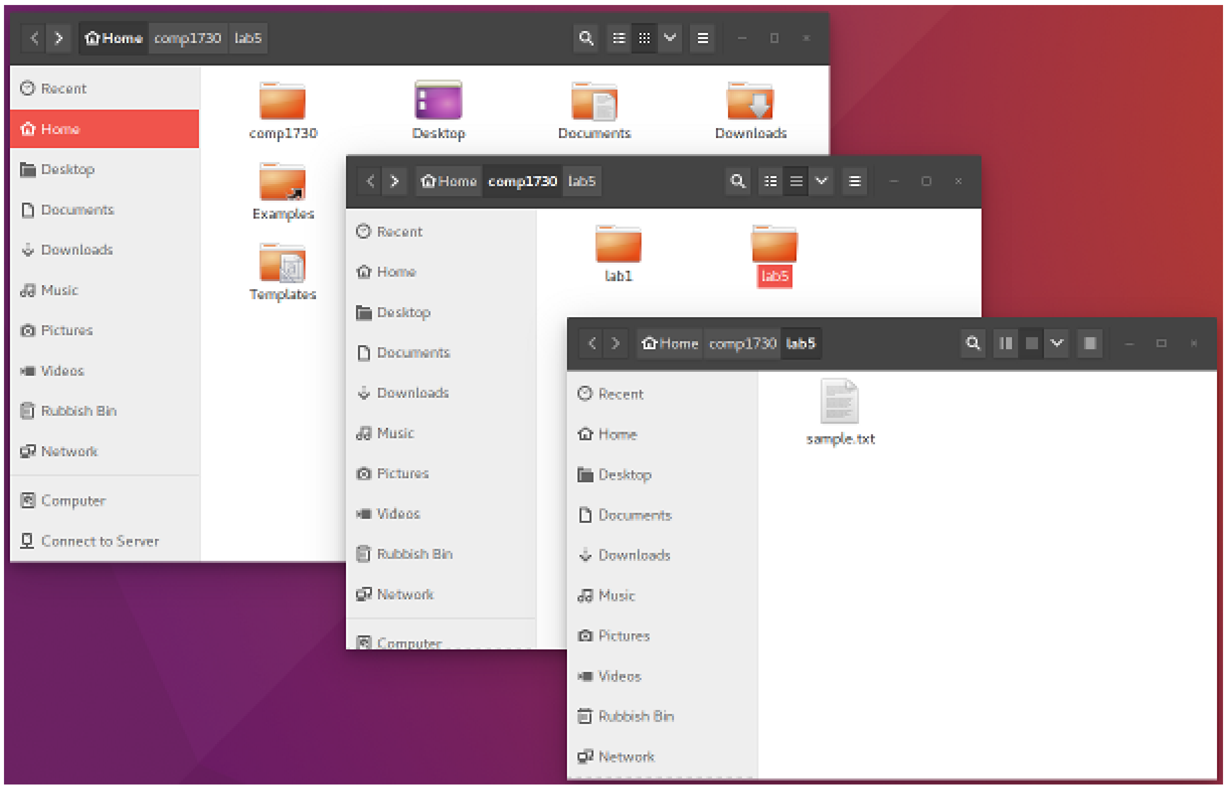

Figure 1: screenshot showing folders and files

You should find that the current working directory is now `'/students/uNNNNNNN/comp1730'`. The `os.chdir` ("change directory") function changes it.

The location given in the example above is *absolute*: it specifies the full path, from the top-level ("root") directory. A *relative* path specifies a location in the directory structure relative to the current working directory. Try

```
In [5]: os.chdir("lab7")

In [6]: os.getcwd()

Out [6]: ...

In [7]: os.chdir("..")

In [8]: os.getcwd()

Out [8]: ...
```

The path `..` means "the parent directory". So, for example, if your current working directory is `/students/uNNNNNNN/comp1730/lab7` and you also have a `lab1` directory in `comp1730`, you can change to it with

```
os.chdir('../lab1')
```

Finally, the `os.listdir` function returns a list of the files and subdirectories in a given directory. If you have created the text file `sample.txt` (and nothing else) in `comp1730/lab7`, then

```
os.listdir('/students/uNNNNNNN/comp1730/lab7')
```

should return `['sample.txt']`, while

```
os.listdir('..')
```

will return a list of the subdirectories and files in the parent of the current working directory.

Exercise 2(a): Reading a text file

To read the sample text file that you created in python, you can do the following:

```
In [1]: fileobj = open("sample.txt", "r")

In [2]: fileobj.readline()
```

```
Out [2]: ...

In [3]: fileobj.readline()

Out [3]: ...
```

(This assumes the current working directory is where the file is located, i.e., `'/students/uNNNNNNN/comp1730/lab7'`. If not, you need to give the (absolute or relative) path to the file as the first argument to `open`.) You can keep repeating `fileobj.readline()` as many times as you wish. Notice that each call returns the next line in the file: the file object keeps track of the next point in the file to read from. When you get to the end of the file, `readline()` returns an empty string. Also notice that each line has a newline character (`'\n'`) at the end, including empty lines in the file.

When you are done reading the file (whether you have reached the end of it or not), you must always close it:

```
In [4]: fileobj.close()
```

Exercise 2(b)

A more convenient way to iterate through the lines of a text file is using a `for` loop. The file object that is returned by the built-in function `open` is *iterable*, which means that you can use a for loop, like this:

```
for line in my_file_obj:
    # do something with the line
```

However, the file object is not a sequence, so you can't index it, or even ask for its length.

Write a function that takes as argument the path to a file, reads the file and returns the number of non-empty lines in it. You should use a `for` loop to iterate through the file.

Remember to *close the file* before the end of the function!

Programming problems

Note: We don't expect everyone to finish all these problems during the lab time. If you do not have time to finish these programming problems in the lab, you should continue working on them later (at home, in the CSIT labs after teaching hours, or on one of the computers available in the university libraries or other teaching spaces).

Reading CSV files

Comma-separated value (csv) files are a common format for storing tabular data in a text file. Each line in a csv file corresponds to one row in the table. (The first line may be a header, which gives the column names.) Cells within the row are separated with a comma. Each row has the same number of cells, which equals the number of columns in the table. As a further complication, values in the cells of a csv file can be enclosed in quotes (double or single); this allows commas to appear in the text in a cell. For a simple csv file (without quoting), we can use the string `split` method to separate each line of the file into cells, like this:

```
cells = line.split(',')
```

To deal with more complex csv files, the python standard library has a module called `csv`, which provides some helpful functions. In particular, it defines a data type called `reader`, which is a wrapper around the file object. This wrapper object is iterable, which means it can be used in a `for` loop. Instead of yielding one line from the file in each iteration, the csv reader yields a list with cell values. To use the csv reader, you can do as follows:

```
import csv
fileobj = open(...) # open file for reading as usual
reader = csv.reader(fileobj)
for row in reader:
    # process the row
    first_column = row[0] # note indexing from 0
    second_column = row[1]
    # ... etc
fileobj.close()
```

Note that when you're finished reading, you close the file object, not the reader object.

Here is a csv file which contains (somewhat outdated) information about countries of the world: countrylist.csv. It includes, among other things:

- The country's common name (column 1).
- Whether the country is an independent state, dependency, Antarctic territory, etc (columns 4 and 5).
- The name of the country's capital (column 6).
- The country's currency name (column 8) and abbreviation (column 7).
- The country's international dialing code (column 9).
- The ISO standard 2-letter (column 10) and 3-letter (column 11) code for the country.
- The country's internet top-level domain name (column 13).

(Column indices are from 0.) Download the file and save it to your working directory.

Making quiz questions Using the data from the countrylist.csv file, write a program that generates multiple-choise geography quiz questions. The output of your program should be questions like the following:

What is the capital of (some country)?

a) (some capital name)

b) (some capital name)

c) (some capital name)

where one of the choices (randomly selected which one) is the right answer and the other two are randomly chosen from among all the capital city names in the file. (If you want more variation, you can also generate reverse questions, like “What country is (some capital city) the capital of?”, or use other fields of information, such as “The currency (some currency name) is used in which of the following countries?” Take care, however, since many countries’ currencies have the same name!)

Your program should *read the file only once*, store the relevant bits of information from it in an appropriate data structure, and then be able to generate any number of quiz questions.

To select countries and cities at random, you will need to use the `random` module. In particular, the following functions are useful:

- `random.randint(a, b)` returns a random integer between `a` and `b`, inclusive. For example, `random.randint(0, 2)` will return 0, 1 or 2.
- `random.sample(a_list, number)` returns a list with `number` elements from the argument list chosen at random, without repetition.

If you want, you can make your program prompt the user to answer the question, and reveal the correct answer afterwards. To do this, use the `input` function, as in

```
answer = input("Your answer:")
```

You can make the program keep asking questions until the user decides to quit (for example, by entering a blank answer).

Reading in reverse

Files can only be read forward. When you read, for example a line from a text file, the *file position* advances to the beginning of the next line.

However, you can move the file position, using the method seek(pos) on the file object. File position 0 is the beginning of the file. The default is that pos is a positive integer offset from the beginning of the file, but there are also other seek modes (see the documentation). The method `tell()` returns the current file position. For example:

```
fileobj = open("my_text_file.txt")
line1 = fileobj.readline() # reads the first line
pos_ln_2 = fileobj.tell()  # file position of beginning of line 2
line2 = fileobj.readline()
line3 = fileobj.readline()
fileobj.seek(pos_ln_2)      # go back
line2b = fileobj.readline() # reads line 2 again
fileobj.seek(0)             # go back
line1b = fileobj.readline() # reads line 1 again
```

Write a program that reads a text file and prints its lines in reverse order. For example, if the file contents are

```
They sought it with thimbles, they sought it with care;
     They pursued it with forks and hope;
They threatened its life with a railway-share;
     They charmed it with smiles and soap.
```

then the output of your program should be

```
     They charmed it with smiles and soap.
They threatened its life with a railway-share;
     They pursued it with forks and hope;
They sought it with thimbles, they sought it with care;
```

- Can you write a program that does this while reading through the file, from beginning to end, only once?
- Can you write a program that does this without storing all lines in memory?

It is possible to do both, but it is not possible to do both at the same time.

Writing CSV files

In one of the earlier lectures, we wrote a program to simulate the first-stage flight of the Falcon 9 rocket. Here is a copy of the program that was written in the lecture.

In each iteration of the simulation, the program prints out some values (the simulation time, velocity and altitude, and remaining fuel mass). Modify the program so that instead of printing this information to the screen, the values for each time step are recorded as a line of comma-separated values in a CSV file. You can print a CSV file by just printing values separated by commas, or you can use the `writer` object from the `csv` module. You program should still print a message to the screen when the simulation begins and ends.

To look at the output, you can open the CSV file that your program has written in a text editor (like the program editor in your IDE) or using a spreadsheet program (such as excel or openoffice).

Reading image files

Portable pixmap, or *ppm*, is a simple, non-compressed image format. A ppm file can be stored in text or binary form; let's start with reading it in text form.

A text-form ppm file starts with the *magic string* `P3`. That means the first two characters in the file are always `P3`. This identifies the file format. After this comes three positive integers (each written out with the digits 0-9): the first two are the width and height of the image, and the third is the maximum colour value, which is less than or equal to 255. Then follows width times height time 3 integers (again, each written with digits 0-9); these represent the red, green and blue value for each pixel. The pixels are written left-to-right by row from the top, meaning the first triple of numbers is the colours of left-most pixel in the top row, then the second from the left in the top row, and so on.

Here is a small example:

```
P3
3 2
255
255   0   0
0   255   0
0     0 255
255 255   0
255 255 255
0     0   0
```

This image has width 3 and height 2, which means it has 6 pixels. All the numbers in the file are separated with whitespace, which can be one or more spaces (`' '`) or a newlines (`'\n'`). The format does not require that all pixels in a row are on one line. In the example above, they are written one pixel per line, but the following would also be a correct representation of the same image:

```
P3
3 2
255
255   0   0 0   255   0 0     0 255
255 255   0 255 255 255 0     0   0
```

To display the image after reading it, you can use the `imshow` function from the `matplotlib.pyplot` module. You will need to create a 3-dimensional `array`, where the sizes of the dimensions are the width, the height, and 3. The array entries at `i,j,0`, `i,j,1` and `i,j,2` are the red, green and blue colour values for the pixel at row `i` column `j` in the image. For example, to show the image above, you can do the following:

```
import numpy as np
import matplotlib.pyplot as mpl
```

```
image = np.zeros((2,3,3))     # create the image array (fill with 0s)
image[0,0,:] = 1.0, 0.0, 0.0  # RGB for top-left (0,0) pixel
image[0,1,:] = 0.0, 1.0, 0.0  # RGB for top-middle (1,0) pixel
image[0,2,:] = 0.0, 0.0, 1.0  # RGB for top-right (2,0) pixel
image[1,0,:] = 1.0, 1.0, 0.0  # RGB for row 2 left (0,1) pixel
image[1,1,:] = 1.0, 1.0, 1.0  # RGB for row 2 middle (1,1) pixel
image[1,2,:] = 0.0, 0.0, 0.0  # RGB for row 2 right (2,1) pixel
mpl.imshow(image, interpolation='none')
mpl.show()
```

Note that each of the colour values above (`1.0` or `0.0`) is the result of taking the corresponding colour value from the image file (255 or 0) and dividing it by the maximum colour value (255). The extra argument `interpolation='none'` to `imshow` disables image interpolation, which it may otherwise do if the image has low resolution.

To display the image read from the file, you need to create an array of the right size (as read from the image file) and replace the part that fills in the values above with some kind of loop that fills in the values read from the file.

Here are two image files (of the kind that the internet is most famous for) that you can test your program on: cat_picture_1.ppm, cat_picture_2.ppm.

Advanced: As mentioned above, the ppm format also has a binary form. It is very similar the text format, except that each colour value is stored as a single byte, rather than written out as text. The magic string for this format is `P6`. The width, heigh and max colour value are still written out with digits, and there should be a newline before the start of the binary image data.

Modify your program so that it can read images in both text and binary format. (To decide which format a given file is, you will need to open it and read the first two characters.) Note that to read the binary format correctly, you will have to open the file in binary mode.

Here are the two image files above encoded in the binary form: cat_picture_1_binary.ppm, cat_picture_2_binary.ppm.

Recursive file listing

As shown above, the `listdir` function from the `os` module returns a list with the names of all files and subdirectories in a given directory. (If called without an argument, `os.listdir()`, it returns the list of files and directories in the current working directory.)

Two other functions in the `os` module are os.path.isfile(path_str) and os.path.isdir(path_str). The first will return `True` if the argument path points to a file (and `False` otherwise) and the second will return `True` if the argument path names a directory. Using `os.listdir`, `os.path.isdir` and `os.path.isfile`, write a function that searches a directory, all its subdirectories, its subdirectories subdirectories, and so on, and returns a list of all the files found.

Hint: Recursion is a convenient way to solve this problem.