Lab 6: Python Strings and Beyond

CS 320 Principles of Programming Languages, Fall Term 2019
Department of Computer Science, Portland State University

Learning Objectives

Upon successful completion, students will be able to:

- Write simple Python programs involving strings, lists, sets, maps, iterators, and file I/O.
- Contrast Python and Java treatment of types.

Instructions

Download the file `lab6.zip` from D2L into any convenient directory and type

```
$ unzip lab6.zip
```

You'll see a new directory `lab6` with a set of sub-directories. As usual, work through these directories, in sequence.

To do this lab, you'll need to have access to a working verion of the `python3` interpreter. This is available at the command line on the linuxlab systems, and is also easy to install on your own system (see `https://www.python.org/downloads/`). Make sure to use Python3 rather than the older (but still widely used) Python2.

A slide deck containing a quick summary of relevant Python features is available at `http://www.cs.pdx.edu/~apt/cs320/Python.pdf`.

This week, *all* the directories contain exercises. The earlier directories contain Python warm-up exercises borrowed from Google's Python class (`https://developers.google.com/edu/python/`). You should attempt all these exercises and then check your solutions against the ones provided. The remaining directories, whose names end in `X`, contain additional exercises for which solutions are not provided; your solutions to these will be graded for correctness in the usual way. When you have completed all the exercises, return to the top-level directory above `lab6` and type

```
$ zip -r sol6.zip lab6/*X*/*
```

Then submit the resulting file `sol6.zip` to D2L.

Strings (00)

Complete the exercises in the file called `strings.py`, and run against the test cases provided by invoking

```
$ python3 strings.py
```

Sample solutions are in the `sols` sub-directory. But don't look at these until you've tried to solve the exercises yourself.

The necessary Python background for these exercises is in the Python tutorial `https://docs.python.org/3/tutorial/` and library documentation `https://docs.python.org/3/library/`. Specific sections to read (or skim) are listed below.

Reading:

- Tutorial 3.1.2 Strings
- Tutorial 4.1 If
- Tutorial 4.6 Functions
- Library 4.7.1 String Methods

These exercises are taken from Google's Python class (`https://developers.google.com/edu/python/`). If you want additional background, you may want to look at the `Python Strings` section of that course and possibly watch the associated video (1.1). Warning: that class uses Python2; there are some minor differences.

Here's a quick reference for some operations on strings:

```
len(str)         # return the length of the specified string
str1 + str2      # concatenate strings
str * n          # make a new string with n copies of given string
x in str         # determine if x appears in the given string
str[i]           # return the ith element in str (starts at i=0)
str[-i]          # return the ith element from the end of str
str[i:]          # the "slice" of str starting at index i
str[:j]          # the "slice" of str up to but not including index j
str[i:j]         # the "slice" of str from index i up to (not including) j
str1.find(str2)  # find the position of str2 in str1 (or return -1)
```

Lists (01)

Similar instructions as for (00), for file `lists.py`.

Reading:

- Tutorial 3.1.3 Lists
- Tutorial 4.2 For
- Tutorial 4.3 The range() function
- Tutorial 5.1 (but not its subsections) Lists
- Tutorial 5.2 Del
- Tutorial 5.3 Tuples and Sequences

The associated Google class material if you need it is in sections `Lists` and `Sorting`, video section 1.2.

Quick reference of operations on lists: Lists support the same operations as strings, but because they are mutable, they also support some additional operations that perform "destructive" updates on list structures:

```
list.append(x)    # add x to the end of the given list
list1 += list2    # append elements in list2 to list1
list.pop()        # delete and return last element
list.pop(i)       # delete and return element i from the given list
list.reverse()    # reverse the elements in the given list
list.sort()       # sort the elements in the given list
list.sort(key=k)  # sort the list using the values returned by
                  #  applying the function k to each list element
```

Note also that the slice operations on lists return new lists and do not modify the original lists. In particular, `list[:]` creates a fresh copy of the given list.

Dicts and Files (02)

Similar instructions as before, for files `wordcount.py` and `mimic.py`

Read:

- Tutorial 5.4 Sets
- Tutorial 5.5 Dicts
- Tutorial 5.6 Looping Techniques
- Tutorial 5.7 More on Conditions
- Tutorial 6.1, 6.1.1. Modules
- Tutorial 7.2 Reading and Writing Files
- Tutorial 7.2.1 Methods of File Objects

The associated Google class material is in `Dicts and Files`, video 1.3.

Graphs, Revisited (03X)

In this exercise, you will revisit the undirected graph manipulation code that you worked on in Java in lab 3, and rewrite parts of it in Python. The purpose of this exercise is to compare the Java and Python approaches to supporting data structures such as lists, sets, and maps.

Roughly speaking, you need to reproduce the functionality of `Graph.java`, `AdjSetGraph.java` and `GraphUtils.java` as they appear in the solution to lab 4 exercises 8X and 10X. This involves being able to read and construct a graph from either the `.ig` or `.sg` formats, and then to produce a corresponding `.dot` file. A sample solution to lab 4 is available on D2L, and you should use that as the basis of your work in this exercise if you have any doubts about the correctness of your own code for lab 4.

It would be possible, and even plausibly Pythonic, to reproduce the class structure of the Java `Graph` in Python, minus an interface file, of course. But to emphasize the point that you can get a lot done in Python without using OO techniques, you should *not* do that here. Instead, just directly construct and use a Python `dict` with vertex keys and (set of vertex) values, where vertices can be type of values (even heterogenous ones!)

A skeleton main program is provided in `graph.py`; you just need to fill in the missing routines. Test on the usual files (provided here again) plus any you invented for lab 4.

CSV Files (04X)

In this section, you will write a Python program `bargains.py` that reads a file of purchase information in CSV (comma separated value) format and reports which purchases are "bargains."

The first line of each data file contains a single positive integer n. This line is followed by n lines each of the form

quantity`,`*name*`,`*grade*`,`*color*`,`*price*

where *quantity* is a positive integer, *name*, *grade*, and *color* are strings (containing arbitrary characters other than commas), and *price* is a floating point number with two digits to the right of the decimal point. Each line describes a *purchase*. For the purposes of this exercise, the *cost* of a purchase is its quantity multiplied by its price, and a purchase is a *bargain* if (i) its cost is no greater than 1.5 times the average cost of all purchases in the file; and (ii) its grade is "`superior`" or its color is "`blue`".

You will find a collection of sample input files, with extension `.input`, in the directory. Corresponding expected output files are also present, with extension name `.bargains`. For example, the expected output for `bargains.py` on file `example1.input` is in `example1.bargains`.

The program should take a single command line argument containing the name of the CSV file to read. You may assume that the input file is well-formed (e.g. that the number of lines is correct, there are four fields on each line after the first, etc.). The program reads all purchases from the input file. The program prints to standard output the *names* of any purchases that are bargains (according to the definition above), with one name per line.

OS files (05X)

Complete the program `find.py` in this subdirectory, by writing the functions `find` and `copy`. The behavior of this program is as follows:

`find root target` takes a directory name `root` and a simple filename (without slashes) `target` and searches in the subtree rooted at `root` for all files whose simple name is `target`. It prints out a list of the absolute pathnames of these files. (If you are not familiar with the concept of absolute path names, try looking at `https://www.geeksforgeeks.org/absolute-relative-pathnames-unix/`). If `root` is not a valid directory name, the program should print a message and exit immediately.

`find --copy newdir root target` does the same search, but in addition to printing out the names, it creates a new directory `newdir` and copies each of the found files to this directory. The target name for the copied file should be formed from the absolute path name by removing the leading slash (`/`) and replacing all other slashes with underscores(`_`). It is ok to raise an exception if `newdir` already exists or cannot be created for some reason.

You will need to consult documentation for the `os` package (including the `os.path` subpackage) and the

`shutil` package. These can be found in the standard Python documentation.

As an example, suppose you have the following directory tree in your filespace:

```
$ ls -R /Users/apt/testdir
/Users/apt/testdir/:
d1

/Users/apt/testdir/d1:
a.txt  b.txt  d2  d3

/Users/apt/testdir/d1/d2:
b.txt  c.txt

/Users/apt/testdir/d1/d3:
a.txt  d.txt  d4  d5

/Users/apt/testdir/d1/d3/d4:
a.txt  c.txt

/Users/apt/testdir/d1/d3/d5:
a.txt  b.txt  d.txt
$
```

Then, assuming that directory `zz` does not already exist in your current working directory, your program should have the following behavior:

```
$ python3 find.py --copy zz /Users/apt/testdir a.txt
/Users/apt/testdir/d1/a.txt
/Users/apt/testdir/d1/d3/a.txt
/Users/apt/testdir/d1/d3/d4/a.txt
/Users/apt/testdir/d1/d3/d5/a.txt
$ ls zz
Users_apt_testdir_d1_a.txt     Users_apt_testdir_d1_d3_d4_a.txt
Users_apt_testdir_d1_d3_a.txt  Users_apt_testdir_d1_d3_d5_a.txt
$
```