SENG265 - Lab 5 - Introduction to Python

Lab files

The files for lab 5 are located in `/home/zastre/seng265/lab-05` folder. Copy them to your git repository as you have done in previous labs.

Python vs Python 2 vs Python 3

The python language has evolved several times, and each major version is differs significantly from the others. We therefore need to be careful which version we're using. In this course, we will exclusively use python3 - whenever we refer to python, we mean python3.

You can see which version of python you're using with the –version option to the command line:

At the command line, try (the dollar sign represents the command prompt):

```
$ python --version
```

produces the output `Python 2.7.5` while the command:

```
$ python3 --version
```

gives us `Python 3.6.8`, which is newer. Always run your scripts using python3 rather than python.

Running the Python interpreter

We can run the python interpreter in order to execute a pre-written *python script*, or in interactive mode (in which commands are exectuted as they are typed).

Try launching the interpreter in interactive mode by running the `python3` program with no arguments:

```
$ python3
```

You'll see the python3 interactive prompt:

```
Python 3.6.8 (default, Nov 16 2020, 16:55:22)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-44)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python is waiting for you to enter a python instruction. Try printing `hello world` (you don't need to enter the `>>>` symbols - they indicate the python prompt):

```
>>> print("hello world")
```

You'll see the output is immediately printed, and the prompt waits for a new command to be typed. This interactive mode is a good way to test out instruction syntax.

Quit the interactive interpreter with `Ctrl-D` or with the function `exit()`

We will usually run the interpreter to execute a *python script*. Create a file named `hello.py` with the following contents:

```
print("hello world")
```

This can be executed with the command at the shell:

```
$ python3 hello.py
```

Making our script executable

In an earlier lab, we saw that we can set a file to *executable* by changing its permissions. Once we do so we can execute it as a program. To do this with our `hello.py` file, we also need to tell the shell that this is a python3 script, and not a shell script (remember that linux doesn't care about file extensions). To do this, we can add the following line to the top of our `hello.py` file:

```
#!/usr/bin/env python3
```

This line (which we call a *bang-path*) tells the shell that this file, when executed, should be run using the installed version of python3. Some scripts you may read use `#!/usr/bin/python3` instead, but this is dangerous - such scripts will fail on machines that don't install python3 to that location. By using the `env` program to tell us where the `python3` binary is, the above bang-path achieves greater portability.

(Bang paths can be used for scripts in other languages than just python; they're common in shell scripting generally).

Once you have the bang-path added to the program and the permissions set properly, you should be able to run `hello.py` at the command line like this:

```
./hello.py
```

Putting our code in functions

Writing all our code outside functions is considered bad programming practice. Let's improve our hello.py program by putting our instruction into a `main()` function, and making sure that `main()` is the first function run when our script is executed:

```
#! /usr/bin/env python3

def main():
    print("hello world")
```

```
if __name__ == "__main__":
    main()
```

Note that there are two underscores before '`name`', two underscores after it, and two both before and after '`main`' as well on the second-to-last line.

Exercise: pythag1.py

Open pythag1.py. This program uses hard-coded variables and the `math.sqrt()` function to compute the length of the hypotenuse. Note that although the variables a and b are integer-typed and c is a floating-point type, we don't need to actually specify the types explicitly - python can infer the types of variables from context.

Also note the two different ways we can mix variable values and string literals in our print statement. The first example shows strings interspersed with integer and floating-point valued variables, separated by commas. The second print statement shows formatted output more similar to C's printf() syntax.

Exercise: pythag2.py

This program is missing a pythag function. Try adding it. You can import the math library, or take advantage of the fact that taking the square-root of a number is equivalent to raising it to 0.5.

Exercise: pythag3.py

Instead of hard-coding the lengths of triangle sides, let's accept them as command line arguments. For example, if the program is run like this:

```
$ ./pythag3.py 3 4
```

The output should be:

```
Sides 3.0 and 4.0, hypotenuse 5.0000
```

Hints:

- you'll need to copy your pythag() function from pythag2.py into this program.
- sys.argv is a `list` of strings that works the same way as C's argv (try printing it to see what's in it).
- you can find the number of items in a list (or any other sequence) with the `len()` function.
- you can use the [] operator to retrieve items from a list, like you'd do in a C array.
- Each of the arguments is initially a string. You'll need get the floating-point number version of the argument using the float() function.

Exercise: split_text.py

Write a Python program to that takes a single string surrounded by single-quotes on the command line. The program should split the string into separate parts, where parts are separated by commas. It should print each part on a separate line. For example, if run like this:

```
$ split_text.py 'this,is a,bit,of text'
```

The program should produce:

```
this
is a
bit
of text
```

Use split_text.py as your foundation.

Hints:

- If you have a string in `s`, you can split it up on some `deliminator` with `s.split(deliminator)`.
- The result of splitting a string like this is a `list`.
- You can write a for loop that traverses a list, giving you each element in turn. For example, if `lines` is the name of a list:

```
for w in lines:
    # do something with w
```

Exercise: sort_unique_words.py

Take a look at `thesaurus.csv`. Each line of this file contains several words, separated by commas. These words are in no particular order, and there are duplicates.

Write a program using sort_unique_words.py as a basis. This program should produce a list of all the unique words in this file, sorted alphabetically, one word per line.

Hints:

- You can iterate through sys.stdin, which is a list of everything in standard input, with one string per line.
- You can add a string to a list with the append() function: For example, to add `word` to `lst`, you can use `lst.append(word)`.
- Lists can be sorted in-place by calling their built-in .sort() function, or you can return a sorted copy of a list with `sorted()`.
- Lists permit duplicates, but sets don't. You can convert back and forth between the two with the list() and set() functions.
- The expected output is in the sort_unique_words_expected.txt. Use the `diff` linux command to compare your output with this file.

Lab Participation

Take a screenshot of your source code for either `split_text.py` or `sort_unique_words.py` (your choice) and upload it to the `Lab 5 screenshot submission` assignment in the `Course Tools -> Assignments` section of Brightspace by midnight on Friday night.