LISTS WITH PYTHON

José M. Garrido
Department of Computer Science

May 2015

College of Computing and Software Engineering
Kennesaw State University

Lists with Python

1 Lists

A list in Python is simply an ordered collection of items each of which can be of any type. A list is a dynamic mutable data structure and this means that items can be added to and deleted from it. The list data structure is the most common data *sequence* in Python. A sequence is a set of values identified by integer indices.

To define a list in Python, the items are written separated by commas and in square brackets. A simple list with name *vv* and n items is defined as follows:

$$vv = [p_1,\ p_2,\ p_3,\ \ldots,\ p_n]$$

For example, the command that follows defines a list with name *vals* and six data items:

```
vals = [1,2,3,4,5,6]
```

1.1 Indexing Lists

An individual item in the list can be referenced by using an index, which is an integer number that indicates the relative position of the item in the list. The values of index numbers always start at zero. In the the list *vals* defined previously, the index values are: $0, 1, 2, 3, 4$ and 5.

In Python, the reference to an individual item of a list is written with the name of the list and the index value or an index variable within brackets. The following Python commands in interactive mode define the list *vals*, reference the first item on the list with index value 0, reference the fourth item with index value 3, then use an index variable *idx* with a value of 4 to reference an item of list *vals*.

```
>>> vals = [1, 2, 3, 4, 5, 6]
>>> vals
[1, 2, 3, 4, 5, 6]
>>> vals[0]
1
>>> vals[3]
4
>>> idx = 4
>>> vals[idx]
5
```

In non-interactive mode, the command `print vals[idx]` is used to display the value of the list item indexed with variable *idx*.

Using an index value of -1 is used to reference the last item of a list and an index value of -2 is used to reference the previous to last item of the list. The following commands also in interactive mode illustrate this.

```
>>> vals[-1]
6
>>> vals[-2]
5
```

Because a list is a mutable data structure, the items of the list can change value by performing assignment on them. The second of the following Python commands assigns the new value of 23.55 to item that has index value 3.

```
>>> vals
[1, 2, 3, 4, 5, 6]
>>> vals [3] = 23.55
>>> vals
[1, 2, 3, 23.55, 5, 6]
```

1.2 Slicing Operations

The slicing operations are used to access a sublist of the list. The colon notation is used to specify the range of index values of the items. The first index value is written before the colon and the last index value is written after the colon. This indicates the range of index values from the start index value up to but not including the last index value specified.

In the following example, which uses Python in interactive mode, the second command specifies a sublist of list *vals*, that includes the items starting with index value 0 up to but not including the item with index value 4. The third command assigns the sublist `vals[2:5]` to variable y; so this command creates a new sublist and assigns it to y.

```
>>> vals
[1, 2, 3, 4, 5, 6]
>>> vals[0:4]
[1, 2, 3, 4]
>>> y = vals[2:5]
>>> y
[3, 4, 5]
```

A range of items can be updated using slicing and assignment. For example, the following command changes the values of items with index 0 and up to but not including the item with index value 2.

```
>>> vals[0:2] = vals[1:3]
>>> vals
[2, 3, 3, 23.55, 5, 6]
```

Using slicing, the second index value can be left out and implies that the range of index values starts from the item with the index value specified to the last item of the list. In a similar manner, the first index value can be left out and implies that the range of items starts with the first item of the list.

```
>>> vals[1:]
[3, 3, 23.55, 5, 6]
>>> vals[:5]
[2, 3, 3, 23.55, 5]
```

The first useful operation on a list is to get the number of items in a list. Function *len* is called to get the number of items from the list specified in parenthesis. In the following commands, the first command gets the length of list *vals* and assigns this value to variable n. The next command shows the value of n, which is 6 because *vals* has six items. The next command calls function *range* to generate another list starting at 0 and the last value is 5 (one before 6). Recall that function *range* was used in the for-loop discussed previously. The next command combines functions *range* and *len* to produce the same result as the previous command.

```
>>> n = len(vals)
>>> n
6
>>> range(n)
[0, 1, 2, 3, 4, 5]
>>> range(len(vals))
[0, 1, 2, 3, 4, 5]
```

1.3 Iterating Over a List with a Loop

Indexing is very useful to access the items of a list iteratively in a loop. A for-loop accesses the items of a list one by one by iterating over the index values of the list.

Listing 1 computes the summation of the items in list *vals2* and selects only the ones that have a value ≤ 3.15. The Python script is stored in file `sumlist.py`.

Listing 1 Python program for computing the summation on a list.

```
# Script: sumlist.py
# Compute the summation of the values in a list
# that are less or equal to 3.15 using a loop
#
vals2 = [2, 3.45, 1.22, 4.87, 0.78, 2.45, 8.76]
nl = range ( len(vals2))
sum = 0.0
for i in nl:
   if vals2[i] <= 3.15 :
      print "Index: ", i
      sum = sum + vals2[i]
print "Summation: ", sum
```

The following output listing shows the shell commands that start the Python interpreter with file `sumlist.py`, and the results computed.

```
$ python sumlist.py
Index:  0
Index:  2
Index:  4
Index:  5
Summation:  6.45
```

Python supports iterating directly over the items of a list using the *for* statement. Listing 2 shows a Python program that computes the same summation of the list discussed in the previous problem. Instead of using indexing, this program iterate over the items of the list *vals2* and the final result is the same as in the previous program. Note that the variable *item* refers to the value of an individual item of the list. This Python script is stored in file `sumlistb.py`.

Listing 2 Python program for computing the summation on a list.

```
# Script: sumlistb.py
# Compute the summation of the values in a list
# that are less or equal to 3.15 by
# using a loop to iterate over the items of a list
#
vals2 = [2, 3.45, 1.22, 4.87, 0.78, 2.45, 8.76]
```

```
sum = 0.0
for item in vals2:
   if item <= 3.15 :
      print "item ", item
      sum = sum + item
print "Summation: ", sum
```

The following output listing shows the shell commands that start the Python interpreter with file `sumlistb.py` and the results.

```
$ python sumlistb.py
item  2
item  1.22
item  0.78
item  2.45
Summation:  6.45
```

1.4 Creating a List Using a Loop

A list can be created starting with an empty list; items can be appended using a for-loop. The *append* method is an operation of a list and is very useful for creating a list. The following command appends the value of a new item v to a list *mlist*:

```
mlist.append(v)
```

The following example builds a list of items with values that are multiples of 5. Listing 3 shows a Python program that builds the list starting with an item with value 5. Line 8 has a *for* statement that defines the loop that iterates with loop counter j starting with 1 upt to *SIZE*. The value of the current item is computed in line 9 and it is placed at the end of the current list using the *append* list method in line 10. Note that the variable *item* refers to the value of an individual item of the list. This Python script is stored in file `blist5.py`.

Listing 3 Python program with a list of values that are multiples of 5.

```
1 # Script: blist5.py
2 # Python script to build a list with
3 # items with values multiple of 5
4
5 SIZE = 15     # number of items in list
6 listmf = []   # create empty list
```

```
# the list starts with 5
for j in range (1, SIZE+1):
     item = j * 5
     listmf.append(item)

print "List is: ", listmf
```

The following output listing shows the shell commands that start the Python interpreter with file `blist5.py` and the results.

```
$ python blist5.py
List is:  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

1.5 Passing Lists to a Function

One or more lists can be passed to a function and the lists are used as arguments in the function call. The lists are specified as parameters in the function definition.

Listing 4 shows a Python program that includes a function definition *buildlf* starting in line 8. The function has two parameters: a list *llist* and simple variable *lsize*. The function builds the list by appending items into it. The function call with two arguments appears in line 15 and the results are displayed by the instruction in line 16.

Listing 4 Program that builds a list with values that are multiples of 5.

```
 1 # Script: blist5f.py
 2 # Python script to build a list with
 3 # items with values multiple of 5
 4 # using function buildlf
 5
 6 #Function that builds a list
 7 # the list starts with 5
 8 def buildlf (llist, lsize):
 9     for j in range (1, lsize+1):
10         item = j * 5
11         llist.append(item)
12
13 SIZE = 10     # number of items in list
14 listmf = []   # create empty list
15 buildlf(listmf, SIZE)
16 print "List is: ", listmf
```

The following output listing shows the shell commands that start the Python interpreter with file `blist5f.py` and the results.

```
$ python blist5f.py
List is:  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

A list can be returned by a function and the list must be defined in the function. Listing 5 shows a Python program that includes a function definition *buildlg* starting in line 8. This function builds a list and then returns it in line 13. Note that the function includes only one parameter: *lsize*. The script calls in the function in an assignment statement in line 16.

Listing 5 Python program that builds a list with values that are multiples of 5.

```
 1 # Script: blist5g.py
 2 # Python script to build a list with
 3 # items with values multiple of 5
 4 # using function buildlg
 5
 6 #Function that builds a list
 7 # the list starts with 5
 8 def buildlg (lsize):
 9     llist = [] # create empty list
10     for j in range (1, lsize+1):
11         item = j * 5
12         llist.append(item)
13     return llist
14
15 SIZE = 10     # number of items in list
16 listmf = buildlg(SIZE)
17 print "List is: ", listmf
```

The results are the same as the previous two Python scripts. The following output listing shows the shell commands that start the Python interpreter with file `blist5g.py` and the results.

```
$ python blist5g.py
List is:  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

1.6 Additional Operations on Lists

In addition to list method *append*, there are several methods for lists provided in Python. The *extend* method appends another list to the current list. The following commands using Python in interactive mode define two lists *listm* and *lst2*, then append the second list to the first list.

```
>>> listm = [5, 10, 15, 20, 25, 30]
>>> lst2 = [35, 40]
>>> listm.extend(lst2)
>>> listm
[5, 10, 15, 20, 25, 30, 35, 40]
```

List method *insert* places an item at a given position of a list. Calling this method requires two arguments, the first is the index value before which the new item is to inserted, the second argument is the value of the item to be inserted to the list. The following command inserts an item with value 33 into list *listm* at position with index value 5.

```
>>> listm.insert(5, 33)
>>> listm
[5, 10, 15, 20, 25, 33, 30, 35, 40]
```

List method *remove* searches for the first item with the specified value and removes from the list. The following command removes the item with value 35 from the list *listm*.

```
>>> listm.remove(35)
>>> listm
[5, 10, 15, 20, 25, 33, 30, 40]
```

List method *pop* removes the last item from the specified list and returns the value of the item. The following command removes and displays the last item from list *listm*.

```
>>> listm.pop()
40
>>> listm
[5, 10, 15, 20, 25, 33, 30]
```

List method *index* finds the first item with the value specified and returns its index value. The following command gets and returns the index value of the item that has value 33.

```
>>> listm.index(33)
5
```

List method *sort* rearranges in ascending order the items of a list. The following commands sort the items (in ascending order) in list *listm* and displays the list again.

```
>>> listm.sort()
>>> listm
[5, 10, 15, 20, 25, 30, 33]
```

List method *reverse* rearranges the items of a list in reverse order. The following command reverses the items in list *listm*.

```
>>> listm.reverse()
>>> listm
[33, 30, 25, 20, 15, 10, 5]
```

2 Temperature Conversion Problem

The temperature conversion problem was discussed in the previous chapter. The description of the revised problem is: given a list of values of temperature in degrees Celsius, compute the corresponding values in degrees Fahrenheit and show this result.

2.1 Mathematical Model

The mathematical representation of the solution to the problem, the formula expressing a temperature measurement F in Fahrenheit in terms of the temperature measurement C in Celsius is:

$$F = \frac{9}{5}\,C + 32$$

The solution to the problem applies the mathematical expression for the conversion of a temperature measurement in Celsius to the corresponding value in

Fahrenheit. The mathematical formula expressing the conversion assigns a value to the desired temperature in the variable *itemF*, the dependent variable. The values of the variable *itemC* can change arbitrarily because it is the independent variable. The mathematical model uses floating-point numbers to represent the temperature readings in various temperature units.

2.2 The Python Implementation

The solution to his problem is implemented in Python using lists. Variable *itemC* refers to a value of the temperature in Celsius and variable *itemF* refers to the corresponding value of the temperature in Fahrenheit. All values of the temperatures in Celsius are placed in list *listC* and all values computed of the temperature in Fahrenheit are placed in list *listF*.

Listing 6 shows a Python program that computes the temperature in Fahrenheit for every value in the list of temperature in Celsius. This program uses a loop in which the two lists are built by appending a new item to the lists. This Python script is stored in file `tconvctfl.py`.

Listing 6 Python program for temperature conversion on a list.

```
# Program     : tconvctfl.py
# Author      : Jose M Garrido
# Date        : 6-02-2014
# Description : Read values of temperature in Celsius
# from console, convert to degrees Fahrenheit, and
# display corresponding values of temperature
# in fahrenheit on screen

SIZE = 15     # number of items in list
listC = []    # create empty list for temp in Celsius
listF = []    # create empty list for temp in Fahrenheit
# listC starts with 5
for j in range (1, SIZE+1):
     itemC = j * 5
     listC.append(itemC)
     itemF = itemC * (9.0/5.0) + 32.0 # temp in Fahrenheit
     listF.append(itemF)

print "Values of temperature in Celsius: "
print listC
print "Values of temperature in Fahrenheit: "
print listF
```

The following listing shows the shell command that starts the Python interpreter with file `tconvctfl.py` and the results.

```
$ python tconvctfl.py
Values of temperature in Celsius:
  [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75]
Values of temperature in Fahrenheit:
  [41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0, 104.0, 113.0,
   122.0, 131.0, 140.0, 149.0, 158.0, 167.0]
```

2.3 Implementation Using a Function

Listing 5 shows a Python script that solves the same problem as the previous script. It defines function *tconvf* in lines $9 - 16$. This function takes a list of values of temperature in Celsius, computes the temperature in Fahrenheit, and returns these values in a new list. This Python script creates a list of values of temperature in Celsius in line 18, calls function *tconvf* in line 19 using the list as the argument, then displays the two lists in lines $20 - 23$. This script is stored in file `tconvs.py`.

Listing 5 Python program calls a function for temperature conversion.

```
 1 #  Program     : tconvs.py
 2 #  Author      : Jose M Garrido
 3 #  Date        : 6-02-2014
 4 #  Description : Given a list of values of temperature in
 5 #  Celsius, convert to degrees Fahrenheit, and return a
 6 #  list of values in Fahrenheit. This script defines and
 7 #  calls function 'tconvf'
 8
 9 def tconvf (listC):
10    # listC list of temperature values in Celsius
11    listF = []    # empty list for temp in Fahrenheit
12    size = len(listC)
13    for j in range (0, size):
14        itemF = listC[j] * (9.0/5.0) + 32.0
15        listF.append(itemF)
16    return listF
17
18 c = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55]
19 f = tconvf(c)
20 print "Values of temperature in Celsius: "
21 print c
22 print "Values of temperature in Fahrenheit: "
23 print f
```

The following listing shows the shell command that starts the Python interpreter with file `tconvs.py` and the results.

```
$ python tconvs.py
Values of temperature in Celsius:
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55]
Values of temperature in Fahrenheit:
[32.0, 41.0, 50.0, 59.0, 68.0, 77.0, 86.0, 95.0, 104.0, 113.0,
 122.0, 131.0]
```

3 List Comprehensions

A list comprehension is a compact notation in Python for generating a list of a given size and with the elements initialized according to the specified expression. The following example generates a list *ll* with 12 elements all initialized with value 1.

```
>>> lsize = 12
>>> ll = [ 1 for j in range(lsize) ]
>>> ll
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

The same list can be generated using a for loop and function *append*, as the following example shows.

```
>>> ll = []
>>> for j in range(lsize):
...    ll.append(1)
...
>>> ll
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

In the notation for list comprehension, the expression appears first followed by one or more *for* clauses and all within brackets. The following example generates a list *ll* of size *lsize* with the elements initialized to a value from the expression $j + 12.5$.

```
>>> ll = [ j+12.5 for j in range(lsize)]
>>> ll
[12.5, 13.5, 14.5, 15.5, 16.5, 17.5, 18.5, 19.5, 20.5,
      21.5, 22.5, 23.5]
```

4 Lists of Lists

Lists of lists are also known as *nested lists* and means that one or more items in a list are also lists. Multidimensional arrays can be defined with nested lists. In the following example, the first command creates list *lst1* with four items. The second command creates list *lst2* and its third item is list *lst1*. The third command displays list *lst2* and the last command shows that the length of list *lst2* is 5.

```
>>> lst1 = [12, 54, 2, 9]
>>> lst2 = [99, 5, lst1, 20, 7]
>>> lst2
[99, 5, [12, 54, 2, 9], 20, 7]
>>> len (lst2)
5
```

To reference an item of a list that is part of a larger list, two indices are required. The first index refers to an item in the outer list, the second index refers to an item in the inner list. In the following example, the first command uses index value 2 to reference the third item of list *lst2* and this item is the inner list *lst1*. The second command uses two index values, the first index value 2 indicates the third item of list *list2* and the second index value 3 references the fourth item of the inner list *lst1*, which has value 9.

```
>>> lst2 [2]
[12, 54, 2, 9]
>>> lst2 [2][3]
9
```

The following commands create a small matrix *smatrix* with two rows and three columns, references the element of the second row and third column, and assign the value to variable *eval*. In a similar manner, the element of the first row and second column is referenced and its value is assigned to variable *fval*.

```
>>> smatrix = [[9, 2, 5], [4, 8, 6]]
>>> smatrix
[[9, 2, 5], [4, 8, 6]]
>>> eval = smatrix[1][2]
>>> eval
6
>>> fval = smatrix[0][1]
>>> fval
2
```

The following example generates a 3 by 5 list, that is, a list with three rows and five columns. The outer *for* loop is used to generate a row list, the inner *for* loop generates all the elements in a row initialized to value 1.

```
>>> nll = [[]]
>>> for i in range(3):
...       row = []
...       for j in range (5):
...           row.append(1)
...       nll.append(row)
...
>>> nll
[[], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

The following command calls list function *pop* to remove the first element from the list, which is an empty list.

```
>>> nll.pop(0)
[]
>>> nll
[[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

Nested list comprehensions are used to generate multi-dimensional lists initialized to a value according to a specified expression. The following example generates a 3 by 5 list, that is, a list with three rows and five columns. The inner (first) *for* clause is used to generate the values in a row, the second *for* clause generates all the rows.

```
>>> lll = [[1 for i in range(5)] for j in range (3)]
>>> lll
[[1, 1, 1, 1, 1], [1, 1, 1, 1, 1], [1, 1, 1, 1, 1]]
```

5 Tuples

A tuple is a Python sequence similar to a list. To create a tuple of items, write the values separated by commas. It is often convenient to enclose the items in parenthesis. For example:

```
>>> xt = (4.5, 6, 78, 19)
>>> xt
(4.5, 6, 78, 19)
```

A tuple is immutable, which means that after creating a tuple it cannot be changed. The value of the elements in a tuple cannot be altered, elements cannot be added or removed from the tuple.

As with lists, the individual elements of a tuple can be referenced by using an index value. In the following example, the third element of tuple *xt* is accessed and its value is assigned to variable *yy*.

```
>>> yy = xt[2]
>>> yy
78
```

Tuples can be nested, which means tuples of tuples can be created. For example, to a create a tuple *xt2* that includes tuple *xt* as its second element.

```
>>> xt2 = (25, xt, 16)
>>> xt2
(25, (4.5, 6, 78, 19), 16)
```

Method *len* can be used to get the number of elements in a tuple. The following assignment statement gets the length of tuple *xt* and assigns this value to variable *lxt*.

```
>>> lxt = len (xt)
>>> lxt
4
```

A tuple can be converted to a list by calling method *list*. For example, the following command converts tuple *xt* to a list *llxt*.

```
>>> llxt = list (xt)
>>> llxt
[4.5, 6, 78, 19]
```

A list can be converted to a tuple by calling method *tuple*. For example, the following commands create list *vals* then convert the list to a tuple *mytuple*.

```
>>> vals = [1, 2, 3, 4, 5, 6]
>>> vals
[1, 2, 3, 4, 5, 6]
>>> mytuple = tuple (vals)
>>> mytuple
(1, 2, 3, 4, 5, 6)
```

It is possible to build lists of tuples and tuples of lists. The following command defines a list *myltt* of tuples.

```
>>> myltt = [(12, 45.25), (45, 68.5), (25, 78.95)]
>>> myltt
[(12, 45.25), (45, 68.5), (25, 78.95)]
```

The following command defines a tuple *mytup* of lists.

```
>>> mytup = ([14, 45.25], [55, 68.5], [28, 78.95])
>>> mytup
([14, 45.25], [55, 68.5], [28, 78.95])
```

6 Dictionaries

Dictionaries are also Python data structures except that these are indexed by *keys*. A dictionary is used as an unordered set of *key* and *value* pairs that are enclosed in curly braces. The key can be any immutable type and must be unique. The corresponding value associated with a key is written after a semicolon and following the key.

The following example creates a dictionary of three key-value pairs separated by commas. Note that the keys are strings in this example. The last command extracts the value of the pair that has given key *'price'*.

```
>>> mydict = {'desc': 'valve 5in', 'price': 23.75, 'quantity': 54}
>>> mydict
{'price': 23.75, 'quantity': 54, 'desc': 'valve 5in'}
>>> mydict['price']
23.75
```

The value of a pair in a dictionary can be updated given the key. The following example changes the value of price to 25.30.

```
>>> mydict['price'] = 25.30
>>> mydict
{'price': 25.3, 'quantity': 54, 'desc': 'valve 5in'}
```

The *in* keyword is used to check whether a key appears on a the given dictionary. The following commands checks the dictionary *mydict* for the key *'desc'*.

```
>>> 'desc' in mydict
True
```

The dictionary method *keys* is used to get a list of all the keys in a given dictionary. The following command gets a list of the keys in *mydict*.

```
>>> mydict.keys()
['price', 'quantity', 'desc']
```

Given a list of two-tuples, it can be converted to a dictionary by calling function *dict*. The following commands define a list of two-tuples (tuples with two values) *myltt* and convert this list to a dictionary.

```
>>> myltt = [(12, 45.25), (45, 68.5), (25, 78.95)]
>>> myltt
[(12, 45.25), (45, 68.5), (25, 78.95)]
>>> mdict3 = dict (myltt)
>>> mdict3
{25: 78.95, 12: 45.25, 45: 68.5}
```

7 Strings

A string is a sequence of text characters in a particular character encoding. Syntactically, a string literal is enclosed in single or double quotes and can be assigned to a variable. Strings are immutable, once defined strings cannot be modified. The following command defines a string literal and assigns the reference to variable *mystr*.

```
>>> mystr = 'State University'
>>> mystr
'State University'
```

One of the most commonly used operations is concatenation. It joins two or more strings and creates a longer string. The concatenation string operator is the plus sign (+). The following command concatenates two strings and the newly created string is assigned to variable *str2*.

```
>>> str2 = 'Kennesaw ' + mystr
>>> str2
'Kennesaw State University'
```

Function *len* gets the length of a string, that is, the number of characters in the string. The following command gets the number of characters in the string referenced by variable *str2*.

```
>>> s = len (str2)
>>> s
25
```

Indexing is used to access particular characters of a string. Index values start at zero and are written withing brackets. The following command prints the character with index 5 of string *str2*, which is `s`.

```
>>> print str2[5]
s
```

The character accessed in a string can be assigned to another variable, as the next example shows.

```
>>> x = str2[5]
>>> x
's'
```

The slicing operation is used to access a subset of a string. This operation requires the slice operator (:) and indexes that specify the range of characters to include. In the following example, the first command prints the subset of string *str2* specified from the second character (with index 1) up to the fourth character (with index 4). The second command assigns to variable *strx* a subset of string *str2* specified from the first character to the fifth character.

```
>>> print str2[1:4]
enn
>>> strx = str2[0:5]
>>> strx
'Kenne'
```

The following example creates a new string that consists of "Kansas" concatenated with a subset of *str2* that starts with ninth character to the end of *str2*.

```
>>> nstrx = "Kansas" + str2[8:]
>>> nstrx
'Kansas State University'
```

The membership operator *in* is used to check whether a character or a substring is contained in another string and returns `True` or `False`. In the following example, the first command checks if the character `'K'` belongs in string *nstrx*. The second command checks if the substring *strx* is contained in the string *str2*.

```
>>> 'K' in nstrx
True
>>> strx in str2
True
```

Some characters in a string and escape characters and are written preceded with a backslash (\). For example when a newline character is part of a string, that indicates that a change of line occurs at that point and the rest of the string appears on a new line. The following example shows a string that is to be displayed on two lines.

```
>>> message = "Start the program now\n click on the icon"
>>> message
'Start the program now\n click on the icon'
>>> print message
Start the program now
 click on the icon
```

When a string is enclosed in double quotes (") but one or more double quotes are also used as part of the string, these have to be escaped with a backslash as in the following example. The same applies with single quotes.

```
>>> mess1 = "The program responds with \"welcome\" then waits"
>>> print mess1
The program responds with "welcome" then waits
```

Python provides several string methods and the general form to call these methods is:

```
string_name.method_name(arguments)
```

For example, method *find* is used to find the index value in string *nstrx* where the substring `"State"` starts, as in the following example.

```
>>> nstrx
'Kansas State University'
>>> nstrx.find("State")
7
```

Another very useful string method is *isdigit* and is used to check whether all characters in a string are decimal digits. The following example checks whether string *nn* contains only decimal digits.

```
>>> nn = "54321"
>>> nn.isdigit()
True
```

8 Simple Numerical Applications Using Lists

This section discusses several simple applications of arrays as lists; a few of these applications perform simple manipulation of arrays, other applications perform slightly more complex operations with arrays such as searching and sorting.

The problems discussed in this section compute the average value and the maximum value in an array named *varr*. The algorithms that solve these problems examine all the elements of the array.

8.1 The Average Value in an Array

To compute the average value in an array, the algorithm is designed to first compute the summation of all the elements in the array, the accumulator variable *sum* is used to store this. Second, the algorithm computes the average value by diving the value of *sum* by the number of elements in the array. The following listing has the pseudo-code description of the algorithm.

1. Initialize the value of the accumulator variable, *sum*, to zero.
2. For every element of the array, add its value to the accumulator variable *sum*.
3. Divide the value of the accumulator variable by the number of elements in the array, *num*.

The accumulator variable *sum* stores the summation of the element values in the array named *varr* with *num* elements. The average value, *ave*, of array *varr* using index j starting with $j = 1$ to $j = n$ is expressed mathematically as:

$$ave = \frac{1}{num} \sum_{j=1}^{num} varr_j$$

Listing 7 shows the Python script that implement the algorithm that computes the average value of the elements in the array. This code is stored in the script file `aver.py`.

Listing 7: Python script file for computing average in a list.

```
# Python script file to compute average value in a list
# This script inputs the array size
# and the elements of the array from the console
# Computes the average value in the array
# File: aver.py
num = input('Enter array size: ');
varr = []  # empty list
for j in range(0, num):
     item = input('Enter array element: ')
     varr.append(item)

# Now compute the average value in list
sum = 0.0
for j in range(0, num):
     print "index: ", j, " value: ", varr[j]
     sum = sum + varr[j]
```

```
ave = sum/num
print "Average value: ", ave
```

The following output listing shows the result of executing the script *aver.py* at the Linux prompt.

```
$ python aver.py
Enter array size: 4
Enter array element: 9.75
Enter array element: 8.34
Enter array element: 7.25
Enter array element: 6.77
index:  0  value:  9.75
index:  1  value:  8.34
index:  2  value:  7.25
index:  3  value:  6.77
Average value:  8.0275
```

8.2 Maximum Value in a List

Consider a problem that deals with finding the maximum value in an array named *varr*. The algorithm with the solution to this problem also examines all the elements of the array.

The variable *max_arr* stores the maximum value found so far. The name of the index variable is *j*. The algorithm description is:

1. Read the value of the array size, *num*, and the value of the array elements.
2. Initialize the variable *max_arr* that stores the current largest value found (so far). This initial value is the value of the first element of the array.
3. Initialize the index variable (value zero).
4. For each of the other elements of the array, compare the value of the next array element; if the value of the current element is greater than the value of *max_arr* (the largest value so far), change the value of *max_arr* to this element value, and store the index value of the element in variable *k*.
5. The index value of variable *k* is the index of the element with the largest value in the array.

Listing 8 contains the Python script that implements the algorithm for finding the maximum value in an array; the script is stored in the file `arrmax.py`. As in the previous examples, the list is first created by reading the values of the elements, in lines 6-10. Finding the maximum value in the list is preformed in lines 13-18

Listing 8: Python script file for finding maximum value in a list.

```
 1 # Python script file to find the maximum value in a list
 2 # This script inputs the array size
 3 # and the elements of the array from the console
 4 # Computes the maximum value in the array
 5 # File: arrmax.py
 6 num = input ('Enter array size: ')
 7 varr = []  # empty list
 8 for j in range(0, num):
 9      item = input('Enter array element: ')
10      varr.append(item)
11
12 # Now find the maximum value in list
13 max_arr = varr[0] # initial value of max_arr
14 for j in range(1, num):
15      print "index: ", j, " value: ", varr[j]
16      if varr[j] > max_arr:
17          k = j
18          max_arr = varr[j]
19
20 print "Index of max value: ", k
21 print "Max value: ", varr[k]
```

Executing the script file `arrmax.py` with the Python interpreter produces the following output.

```
$ python arrmax.py
Enter array size: 4
Enter array element: 5.56
Enter array element: 7.87
Enter array element: 3.78
Enter array element: 2.7
index:  1  value:  7.87
index:  2  value:  3.78
index:  3  value:  2.7
Index of max value:  1
Max value:  7.87
```

8.3 Searching

Looking for an array element with a particular value, known as the *key*, is called searching and involves examining some or all elements of an array. The search ends when and if an element of the array has a value equal to the requested value. Two general techniques for searching are: linear search and binary search.

8.3.1 Linear Search

A linear search algorithm examines the elements of an array in a *sequential* manner starting with the first element. The algorithm examines the first element of the array, then the next element and so on until the last element of the array. Every array element is compared with the key value, and if an array element is equal to the requested value, the algorithm has found the element and the search terminates. This may occur before the algorithm has examined all the elements of the array.

The result of this search is the index of the element in the array that is equal to the key value given. If the key value is not found, the algorithm indicates this with a negative result or in some other manner. The following is an algorithm description of a general linear search using a search condition of an element equal to the value of a *key*.

1. Repeat for every element of the array:
   - (a) Compare the current element with the requested value or key. If the value of the array element satisfies the condition, store the value of the index of the element found and terminate the search.
   - (b) If values are not equal, continue search.
2. If no element with value equal to the value requested is found, set the result to value -1.

The algorithm outputs the index value of the element that satisfies the search condition, whose value is equal to the requested value *kval*. If no element is found that satisfies the search condition, the algorithm outputs a negative value.

The Python script is stored in file `lsearch.py`. Listing 9 shows the Python commands that implement the algorithm that searches the list *llist* for the key value, *key*. In line 18 list *llist* is created. In line 20, there is a function call to *lsearchf* using two arguments: the list and the key value to search. The function definition appears in lines 8-16. The function returns the index value of the element found that is equal to the key value, or -1 if not found.

Listing 9: Script file for computing linear search in a list

```
# Python script for linear search
# it performs a linear search of the array varr
# looking for the value kvar
#    The algorithm sets the result, the index value of
#    the element found, or -1 if not found.
# File: lsearch.py

def lsearchf (varr, kval):
      # find the element in varr equal to kval
      found = False
      num = len (varr)
      for j in range(0, num):
            if found == False and varr [j] == kval:
               found = True
               return j
      return -1

llist = [23, 12, 19, 35, 22, 81, 14, 8, 33]
key = input ("Enter the key value: ")
result = lsearchf(llist, key)
if result >= 0:
     print 'Result index is: ', result
else:
     print 'Key not found'
```

Executing the script file `lsearch.py` with the Python interpreter produces the following output.

```
$ python lsearch.py
Enter the key value: 33
Result index is:  8
$ python lsearch.py
Enter the key value: 45
Key not found
```

8.3.2 Binary Search

Binary search is a more complex search method and is very efficient, compared to linear search because the number of comparisons is smaller.

A prerequisite for the binary search technique is that the element values in the array to be searched be sorted in ascending order. The array elements to search are split into two halves or partitions of about the same size. The middle element is compared with the key (requested) value. If the element with this value is not found, the search is continued on only one partition. This partition is again split into two smaller partitions until the element is found or until no more splits are possible because the element is not found.

With a search algorithm, the efficiency of the algorithm is determined by the number of compare operations with respect to the size of the array. The average number of comparisons with linear search for an array with N elements is $N/2$, and if the element is not found, the number of comparisons is N. With binary search, the number of comparisons is $\log_2 N$. The informal description of the algorithm is:

1. Assign the lower and upper bounds of the array to *lower* and *upper*.
2. While the lower value is less than the upper value, continue the search.
   (a) Split the array into two partitions. Compare the middle element with the key value.
   (b) If the value of the middle element is equal to the key value, terminate search and the result is the index of this element.
   (c) If the key value is less than the middle element, change the upper bound to the index of the middle element minus 1. Continue the search on the lower partition.
   (d) If the key value is greater or equal to the middle element, change the lower bound to the index of the middle element plus 1. Continue the search on the upper partition.
3. If the key value is not found in the array, the result is -1.

Listing 10 shows the Python script that implements the binary search algorithm. The script commands is stored in command file `bsearch.py`.

Listing 10: Python script for searching for a key using binary search.

```
# Python script that implements a binary search
# of list (array) llist using key value key.
# The result is the index value of
# the element found, or -1 if not found.
# File: bsearch.py

def bsearchf (varr, kval):
      # find the element in varr equal to kval
      num = len (varr)
```

```
      lower = 0
      upper = num - 1
      while lower <= upper :
            middle = (lower + upper) / 2
            if kval == varr[middle]:
                return middle;  # result
            else:
                if kval < varr[middle]:
                    upper = middle - 1
                else:
                    lower = middle + 1
      return -1  # not found

llist = [9, 10, 13, 61, 72, 82, 89, 95, 102]
key = input('Enter key value: ')
result = bsearchf(llist, key)
if result >= 0:
     print 'Result index is: ', result
else:
     print 'Key not found'
```

Executing the *bsearch* script with the Python interpreter produces the following output listing.

```
$ python bsearch.py
Enter key value: 81
Key not found
$ python bsearch.py
Enter key value: 72
Result index is:  4
```

Note that because of the limited precision of digital computers, it is not recommended that two floating-point values be tested for equality. Instead, a small fixed constant value is used to compare with the absolute difference of the two values.

The following example defines the symbol *EPSIL* as a constant with a relatively small value 0.000001 and is used to compare with the difference of the values of variables *aa* and *bb*. In this case, the two variables are not considered equal because their difference is not less or equal to *EPSIL*.

```
>>> EPSIL = 0.000001
>>> aa = 46.005
>>> bb = 46.0055
```

```
>>> aa - bb
-0.0004999999999995282
>>> abs(aa - bb) <= EPSIL
False
```