New York University
School of Continuing and Professional Studies
Management and Information Technology

Advanced Python

Homework Topic Discussion, Session 2

2.1 Use set comparisons to perform argument validation.

a. **sys.argv** allows arguments to be passed at the command line.

At the command line (this particular example is unrelated to the homework):

```
$ python myscript.py arg1 arg2 arg2
```

In the script:

```
import sys

print sys.argv          # ['myscript.py', 'arg1', 'arg2', 'arg3']
```

As you can see, the arguments are collected in a list, called **sys.argv**. The first element of the list is the program name itself, and the subsequent elements are the arguments passed at the command line.

Also note that arguments are delimited on the command line by whitespace. **sys.argv** uses whitespace to separate arguments into list items.

b. The argument splitting and loading into your arg dict should be done in as few lines as possible. Here are the features you'll need (**variable names have been chosen randomly**):

```
# splitting on '='
key, val = arg.split('=')

# setting and key and val in a dict
argdict[key] = val
```

Of course, the user may make a mistake which would mess up the program's proper functionining if you don't account for the possibility. Use **try:** with **except SomeException:** to trap the error that would occur if the user puts in bad arguments (**however, be sure to specify the proper exception in place of SomeException**).

As mentioned, it is possible to do this arg splitting into a dict using a ***single list comprehension*** *along with the* ***dict()*** *function.* Can you do it for bragging rights?

c. Validating the arguments: remember that **set.difference()** will tell you which elements are in a set that are not in another set.

```
x = set(['a', 'b', 'c'])
y = ['b', 'c']
x.difference(y)            # set(['a'])
```

We have two validations to perform: detect fields that are required (i.e., present in the set of required fields) that are not present in the user input, and detect fields that are present in the user input, but not present in the set of valid fields. You should be able to use **set.difference()** to perform these detections **without looping**.

Hint: remember that anytime you use a **dict** in a "listy" sort fo way, it will return its keys. So it's pretty easy to convert a dict's keys into a set.

d. Adding 'subject' and 'body' key/values to your arg dict if they aren't specified by the user: you can test to see if a dict key is in a dict with in:

```
if 'subject' not in mydict:
```

Or perhaps better, consider that adding a duplicate key to a dict overwrites that key. Maybe set 'subject' and 'body' in the arg dict first, and *then* load it with the user's specified args?

e. Reading from STDIN: as you know our program reads from STDIN using **sys.stdin**. Consistent with Python's "unity of syntax", **sys.stdin** can be read much like a file -- it supports **read()**, **readlines()** and **for** looping. If no body is passed to STDIN, issue a warning to **sys.stderr** and continue with an 'empty body'.

f. Issuing a warning: if there is no body specified, we don't need to exit() -- instead, we can simply issue a warning. However, warnings must be output through the **STDERR** output stream. To do this, write to **sys.stderr**.

g. Building the **sendmail** string: in order to create a string with newlines in it, it's usually easiest to use a triple-quoted string, which allows multi-line strings directly in your code. Consider this example:

```
template = """Line {0}
Line {1}
Line {2}"""

mystr = template.format('a', 5, 'hello')
print mystr           ###  Line a
                      ###  Line 5
                      ###  Line hello
```

2.2 Show "top n" files in a directory (or directory tree), sorted by file criteria.

a. Building a dict of dicts: the key for your "outer" dict should be the unique pathname to the file. The "inner" dict will contain keys for basename (i.e., the name of the file without the directory), size and modification time. Since you will know all of these at the same time, you can create the "inner" dict in one statement and in fact can also assign it to the "outer" dict (which contains filepaths associated with these "inner" dicts) in the same statement. Here's the pattern:

```
outerdict[filepath] = { 'name': filename,
                        'size':  filesize,
                        'mtime':  filemodtime }
```

b. To capture arguments, use **sys.argv** if desired, or for more control over arguments use the **argparse** module. See the description for **argparse** in the **File and Directory I/O** slide deck. **argparse** can:

   i. require some arguments and allow others to be optional

   ii. require that an argument be of a certain type

   iii. require than argument be one of a list of values

   iv. declare a default for a non-optional argument

c. Remember that when working with filenames obtained from **os.listdir()** or **os.walk()** you must prepend a filename with its directory. Otherwise Python will not know where to find the file based on its basename alone (which won't work unless your current directory and the file directory are the same). We use **os.path.join()** for this purpose.

```
import os
searchdir = '/my/home'
for file in os.listdir(searchdir):
    fullpath = os.path.join(searchdir, file)
```

d. Sorting a dict of dicts by inner value: sorting a dictionary means sorting the keys. Once we have the sorted keys we can then loop through and print keys and values. For example, this statement returns the keys of **mydict**, sorted alphabetically:

```
mydict = {'z': 1, 'b': 2}
sorted_keys = sorted(mydict)
print sorted_keys                  ['b', 'z']
```

When writing a sort modification function, remember that such a function takes *the thing to be sorted* (in this case, a key) and returns *the value by which it should be sorted*.

This means that the sort modification function will take a key from the dict as the argument and return the inner value associated with name, or size, or mtime.

If this is confusing, remember that the sort modification function doesn't actually do any sorting. It simply takes an item (i.e., any item, but only one) to be sorted and returns the value by which it should be sorted.

e. Converting epoch seconds to readable time format -- look up **time.ctime()**

f. Suggested functions -- these are provided as suggestions, not a requirement. You are free to develop your own design ideas, keeping in mind the advantages of modularity and clarity, and also feel free to use one or more of these suggestions in your design.

i. a function to acquire and validate arguments. It's likely that if you use **argparse** that the configuration values you set with **parser.add_argument()** can provide all the needed validation. This function would read from **argparse** and return an object containing the argument values (as described in the **argparse** discussion).

ii. a function to get a list of files to be sorted. This function would take the directory to be searched, and then could do the work of reading from the directory (handling an unreadable directory) and appending each filename to the filepath for that file. It would return a list of filepaths.

iii. a function for building the dict of filepaths associated with a dict of attributes for that file (see **a.** above): this would take the list of filepaths and return the dict of dicts.

iv. a function for displaying sorted results: this would take the dict of dicts, the sort type (i.e. by size, name, mtime), the number of results and the sort direction (ascending or descending) and would simply print out the results (since you can't sort a dictionary, it doesn't make sense to try to return the "sorted" dict).

To summarize, here is my **main()** function based on the suggestions above. I want to stress that this is just one way to do it -- it may make more sense to you to build out even more functions, or use fewer, etc. Your choice!

```
def main():
    args = get_args()
    files = get_files(args.dir)  # or args['dir'] if building
                                 # a dict from sys.argv
    file_dod = get_file_attrs(files)
    display_sorted_results(file_dod, args.by, args.results, args.direction)
```

2.3 (extra credit) Aggregate bit.ly data.

a. The bitly.tsv file is tab-separated data. This means that each line must be split on a *tab*, not just on whitespace:

```
for line in open('bitly.tsv'):
    els = line.split('\t')
```

Be careful not to **rstrip()** the line on whitespace, or you'll eliminate any tabs at the end! Instead, you may **rstrip('\n')**, which will strip just the newline character.

b. A **set** container by its very nature will always contain a unique collection of values. So simply using **set.add()** to add elements to a set will render a unique set. There

is no need to test for existing elements because the **set** will by default elimiate duplicates.

c. The standard idiom for creating a "key count" in a dictionary is to add 1 to the value associated with the key if the key exists, or to set the value associated to the key as 1 if the key didn't previousy exist:

```
if key in mydict:
    mydict[key] += 1
else:
    mydict[key] = 1
```

d. Standard sorting will sort a sequence by numerical value or by string value (depending on the value type). A **dict** will sort by its keys, but can be made to sort its keys by its associated values using the **dict.get()** method, which is passed as a method (**not** as an actual call) to the **sorted()** function:

```
sorted_keys_by_value = sorted(mydict, key=mydict.get)
```

e. Deriving the **machine_name** from the **long_url** variable would be most elegantly achieved using regular expressions, but it should be just as easily done with **str.split()** using the forward slash as the delimeter.

```
long_url = 'http://www.nasa.gov/mlmda/vdglry/index.html'
machine = long_url.split('/')[2]
print machine                          # www.nasa.gov
```