CSE 130 - Programming Assignment #5

Python

130 points

(see submission instructions below)

(click your browser's refresh button to ensure that you have the most recent version)

(Programming Assignment #5 FAQ)

Note: To download and install Python version 2.7.1 on your home machines see this page. Remember that this is only to enable you to play with the assignment at home: The final version turned in must work on the ACS Linux machines. While you can use Windows to begin working with Python, the code you turn in must be that required for the Linux environment.

Integrity of Scholarship

University rules on integrity of scholarship will be strictly enforced. By completing this assignment, you implicitly agree to abide by the UCSD Policy on Integrity of Scholarship described in the Academic Regulations, in particular "all academic work will be done by the student to whom it is assigned, without unauthorized aid of any kind."

You are expected to do your own work on this assignment; there are no group projects in this course. You may (and are encouraged to) engage in general discussions with your classmates regarding the assignment, but specific details of a solution, including the solution itself, must always be your own work. Incidents that violate the University's rules on integrity of scholarship will be taken seriously: In addition to receiving a zero (0) on the assignment, students may also face other penalties, up to and including, expulsion from the University. Should you have any doubt about the moral and/or ethical implications of an activity associated with the completion of this assignment, please see the instructors.

Code Documentation and General Requirements

Code for all programming assignments should be **well documented**. A working program with no comments will **receive only partial credit**. Documentation entails providing documentation strings for all methods, classes, packages, etc., and comments throughout the code to explain the program logic. Comments in Python are preceded by `#` and extend to the end of the line. Documentation strings are strings in the first line of a function, method, etc., and are accessible using `help(foo)`, where foo is the name of the method, class, etc. It is understood that some of the exercises in this programming assignment require extremely little code and will not require extensive comments.

While few programming assignments pretend to mimic the "real" world, they may, nevertheless, contain some of the ambiguity that exists outside the classroom. If, for example, an assignment is amenable to differing interpretations, such that more than one algorithm may implement a correct solution to the

assignment, it is incumbent upon the programmer to document not only the functionality of the algorithm (and more broadly his/her interpretation of the program requirements), but to articulate **clearly** the reasoning behind a particular choice of solution.

Assignment Overview

The overall objective of this assignment is to introduce you to Python. Emphasis will be placed on text and string manipulations and should use some of Python's facilities for iterating over structures. The assignment is spread over three python files misc.py, crack.py, test.py, and three files words, passwd, and news.txt that you need to download. The first three files contain several skeleton Python functions, with missing bodies, i.e. expressions, which currently contain the text `raise Failure("to be written")`. Your task is to replace the text in those files with the the appropriate Python code for each of those expressions. An emphasis should be placed on writing concise easy to read code.

Assignment Testing and Evaluation

Your functions/programs **must** compile and/or run on a **Linux** ACS machine (e.g. `ieng6.ucsd.edu`), as this is where the verification of your solutions will occur. While you may develop your code on any system, ensure that your code runs as expected on an ACS machine prior to submission. You should test your code in the directories from which the zip files (see below) will be created, as this will approximate the environment used for grading the assignment.

Most of the points, except those for comments and style, will be **awarded automatically, by evaluating your functions against a given test suite**. The fourth file, `test.py` contains a very small suite of tests which gives you a flavor of these tests. At any stage, by typing at the UNIX shell :

```
python < test.py | grep "130>>" > log
```

you will get a report on how your code stacks up against the simple tests.

The last (or near the bottom) line of the file `log` **must contain the word "Compiled" otherwise you get a zero for the whole assignment**. If for some problem, you cannot get the code to compile, leave it as is with the `raise ...`, with your partial solution enclosed below as a comment. **There will be no exceptions to this rule**. The second last line of the log file will contain your overall score, and the other lines will give you a readout for each test. You are encouraged to try to understand the code in `test.py`, and subsequently devise your own tests and add them to `test.py`, but you will not be graded on this.

Alternately, inside the Python shell, type (user input is in red):

```
>>> import test
.
.
.
130>> Results: ...
130>> Compiled
```

and it should print a pair of integers, reflecting your score and the max possible score on the sample tests. If instead an error message appears, your code will receive a zero.

Submission Instructions

1. Create the zip file for submission

Your solutions to this assignment will be stored in separate files under a directory called `pa5_solution/`, inside which you will place the files: `misc.py, crack.py` . These two files listed are the versions of the corresponding supplied files that you will have modified. There should be **no other files in the directory**.

After creating and populating the directory as described above, create a zip file called `<LastName>_<FirstName>_cse130_pa5.zip` by going into the directory `pa5_solution` and executing the UNIX shell command:

```
zip <LastName>_<FirstName>_cse130_pa5.zip *
```

2. Test the zip file to check for its validity

Once you've created the zip file with your solutions, you will use the `validate_pa5` program to see whether your zip file's structure is well-formed to be inspected by our grading system by executing the UNIX shell command:

```
validate_pa5 <LastName>_<FirstName>_cse130_pa5.zip
```

The `validate_pa5` program will output `OK` if your zip file is well-formed and your solution is compiled. Otherwise, it will output some error messages. Before going to step 3, make sure that your zip file passes `validate_pa5` program. **Otherwise you get a zero for the whole assignment.** If you have any trouble with this, refer to the instructions in step 1.

3. Submit the zip file via the `turnin_pa5` program

Once you've created the zip file with your solutions, you will use the `turnin_pa5` program to submit this file for grading by going into the directory `pa5_solution/` and executing the UNIX shell command:

```
turnin_pa5 <LastName>_<FirstName>_cse130_pa5.zip
```

The `turnin_pa5` program will provide you with a confirmation of the submission process; make sure that the size of the file indicated by `turnin_pa5` matches the size of your zip file. (turnin_pa5 is a thin wrapper script around the ACMS command `turnin` that repeats validation and ensures that the propper assignment name is passed). Note that you may submit multiple times, but your latest submission overwrites previous submissions, and will be the ONLY one we grade. If you submit before the assignment deadline, and again afterwards, we will count it as if you only submitted after the deadline.

Hints

Using .py Files

To load `foo.py` into python, where `bar()` is defined in `foo.py` do the following:

```
import foo
foo.bar()
```

Modify `foo.py`, then to reload it:

```
import imp
imp.reload(foo)
foo.bar()
```

If you use the following syntax instead, you have to restart python to reload it.

```
from foo import *
bar()
```

Useful Links

- Python Tutorial
- Python FAQ
- Python Library Reference

Quirks / Features of Python

- `None` doesn't print a line when it is the return value
- `None, 0`, and `[]` all evaluate to false when used as a predicate.
- Many objects can be iterated over with `for` loops, not just lists.

Problem #1: Warm-Up (`misc.py`)

(a) 10 points

Write a function `closest_to(l,v)` , which returns the element of the list `l` closest in value to `v`. In the case of a tie, the first such element is returned. If l is empty, None is returned. Once implemented you should get the following behavior at the Python prompt:

```
>>> from misc import * # this will load everything in misc.py
>>> closest_to([2,4,8,9],7)
8
>>> closest_to([2,4,8,9],5)
4
```

(b) 10 points

Now, write a function `make_dict(keys,vals)` which takes a list of keys and a list of values and returns a dictionary (`dict`) pairing keys to corresponding values. Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> make_dict(["foo","baz"],["bar","blah"])
{'foo': 'bar', 'baz': 'blah'}
>>> make_dict([1],[100])
{1: 100}
```

(c) 10 points

Write a Python function `word_count(filename)` that takes a string, `filename`, and returns a dictionary mapping words to the number of times they occur in the file `filename`. For this function, a word is defined as a sequence of alphanumeric characters and underscore. All other characters should be ignored. Words should be returned in lower case, and case should be ignored when counting occurrences. Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> word_count("news.txt")
{'all': 2, 'code': 2, 'gupta': 1, 'results': 1, 'four': 1, 'edu': 2, ...}
```

Problem #2: Password Cracking (`crack.py`)

For this problem, you will write a simple dictionary based password cracker. This should primarily be an exercise in string manipulation and data structures. You will be attempting to crack as many passwords as possible in a UNIX password file. The passwords are encrypted using the UNIX `crypt()` function. For more information see `man 5 passwd` and `man 3 crypt` on UNIX or Linux. In addition, more information will be provided when used.

(a) 10 points

Write a Python function `load_words(filename,regexp)` which loads the words from the file `filename` which match the regular expression `regexp`. There will be one word per line in the input file. The resulting words should be returned in a list in the same order they occur in the input file.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> from crack import *
>>> load_words("words",r"^[A-Z].{2}$")
['A-1', 'AAA', 'AAE', ...]
>>> load_words("words",r"^xYx.*$")
[]
```

(b) 5 + 10 + 15 points

Write three python functions to implement the following string transformations: `transform_reverse`, `transform_capitalize`, and `transform_digits`. Each will take a string as an argument and return a list of strings. The order of returned strings does not matter.

The first function, `transform_reverse(str)` should return a list with the original string and the reversal of the original string.

The second function, `transform_capitalize(str)` should return a list of all the possible ways to capitalize the input string.

Finally, the third function, `transform_digits(str)` should return return a list of all possible ways to replace letters with similar looking digits according to the following mappings. This should be done without regard to the capitalization of the input string, however when a character is not replaced with a digit, it's capitalization should be preserved.

o -> 0	i,l -> 1
z -> 2	e -> 3
a -> 4	s -> 5
b -> 6	t -> 7
b -> 8	g,q -> 9

Once you have implemented the function, you should get the following behavior at the Python prompt (order of list elements is not important):

```
>>> transform_reverse("Moose")
['Moose','esooM']
>>> transform_capitalize("foo")
['foo', 'Foo', 'fOo', 'FOo', 'foO', 'FoO', 'fOO', 'FOO']
>>> transform_digits("Bow")
['Bow', 'B0w', '6ow', '60w', '8ow', '80w']
```

(c) 10 points

Write a function `check_pass(plain,enc)` which will take a plain-text password, `plain`, and an encrypted password `enc` and will return `True` if `plain` encrypts to `enc` using the function crypt.

Crypt is available in Python in the `crypt` module. You can load this modules with `import crypt`. Then the crypt function is available as `crypt.crypt()`. See `help(crypt.crypt)` in Python. When crypt is used, a salt must be provided to perturb the algorithm. This salt is a random two character string which is returned in the beginning of the password. Since you are going in the other direction, you should extract the salt from the first two characters of the encrypted password (or see here for more information).

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> check_pass("asarta","IqAFDoIjL2cDs")
True
>>> check_pass("foo","AAbcdbcdzyxzy")
False
```

(d) 10 points

Now, write a Python function `load_passwd(filename)` which takes a string as input and returns a list of dictionaries with fields "account", "password", "UID", "GID", "GECOS", "directory", and "shell", each mapping to the corresponding field of the file. The UID and GID fields should be returned as integers. The password file contains lines of the following format (see here for more information):

```
account:password:UID:GID:GECOS:directory:shell
```

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> load_passwd("passwd")
[{'account': 'root', 'shell': '/bin/bash', 'UID': 0, 'GID': 0, 'GECOS': 'Corema
Latterll', 'directory': '/home/root', 'password': 'VgzdTLne0kfs6'},
  ... ]
```

Hint: The function `re.split` may be of use.

(e) 15 points + 25 points fractional credit

Finally, write a function `crack_pass_file(filename,words,out)` which takes two strings corresponding to a password file and a file with a list of words and a string corresponding to an output file. Attempt to crack as many of the passwords in the password file as possible. As passwords are discovered, they should be written to the output file in the following format:

username=pass

for example,

```
checani=asarta
```

After each line is written, the output file should be flushed. Your function will only be allowed to run for a limited amount of time (several minutes). Thus, you should attempt to crack the easiest passwords first. All passwords in the input file are between 6 and 8 characters and can be cracked using words in the provided dictionary along with a combination of the transformations above. The base points will be provided if all the passwords which are untransformed strings can be cracked. Fractional credit will be proportional to the number of additional passwords cracked, with *the student cracking the most extra words getting the full 25 points*.

Once you have implemented the function, you should get the following behavior at the Python prompt:

```
>>> crack_pass_file("passwd","words","out")
Several minutes pass...
Ctrl-C
Traceback (most recent call last):
...
KeyboardInterrupt
```

The file `out` should contain something like the following:

```
checani=asarta
root=...
...=...
...=...
```