PythonContainers_1

October 18, 2015

Contents

1 Overview

Containers are Python objects that hold (contain) other objects. Because they are comprised of multiple objects, containers are often called compound objects. The objects within a container might be all of the same type, or they might have different types. Thus a container might be a collection of nothing but ints or nothing but floats, or it might be a collection of mixed types with integers, floats, and even other types within the same container.

We will cover four types of containers: strings, lists, tuples and dictionaries. The first three categories are sequences, so named because their subparts are ordered. Because of the ordering within a sequence, we can talk meaningfully about the first member, or the second, or the 10th. On the other hand, objects within dictionaries are not sequenced or ordered. That is, we as programmers have no knowledge of the order Python might use when objects are stored, and we cannot refer to the position of an object relative to any other object in the container.

As will be seen, strings, lists, and tuples are very similar in terms of how members are accessed within the container. Dictionaries are quite different, and will be discussed after the sequences. Here in Topic 7 we will cover strings and lists. The following topic presents tuples and dictionaries.

2 Strings

As you know already, strings are sequences of text characters, such as 'JohnSmithThe2nd'. Why should we learn about strings? Are we going to write a text editor or word processor? Hardly, but there are many other uses, such as parsing user input, or manipulating file names and paths. Working with file and folder names is especially important—we often need to extract the file name, subdirectory, and/or file extension from a path. This calls for knowing how to use and manipulate character strings. Some basic points about strings:

Strings can contain any Ascii character. This means: 1. All the usual printable alphabetic characters and numerals: A-Z, a-z, 0-9 2. Other keyboard symbols: - @ # ; : etc. 3. Other non-printing characters: tab, line feed, etc. Don't try using the keyboard to insert these! (see below) Strings can also hold Unicode characters. Unicode is a standard designed to support almost every writing system in use, including those that are not based on the familiar Roman alphabet (e.g., Chinese, Arabic, Hebrew). The Unicode character set has many more characters than the 256 contained in the Ascii standard. Indeed, at present there are more 110,000 Unicode characters. There is a small chance you will see Unicode used in a programs, and for that reason it deserves mention. But except for this, we won't discuss strings of Unicode characters. Much of what we learn will carry over to other sequences

2.1 String Creation and Assignment

Strings are delimited by matching single or double quotes:

```
In [ ]: a = 'text'
        a = "text"
```

What if we want a quote (" or ') inside a string? Answer: use the other quote type delimiters:

```
In [48]: a = "Somebody's in big trouble"
         print a
         a = 'Uh-oh, "somebody" is in trouble'
         print a
         a = ''  #an empty string
         print a
         a = ""  #another empty string
         print a
```

```
Somebody's in big trouble
Uh-oh, "somebody" is in trouble
```

We've discussed the " before paranthetically. It is a special character: * \t means tab, e.g., `a = 'string with a \t tab inside'` * \n means newline (CR+LF in Windows) * \' means one single quote, e.g., `a = 'Mom\'s apple pie'` * \" means one double quote * \\ means a literal (so if you wanted to print out "either\or" you'd have to type "either\\or"

Some examples:

This is a single variable (a),but will print as two lines

```
In [ ]: a = 'Line 1\nLine 2'
```

This is a properly formed path:

```
In [ ]: g387FolderName = 'c:\\class\\Geog378'

        a = r'string' #creates a raw string, meaning nothing is special
           For example, this is a properly formed path:

        tempFolder = r'c:\temp\new folder'
```

Without the leading `r`, the string would contain a tab (`\t`) and a newline (`\n`).
To make a string spanning multiple lines, use a Triple quote (repeated ' or "):

```
In [ ]: x = '''Line 1
        Line 2
        Line 3'''

        print x
```

x includes 2 newlines: one after the "1", another after the "2".
In a verbatim string like this nothing is special.

2.2 Accessing String Elements

Characters within a string are ordered, thus they have a definite position within the string. As with most languages (but not R), the first position is considered position 0, the 2nd is character in in position number 1, and so on. If the string has 10 characters, the final character is located at position 9.

This takes some getting used to. Think of a character's position as the distance it lies from the start of the string. In this way the first character is at the start, thus its distance from the start is zero. The next character is 1 away from the start of the string. The 10th character in a string is found 9 characters away from the start. Computer scientist use the word offset to mean the distance from the start. So we will refer to a character's position as its offset.

To access the individual characters in a string, we put the offset (position) inside square brackets, such as `x[9]`. For example:

```
In [3]: x = 'Hello'
        print '\'x[0]\' is', x[0], 'and \'x[4]\' is', x[4]

        y = 'abcdefghih'
        print 'If you want to get the third character remember that the index starts at 0:', y[2]
```

```
'x[0]' is H and 'x[4]' is o
If you want to get the third character remember that the index starts at 0: c
```

Use `x[j:k]` for slice running from `x[j]` to `x[k-1]`

```
In [5]: x = '0123456789'
        print 'The third to sixth characters are', x[2:5]
```

```
The third to sixth characters are 234
```

Use `[j:]` for j thru end Use `[:j]` for 0 thru j-1 Use `[i:j:k]` to jump from i to j in steps of k (ver. 2.3 onward)
Use `[-j]` for jth from the last (e.g., `[-2]` is the second from last object):

```
In [4]: x = 'Hello'
        print x[-2], x[-1], x[-3], ". . . you get the picture."
```

```
l o l . . . you get the picture.
```

Very important: Even though strings are similar to tuples or lists, the elements within a string are immutable. They cannot be changed by assignment, you will get an `item assignment` error if you try this. There are other ways to substitute characters,

```
In [8]: x[3] = 'b' #illegal
```

```
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-8-2d2d1d2afd62> in <module>()
----> 1 x[3] = 'b' #illegal


TypeError: 'str' object does not support item assignment
```

2.3 Exercise 7.1:

a) Create a string variable named x consisting of the digits 0 through 9 in ascending order.

b) Create a string variable named y consisting of the digits 0 through 9 in descending order.

c) For what value of j is x[j] == y[j+1]?

Go to the Exercise 7.1. PDF on Learn@UW

2.4 String Operators and Functions

Python has many built-in operators and functions that work with strings.

Operators You have seen + and * before (note the use of the ' here to give us quotes in the `print` statement):

```
In [12]: print '\'1\' + \'2\' == \'12\' is', '1'  +  '2'   ==  '12'
         print '3 * \'2\' == \'222\' is', 3   *  '2'   ==  '222'
```

```
'1' + '2' == '12' is True
3 * '2' == '222' is True
```

Note the mix of string and numeric operands for the * operator.

Comparison operators are defined for strings, e.g., `a < b`, `a > b`, etc. The comparison is similar to the way we normally compare words, like when determining word order in a dictionary. In particular: * The comparison is character-by-character * The order of first two differing elements determines the result of the comparison

```
In [16]: a = 'Introduction to geocomputing'
         b = 'Introduction to Geocomputing - Super Magnificent!'

         a > b    #the result is True because upper case letters are 'lower' than lowercase letters in
```

```
Out[16]: True
```

Even though `b` is longer than `a` we see that the capital "G" is the first character difference between the two strings, and we know that upper case letters are 'lower' in the order than lower case letters (recall the ASCII table:).

In the next example we see that the corresponding characters are the same in both strings, and Python therefore compares their lengths.

```
In [17]: a = 'Introduction to geocomputing'
         b = 'Introduction to geocomputing - Super Magnificent!'

         a > b     # the result is False because the fist different character is either NULL (the string
                   # space, and the space is bigger than nothing.
```

```
Out[17]: False
```

`in` and `not in` operators (return True/False):

```
In [ ]: 'J' in 'Jane'      #True
        'j' in 'Jane'      #False
        'j' not in 'Jane' #True
        'Ja' in 'Jane'     #True
```

2.4.1 Example 7.1:

```
In [ ]: baseFolder  = '\\Wisconsin''
        imageFolder = baseFolder + '\\images'

        landsatFolder = imageFolder + '\\landsat'
        modisFolder   = imageFolder + '\\modis'
```

This example builds paths to LandSat and MODIS images within a directory tree. If the state were to change from Wisconsin to something else, only one line in the program needs to change.

2.5 Exercise 7.2:

Create a string with 6 characters. Make a new string that is a concatenation of the first and last 2 characters.

Go to the Exercise 7.2. PDF on Learn@UW

2.6 Format operator: %.

[Skim this section. It is mentioned here only for completeness.]

This is another operator that takes a mix of types resulting in a new string. This operator is used for printing when we want to format values in a particular way. The operator is given a sequence of values to format and a string saying how to format thoese values. The syntax is

`formatString % sequenceOfValuesToConvert`

Example:

```
In [53]: import math

         print 2*math.pi # Very long, and not particularly meaningful output.

         TwoPi  = 2*math.pi  #this is a value (~pi * 2 )
         format = '%10.3f'   #this is a format string

         s = format % TwoPi     #TwoPi is converted to a string

         print TwoPi
         print s
```

```
6.28318530718
6.28318530718
     6.283
```

Shorter version of above:

```
In [57]: # Note the brackets.  We need to follow orger of operations, we'd have an error otherwise.
         print "%10.3f" % (2 * math.pi)
```

```
6.283
```

Some format strings: * %d #format as int
* %f #format as float * %e #format using exponential notation * %s #format as string
* %wd #format as int w digits wide * %w.pf #format as float with w.p digits * %-8d #format as int, left justify
More than one value can be formatted if placed in a tuple:

```
In [ ]:  values = ('Jones', 'J.', 367, 23, 459)
         fmt    = 'Name: %s, %s  SSN: %3d-%2d-%3d'
         print fmt % values
```

Built-in string functions:

```
In [22]: print 'an example with \'len\':', len('abc')    #returns 3
         print 'an example with \'max\':', max('cba')    #returns 'c'
         print 'an example with \'min\':', min('cba')    #returns 'a'
         print 'an example with \'str\':', str(4.6)      #returns '4.6'
         print 'an example with \'chr\':', chr(65)       #returns 'A'
         print 'an example with \'ord\':', ord("A")      #returns 65

         print 'multiplying two \'len\' arguments together:', len('abc') * len('12')
```

```
an example with 'len': 3
an example with 'max': c
an example with 'min': a
an example with 'str': 4.6
an example with 'chr': A
an example with 'ord': 65
multiplying two 'len' arguments together: 6
```

2.6.1 Example 7.2: Find and print all duplicate characters in a string.

Idea: compare every character in string with all characters to "right"and report if duplicate. Start at first character, stop at 2nd to last character.

Pseudo-code: for every character in string a[i] except the last" for all characters a[j] where j > i, report if a[i] == a[j]

```
In [24]: a = 'Hopefully we all learn Python'

         #print a 'ruler' and the string
         print '01234567890123456789012345678 9'
         print a,'\n'

         for i in xrange(len(a) - 1):
           for j in range(i+1,len(a)):
              if a[i] == a[j]:
                 print 'characters',i,'and',j,'are both \'',a[i],'\''
```

```
01234567890123456789012345678 9
Hopefully we all learn Python
```

```
characters 1 and 27 are both ' o '
characters 3 and 11 are both ' e '
characters 3 and 18 are both ' e '
characters 6 and 7 are both ' l '
characters 6 and 14 are both ' l '
characters 6 and 15 are both ' l '
characters 6 and 17 are both ' l '
characters 7 and 14 are both ' l '
characters 7 and 15 are both ' l '
characters 7 and 17 are both ' l '
characters 8 and 24 are both ' y '
characters 9 and 12 are both '   '
characters 9 and 16 are both '   '
characters 9 and 22 are both '   '
characters 11 and 18 are both ' e '
characters 12 and 16 are both '   '
characters 12 and 22 are both '   '
characters 13 and 19 are both ' a '
characters 14 and 15 are both ' l '
characters 14 and 17 are both ' l '
characters 15 and 17 are both ' l '
characters 16 and 22 are both '   '
characters 21 and 28 are both ' n '
```

2.7 Exercise 7.3:

Write a function to count occurrences of a particular character in a string. Both the string and the character are parameters of the function. Test the function.

Go to the Exercise 7.3. PDF on Learn@UW

2.8 String Methods

Strings are objects, which means they consists of data and procedures. Of course, the most important "data" of a string are its characters. The procedures are formally known as methods, and they accomplish some particular task, such as finding the position of a given character. To invoke a string method we use a dot notation, e.g., `a.lower()`

String methods return something to program: it might be a value, or it could be another string. There are many string methods available, and we will cover just some of the more helpful ones:

```
In [25]: a = " Super Potato "
         print '\'', a.lower(), '\''        # returns lower case equivalent of a
         print '\'', a.upper(), '\''        # returns upper case equivalent
         print '\'', a.strip(), '\''        # returns the same string but without leading or trailing
                                            # blanks or other such "whitespace"
```

```
super potato
SUPER POTATO
Super Potato
```

[aside: the exact def of "whitepace" depends on system. Typically it includes the following characters: space, tab, linefeed, return, formfeed, and vertical tab]

These 3 return a list:

```
In [26]: print a.split()         #split the string at blanks and other whitespace
         print a.split(',')      #split the string at commas (but there are no commas, so it's a list wi
         print a.splitlines()    #split the string at newlines
```

```
['Super', 'Potato']
['Super Potato']
['Super Potato']
```

These 3 return an integer corresponding to the first occurrence of a character or string.

```
In [31]: print a.find('l')              #returns first index of 'l', -1 if absent
         print a.find('Pot',3,-1)         #look for 'ly' starting at position 3, stop at position 6
         print a.rfind('ly')            #earch from end to beginning

         print a.replace('&','amp;')    #returns new string with all
                                        #Occurrences of "&" replaced by 'amp;"
         print a.isdigit()              #returns True/False
         print a.isalpha()              #returns True/False
         print a.endswith('.jpg')       #returns True/False
```

```
-1
6
-1
Super Potato
False
False
False
```

```
In [ ]:
```

2.8.1 Example 7.3:

The USGS makes its scanned topographic quadrangles available as geopdf files on the National Map. These files go by names such as "WI_Madison West_320068_1990_24000_geo.pdf" or "NV_Silverado Mountain_320068_1990_24000_geo.pdf". However, users generally know the maps by the quadrangle name, "Madison West" or "Silverado Mountain".

As part of a larger application we want to simplify the act of selecting quadrangles, providing users with a list of Quadrangle names only (without the extra numbers and underscores). It would be straightforward to go in and edit this by hand, but there are an enormous number of maps, so while we could do this by hand it would be an awful job.

Problem: write a function that finds the quadrangle name within the filename. If a quadrangle name can't be found, return an empty string. If a name is found replace any blanks with underscores.

Idea: We notice that the quad name is bracketed by underscores, and also that underscores are not used in a quad name. So, we will search for the two underscores and return whatever is within.

```
In [6]: def getQuadname(fname):
            #Extract quadname from USGS geopdf file name

            i = fname.find('_') + 1
            j = fname.find('_',i+1)

            if i < 0 or j < 0:
              print "Couldn't find '_' brackets around quad name"
              quadname = ""

            else: quadname = (fname[i:j]).replace(' ','_')

            return quadname

        f1 = 'NV_Silverado Mountain_320068_1990_24000_geo.pdf'
```

```
        f2 = 'WI_Not a valid nameGeo.pdf'
        f3 = 'Another invalid_nameGeo.pdf'

        print getQuadname(f1)
        print getQuadname(f2)
        print getQuadname(f3)
```

```
Silverado_Mountain
Couldn't find '_' brackets around quad name

Couldn't find '_' brackets around quad name
```

2.9 Exercise 7.4:

Change the code in Example 7.3 to print the quadrangle name with State name. For example, "WI_Madison West_320068_1990_24000_geo.pdf" would become "WI_Madison_West"

Go to the Exercise 7.4. PDF on Learn@UW

More string method examples:

2.9.1 Example 7.4:

```
In [7]: x = 'Hopefully we all learn Python'
        print x.lower()     # Note that none of these change x explicitly, x retains the same value it'
        print x.upper()
        print x
        print x.split()
        print x.split('e')
        print x.split('fully')

        x += '\nAnd fast.' # now we're adding this on to the end of the existing string.
        print x
        print x.splitlines()

        print x.find('ly')
        print x.find('ly',3,7)
        print x.rfind('e')
```

```
hopefully we all learn python
HOPEFULLY WE ALL LEARN PYTHON
Hopefully we all learn Python
['Hopefully', 'we', 'all', 'learn', 'Python']
['Hop', 'fully w', ' all l', 'arn Python']
['Hope', ' we all learn Python']
Hopefully we all learn Python
And fast.
['Hopefully we all learn Python', 'And fast.']
7
-1
18
```

Notes: All of above return a value. Why? E.g., why doesn't `x.lower()` actually convert `x` to lower case? The split characters disappear from the list returned by `x.split()`

To see all the string methods, print the directory of a string object:

```
print dir('q')
```

```
print dir('')
s = 'abc'
print dir(s)
```

3 Lists

Lists are sequences like strings and so much carries over, but: * Unlike strings, elements of a list need not be characters * Unlike strings, lists are mutable. New members can be added to a list, and existing elements can be removed or replaced with something else.

You've already seen `tuples` before, and we'll use them later. In some senses they are like lists (except they've got rounded brackets), they have members that can take on different values, but they're also like strings, the position of the value is as meaningful as its value. So, if the position of the numbers is to be meaningful (e.g., compare `(2015,10,31)` to `[31,2015,10]` as a way to store dates), then you'd want to use a tuple. Otherwise a list is fine.

a. `List creation and assignment`

Delimited by square brackets, with items separated by commas `a = [117, 155, 222]`

Members can be of differing types `a = ['Jones', 'Jane','P.', 37215,"12-14-192",(424, 17, 5923)]`

Lists can contain lists `a = ["Jones", "Jane",'P.', 37215,"12-14-192", [424, 17, 5923] ]`

Lists can be empty (as we saw in an earlier exersise) `a = []`

Lists can be constructed from variables. The values used are those at time of list creation. So even if a variable changes, the values stored in the list does not change. In other words, there is no dynamic update. Example 1 illustrates this.

3.0.2 Example 7.5:

```
In [8]: firstName = 'Jane'
        midName   = 'P'
        lastName  = 'Jones'

        x = [firstName, midName, lastName]

        lastName = 'Smith'

        print 'x is:',x
        print 'lastName is',lastName


x is: ['Jane', 'P', 'Jones']
lastName is Smith
```

Lists are also created by the range() function

```
In [ ]: a = range(10)      #returns [0,1,2,3,4,5,6,7,8,9]
        a = range(1,10,3) #returns [1,4,7]
```

Note that `xrange()` does not return a list. Example:

```
In [9]: a = range(10)
        b = xrange(10)
        print type(a)
        print type(b)

<type 'list'>
<type 'xrange'>
```

It is fine to use `xrange()` for looping, but not when you need a list. `xrange()` is more efficient for a loop because `xrange` doesn't construct a list with all its items; it constructs something ideally suited to moving over a container, known as an "iterator". That said, usually we are not overly concerned with efficiency in Python programs, and use of `range()` is fine for looping.

Lists are also created by the `list()` function from a given sequence

```
In [ ]: a = list('abcd')          #returns ['a', 'b', 'c', 'd']
        a = list( (9,7,'abc') )   #returns [9, 7, 'abc']
```

list() requires a sequence

```
In [10]: a = list( 9,7,'abc' )     #this bombs!


         ---------------------------------------------------------------------------

         TypeError                                 Traceback (most recent call last)

         <ipython-input-10-f0a758c2c963> in <module>()
    ----> 1 a = list( 9,7,'abc' )     #this bombs!


         TypeError: list() takes at most 1 argument (3 given)
```

3.1 Accessing Lists

Lists use the same indexing syntax as strings, with each element accessed by its positional index

```
In [12]: a = [1,2,3,4,5]
         print a[0]          #equals 1
         print a[-2]         #equals 4
         print 'And now some slices'
         print a[1:3]        #equals [2,3]
         print a[2:]         #equals [3,4,5]

1
4
And now some slices
[2, 3]
[3, 4, 5]
```

List items within lists are accessible (a "nested" list), where the first square backets refer to the outermost list, and the second set of square brackets is the index in the next list in the hierarchy.

```
In [15]: a = ['abcde', ['a','b','c','d'], 2, 3]

         print a[1][2]       #equals 'c'
         print a[0][0]       # Equals 'a'

c
d
```

In this way we can simulate multi-dimensional arrays. Consider a 2x3 array:

```
In [17]: a = [  [1,2,3],      #the 1st "row" has 3 elements
                [4,5,6]   ]   #the 2nd "row" has 3 elements

         print "row 0, column 2:",a[0][2]
         print "row 1, column 1:",a[1][1]
```

```
row 0, column 2: 3
row 1, column 1: 5
```

As will be seen later, there are better ways to store arrays of numbers, especially large arrays.

Unlike strings, lists *are* mutable: we can change individual elements

a = [1,2,3,4] a[1] = 'text' #a now equals [1,'text',3,4] a[1:3] = 'ab' #a now equals [1,'a','b',4] del a[1] #a now equals [1,'b',4]

3.2 List Operators and Functions

3.2.1 Concatenation

```
In [18]: a = [1,2,3,4]
         print a + [5,6,7]
```

```
[1, 2, 3, 4, 5, 6, 7]
```

3.2.2 Replication

print a*2

3.2.3 Membership (`in` and `not in`)

```
In [19]: print 3 in a       # Remember a is a vector of numbers, as defined above.
         print 'z' in a
         print '3' not in a

         print len([10,21,17])    #returns 3
         print max([10,21,17])   #returns 21
         print min(['a','Z','z']) #returns 'Z' (ascii Z comes before a...z)
         print sum([1.5, 2, 3.5]) #returns 7.0
```

```
True
False
True
3
21
Z
7.0
```

3.2.4 Example 7.6:

Write a program to get a telephone number from a use and delete anything that is not a digit. We wil use a list because strings aren't mutable. Idea: test every character in user's string using the string method isdigit(). If not a digit, delete the character and move on to next.

Pseudo-code: 1. Get string from user and convert to list of characters 2. For every member of the list (first through last character): 1. Check if digit 2. If not delete that member of the list 3. Print the list

Try this:

```
In [22]: #first try
         tlist = list(raw_input('Enter telephone number > '))
         for i in range(len(tlist)):
             if not tlist[i].isdigit(): del tlist[i]

         print tlist
```

```
Enter telephone number > (608)555-1212
```

```
        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-22-aeb5099864a9> in <module>()
          2 tlist = list(raw_input('Enter telephone number > '))
          3 for i in range(len(tlist)):
    ----> 4     if not tlist[i].isdigit(): del tlist[i]
          5
          6 print tlist


        IndexError: list index out of range
```

Why does the program fail?

```
In [23]: #2nd try
         tlist = list(raw_input('Enter telephone number >'))
         print tlist

         for i in range(len(tlist)):
             print 'ready to examine element',i,'of tlist:',tlist
             if not tlist[i].isdigit(): del tlist[i]
```

```
Enter telephone number >608-555-1212
['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 0 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 1 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 2 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 3 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 4 of tlist: ['6', '0', '8', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 5 of tlist: ['6', '0', '8', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 6 of tlist: ['6', '0', '8', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 7 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 8 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 9 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 10 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
```

```
        ---------------------------------------------------------------------------

        IndexError                                Traceback (most recent call last)

        <ipython-input-23-557395a1a35f> in <module>()
```

```
      5 for i in range(len(tlist)):
      6     print 'ready to examine element',i,'of tlist:',tlist
----> 7     if not tlist[i].isdigit(): del tlist[i]


IndexError: list index out of range
```

Now we see why it bombed. Each time we delete an element of the list the length of the list becomes shorter by one element. The problem is that we initialized the loop using the list length returned from the full phone number. At some point our loop index becomes higher than the length of the reduced list. We could solve the problem by working from the end of the list to the start. This means we need to modify our call to `range` so that we're counting backwards:

```
In [24]: #3rd try (Example7.6.py)
         tlist = list(raw_input('Enter telephone number >'))
         print tlist

         print 'will examine ',len(tlist),' elements:',range(len(tlist)-1, -1, -1)

         for i in range(len(tlist)-1, -1, -1):
             print 'ready to examine element',i,'of tlist:',tlist
             if not tlist[i].isdigit(): del tlist[i]

         print tlist
```

```
Enter telephone number >608-555-1212
['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
will examine  12  elements: [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
ready to examine element 11 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 10 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 9 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 8 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 7 of tlist: ['6', '0', '8', '-', '5', '5', '5', '-', '1', '2', '1', '2']
ready to examine element 6 of tlist: ['6', '0', '8', '-', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 5 of tlist: ['6', '0', '8', '-', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 4 of tlist: ['6', '0', '8', '-', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 3 of tlist: ['6', '0', '8', '-', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 2 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 1 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
ready to examine element 0 of tlist: ['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
['6', '0', '8', '5', '5', '5', '1', '2', '1', '2']
```

Takeaways: 1. You need to carefully evaluate your pseudo-code 2. You need good test data 3. Verification can be a nuisance because Idle doesn't have an inspector that lets us see values as the program executes (must insert print statements) - **but** 4. Verification is critical to make sure your programs run as expected.

3.3 Exercise 7.5:

In our third try, we examined elements of the list from the end to the beginning in order to avoid the "out of range" error. Can you change the code to start from the beginning to the end but also avoid the range error? [Hint: how about building a new list consisting of only valid digits?]

Go to the Exercise 7.5. PDF on Learn@UW

3.4 List Methods

Many list methods available. To see them all, ask for a "directory" of any list object:

```
a = [1,2,3]
dir(a)
dir([1])
dir([])
```

Some useful list methods, nothing is returned except as noted, but, unlike with the string methods earlier, many of the methods return nothing, but change the object itself. The act of calling the method changes the object associated with the method. For example:

```
In [34]: print '### Here is the string method ###'
         aa = 'abcd'
         print 'The return value is:',aa.upper()
         print 'The value of the variable is now:', aa

         print '\n### Here is the string method ###'
         aa = ['abcd']
         print 'The return value is:', aa.append('ABCD')
         print 'The value of the variable is now:', aa

### Here is the string method ###
The return value is: ABCD
The value of the variable is now: abcd

### Here is the string method ###
The return value is: None
The value of the variable is now: ['abcd', 'ABCD']

In [30]: object = 'robots'
         a = ['summer', 'winter', 'fall', 'spring', 'robots', 'pumpkin']

         print a.append(object)  #append object to list e.g. a.append('end')

         j = 6
         print a.insert(j, object) #insert object at position j

         print a.remove(object)   #remove object from list
                                  #bombs if object not found in list

         print a.pop(2)           #remove and return object at position j

         print a.reverse()         #reverse order of items in list

         print a.count(object)    #return count of object in list

         print a.index(object)    #return 1st index of object
                                  #bombs if object not found

         i = 1
         print a.index(object,i,j) #return 1st index of object b/t i and j
                                   #bombs if object not found

         sequence = range(0, 10, 2)
         print a.extend(sequence) #append all items in sequence to list
                           # Note:  a.extend('abc')is different than a.append('abc')
```

```
        print a.sort()          #order items in ascending order

        print a.sort(reverse=True)#version 2.4 added reverse flag
```

```
None
None
None
fall
None
2
0
1
None
None
None
```

3.4.1 Example 7.7:

Sort example program:

```
In [35]: p = [-9,7,-5,2]
         print "p   unsorted:",p

         p.sort()
         print "p  ascending:",p

         p = [-9,7,-5,2]
         p.sort(reverse=True)
         print "p descending:",p
```

```
p   unsorted: [-9, 7, -5, 2]
p  ascending: [-9, -5, 2, 7]
p descending: [7, 2, -5, -9]
```

What if we want to control how Python decides the sort order of objects? In this case we can supply the name of a function to the sort method. The function itself is something that we program. Its only job is to compare to members of the list being sorted and return an integer that tells which of the two is larger (or zero if they are equal). Here we are going to write our own function that will sort integers based on:

1. how many numbers they have (the actual length of the number)
2. whether they're even or not (even numbers listed first)
3. the actual value of the number.

I will freely admit this is a ridiculous way of sorting, but there are probably cases where you may well want to sort on strange characteristics.

```
In [44]: def cmpFunc(a,b) :
             # The function must return a -1 (if a is lower than b), a 0 if
             # a and b are equivalent, or a 1 if a is after b
             if len(str(a)) == len(str(b)):
                if a % 2 == b % 2:
                    if a < b:   return -1
                    elif a > b: return 1
                    else:       return 0
                elif a % 2 == 0:     return -1
                else: return 1
```

```
            if len(str(a)) < len(str(b)):   return -1
            if len(str(a)) > len(str(b)):   return 1

        a = [2,10, 23, 43, 12, 10, -2, 100, 73]

        print a
        a.sort(cmpFunc)    #sort based on cmpFunc comparison function, doesn't return anything.
        print a
```

```
[2, 10, 23, 43, 12, 10, -2, 100, 73]
[2, -2, 10, 10, 12, 23, 43, 73, 100]
```

So, lots of nested `if` statements here, the most important one is the first sort - do the numbers have the same number of digits. Then we see that even numbers are first and odd numbers are second, and then that they're sorted in numerical order. One issue is that the negative numbers are counted as 'two digit' numbers. Are we okay with this? How might we fix it?

3.4.2 Example 7.8:

Suppose we want to sort on absolute value. We define a comparison function that examines the absolute values:

```
In [45]: def absComp(a,b):
           if(a == b)           : return 0
           elif abs(a) < abs(b) : return -1
           else                 : return 1
```

Every time the sort method needs to compare two objects a and b, it will use absComp to make the comparison.

```
In [46]: a = [-9,-5,2,7]
         a.sort(absComp)
         print a
```

```
[2, -5, 7, -9]
```

The `cmpFunc` parameter has gone away in Python 3 (and there are a large number of methods that have changed, so it's worth being aware that Python 3 breaks a lot of backward compatability). You will the cmpFunc option often, and need to know about it. But in new code that you write, perhaps use the `key` parameter instead. This parameter specifies a function to be called prior to the comparison. The function returns a value that is used as the basis for comparison. For example, the absolute value sort could be done by:

def cmpval(x): return abs(x)
a.sort(key = cmpval)
More compactly, we could define the key function on the sort line:
a.sort(key = lambda x: abs(x))

The keyword `lambda` is used to create an "anonymous" function. It is generated specifically for use in this function. In this case the function has one argument named x. The function returns the absolute value of x. This mirrors our ordinary function cmpval. Please note: there are many other uses for `lambda`. It is not tied to sort.

3.4.3 Example 7.9:

Suppose land parcels are objects with a parcel ID number, an owner name (string), and a size in acres. How can we sort a list of parcels by size?

```
In [47]: #create 3 individual parcels w/ ID, owner, size
         p1 = [711,'Jones',3.0]
         p2 = [1190,'Smith',2.8]
         p3 = [702,'Willy',4.0]
```

```
        #here's our list of parcels
        parcels = [ p1, p2 , p3]


        #this will compare based on element 2: size
        def cmpParcelSize(a,b):
          #print 'comparing',a,'and',b
          if   a[2] <  b[2]: return -1
          elif a[2] == b[2]: return 0
          else             : return 1

        print '       before sort:',parcels
        parcels.sort()
        print 'after default sort:',parcels
        parcels.sort(cmpParcelSize)

        print ' after custom sort:',parcels
```

```
before sort: [[711, 'Jones', 3.0], [1190, 'Smith', 2.8], [702, 'Willy', 4.0]]
after default sort: [[702, 'Willy', 4.0], [711, 'Jones', 3.0], [1190, 'Smith', 2.8]]
 after custom sort: [[1190, 'Smith', 2.8], [711, 'Jones', 3.0], [702, 'Willy', 4.0]]
```

3.5 Exercise 7.6:

The list below contains 4 rectangles defined by width and height. Sort the list based on the area of each rectangle using an appropriate comparison function. rects = [(5,2), (7,13), (9,4), (17,4)]

Go to the Exercise 7.6. PDF on Learn@UW