Lab 6: Collections

CSE/IT 107

NMT Computer Science

"All thought is a kind of computation."

— D. Hobbes

"Vague and nebulous is the beginning of all things, but not their end."

— K. Gibran

"It [programming] is the only job I can think of where I get to be both an engineer and artist. There's an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation."

— A. Hertzfeld

---

1 Introduction

In previous labs, we have used lists to be able to enclose data in a certain structure and manipulate it. Lists gave us an easy way to find sums of numbers, to store things, to sort things, and much more. Other structures like this are commonly referred to as collections: collections collect data and encapsulate it. They give us useful methods to manipulate that data.

A list in Python is an ordered container of any elements you want indexed by whole numbers starting at 0. Lists are mutable: this means you can add elements, change elements, and remove elements from a list. In this lab, we will introduce you to three other collections: Sets, tuples, and dictionaries.

Please read Chapters 7 and 9 for this lab. Another great resource on lists, sets, tuples, and dictionaries is the Python Tutorial:

`https://docs.python.org/3.3/tutorial/datastructures.html`

Be sure to use the Python 3 version of the Python Tutorial.

This lab will be due in *two* weeks, so you have time to study for your midterm.

2 Collections Summary

Data Structure	Mutable	Mutable elements	Indexing	Ordered	Other properties
List `[]`	yes	yes	by whole numbers	yes	can contain elements more than once
Sets `{}`	yes	no	not indexable	no	no element can appear more than once
Tuples `()`	no	yes	by whole numbers	yes	can contain elements more than once
Dictionary `{}`	yes	yes	by anything "hashable" (immutable collections or basic types)	no	

Table 1: Summary of Data Structures in Python

For syntax, please refer to the Python Tutorial or Chapters 7 and 9 in your book.

`https://docs.python.org/3.3/tutorial/datastructures.html`

Be sure to use the Python 3 version of the Python Tutorial.

3 Stacks

Stacks are an important data structure in the world of computer science. They are at the heart of every operating system and used in many, many pieces of software.

For a stack, imagine a stack of plates. You can only add plates to the top and remove plates from the top. We call this a "Last-In-First-Out" data structure: If you add three plates to your stack, the last one you added will be the first one you remove.

Similar to that, in Python you can say that a stack is a list where you can only add elements to the end or remove elements from the end. In Python, this is accomplished using the `.pop()` and `.append(element)` methods on lists. When given no arguments, the pop method will remove the last element of the list and return it. The append method will add an element to the end of the list.

When we ask you to use a stack in Python, you should use a list and only use these two methods and the array index `[-1]` to inspect the last element of the list. For example:

```
>>> somelist = [1, 2, 3]
>>> a = somelist.pop()
>>> print(a)
3
>>> print(somelist)
[1, 2]
>>> somelist.append(4)
>>> print(somelist)
[1, 2, 4]
>>> somelist[-1]
4
```

Interacting with the elements of the list in any other way violates the idea of the list being a stack.

4 A Note on Sequence Types

A sequence is a data type that stores data in a structured order. The most familiar example of a sequence is the list, but other sequences include tuples, strings, and the value given by `range()`. For the most part we will care only about lists, strings, and tuples. Lists and strings have already been covered fairly heavily in previous labs, so for this section we will be focusing on tuples.

In many ways, tuples behave similarly to lists. We create a tuple in the same way that we would create a list, though we use parentheses instead of square brackets to surround the elements of the tuple.

```
>>> tup = (1, 7, 3, 2, 6)
>>> print(tup)
(1, 7, 3, 2, 6)
```

The primary difference between a list and a tuple is that a tuple is immutable. This means that, once it has been created, its elements cannot be added, removed, or replaced.

```
>>> tup = (1, 7, 3, 2, 6)
>>> tup[0] = 5
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup.append(10)
Traceback (most recent call last):
  File ''<stdin>'', line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Beyond this change, most list operations will work perfectly normally on tuples.

```
>>> tup = (1, 7, 3, 2, 6)
>>> tup[2:5]
(3, 2, 6)
>>> tup += (2, 3, 4)
>>> tup
(1, 7, 3, 2, 6, 2, 3, 4)
>>> min(tup)
1
>>> 5 in tup
False
>>> len(tup)
8
```

It should be noted that in the case of `tup += (2, 3, 4)` we are not modifying the existing tuple. Instead, we are creating a new tuple that contains the elements of the two tuples we are joining, then assigning it to the same variable as the original tuple.

It should also be noted that we need a bit of special syntax to create a tuple with only one element. If we want a tuple that only contains `2`, then we must say `(2, )` rather than simply `(2)`. This is because Python will interpret `(2)` as a simple mathematical statement and simplify it to `2` rather than a tuple containing `2`.

Knowing when to use a tuple and when to use a list can sometimes be difficult. The general rule is that a list should be used when the elements are homogeneous (meaning they represent the same type of data) while a tuple should be used when the data being stored is heterogeneous.

As an example, let's assume we want to store birthdays. In order to store a single person's birthday, we need three integers. Even though all three values are integers, they are not homogeneous: one integer represents a day, one represents a month, and one represents a year. Therefore, a tuple is a better choice than a list.

```
>>> birthday = (21, 12, 1948)
```

Now assume we want to store a large number of birthdays. Since each element of this collection will be of the same type – a birthday – we can consider them to be homogeneous. Therefore, we will use a list.

```
>>> birthday = (21, 12, 1948)
>>> birthdays = []
>>> birthdays.append(birthday)
>>> birthdays.append((27, 3, 1963))
>>> birthdays.append((19, 3, 1955))
>>> birthdays.append((14, 4, 1961))
>>> birthdays.append((13, 12, 1957))
>>> print(birthdays)
[(21, 12, 1948), (27, 3, 1963), (19, 3, 1955), (14, 4, 1961), (13, 12, 1957)]
```

In each tuple that we created, each index represents the same value as in the other tuples. This is important since they represent different data types: changing the order of the elements in a tuple would ruin our interpretation of that data.

In the list, however, the order is purely a method of organizing the data. We can change the order based on how we wish to view it – currently it is ordered by when elements were added to the list, but we could easily sort it based on birth year – and it has no impact on the data itself.

5 Exercises

Boilerplate

Remember that this lab *must* use the boilerplate syntax introduced in Lab 5, including the review exercises.

5.1 Review for Midterm

Reviewing: for, while, if, if-elif-else, functions, file I/O, strings, lists.

piglatin.py Write a program that will take a string of words and convert each word to pig latin.

The rules for pig latin are as follows:

For a word that begins with consonant sounds, the initial consonant or consonant cluster is moved to the end of the word and "ay" is added as a suffix:

- "happy" → "appyhay"
- "glove" → "oveglay"

For words that begin with vowels or silent letters, you add "way" to the end of the word:

- "egg" → "eggway"
- "inbox" → "inboxway"

For your program, you *must* write a function that takes in one individual word and returns the translation to pig latin. Write another function that takes a string, which may be sentences (may contain the characters "a-zA-Z,.-;!?()" and space), and returns the translation of the sentence to pig latin.

piglatin_file.py Use the two functions you previously wrote for `piglatin.py` and write a program that asks the user for a filename, reads the contents of that file and translates it to pig latin. The pig latin must be written to a file name `originalfilename_piglatin.txt`.

luhns.py Luhn's algorithm (`http://en.wikipedia.org/wiki/Luhn_algorithm`) provides a quick way to check if a credit card is valid or not. The algorithm consists of three steps:

1. Starting with the second to last digit (ten's column digit), multiply every other digit by two.
2. Sum all the digits of the resulting number.
3. If the total sum modulo by 10 is zero, then the card is valid; otherwise it is invalid.

For example, to check that the Diners Club card 38520000023237 is valid, you would start at 3, double it and double every other digit to give, writing the credit card number as separate digits:

3 8 5 2 0 0 0 0 0 2 3 2 3 7

6 8 10 2 0 0 0 0 0 2 6 2 6 7

Next you would sum all the digits of the resulting number:

$$6+8+(1+0)+2+0+0+0+0+0+2+6+2+6+7=40$$

Note that for 10, you also sum its digits – $(1+0)$.

The last step is to check if $40 \mod 10 = 0$, which is true. So the card is valid.

Write a program that implements Luhn's Algorithm for validating credit cards. It should ask the user to enter a credit card number and tell the user whether it is valid or not. It should also have a separate function to validate the card number.

anagrams.py Write a function that takes in two strings. If the strings are anagrams of one another, return `True`. If not, return `False`.

5.2 New Material

fractions.py Any fraction can be written as the division of two integers. You could express this in Python as a tuple – (numerator, denominator).

Write functions for each of the following. They must use the tuple representation to return fractions.

1. Given two fractions as tuples, multiply them.
2. Given two fractions as tuples, divide them.
3. Given a list of fractions as a tuple, return the one that is smallest in value.

(For midterm review, you can do this problem using lists instead of tuples to represent fractions. When you turn it in, please have it use tuples though.)

days.py Write a function that takes four arguments – day, month, and year as numbers, and weekday as MTWRFSU – and converts this date to a human-readable string. Have the program be called with user-specified input. For example:

```
Enter day: 28
Enter month: 9
Enter year: 2014
Enter weekday: U
Date is: Sunday, September 28, 2014
```

Use dictionaries to convert weekdays to their long name and months to their long name.

rpn_calculator.py Your task is to write a reverse Polish notation calculator. In reverse Polish notation (also HP calculator notation), mathematical expressions are written with the operator following both operands. For example, $3+4$ becomes 3 4 +.

To write $3+(4*2)$, we would have to write 4 2 * 3 + in RPN. The expressions are evaluated from left to right.

A longer example of an expression is this:

5 1 2 / 4 * + 3 –

which translates to

$$5 + ((1/2) * 4) - 3$$

If you were to try to "parse" the RPN expression from left to right, you would probably "remember" values until you hit an operator, at which point you take the last two values and use the operator on them. In the example expression above, you would store 5, then 1, then 2, then see the division operator (/) and take the last two values you stored (1 and 2) to do the division. Then, you would store the result of that (0.5) and encounter 4, which you store. When you encounter the multiplication sign (*), you would take the last two values stored and do the operation $(4 * 0.5)$ and store that.

Writing this algorithm for evaluating RPN in pseudo code, we get:

1. Read next value in expression.
2. If number, store.
3. If operator:
   (a) Remove last two numbers stored.
   (b) Do operation with these last two numbers.
   (c) Store the result of the operation as last number.

If you keep repeating this algorithm, you will eventually just end up with one number stored unless the RPN expression was invalid.

Your task is to write an RPN calculator which asks the user for an RPN expression and prints the result of that expression. You *must* use a stack (see Section 3). The RPN algorithm has to be in a separate function (not main).

Please see the example file and output below for expressions you can test with.

rpn_file.py Write another version of the RPN calculator that reads RPN expressions from a file (one per line) and prints the answers to them (one per line). Use the function you previously wrote. You must ask the user which file they want to read from.

Example file:

```
4 3 +
4 3 -
3 4 -
5 1 2 / 4 * + 3 -
5 12 50 5 * / 5 + +
```

Answers:

```
7
1
-1
14
15.0083333333333
```

6 Submitting

We will be adding more exercises later. We have just not had the time to finish them. You will get an email about them.

Files to submit:

- piglatin.py (Section 5.1)
- piglatin_file.py (Section 5.1)
- luhns.py (Section 5.1)
- anagrams.py (Section 5.1)
- fractions.py (Section 5.1)
- days.py (Section 5.2)
- rpn_calculator.py (Section 5.2)
- rpn_file.py (Section 5.2)

You may submit your code as either a tarball (instructions below) or as a .zip file. Either one should contain all files used in the exercises for this lab. The submitted file should be named either `cse107_firstname_lastname_lab6.zip` or `cse107_firstname_lastname_lab6.tar.gz` depending on which method you used.

For Windows, use a tool you like to create a `.zip` file. The TCC computers should have `7z` installed. For Linux, look at lab 1 for instructions on how to create a tarball or use the "Archive Manager" graphical tool.

Upload your tarball or .zip file to Canvas.