Dictionaries and Sets, Types

CSE/IT 107L

NMT Department of Computer Science and Engineering

"Talk is cheap. Show me the code."

— Linus Torvalds

"We shall do a much better programming job, provided we approach the task with a full appreciation of its tremendous difficulty, provided that we respect the intrinsic limitations of the human mind and approach the task as very humble programmers."

— Alan Turing

"If you were plowing a field, which would you rather use? Two strong oxen or 1024 chickens?"

— Seymour Cray

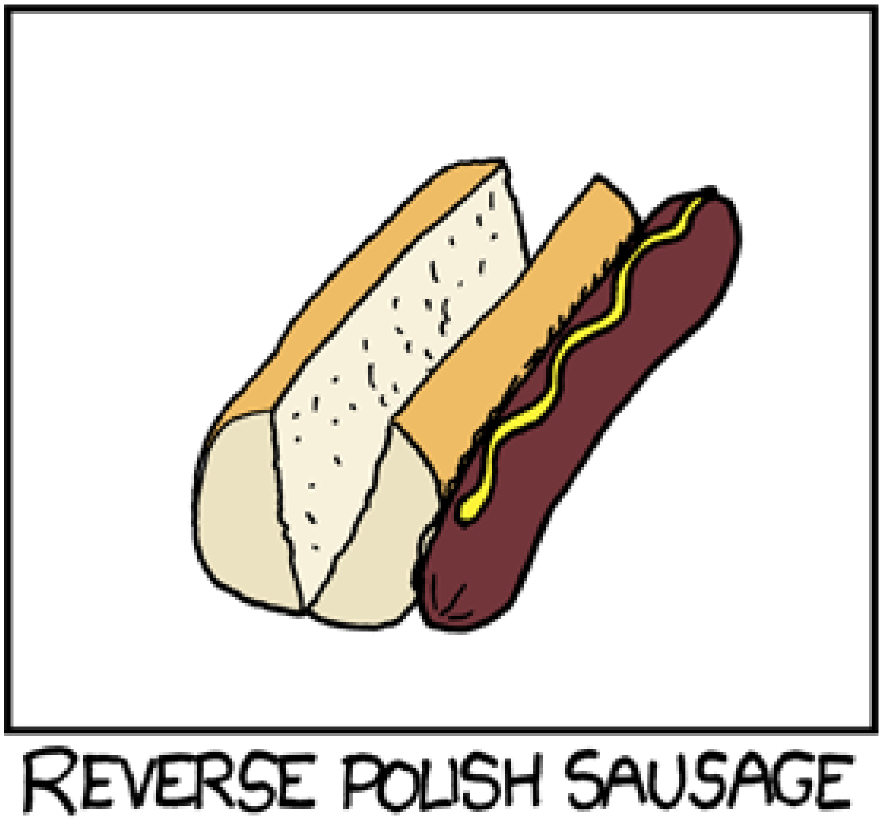

Figure 1: `http://xkcd.com/1537`

Introduction

Sets are collections of unique, and unordered (non-repeated) values. They support fast combination operations, like finding the elements that appear in two sets. Sets are similar to lists and can be used in similar ways, but contain key distinctions and limitations. This lab will introduce what sets are and how they can be used within your programs. We will define the *types* of Python values and describe how to convert between different types.

Contents

1 Sets: Collections With Unique Elements

1.1 Creating Sets

Sets are a lot like lists, but *no element can appear twice* and a set's elements are *unordered*. Additionally, a set cannot contain mutable Python values (like lists). Sets are mutable.

A set is made using curly braces by converting a list:

```
>>> a = {5, 5, 4, 3, 2} # duplicates ignored
>>> print(a)
{2, 3, 4, 5}
>>> b = set([5, 5, 4, 3])
>>> print(b)
{3, 4, 5}
```

Because sets are unordered, they cannot be indexed. That means you cannot index a set using the [] operator.

```
>>> a = {5, 4, 3, 2}
>>> a[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

1.2 Combining Sets

The | operator can be used to join two sets together. This creates a new set that contains the elements in each of the two sets. This is called the *union* of the two sets. Since the result is also a set, duplicate values will only appear once.

```
>>> a = {1, 2, 4, 8, 16}
>>> b = {1, 3, 9, 27}
>>> a | b
{1, 2, 3, 4, 8, 9, 16, 27}
```

The & operator can be used to find the *intersection* of two sets. This is the set containing all elements that are in both sets.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a & b
{12, 6}
```

The - operator can be used to find the *difference* of two sets. This set contains all of the elements of the first set excluding the elements in the second set.

```
>>> a = {2, 4, 6, 8, 10, 12}
>>> b = {3, 6, 9, 12, 15, 18}
>>> a - b
{8, 2, 10, 4}
```

The ^ operator can be used to find the *symmetric difference* of two sets. This is the set containing all elements in either of the original sets, but not in both.

```
>>> a = {'one', 'eight'}
>>> b = {'two', 'four', 'six', 'eight'}
>>> a ^ b
{'one', 'six', 'two', 'four'}
```

Operator	Meaning	Definition
`a	b`	in either	union of `a` and `b`
`a & b`	in both	intersection of `a` and `b`
`a - b`	all not in `b`	difference of `a` and `b`
`a ^ b`	all elements not in common	symmetric difference of `a` and `b`

1.3 Conversions Using `set()`, `list()`

A list can be converted into a set using the `set()` function:

```
>>> a = [1, 5, 2, 2.3, 5, 2]
>>> set(a) # duplicate removed!
{1, 2, 2.3, 5}
```

Similarly, a set can be converted into a list using the `list()` function:

```
>>> a = {1, 2, 2, 5, 3, 2}
>>> a
{1, 2, 5, 3}
>>> list(a)
[1, 2, 3, 5]
```

1.4 Adding Elements Using `set.add()`

To add an element to a set, use the `set.add(element)` function. A sample:

```
>>> a = {1, 2, 5, 3}
>>> a.add(5)
>>> a
{1, 2, 3, 5}
>>> a.add(7)
>>> a
{1, 2, 3, 5, 7}
```

1.5 Removing Elements Using `set.remove()` or `set.pop()`

To remove an element that is present in the set, use the `set.remove(element)` function. To remove an arbitrary element from the set, use the `set.pop()` function. Python raises a `KeyError` if the set is empty or the element is not in the set. A sample:

```
>>> a = {1, 2, 5, 3}
>>> an_element = a.pop()
>>> an_element, a
(1, {2, 3, 5})
>>> a.remove(5)
>>> a
{2, 3}
>>> a.remove(10)
Traceback (most recent call last):
  File "<doctest lab7.tex[76]>", line 1, in <module>
    a.remove(10)
KeyError: 10
```

1.6 Traversing Sets With For Loops

You may also iterate over a set using a for-loop. Order of iteration is unspecified.

```
>>> companions_nine = {'rose', 'jack'}
>>> companions_ten = {'rose', 'mickey', 'jack', 'donna', 'martha', 'wilf'}
>>> for companion in companions_nine:
...     print(companion)
jack
rose
```

1.7 Creating an Empty Set and an Empty Dictionary

Note that empty curly braces create an empty dictionary. To create an empty set, use the `set()` function:

```
>>> type({})
<class 'dict'>
>>> type(set())
<class 'set'>
```

2 Ordered Access With Stacks

Imagine a stack of plates. You can only add plates to the top and remove plates from the top. This a "Last-In-First-Out" data structure: If you add three plates to the stack, the last one you added will be the first one you remove. *Stacks* are a specialized data structure. They are at the heart of every operating system and used in many pieces of software.

A stack is similar to a Python list where you can only add elements to the end or remove elements from the end. This is accomplished using the `list.pop()` and `list.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. The array index `list[-1]` is used to inspect the top of the stack; or the last element of the list. For example:

```
>>> stack = [1, 2, 3]
>>> a = stack.pop()
>>> print(a)
3
>>> print(stack)
[1, 2]
>>> stack.append(4)
>>> print(stack)
[1, 2, 4]
>>> stack[-1]  # top!
4
```

Interacting with the elements of the list in any other way violates the idea of the list being a stack. By adhering to this restriction, we can effectively invent a new data structure based on existing Python data structures. There are more effective ways of creating data structures for organizing data according to specific needs. For example, you could compose two data structures. If you needed a dictionary with ordered key-value pairs, you could use a list of tuples where the first element in the tuple is the key and the second is the value.

3 List Comprehensions: Concise Iteration

3.1 Modifying Collections

List comprehensions can be used to create new lists by doing something to each element of a currently existing list or range. For example, if we want to make every string in a list of strings lowercase we could use a for-loop to call `.lower()` on each string.

```
>>> strings = ['Python', 'N.M.']
>>> result = []
>>> for string in strings:
...     result.append(string.lower())
>>> result
['python', 'n.m.']
```

List comprehensions are a more concise way of doing this.

```
>>> strings = ['Python', 'N.M.']
>>> result = [string.lower() for string in strings]
>>> print(result)
['python', 'n.m.']
```

We can process more complicated elements with list comprehensions. If we have a list of dictionaries containing lists, we can find the maximum element in each sublist by using this list comprehension:

```
>>> data = [{'d': [1,2,3]}, {'d': [4,5]},
...         {'d': [0,0,1]}]
...
>>> [max(dictionary['d']) for dictionary in data]
[3, 5, 1]
```

3.2 Filtering Elements

What if we want to exclude elements? In this code, we take a list of words and keep all words of length less than 2.

```
>>> strings = ['a', 'ab', 'abb', 'aab', 'aaa']
>>> result = []
>>> for string in strings:
...     if len(string) <= 2:
...         result.append(string)
...
>>> result
['a', 'ab']
```

This can be written more concisely by using an `if` statement within a list comprehension.

```
>>> strings = ['a', 'ab', 'abb', 'aab', 'aaa']
>>> [x for x in strings if len(x) <= 2]
['a', 'ab']
```

3.3 Nested Comprehensions

We can also write multiple `for` statements and `if` statements in a list comprehension. This example creates a list of floating point numbers by dividing two integers.

```
>>> result = [y / x for x in range(3) if x != 0 for y in range(3)]
>>> print(result)
[0.0, 1.0, 2.0, 0.0, 0.5, 1.0]
```

These statements should be read from right to left. So the previous list comprehension is the same as these nested statements:

```
result = []
for x in range(10):
    if x != 0:
        for y in range(10):
            result.append(y / x)
```

4 Types

In Python, every value has a type. We have already seen a few types in action: integers, floating-point numbers, strings, and boolean values, lists, tuples, sets, and dictionaries. The type of a value or variable restricts what it can represent. Here is a list of some types and example values.

Name	Description	Examples	Convert y
Int	Integer, whole number	`1, 123, -12`	`int(y)`
Float	Floating-point number	`1.0, 3.1415, -0.01`	`float(y)`
String	Sequence of characters	`"", "1", "abc 123\n"`	`str(y)`
Boolean	True or false	`True, False`	`bool(y)`

Most programming languages have types for a good reason: for one, operations (such as +, -, . . .) have different effects on different types. For example, an integer ∗ an integer results in an integer (the multiplication of the two *operands*), but an integer ∗ a string results in the string repeated. However, a string ∗ a string results in an error:

```
>>> 5*3
15
>>> 5*'hi'
'hihihihihi'
>>> 'hi'*'hi'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Also, Python can speed up your programs by picking efficient ways of dealing with different types. Multiplying integers and multiplying floating point numbers require very different techniques. Python may optimize code if the type of the number is known.

4.1 Types of Collections

In the previous labs, we have described and demonstrated several data structures. Each of them has its own type.

Name	Description	Examples	Convert y
List	Sequence of values	`[], [0, 1], [True, True]`	`list(y)`
Tuple	Immutable sequence of values	`(0, 1), ('a', 'b')`	`(y)`
Dictionary	Relationship between values	`{}, {1:2}, {'a':True}`	`dict(y)`
Set	Collection of unique values	`{0,1}, {True}`	`set(y)`

4.2 Checking Types with the `type()` Function

The `type()` function returns the type of a value.

```
>>> type(5)
<class 'int'>
```

```
>>> x = 10
>>> type(x)
<class 'int'>
>>> type('Allons-y!')
<class 'str'>
>>> type(True)
<class 'bool'>
```

4.3 Converting Values to Different Types

There are several functions that can convert the types of Python values. Only values have types, variables can store values of any type.

```
>>> float("5.5") # convert string with 5.5 to a float
5.5
>>> int("5") # integer containing 5
5
>>> str(5) # string containing 5
'5'
>>> # Note that an impossible conversion will throw an error
>>> int("55a")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '55a'
```

There are several functions that work on different types of collections. The `len()` function can be used on any collection object in order to find the number of elements. All collections can be iterated over with `for` loops, All collections can use the `in` keyword to test if an element is in that collection. For dictionaries, this will compare against the list of keys, not the values.

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 "hash-able" (immutable collections or basic types)	no	

Table 1: Summary of Data Structures in Python

Tuples, sets, and lists can all be freely converted from one to another using the `tuple()`, `set()`, `list()`, `str()`, and `dict()` functions.

```
>>> food = ('nothing', 'cereal', 'lemonheads')
>>> list(food)
['nothing', 'cereal', 'lemonheads']
>>> str(food) # not particularly useful
"('nothing', 'cereal', 'lemonheads')"
>>> s = "The cat in the"
>>> list(s)
['T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e']
>>> tuple(s)
('T', 'h', 'e', ' ', 'c', 'a', 't', ' ', 'i', 'n', ' ', 't', 'h', 'e')
```

4.4 Converting Sequences to Strings With `str.join()`

Note `str()` cannot undo the action of `list()` on a string. That is, `str(list("ABC"))` is not equal to `"ABC"`. Use `"".join(list("ABC"))` for that.

To add spaces between the elements of a list, change the string at the beginning.

```
>>> "".join(['c', 'a', 't'])
'cat'
>>> " ".join(['c', 'a', 't'])
'c a t'
```

The elements of the sequence can be of any type.

```
>>> ' '.join(("dinosaurs", "roamed the", "earth"))
'dinosaurs roamed the earth'
>>> print('\n> '.join(("dinosaurs", "roamed the", "earth")))
dinosaurs
> roamed the
> earth
```

5 Sample Program

```
import turtle

# All units in kilometers.
planets8 = [
    {"name": "mercury", "radius": 2439.7, "color": "lightgrey", "moons": []},
    {"name": "venus", "radius": 6051.8, "color": "orange", "moons": []},
    {"name": "earth", "radius": 6371, "color": "blue",
     "moons": [{"name": "moon", "radius": 1737.4, "color": "lightgrey"}]},
    {"name": "mars", "radius": 3389.5, "color": "red",
     "moons": [{"name": "phobos", "radius": 11.1, "color": "grey"},
               {"name": "deimos", "radius": 6.3, "color": "grey"}]},
    {"name": "jupiter", "radius": 69911, "color": "orange",
     "moons": [{"name": "ganymede", "radius": 2631.2, "color": "grey"},
                {"name": "callisto", "radius": 2410.3, "color": "darkgrey"},
                {"name": "io", "radius": 1830, "color": "lightgreen"},
                {"name": "europa", "radius": 1560.8, "color": "green"}]},
     {"name": "saturn", "radius": 58232, "color": "yellow",
      "moons": [{"name": "titan", "radius": 2574, "color": "orange"},
                {"name": "rhea", "radius": 763.5, "color": "grey"},
                {"name": "iapets", "radius": 734.3, "color": "darkgrey"},
                {"name": "dione", "radius": 561.4, "color": "grey"}]},
     {"name": "uranus", "radius": 25362, "color": "lightblue",
      "moons": [{"name": "titania", "radius": 788.4, "color": "grey"},
                {"name": "oberon", "radius": 761.4, "color": "grey"},
                {"name": "umbriel", "radius": 584.7, "color": "darkgrey"},
                {"name": "ariel", "radius": 578.9, "color": "grey"}]},
     {"name": "neptune", "radius": 24622, "color": "blue",
      "moons": [{"name": "triton", "radius": 1352.6, "color": "salmon"},
                {"name": "proteus", "radius": 210, "color": "grey"},
                {"name": "nereid", "radius": 170, "color": "grey"},
                {"name": "larissa", "radius": 97, "color": "grey"}]},  # :(
]

def draw_body(body, zoom):
    """This function draws a filled-in circle based on the 'radius' and
    'color' keys of the dictionary `body`. The drawing is scaled by the
    factor `zoom`."""
    turtle.dot(2 * body["radius"] * zoom, body["color"])

def separate_bodies(body, prev_body_radius, zoom, sep):
    """This function moves the turtle the appropriate distance to avoid
    overlap between two drawings. Minimal separation given by `sep`."""
    distance = (body["radius"] + prev_body_radius) * zoom + sep
    turtle.forward(distance)
    return distance

def draw_planets(system, zoom):
    separation = 20  # Distance between planets.
    moon_separation = 10  # Distance between moons and their planets.

    prev_planet_radius = 0
    for index, planet in enumerate(system):
        if index > 0:
            prev_planet_radius = system[index - 1]["radius"]
        separate_bodies(planet, prev_planet_radius, zoom, separation)
        draw_body(planet, zoom)

        # Draw moons
        turtle.left(90)
        distance = 0  # How far has the turtle moved to draw the moons?
        prev_radius = planet["radius"]
        for index2, moon in enumerate(planet["moons"]):
            if index2 > 0:
                prev_radius = planet["moons"][index2 - 1]["radius"]
            distance += separate_bodies(moon, prev_radius, zoom, moon_separation)
            draw_body(moon, zoom)
        turtle.forward(-distance)
        turtle.right(90)

def draw_system(system, zoom):
    # Program inputs.
    print('Welcome to the 107L2 Radially-Precise Planetarium!')
    turtle.tracer(False)

    turtle.bgcolor('black')  # Black background
    turtle.penup()  # Hide pen

    # Draw (2 * decoration) ** 2 stars.
    decoration = 15
    scale = max(turtle.window_height(), turtle.window_width())
    for col in range(-decoration, decoration):
        for row in range(-decoration, decoration):
            # row / decoration gives is a number between -1 and 1
            # the `scale` variable determines the space between stars
            y = row / decoration * scale
            # shift every other row of stars by 50%
            x = ((col + 0.5 * (row % 2)) / decoration) * scale

            turtle.goto(x, y)

            # make every other star's radius larger
            turtle.dot(1 + col % 2, 'white')
    turtle.goto(0, 0)

    # Draw planets starting near the screen's edge.
    turtle.forward(-turtle.window_width() / 2 * 0.9)
    draw_planets(system, zoom)

    turtle.done()

if __name__ == "__main__":
    draw_system(planets8, 0.003)
```

6 Exercises

Exercise 6.1 (lettercount.py, design).
Before attempting to program `lettercount.py`, create a file called `design` that includes some analysis on how you think the problem will be solved. Examples include: a flowchart of events in the program, pseudocode, or a step by step process written in plain english. If you choose to scan your design please make sure that it is legible.

Next write a program that reads in a string on the command line and returns a table of the letters of the alphabet in alphabetical order which occur in the string together with the number of times each letter occurs. Case should be ignored. A sample run of the program would look this:

```
Enter some letters >>> The cat in the hat
a 2
c 1
e 2
h 3
i 1
n 1
t 4
```

This should involve writing a function called `count_letters` that takes in a string and returns a dictionary with these letters and counts.

Supplementary Files

The file `test_letter_count.py` is available on Canvas. Open it to see more examples.

Exercise 6.2 (date.py, horoscope.py).
First write the program `date.py`, which should take in a month, day of the month and year, and convert it to the corresponding day of the year. Assume that leap years exist. Check that each date is correct (ie. `March 56, 2017` is incorrect). For incorrect dates the conversion function should return `-1`, and you should print an error from `main`. For both programs, use dictionaries to store whatever data you might need (the number of days in each month, etc.)

For example:

```
python3 date2.py
Type stop or enter a month, day, and year.
> March, 14, 2015
Day of the year: 73
> December, 14, 2016
Day of the leap year: 349
> Oktober, 91, 2015
Invalid date entered.
> stop
```

Use the following boolean expression to check if a year is a leap year or not:

```
is_leap = (year % 4 == 0 and year % 100 != 0) or year % 400 == 0
```

Next, the `horoscope.py` program should take in a month, day, and year. It should print an error message if the date is invalid, otherwise it should print the horoscope of someone who was born on that day. You should use the `import` statement to make use of the code you wrote in `date.py` for determining the user's astrological sign. The horoscopes themselves are completely up to you, as long as they start with the correct sign.

For example:

```
python3 horoscope2.py
Type stop or enter a month, day, and year.
> March, 14, 2015
Pisces: Tui and La, push and pull, commit and rebase...
> May, 5, 1848
Taurus: today you must choose between earth and sea, C and miniKanren...
> February, 29, 1847
Invalid date entered.
> stop
```

For a reference on the zodiac dates, see the "Tropical zodiac" column from this table: `https://en.wikipedia.org/wiki/Zodiac#Table_of_dates`. Watch out for Capricorn! That zodiac spans from December 22nd to January 20th, which for this program means that it will have two different date ranges, 1 to 20 and 356 to 365.

Exercise 6.3 (rpn_calculator.py).

Write a reverse Polish notation calculator. In reverse Polish notation (also known as HP calculator notation), mathematical expressions are written with the operator following its operands. For example, $3 + 4$ becomes $3\ 4\ +$.

Order of operations is entirely based on the ordering of the operators and operands. To write $3 + (4 * 2)$, we would 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.

Following this through step by step, the steps would look something like this (the bold number is the most recently computed value):

1. 5 1 2 / 4 * + 3 -
2. 5 **0.5** 4 * + 3 -
3. 5 **2** + 3 -
4. **7** 3 -
5. **4**

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 **??**). The RPN algorithm has to be in a separate function (not main). You need to support the four basic operators (+, -, *, and /).

You should detect and display messages for the following errors:

- Operand is used when not enough numbers are stored.
- More or less than one number stored after the expression is evaluated.

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

RPN Expression	Output
`5 1 2 / 4 * + 3 -`	4
`312 999 +`	1311
`4 2 + 1 5 + * +`	Not enough operands for +.
`2 100 3 * 5 + 2 2 2 + + * *`	3660

Note that you need to support multi-digit numbers. You cannot expect all input to be single digits.

Exercise 6.4 (README).
For the following labs you will be required to create aand submit a README file.

A README file is a simple text file that contains basic information regarding the purpose and use of the software program, utility, or game. README files often contain instructions or additional help regarding the software it accompanies.

For this lab you must have the following in your README file::

- **Purpose**: Describes what each program does and what problem it solves. You can keep this breif.
- **Conclusion**:
  - What you learned during the lab? What new aspect of programming did you learn from the lab? Be analytical about what you learned.
  - Did pair programming help in solving the problems and completing the prelab? Did you have problems with your budy?
  - Did you work with your budy on the lab? What sections did you discuss? Did you and your budy carry out a review session with each others' code?
  - Did you encounter any problems? How did you fix those problems?
  - What improvements could you make?

The conclusion does not have to be lengthy, but it should be thorough. You will include the README file with your submissions.

Index of New Functions and Methods

Submitting

You should submit your code as a `.zip` that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab7.zip`

Upload your .zip file to Canvas.

List of Files to Submit

Exercises start on page 11.