Data Structures and Algorithms

- part of the "science" in computer science is the design and use of data structures and algorithms
- as you progress in CS, you will learn more about these two areas

Data Structures

- data structures are particular ways of storing data to make some operation easier or more efficient; i.e., 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

Two kinds of data structures:
- built-in data structures – so common as to be provided by default
- user-defined data structures (classes in object oriented programming) – designed for a particular task

Python Built-in Data Structures

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

- 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)?

Creating Lists

- lists have a **constructor**, with the same name as the data structure
  - the constructor receives an iterable data structure and *adds each item* to the list
- lists can use square brackets [ ] to include explicit items

Making Lists

```python
>>> a_list = [1, 2, 'a', 1.1459]
>>> week_days_list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> list_of_lists = [[1, 2], ['a', 'b', 'c']]
>>> list_from_collection = list('Hello!')
```

Similarities with Strings

- concatenate/+ (but only of lists)
- repeat/*
- indexing (using the [ ] operator)
- slicing ([:])
- membership (using 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 by index; first difference determines the result
Differences Between Lists and Strings

- lists can contain a mixture of any python objects; strings can only hold characters  
  e.g., [1,"bill",1.2345, True]
- lists are mutable, their values can be changed, while strings are immutable
- lists are designated with [], with elements separated by commas; strings use "" or ''

Indexing

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

List of Lists

- what is the second element (index 1) of that list? another list
  my_list = ['a', [1, 2, 3], 'z']

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 = [1,3,4,8]
>>> for element in my_list:  # iterate through list elements
...     print(element, end='') # prints on one line
1 3 4 8
```
Mutability

Strings are immutable; i.e., 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
my_str = 'abc'
my_str[0] = 'z'  # cannot do!
# instead, make new str
new_str = my_str.replace('a','z')
```

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 (in parentheses), and returns a 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).

```python
my_list = ['a',1,True]
my_list.append('z')
```

Some New Methods

- List methods that change the list
  ```python
  my_list[0] = 'a'  # index assignment
  my_list.append()  # append el to list
  my_list.extend()  # append list as els
  my_list.pop()     # remove/return el
  my_list.insert()  # put el at loc
  my_list.remove()  # delete el
  my_list.sort()    # sort
  my_list.reverse() # reverse
  ```
List Methods

• most of these methods do not return a value
• since lists are mutable, the methods modify the list directly; no need to return anything

Unusual Results

my_list = [4, 7, 1, 2]
my_list = my_list.sort()
my_list ⇒ None    # what happened?

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

The split Method

• 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)

split_list = 'this is a test'.split() 
split_list ⇒ ['this', 'is', 'a', 'test']

Sorting

• only lists have a built in sorting method
• thus, you often convert your data to a list if it needs sorting

my_list = list('xyzabc')
my_list ⇒ ['x','y','z','a','b','c']
my_list.sort()    # no return
my_list ⇒ ['a','b','c','x','y','z']

Reverse Words in a String

• join method of string places the argument between every element of a list

>>> my_str = 'This is a test'
>>> string_elements = my_str.split()    # list of words
>>> reversed_elements = []
>>> for element in string_elements:
...     reversed_elements.append(element[::-1])    # reverse, append
... >>> reversed_elements
['this', 'is', 'a', 'test']
>>> new_str = ' '.join(reversed_elements)    # join with space separator
>>> new_str
'isThis a test'

sorted Function

• the sorted function breaks a sequence into elements and sorts the sequence, placing the results in a list

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; e.g., 'iceman' and 'cinema'
- a strategy to identify anagrams is to take the letters of a word, sort those letters, then compare the sorted sequences; anagrams should have the same sorted sequence

```python
def are_anagrams(word1, word2):
    # Return True, if words are anagrams.
    # 1. Sort the characters of the words.
    word1_sorted = sorted(word1)  # sorted returns a sorted list
    word2_sorted = sorted(word2)
    # 2. Check that the sorted words are identical.
    if word1_sorted == word2_sorted:  # compare sorted lists
        return True
    else:
        return False
```

```python
def are_anagrams(word1, word2):
    # Return True, if words are anagrams.
    # 1. Sort the characters of the words.
    word1_sorted = sorted(word1)  # sorted returns a sorted list
    word2_sorted = sorted(word2)
    # 2. Check that the sorted words are identical.
    return word1_sorted == word2_sorted

print("Anagram True")
# 1. Input two words.
two_words = input("Enter two space separated words: ")
word1, word2 = two_words.split()  # split into a list of words
if are_anagrams(word1, word2):  # return True or False
    print("The words are anagrams.")
else:
    print("The words are not anagrams.")
```
Checking Valid Input

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

only runs when no error; otherwise, try again

Code Listing 7.4
Check those errors

def are_anagram(word1, word2):
    """Return True, if words are anagrams. """
    # Sort the characters of the words.
    word1_sorted = sorted(word1)
    word2_sorted = sorted(word2)
    # Check that the sorted words are identical.
    return word1_sorted == word2_sorted

# 1. Input two words; checking for errors now.
valid_input_bool = False
while not valid_input_bool:
    try:
        two_words = input("Enter two space separated words: ")
        word1, word2 = two_words.split()  # Split the input string into a list of words.
        valid_input_bool = True
        except ValueError:
            print("Bad Input")
    if are_anagram(word1, word2):  # function returned True or False
        print("The words {} and {} are anagram.").format(word1, word2)
    else:
        print("The words {} and {} are not anagram.").format(word1, word2)

Code Listing 7.5
Words from text file

def make_word_list(fn):
    """Create a list of words from the file. """
    word_list = []  # list of speech words: initialized to be empty.
    for line in fn:
        line_list = line.split()  # split each line into a list of words.
        for word in line_list:
            if word != " ":  # if the word is not a space,
                word_list.append(word)  # add the word to the speech list.
    return word_list
More about Mutables

Assignment

- Assignment takes an object (the final object after all operations) from the RHS and associates it with a variable on the LHS.

- When you assign one variable to another, you **share the association** with the same object.

Immutables

- 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.

```python
# Getting a list of words

def make_word_list(file):
    """Return a list of words from the file."""
    word_list = []
    for line in file:
        words = line.split()
        for word in words:
            if word != "":
                word_list.append(word)
    return word_list

# Example usage

with open('example.txt', 'r') as file:
    word_list = make_word_list(file)
```
Mutability

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

```
my_list = [1, 2, 3]
newLst = my_list[:]
```

Copy

- if we copy, does that solve the problem?

```
my_list = [1, 2, 3]
newLst = my_list[:]
```

Copy

- 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, only the association gets copied (a **shallow copy**).
Shallow vs. Deep Copy

- Regular copy, using `[:]`, only copies the top level reference/association
- If you want a full copy, you can use `deepcopy`

```python
>>> a_list = [1, 2, 3]
>>> b_list = [5, 6, 7]
>>> a_list.append(b_list)
>>> a_list
[1, 2, 3, [5, 6, 7]]
```

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

- tuples are simply immutable lists
- they are printed with (, )

```
>>> 10,12  # Python creates a tuple
(10, 12)
>>> tuple = 2,3  # assigning a tuple to a variable
>>> tuple
(2, 3)
>>> (1)  # not a tuple, a grouping
1
>>> (1,1)  # comma makes it a tuple
(1, 1)
>>> x,y = "a",3.14159  # from on right, multiple assignments
>>> x
'a'
>>> y
3.14159
>>> x,y  # create a tuple
('a', 3.14159)
```

Lists and Tuples

- everything that works with a list also 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`

Tuples

- why have an immutable list, a tuple, as a separate type?
- because an immutable list gives you a data structure with some integrity, some permanent-ness if you will
- you know you cannot accidentally change it

Commas Create a Tuple

- for tuples, you can think of a comma as the operator that creates 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)
```

Data Structures in General
Organization of Data

- so far, we have seen strings, lists and tuples
- each is an organization of data that is useful for some things, but not as useful for others

Marks of Good Data Structures

- 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

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
- Python therefore provides some pretty powerful support to make common list tasks easier

Constructing Lists

- one way is a "list comprehension"
  
  \[
  [n \text{ for } n \text{ in range}(1,5)]
  \]

  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 the Collected Items

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

- returns \[1,4,9,16,25\]
- note that we can only change the values we are iterating over, in this case \(n\)

Multiple Collects

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

- it is as if we had done the following:
  \[
  \text{my_list = [ ]}
  \text{for } x \text{ in range}(1,4):
  \text{for } y \text{ in range}(1,4):
  \text{my_list.append(x+y)}
  \]
  \[
  \Rightarrow [2,3,4,3,4,5,4,5,6]
  \]

Modifying What Gets Collected

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

- the \(\text{if}\) part of the comprehensive controls which of the iterated values is collected
- only those values which make the \(\text{if}\) part true will be collected
  \[
  \Rightarrow ['H', 'T', 'M']
  \]

Rules

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