Data Structures and algorithms

• Part of the "science" in computer science is the design and use of data structures and algorithms
• As you go on in CS, you will learn more and more about these two areas

Data Structures

• Data structures are particular ways of storing data to make some operation easier or more efficient. That is, they are tuned for certain tasks
• Data structures are suited to solving certain problems, and they are often associated with algorithms.

Kinds of data structures

Roughly two kinds of data structures:
• built-in data structures, data structures that are so common as to be provided by default
• user-defined data structures (classes in object-oriented programming) that are designed for a particular task

Python built in data structures

• Python comes with a general set of built in data structures:
  - lists
  - tuples
  - string
  - dictionaries
  - sets
  - others...
Lists

The Python List Data Structure

- a list is an ordered sequence of items.
- you have seen such a sequence before in a string. A string is just a particular kind of list (what kind)?

Make a List

- Like all data structures, lists have a constructor, named the same as the data structure. It takes an iterable data structure and adds each item to the list
- It also has a shortcut, the use of square brackets [ ] to indicate explicit items.

```python
>>> m_list = [1, 2, 'a', 3, 4, 5]
>>> week_days_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists = [(1, 2), ['a', 'b', 'c']]
>>> list_from_collection = list(set([1, 2, 3]))
>>> x_list
[1, 2, 'a', 3, 4, 5]
>>> week_days_list
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists
[(1, 2), ['a', 'b', 'c']]
>>> list_from_collection
[1, 2, 3]
>>> []
[]
```

Similarities with strings

- concatenate/ + (but only of lists)
- repeat/*
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)

Operators

- `1, 2, 3 + [4]` ⇒ `[1, 2, 3, 4]`
- `1, 2, 3 * 2` ⇒ `[1, 2, 3, 1, 2, 3]`
- `1 in [1, 2, 3]` ⇒ `True`
- `1, 2, 3 < [1, 2, 4]` ⇒ `True`
  - compare index to index, first difference determines the result
differences between lists and strings

- lists can contain a mixture of any python object, strings can only hold characters
- lists are mutable, their values can be changed, while strings are immutable
- lists are designated with [], with elements separated by commas, strings use "" or '

```
mylist = [1, 'a', 3.14159, True]
```

```
myList
<table>
<thead>
<tr>
<th>1</th>
<th>'a'</th>
<th>3.14159</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>
```

Index forward
Index backward

mylist[1] → 'a'
mylist[1:3] → [1, 'a', 3.14159]

**Figure 7.1** The structure of a list.

Indexing

- can be a little confusing, what does the [] mean, a list or an index?
- [1, 2, 3][1] ⇒ 2
- Context solves the problem. Index always comes at the end of an expression, and is preceded by something (a variable, a sequence)

List of Lists

```
my_list = ['a', [1, 2, 3], 'z']
```

- What is the second element (index 1) of that list? Another list.

```
my_list[1][0] # apply left to right
my_list[1] ⇒ [1, 2, 3]
[1, 2, 3][0] ⇒ 1
```

List Functions

- `len(lst)`: number of elements in list (top level).
  - `len([1, [1, 2], 3]) ⇒ 3`
- `min(lst)`: smallest element. Must all be the same type!
- `max(lst)`: largest element, again all must be the same type
- `sum(lst)`: sum of the elements, numeric only

Iteration

You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1, 2, 4, 6]
```

```
for element in my_list:
    print(element, end=' ') # prints on one line
```

```
1 2 4 6
```
Change an object's contents

- strings are immutable. Once created, the object's contents cannot be changed. New objects can be created to reflect a change, but the object itself cannot be changed.

```python
def change_str():
    my_str = 'abc'
    my_str[0] = 'z'  # cannot do!
    # instead, make new str
    new_str = my_str.replace('a', 'z')
    return new_str
```

Lists are mutable

Unlike strings, lists are mutable. You can change the object's contents!

```python
my_list = [1, 2, 3]
my_list[0] = 127
print(my_list) => [127, 2, 3]
```

List methods

- Remember, a function is a small program (such as `len`) that takes some arguments, the stuff in the parenthesis, and returns some value.
- A method is a function called in a special way, the *dot call*. It is called in the context of an object (or a variable associated with an object).

Again, lists have methods

```python
my_list = ['a', 'b', True]
my_list.append('z')
```

Some new methods

- A list is mutable and can change:
  - `my_list[0] = 'a'`  # index assignment
  - `my_list.append()`, `my_list.extend()`
  - `my_list.pop()`
  - `my_list.insert()`, `my_list.remove()`
  - `my_list.sort()`
  - `my_list.reverse()`
More about list methods

- most of these methods do not return a value
- This is because lists are mutable, so the methods modify the list directly. No need to return anything.
- Can be confusing

Unusual results

```python
my_list = [4, 7, 1, 2]
my_list = my_list.sort()
my_list  # what happened?
```

What happened was the sort operation changed the order of the list in place (right side of assignment). Then the sort method returned None, which was assigned to the variable. The list was lost and None is now the value of the variable.

Range

- We have seen the range function before. It generates a sequence of integers.
- In fact what it generates is a list with that sequence:
  ```python
  myList = range(1, 5)
  myList is [1, 2, 3, 4]
  ```

Split

- The string method split generates a sequence of characters by splitting the string at certain split-characters.
- It returns a list (we didn't mention that before)
  ```python
  split_list = 'this is a test'.split()
  split_list  # what happened?
  ```

Sorting

Only lists have a built-in sorting method. Thus you often convert your data to a list if it needs sorting

```python
my_list = list('zyxabc')
my_list  # what happened?
my_list.sort()  # no return
```

reverse words in a string

```python
join method of string places the calling string between every element of a list
```

```python
my_str = 'this is a test'
my_str = my_str.split()  # list of words
my_list = ['this', 'is', 'a', 'test']
# for each word
for word in my_list:
  reversed_elements.append(word[::-1])  # reverse, append
reversed_elements
  # join with space separator
res_str = ' '.join(reversed_elements)
res_str  # what happened?
```
Sorted function

The `sorted` function will break a sequence into elements and sort the sequence, placing the results in a list.

```python
sort_list = sorted('hi mom')
sort_list => ['h', 'i', 'm', 'm', 'o']
```

Anagram example

- Anagrams are words that contain the same letters arranged in a different order. For example: 'iceman' and 'cinema'.
- Strategy to identify anagrams is to take the letters of a word, sort those letters, than compare the sorted sequences. Anagrams should have the same sorted sequence.

```python
def are_anagrams(words1, words2):
    # Return True, if words are anagrams.
    words1_sorted = sorted(words1)
    words2_sorted = sorted(words2)

    # Check that the sorted words are identical.
    if words1_sorted == words2_sorted:  # assume sorted lists
        return True
    else:
        return False
```

Some Examples

Code Listing 7.1

Code Listing 7.3

Full Program
Code Listing 7.4
Check those errors

repeat input prompt for valid input

valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two words: ")
        word1, word2 = two_words.split()
        valid_input_bool = True
    except ValueError:
        print("Bad Input")

Code Listing 7.5
Words from text file

def make_word_list(file):
    word_list = []
    for line_str in file:
        word_list.append(line_str)
Code Listing 7.7
Unique Words, Gettysburg Address

```python
# frequency adder routine

def word_freq_init(list):
    my_dict = {}
    for i in list:
        if i in my_dict:
            my_dict[i] += 1
        else:
            my_dict[i] = 1
    return my_dict

# unique words

def unique_word_init(list):
    my_list = []
    for i in list:
        if i not in my_list:
            my_list.append(i)
    return my_list
```

Reminder, assignment

- Assignment takes an object (the final object after all operations) from the RHS and associates it with a variable on the left hand side
- When you assign one variable to another, you share the association with the same object

```python
my_int = 27
your_int = my_int
```

Immutable

- Object sharing, two variables associated with the same object, is not a problem since the object cannot be changed
- Any changes that occur generate a new object
my_list = [1, 2, 3]
my_list[0] = 1
my_list[1] = my_list[1] + 1

Values
my_int
27

your_int = my_int
your_int = your_int + 1

Values
your_int
28

Mutability

- If two variables associate with the same object, then both reflect any change to that object.

a_list = [1, 2, 3]
b_list = a_list

Values
a_list
[1, 2, 3]
b_list

a_list.append(27)

Values
a_list
[1, 2, 3, 27]
b_list

Copying

If we copy, does that solve the problem?

my_list = [1, 2, 3]
newList = my_list[:]

Values
a_list
[1, 2, 3, 27]
b_list

Values
[1, 2, 3]

FIGURE 7.6 Making a distinct copy of a mutable object.
Sort of/depends - what gets copied?

The big question is, what gets copied?

• What actually gets copied is the top level reference. If the list has nested lists or uses other associations, the association gets copied. This is termed a shallow copy.

\[ \text{a\_list} = \{1, 2, 3\} \]
\[ \text{a\_list.append(a\_list)} \]
\[ \text{print(a\_list)} \rightarrow \{1, 2, 3, [\ldots]\} \]

**FIGURE 7.7** Self-referencing.

\[ \text{a\_list} = \{1, 2, 3\} \]
\[ \text{b\_list} = \{5, 6, 7\} \]
\[ \text{a\_list.append(b\_list)} \]

**FIGURE 7.8** Simple lists before append.

\[ \text{a\_list} = \{1, 2, 3\} \]
\[ \text{b\_list} = \{5, 6, 7\} \]
\[ \text{a\_list.append(b\_list)} \]

**FIGURE 7.9** Lists after append.

shallow vs deep

Regular copy, the [::] approach, only copies the top level reference/association

• If you want a full copy, you can use deepcopy

```
>>> a_list = [1, 2, 3]
>>> b_list = [5, 6, 7]
>>> a_list.append(b_list)
>>> import copy
>>> c_list = copy.deepcopy(a_list)
>>> b_list[0] = 1000
>>> a_list
[1, 2, 3, [5, 6, 7]]
>>> c_list
[3, 3, 3, [5, 6, 7]]
```
Tuples

- Tuples are simply immutable lists
- They are printed with (, )

```
>>> (1, 2, 3)
1, 2, 3
>>> tup = (2, 3)
>>> x, y = tup  # assigning a tuple to a variable
>>> x
2
>>> y
3
>>> (x, y)  # using tuples as a grouping
(2, 3)
```

Commases make a tuple

For tuples, you can think of a comma as the operator that makes a tuple, where the ( ) simply acts as a grouping:

```
myTuple = 1, 2  # creates (1, 2)
myTuple = (1,)  # creates (1)
myTuple = (1)  # creates 1 not (1)
myTuple = 1,  # creates (1)
```

Lists and Tuple

- Everything that works with a list works with a tuple except methods that modify the tuple
- Thus indexing, slicing, len, print all work as expected
- However, none of the mutable methods work: append, extend, del

The question is, Why?

- The real question is, why have an immutable list, a tuple, as a separate type?
- An immutable list gives you a data structure with some integrity, some permanent-ness if you will
- You know you cannot accidentally change one.
Data Structures in General

Organization of data
- We have seen strings, lists and tuples so far
- Each is an organization of data that is useful for some things, not as useful for others.

A good data structure
- Efficient with respect to us (some algorithm)
- Efficient with respect to the amount of space used
- Efficient with respect to the time it takes to perform some operations

EPA Example

EPA Example
- epaData.csv
- program7-9.py - find Ferraris
- program7-10.py - list of gas mileage
- program7-11.py - int mileage data
- program7-12.py - find max, min
- program7-13.py - list of cars
- program7-15.py - nicer output

List Comprehensions
Lists are a big deal!

- The use of lists in Python is a major part of its power.
- Lists are very useful and can be used to accomplish many tasks.
- Therefore Python provides some pretty powerful support to make common list tasks easier.

Constructing lists

One way is a "list comprehension"

\[ \{ n \text{ for } n \text{ in range(1,5)} \} \]

mark the comp with \( \{ \)
\[ \{ n \text{ for } n \text{ in range(1,5)} \} \]

returns \{1,2,3,4\} what we iterate through. Note that we iterate over a set of values and collect some (in this case all) of them

modifying what we collect

\[ \{ n**2 \text{ for } n \text{ in range(1,6)} \} \]

returns \{1,4,9,16,25\}. Note that we can only change the values we are iterating over, in this case \( n \)

multiple collects

\[ \{ x+y \text{ for } x \text{ in range(1,4) for } y \text{ in range(1,4)} \} \]

It is as if we had done the following:

\[
\begin{align*}
\text{my_list} & = [] \\
\text{for } x \text{ in range(1,4)}: & \\
\text{for } y \text{ in range(1,4)}: & \\
& \text{my_list.append(x+y)} \\
\Rightarrow & \{2,3,4,3,4,5,4,5,6\}
\end{align*}
\]

modifying what gets collected

\[ \{ c \text{ for } c \text{ in } "Hi There Mom" \text{ if } c\.isupper() \} \]

- The if part of the comprehensive controls which of the iterated values is collected at the end. Only those values which make the if part true will be collected

\[ \{ "H", "T", "M" \} \]

Reminder, rules so far

- Think before you program!
- A program is a human-readable essay on problem solving that also happens to execute on a computer.
- The best way to improve your programming and problem solving skills is to practice!
- A foolish consistency is the hobgoblin of little minds
- Test your code, often and thoroughly
- If it was hard to write, it is probably hard to read. Add a comment.
- All input is evil, unless proven otherwise.
- A function should do one thing.