1. Basic Python Exercises

Basic Python Exercises

There are 3 exercises that go with the first sections of Google's Python class. They are located in the "basic" directory within the google-python-exercises directory. Download the google-python-exercises.zip if you have not already (see the Set-Up page for details).

- **string1.py** -- complete the string functions in string1.py, based on the material in the Python Strings section (additional exercises available in string2.py)
- **list1.py** -- complete the list functions in list1.py, based on the material in the Python Lists and Python Sorting sections (additional exercises available in list2.py)
- **wordcount.py** -- this larger, summary exercise in wordcount.py combines all the basic Python material in the above sections plus Python Dicts and Files (a second exercise is available in mimic.py)

With all the exercises, you can take a look at our solution code inside the solution subdirectory.

Last updated December 13, 2012.

2. Baby Names Python Exercise

Source URL: https://developers.google.com/edu/python/exercises/baby-names

Baby Names Python Exercise

The Social Security administration has this neat data by year of what names are most popular for babies born that year in the USA (see social security baby names).

The files for this exercise are in the "babynames" directory inside google-python-exercises (download the google-python-exercises.zip if you have not already, see Set Up for details). Add your code in babynames.py. The files baby1990.html baby1992.html ... contain raw html, similar to what you get visiting the above social security site. Take a look at the html and think about how you might scrape the data out of it.

Part A

In the babynames.py file, implement the extract_names(filename) function which takes the filename of a baby1990.html file and returns the data from the file as a single list -- the year string at the start of the list followed by the name-rank strings in alphabetical order. ['2006', 'Aaliyah 91', 'Abagail 895', 'Aaron 57', ...]. Modify main() so it calls your extract_names() function and prints what it returns (main already has the code for the command line argument parsing). If you get stuck working out the regular expressions for the year and each name, solution regular expression patterns are shown at the end of this document. Note that for parsing webpages in general, regular expressions don't do a good job, but these webpages have a simple and consistent format.

Rather than treat the boy and girl names separately, we'll just lump them all together. In some years, a name appears more than once in the html, but we'll just use one number per name. Optional: make the algorithm smart about this case and choose whichever number is smaller.

Build the program as a series of small milestones, getting each step to run/print something before trying the next step. This is the pattern used by experienced programmers -- build a series of incremental milestones, each with some output to check, rather than building the whole program in one huge step.

Printing the data you have at the end of one milestone helps you think about how to re-structure that data for the next milestone. Python is well suited to this style of incremental development. For example, first get it to the point where it extracts and prints the year and calls sys.exit(0). Here are some suggested milestones:

- Extract all the text from the file and print it
- Find and extract the year and print it
- Extract the names and rank numbers and print them
- Get the names data into a dict and print it
- Build the [year, 'name rank', ...] list and print it
- Fix main() to use the ExtractNames list

Earlier we have had functions just print to standard out. It's more re-usable to have the function *return* the extracted data, so then the caller has the choice to print it or do something else with it. (You can still print directly from inside your functions for your little experiments during development.)

Have main() call extract_names() for each command line arg and print a text summary. To make the list into a reasonable looking summary text, here's a clever use of *join*: `text = '\n'.join(mylist) + '\n'`

The summary text should look like this for each file:

```
2006
Aaliyah 91
Aaron 57
Abagail 895
Abbey 695
Abbie 650
...
```

Part B

Suppose instead of printing the text to standard out, we want to write files containing the text. If the flag --summaryfile is present, do the following: for each input file 'foo.html', instead of printing to standard output, write a new file 'foo.html.summary' that contains the summary text for that file.

Once the --summaryfile feature is working, run the program on all the fiels using * like this: "./babynames.py --summaryfile baby*.html". This generates all the summaries in one step. (The standard behavior of the shell is that it expands the "baby*.html" pattern into the list of matching filenames, and then the shell runs babynames.py, passing in all those filenames in the sys.argv list.)

With the data organized into summary files, you can see patterns over time with shell commands, like this:

```
$ grep 'Trinity ' *.summary
$ grep 'Nick ' *.summary
$ grep 'Miguel ' *.summary
$ grep 'Emily ' *.summary
```

Regular expression hints -- year: r'Popularity\sin\s(\d\d\d\d)' names: r'<td>(\d+)</td><td>(\w+)</td>\<td>(\w+)</td>'

Last updated January 21, 2015.

3. Copy Special Python Exercise

Source URL: https://developers.google.com/edu/python/exercises/copy-special

Copy Special Python Exercise

The Copy Special exercise goes with the file-system and external commands material in the Python Utilities section. This exercise is in the "copyspecial" directory within google-python-exercises (download google-python-exercises.zip if you have not already, see Set Up for details). Add your code in copyspecial.py.

The copyspecial.py program takes one or more directories as its arguments. We'll say that a "special" file is one where the name contains the pattern __w__ somewhere, where the w is one or more word chars. The provided main() includes code to parse the command line arguments, but the rest is up to you. Write functions to implement the features below and modify main() to call your functions.

Suggested functions for your solution(details below):

- get_special_paths(dir) -- returns a list of the absolute paths of the special files in the given directory
- copy_to(paths, dir) given a list of paths, copies those files into the given directory
- zip_to(paths, zippath) given a list of paths, zip those files up into the given zipfile

Part A (manipulating file paths)

Gather a list of the absolute paths of the special files in all the directories. In the simplest case, just print that list (here the "." after the command is a single argument indicating the current directory). Print one absolute path per line.

```
$ ./copyspecial.py .
/Users/nparlante/pycourse/day2/xyz__hello__.txt
/Users/nparlante/pycourse/day2/zz__something__.jpg
```

We'll assume that names are not repeated across the directories (optional: check that assumption and error out if it's violated).

Part B (file copying)

If the "--todir dir" option is present at the start of the command line, do not print anything and instead copy the files to the given directory, creating it if necessary. Use the python module "shutil" for file copying.

```
$ ./copyspecial.py --todir /tmp/fooby .
$ ls /tmp/fooby
xyz__hello__.txt        zz__something__.jpg
```

Part C (calling an external program)

If the "--tozip zipfile" option is present at the start of the command line, run this command: "zip -j zipfile <list all the files>". This will create a zipfile containing the files. Just for fun/reassurance, also print the command line you are going to do first (as shown in lecture). (Windows note: windows does not come with a program to produce standard .zip archives by default, but you can get download the free and open zip program from www.info-zip.org.)

```
$ ./copyspecial.py --tozip tmp.zip .

Command I'm going to do:zip -j tmp.zip /Users/nparlante/pycourse/day2/xyz__hello__.txt
/Users/nparlante/pycourse/day2/zz__something__.jpg
```

If the child process exits with an error code, exit with an error code and print the command's output. Test this by trying to write a zip file to a directory that does not exist.

```
$ ./copyspecial.py --tozip /no/way.zip .

Command I'm going to do:zip -j /no/way.zip /Users/nparlante/pycourse/day2/xyz__hello__.txt
/Users/nparlante/pycourse/day2/zz__something__.jpg


zip I/O error: No such file or directory

zip error: Could not create output file (/no/way.zip)
```


Last updated December 13, 2012.

4. Log Puzzle Python Exercise

Source URL: https://developers.google.com/edu/python/exercises/log-puzzle

Log Puzzle Python Exercise

For the Log Puzzle exercise, you'll use Python code to solve two puzzles. This exercise uses the urllib module, as shown in the Python Utilities section. The files for this exercise are in the "logpuzzle" directory inside google-python-exercises (download the google-python-exercises.zip if you have not already, see Set Up for details). Add your code to the "logpuzzle.py" file.

An image of an animal has been broken it into many narrow vertical stripe images. The stripe images are on the internet somewhere, each with its own url. The urls are hidden in a web server log file. Your mission is to find the urls and download all image stripes to re-create the original image.

The slice urls are hidden inside apache log files (the open source apache web server is the most widely used server on the internet). Each log file is from some server, and the desired slice urls are hidden within the logs. The log file encodes what server it comes from like this: the log file animal_code.google.com is from the code.google.com server (formally, we'll say that the server name is whatever follows the first underbar). The animial_code.google.com log file contains the data for the "animal" puzzle image. Although the data in the log files has the syntax of a real apache web server, the data beyond what's needed for the puzzle is randomized data from a real log file.

Here is what a single line from the log file looks like (this really is what apache log files look like):

```
10.254.254.28 - - [06/Aug/2007:00:14:08 -0700] "GET /foo/talks/ HTTP/1.1"
200 5910 "-" "Mozilla/5.0 (X11; U; Linux i686 (x86_64); en-US; rv:1.8.1.4) Gecko/20070515 Firefox/2.0.0.4"
```

The first few numbers are the address of the requesting browser. The most interesting part is the "GET *path* HTTP" showing the path of a web request received by the server. The path itself never contain spaces, and is separated from the GET and HTTP by spaces (regex suggestion: \S (upper case S) matches any non-space char). Find the lines in the log where the string "puzzle" appears inside the path, ignoring the many other lines in the log.

Part A - Log File To Urls

Complete the read_urls(filename) function that extracts the puzzle urls from inside a logfile. Find all the "puzzle" path urls in the logfile. Combine the path from each url with the server name from the filename to form a full url, e.g. "http://www.example.com/path/puzzle/from/inside/file" . Screen out urls that appear more than once. The read_urls() function should return the list of full urls, sorted into alphabetical order and without duplicates. Taking the urls in alphabetical order will yield the image slices in the correct left-to-right order to re-create the original animal image. In the simplest case, main() should just print the urls, one per line.

```
$ ./logpuzzle.py animal_code.google.com
http://code.google.com/something/puzzle-animal-baaa.jpg
http://code.google.com/something/puzzle-animal-baab.jpg
...
```

Part B - Download Images Puzzle

Complete the download_images() function which takes a sorted list of urls and a directory. Download the image from each url into the given directory, creating the directory first if necessary (see the "os" module to create a directory, and "urllib.urlretrieve()" for downloading a url). Name the local image files with a simple scheme like "img0", "img1", "img2", and so on. You may wish to print a little "Retrieving..." status output line while downloading each image since it can be slow and its nice to have some indication that the program is working. Each image is a little vertical slice from the original. How to put the slices together to re-create the original? It can be solved nicely with a little html (knowledge of HTML is not required).

The download_images() function should also create an index.html file in the directory with an "img" tag to show each local image file. The img tags should all be on one line together without separation. In this way, the browser displays all the slices together seamlessly. You do not need knowledge of HTML to do this; just create an index.html file that looks like this:

```
<verbatim>
<html>
<body>
<img src="/edu/python/exercises/img0"><img src="/edu/python/exercises/img1"><img src="/edu/python/exercises/img2">...
</body>
</html>
```

Here's what it should look like when you can download the animal puzzle:

```
$ ./logpuzzle.py --todir animaldir animal_code.google.com
$ ls animaldir
img0  img1  img2  img3  img4  img5  img6  img7  img8  img9  index.html
```

When it's all working, opening the index.html in a browser should reveal the original animal image. What is the animal in the image?

Part C - Image Slice Descrambling

The second puzzle involves an image of a very famous place, but depends on some custom sorting. For the first puzzle, the urls can be sorted alphabetically to order the images correctly. In the sort, the whole url is used. However, we'll say that if the url ends in the pattern "*-wordchars-wordchars*.jpg", e.g. "http://example.com/foo/puzzle/bar-abab-baaa.jpg", then the url should be represented by the **second** word in the sort (e.g. "baaa"). So sorting a list of urls each ending with the word-word.jpg pattern should order the urls by the second word.

Extend your code to order such urls properly, and then you should be able to decode the second place_code.google.com puzzle which shows a famous place. What place does it show?