Lab 6: Collections

CSE/IT 107

NMT Computer Science

"All thought is a kind of computaton."

— D. Hobbes

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

— K. Gibran

"It's [programming] 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 be able to refer to it later. Other structures like this are commonly referred to as collections: collections collect data and encapsulate it.

A list in python is an ordered container of any elements you want to add to it indexed by whole numbers starting at 0. Lists are mutable: this means you can add elements, change elements, and remove elements in 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.

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 twice
Tuples	no	yes	by whole numbers	yes	can contain elements more than once
Dictionary	yes	yes	by anything immutable	no	

Table 1: Summary of Data Structures in Python

2.1 List Syntax

```
semester_gpas = [4.0, 3.5, 2.0, 1.5, 4.0, 3.7]
if 4.0 in semester_gpas:
  print("There was a 4.0 in the semester_gpas list.")
if 1.0 not in semester_gpas:
  print("1.0 is not in the semester_gpas list.")
sum_gpas = 0
for gpa in semester_gpas:
  sum_gpas += gpa
third_semester = semester_gpas[2] # 2.0
first_two_gpas = semester_gpas[:2] # [4.0, 3.5]
```

2.2 Set Syntax

```
>>> companions_nine = {'rose', 'jack'}
>>> companions_ten = {'rose', 'mickey', 'jack', 'donna', 'martha', 'wilf'}
# intersection
>>> companions_of_both = companions_nine & companions_ten
>>> print(companions_of_both)
{'jack', 'rose'}
# union
>>> print(companions_nine | companions_ten)
{'donna', 'wilf', 'rose', 'martha', 'mickey', 'jack'}
# testing membership
>>> 'rose' in companions_nine
True
# iterating over elements
>>> for companion in companions_nine:
      print(companion, end=', ')

rose, jack,
# adding an element to set
```

```
>>> companions_ten.add('sarah jane')
>>> print(companions_ten)
{'donna', 'sarah jane', 'wilf', 'rose', 'martha', 'mickey', 'jack'}
```

2.3 Tuple Syntax

```
>>> ordered_pairs = [(1,2), (2,1), (3,3), (4,5), (5,4)] # list of tuples
```

2.4 Dictionary Syntax

```
ages = {'chris': 20, 'tyler': 22, 'randomperson': 34} # dictionary
```

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. 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
```

4 Plotting in Python

5 Exercises

Boilerplate

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

5.1 Review for Midterm

stuff.py Review exercises. Roadmap: need for, while, if, ifelse, functions, file IO, strings, lists. Do simple algorithms combining these. Look at scott's old labs to find some good ones, or google python class.

luhns.py Copy from lab4 of old labs, Luhn's algorithm. (Reword.)

anagrams.py Do this. Make sure not in book?

caesar.py Caesar cipher on strings. Write encrypt and decrypt function.

fractions.py Express a fraction as a list – [numerator, denominator].

5.2 New Material

addresses.py Classic address book dict exercise.

zombies.py Zombie lab from 113.

words.py Write program that takes a file and

1. finds how many unique words there are
2. finds the unique words and writes them to `originalfile_unique.txt`
3. calculate the percentage of unique words
4. MORE

data.py Some data analysis exercise. Chris will fill in. Use tuples, use matplotlib. Use set to remove duplicates in data.

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.

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). 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

Files to submit:

- rpn_calculator.py (Section 5)
- rpn_file.py (Section 5)

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.