```
>>> a
[1, 2, 3, -8, 2, 4, 2, 5]
```

Remove occurrences of 3:

```
>>> a.remove(3)
>>> a
[1, 2, -8, 2, 4, 2, 5]
```

Remove occurrences of 1:

```
>>> a.remove(1)
>>> a
[2, -8, 2, 4, 2, 5]
```

Append 19 to the list:

```
>>> a.append(19)
>>> a
[2, -8, 2, 4, 2, 5, 19]
```

Print the index of 19 in the list:

```
>>> a.index(19)
6
```

Reverse the list:

```
>>> a.reverse()
>>> a
[19, 5, 2, 4, 2, -8, 2]
```

Sort the list:

```
>>> a.sort()
>>> a
[-8, 2, 2, 2, 4, 5, 19]
```

Extend list a with list b:

```
>>> b = [100,200,300]
>>> a.extend(b)
>>> a
[-8, 2, 2, 2, 4, 5, 19, 100, 200, 300]
```

Remove the first occurrence of 2:

```
>>> a.pop(2)
2
>>> a
```

```
[-8, 2, 2, 4, 5, 19, 100, 200, 300]
```

Remove the last item of the list:

```
>>> a.pop()
300
>>> a
[-8, 2, 2, 4, 5, 19, 100, 200]
```

Now that you understand how to use list-related operations, the next section shows you how to use a Python list as a stack.

3.12 Using a List as a Stack and a Queue

A stack is a LIFO ("Last In First Out") data structure with `push()` and `pop()` functions for adding and removing elements, respectively. The most recently added element in a stack is in the top position, and therefore the first element that can be removed from the stack.

The following code block illustrates how to create a stack and also remove and append items from a stack in Python. Create a Python list (which we'll use as a stack):

```
>>> s = [1,2,3,4]
```

Append 5 to the stack:

```
>>> s.append(5)
>>> s
[1, 2, 3, 4, 5]
```

Remove the last element from the stack:

```
>>> s.pop()
5
>>> s
[1, 2, 3, 4]
```

A queue is a FIFO ("First In First Out") data structure with `insert()` and `pop()` functions for inserting and removing elements, respectively. The most recently added element in a queue is in the top position, and therefore the last element that can be removed from the queue.

The following code block illustrates how to create a queue and also insert and append items to a queue in Python.

Create a Python list (which we'll use as a queue):

```
>>> q = [1,2,3,4]
```

Insert 5 at the beginning of the queue:

```
>>> q.insert(0,5)
>>> q
[5, 1, 2, 3, 4]
```

Remove the last element from the queue:

```
>>> q.pop(0)
1
>>> q
[5, 2, 3, 4]
```

The preceding code uses `q.insert(0, 5)` to insert in the beginning and `q.pop()` to remove from the end. However, keep in mind that the `insert()` operation is slow in Python: insert at `0` requires copying all the elements in underlying array down one space. Therefore, use `collections.deque` with `coll.appendleft()` and `coll.pop()`, where `coll` is an instance of the `Collection` class.

The next section shows you how to work with vectors in Python.

3.13 Working with Vectors

A vector is a one-dimensional array of values, and you can perform vector-based operations, such as addition, subtraction, and inner product. Listing 3.6 displays the contents of `MyVectors.py` that illustrates how to perform vector-based operations.

Listing 3.6: MyVectors.py

```
v1 = [1,2,3]
v2 = [1,2,3]
v3 = [5,5,5]

s1 = [0,0,0]
d1 = [0,0,0]
p1 = 0

print("Initial Vectors"
print('v1:',v1)
print('v2:',v2)
```

```
print('v3:',v3)

for i in range(len(v1)):
    d1[i] = v3[i] - v2[i]
    s1[i] = v3[i] + v2[i]
    p1    = v3[i] * v2[i] + p1

print("After operations")
print('d1:',d1)
print('s1:',s1)
print('p1:',p1)
```

Listing 3.6 starts with the definition of three lists in Python, each of which represents a vector. The lists `d1` and `s1` represent the difference of `v2` and the sum `v2`, respectively. The number `p1` represents the "inner product" (also called the "dot product") of `v3` and `v2`. The output from Listing 3.6 is here:

```
Initial Vectors
v1: [1, 2, 3]
v2: [1, 2, 3]
v3: [5, 5, 5]
After operations
d1: [4, 3, 2]
s1: [6, 7, 8]
p1: 30
```

3.14 Working with Matrices

A two-dimensional matrix is a two-dimensional array of values, and you can easily create such a matrix. For example, the following code block illustrates how to access different elements in a 2D matrix:

```
mm = [["a","b","c"],["d","e","f"],["g","h","i"]];
print 'mm:      ',mm
print 'mm[0]:   ',mm[0]
print 'mm[0][1]:',mm[0][1]
```

The output from the preceding code block is here:

```
mm:       [['a', 'b', 'c'], ['d', 'e', 'f'], ['g', 'h',
'i']]
mm[0]:    ['a', 'b', 'c']
mm[0][1]: b
```

Listing 3.7 displays the contents of My2DMatrix.py that illustrates how to create and populate 2 two-dimensional matrix.

Listing 3.7: My2DMatrix.py

```
rows = 3
cols = 3

my2DMatrix = [[0 for i in range(rows)] for j in
range(rows)]
print('Before:',my2DMatrix)

for row in range(rows):
  for col in range(cols):
    my2DMatrix[row][col] = row*row+col*col
print('After: ',my2DMatrix)
```

Listing 3.7 initializes the variables rows and cols and then uses them to create the rows x cols matrix my2DMatrix whose values are initially 0. The next part of Listing 3.7 contains a nested loop that initializes the element of my2DMatrix whose position is (row,col) with the value row*row+col*col. The last line of code in Listing 3.7 prints the contents of my2DArray. The output from Listing 3.7 is here:

```
Before: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
After:  [[0, 1, 4], [1, 2, 5], [4, 5, 8]]
```

3.15 The NumPy Library for Matrices

The NumPy library (which you can install via pip) has a matrix object for manipulating matrices in Python. The following examples illustrate some of the features of NumPy.

Initialize a matrix m and then display its contents:

```
>>> import numpy as np
>>> m = np.matrix([[1,-2,3],[0,4,5],[7,8,-9]])
>>> m
matrix([[ 1, -2, 3],
        [ 0,  4,  5],
        [ 7,  8, -9]])
```

The next snippet returns the transpose of matrix m:

```
>>> m.T
```

```
matrix([[ 1, 0, 7],
        [-2,  4,  8],
        [ 3,  5, -9]])
```

The next snippet returns the inverse of matrix `m` (if it exists):

```
>>> m.I
matrix([[ 0.33043478, -0.02608696, 0.09565217],
        [-0.15217391,  0.13043478,  0.02173913],
        [ 0.12173913,  0.09565217, -0.0173913 ]])
```

The next snippet defines a vector `y` and then computes the product `m*v`:

```
>>> v = np.matrix([[2],[3],[4]])
>>> v
matrix([[2],[3],[4]])
>>> m * v
matrix([[ 8],[32],[ 2]])
```

The next snippet imports the `numpy.linalg` subpackage and then computes the determinant of the matrix `m`:

```
>>> import numpy.linalg
>>> numpy.linalg.det(m)
-229.99999999999983
```

The next snippet finds the eigenvalues of the matrix `m`:

```
>>> numpy.linalg.eigvals(m)
array([-13.11474312, 2.75956154, 6.35518158])
```

The next snippet finds solutions to the equation `m*x = v`:

```
>>> x = numpy.linalg.solve(m, v)
>>> x
matrix([[ 0.96521739],
        [ 0.17391304],
        [ 0.46086957]])
```

In addition to the preceding samples, the `NumPy` package provides additional functionality, which you can find by performing an Internet search for articles and tutorials.

3.16 Queues

A queue is a FIFO ("First In First Out") data structure. Thus, the oldest item in a queue is removed when a new item is added to a queue that is already full.

Earlier in the chapter you learned how to use a Python List to emulate a queue. However, there is also a queue object in Python. The following code snippets illustrate how to use a queue in Python.

```
>>> from collections import deque
>>> q = deque('',maxlen=10)
>>> for i in range(10,20):
...   q.append(i)
...
>>> print q
deque([10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
maxlen=10)
```

The next section shows you how to use tuples in Python.

3.17 Tuples (Immutable Lists)

Python supports a data type called a *tuple* that consists of comma-separated values without brackets (square brackets are for lists, round brackets are for arrays, and curly braces are for dictionaries). Various examples of Python tuples are here:

https://docs.python.org/3.6/tutorial/datastructures.html#tuples-and-sequences

The following code block illustrates how to create a tuple and create new tuples from an existing type in Python.

Define a Python tuple `t` as follows:

```
>>> t = 1,'a', 2,'hello',3
>>> t
(1, 'a', 2, 'hello', 3)
```

Display the first element of t:

```
>>> t[0]
1
```

Create a tuple `v` containing 10, 11, and `t`:

```
>>> v = 10,11,t
>>> v
(10, 11, (1, 'a', 2, 'hello', 3))
```

Try modifying an element of `t` (which is immutable):

```
>>> t[0] = 1000
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

Python "deduplication" is useful because you can remove duplicates from a set and obtain a list, as shown here:

```
>>> lst = list(set(lst))
```

Note: The "in" operator on a list to search is O(n) whereas the "in" operator on set is O(1).

The next section discusses Python sets.

3.18 Sets

A Python set in Python is an unordered collection that does not contain duplicate elements. Use curly braces or the `set()` function to create sets. Set objects support set-theoretic operations such as union, intersection, and difference.

Note: `set()` is required in order to create an empty set because `{}` creates an empty dictionary.

The following code block illustrates how to work with a Python set.

Create a list of elements:

```
>>> l = ['a', 'b', 'a', 'c']
```

Create a set from the preceding list:

```
>>> s = set(l)
>>> s
set(['a', 'c', 'b'])
```

Test if an element is in the set:

```
>>> 'a' in s
True
>>> 'd' in s
False
>>>
```

Create a set from a string:

```
>>> n = set('abacad')
>>> n
```

```
set(['a', 'c', 'b', 'd'])
>>>
```

Subtract n from s:

```
>>> s - n
set([])
```

Subtract s from n:

```
>>> n - s
set(['d'])
>>>
```

The union of s and n:

```
>>> s | n
set(['a', 'c', 'b', 'd'])
```

The intersection of s and n:

```
>>> s & n
set(['a', 'c', 'b'])
```

The exclusive-or of s and n:

```
>>> s ^ n
set(['d'])
```

The next section shows you how to work with Python dictionaries.

3.19 Dictionaries

Python has a key/value structure called a "dict" that is a hash table. A Python dictionary (and hash tables in general) can retrieve the value of a key in constant time, regardless of the number of entries in the dictionary (and the same is true for sets). You can think of a set as essentially just the keys (not the values) of a `dict` implementation.

The contents of a `dict` can be written as a series of key:value pairs, as shown here:

```
dict1 = {key1:value1, key2:value2, ... }
```

The "empty dict" is just an empty pair of curly braces `{}`.

3.19.1 Creating a Dictionary

A Python dictionary (or hash table) contains of colon-separated key/value bindings inside a pair of curly braces, as shown here:

```
dict1 = {}
dict1 = {'x' : 1, 'y' : 2}
```

The preceding code snippet defines `dict1` as an empty dictionary, and then adds two key/value bindings.

3.19.2 Displaying the Contents of a Dictionary

You can display the contents of `dict1` with the following code:

```
>>> dict1 = {'x':1,'y':2}
>>> dict1
{'y': 2, 'x': 1}
>>> dict1['x']
1
>>> dict1['y']
2
>>> dict1['z']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'z'
```

Note: Key/value bindings for a `dict` and a set are not necessarily stored in the same order that you defined them.

Python dictionaries also provide the `get` method in order to retrieve key values:

```
>>> dict1.get('x')
1
>>> dict1.get('y')
2
>>> dict1.get('z')
```

As you can see, the Python `get` method returns `None` (which is displayed as an empty string) instead of an error when referencing a key that is not defined in a dictionary.

You can also use `dict` comprehensions to create dictionaries from expressions, as shown here:

```
>>> {x: x**3 for x in (1, 2, 3)}
{1: 1, 2: 8, 3: 37}
```

3.19.3 Checking for Keys in a Dictionary

You can easily check for the presence of a key in a Python dictionary as follows:

```
>>> 'x' in dict1
True
>>> 'z' in dict1
False
```

Use square brackets for finding or setting a value in a dictionary. For example, `dict['abc']` finds the value associated with the key `'abc'`. You can use strings, numbers, and tuples work as key values, and you can use any type as the value.

If you access a value that is not in the `dict`, Python throws a `KeyError`. Consequently, use the "in" operator to check if the key is in the dict. Alternatively, use `dict.get(key)` which returns the value or `None` if the key is not present. You can even use the expression `get(key, not-found-string)` to specify the value to return if a key is not found.

3.19.4 Deleting Keys from a Dictionary

Launch the Python interpreter and enter the following commands:

```
>>> MyDict = {'x' : 5,  'y' : 7}
>>> MyDict['z'] = 13
>>> MyDict
{'y': 7, 'x': 5, 'z': 13}
>>> del MyDict['x']
>>> MyDict
{'y': 7, 'z': 13}
>>> MyDict.keys()
['y', 'z']
>>> MyDict.values()
[13, 7]
>>> 'z' in MyDict
True
```

3.19.5 Iterating through a Dictionary

The following code snippet shows you how to iterate through a dictionary:

```
MyDict = {'x' : 5,  'y' : 7, 'z' : 13}

for key, value in MyDict.iteritems():
    print key, value
```

The output from the preceding code block is here:

```
y 7
x 5
```

```
z 13
```

3.19.6 Interpolating Data from a Dictionary

The % operator substitutes values from a Python dictionary into a string by name. Listing 3.8 contains an example of doing so.

Listing 3.8: InterpolateDict1.py

```
hash = {}
hash['beverage'] = 'coffee'
hash['count'] = 3

# %d for int, %s for string
s = 'Today I drank %(count)d cups of %(beverage)s' %
hash
print('s:', s)
```

The output from the preceding code block is here:

```
Today I drank 3 cups of coffee
```

3.20 Dictionary Functions and Methods

Python provides various functions and methods for a Python dictionary, such as `cmp()`, `len()`, and `str()` that compare two dictionaries, return the length of a dictionary, and display a string representation of a dictionary, respectively.

You can also manipulate the contents of a Python dictionary using the functions `clear()` to remove all elements, `copy()` to return a shall copy, get() to retrieve the value of a key, items() to display the (key,value) pairs of a dictionary, keys() to displays the keys of a dictionary, and values() to return the list of values of a dictionary.

3.21 Dictionary Formatting

The % operator works conveniently to substitute values from a `dict` into a string by name:

```
#create a dictionary
>>> h = {}
#add a key/value pair
>>> h['item'] = 'beer'
```

```
>>> h['count'] = 4
#interpolate using %d for int, %s for string
>>> s = 'I want %(count)d bottles of %(item)s' % h
>>> s
'I want 4 bottles of beer'
```

The next section shows you how to create an ordered Python dictionary.

3.22 Ordered Dictionaries

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Python 2.7 introduced a new `OrderedDict` class in the collections module. The `OrderedDict` application programming interface (API) provides the same interface as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted:

```
>>> from collections import OrderedDict
>>> d = OrderedDict([('first', 1),
...                  ('second', 2),
...                  ('third', 3)])
>>> d.items()
[('first', 1), ('second', 2), ('third', 3)]
```

If a new entry overwrites an existing entry, the original insertion position is left unchanged:

```
>>> d['second'] = 4
>>> d.items()
[('first', 1), ('second', 4), ('third', 3)]
```

Deleting an entry and reinserting it will move it to the end:

```
>>> del d['second']
>>> d['second'] = 5
>>> d.items()
[('first', 1), ('third', 3), ('second', 5)]
```

3.22.1 Sorting Dictionaries

Python enables you to support the entries in a dictionary. For example, you can modify the code in the preceding section to display the alphabetically sorted words and their associated word count.

3.22.2 Python Multidictionaries

You can define entries in a Python dictionary so that they reference lists or other types of Python structures. Listing 3.9 displays the contents of `MultiDictionary1.py` that illustrates how to define more complex dictionaries.

Listing 3.9: MultiDictionary1.py

```
from collections import defaultdict

d = {'a' : [1, 2, 3], 'b' : [4, 5]}
print 'firsts:',d

d = defaultdict(list)
d['a'].append(1)
d['a'].append(2)
d['b'].append(4)
print 'second:',d

d = defaultdict(set)
d['a'].add(1)
d['a'].add(2)
d['b'].add(4)
print 'third:',d
```

Listing 3.9 starts by defining the dictionary `d` and printing its contents. The next portion of Listing 3.9 specifies a list-oriented dictionary, and then modifies the values for the keys `a` and `b`. The final portion of Listing 3.9 specifies a set-oriented dictionary, and then modifies the values for the keys `a` and `b`, as well.

The output from Listing 3.9 is here:

```
first: {'a': [1, 2, 3], 'b': [4, 5]}
second: defaultdict(<type 'list'>, {'a': [1, 2], 'b':
[4]})
third: defaultdict(<type 'set'>, {'a': set([1, 2]), 'b':
set([4])})
```

The next section discusses other Python sequence types that have not been discussed in previous sections of this chapter.

3.23 Other Sequence Types in Python

Python supports 7 sequence types: `str`, `unicode`, `list`, `tuple`, `bytearray`, `buffer`, and `xrange`.

You can iterate through a sequence and retrieve the position index and corresponding value at the same time using the `enumerate()` function.

```
>>> for i, v in enumerate(['x', 'y', 'z']):
...     print i, v
...
0 x
1 y
2 z
```

`Bytearray` objects are created with the built-in function `bytearray()`. Although buffer objects are not directly supported by Python syntax, you can create them via the built-in `buffer()` function.

Objects of type `xrange` are created with the `xrange()` function. An `xrange` object is similar to a buffer in the sense that there is no specific syntax to create them. Moreover, `xrange` objects do not support operations such as slicing, concatenation or repetition.

At this point you have seen all the Python type that you will encounter in the remaining chapters of this book, so it makes sense to discuss mutable and immutable types in Python, which is the topic of the next section.

3.24 Mutable and Immutable Types in Python

Python represents its data as objects. Some of these objects (such as lists and dictionaries) are mutable, which means you can change their content without changing their identity. Objects such as integers, floats, strings and tuples are objects that cannot be changed. The key point to understand is the difference between changing the value versus assigning a new value to an object; you cannot change a string but you can assign it a different value. This detail can be verified by checking the `id` value of an object, as shown in Listing 3.10.

Listing 3.10: Mutability.py

```
s = "abc"
print('id #1:', id(s))
print('first char:', s[0])

try:
  s[0] = "o"
```

```
except:
  print('Cannot perform reassignment')

s = "xyz"
print('id #2:',id(s))
s += "uvw"
print('id #3:',id(s))
```

The output of Listing 3.x is here:

```
id #1: 4297972672
first char: a
Cannot perform reassignment
id #2: 4299809336
id #3: 4299777872
```

Thus, a Python type is immutable if its value cannot be changed (even though it's possible to assign a new value to such a type), otherwise a Python type is mutable. The Python immutable objects are of type `bytes`, `complex`, `float`, `int`, `str`, or `tuple`. On the other hand, dictionaries, lists, and sets are mutable. The key in a hash table must be an immutable type.

Since strings are immutable in Python, you cannot insert a string in the "middle" of a given text string unless you construct a second string using concatenation. For example, suppose you have the string:

```
"this is a string"
```

and you want to create the following string:

```
"this is a longer string"
```

The following Python code block illustrates how to perform this task:

```
text1 = "this is a string"
text2 = text1[0:10] + "longer" + text1[9:]
print 'text1:',text1
print 'text2:',text2
```

The output of the preceding code block is here:

```
text1: this is a string
text2: this is a longer string
```

3.25 The `type()` Function

The `type()` primitive returns the type of any object, including Python primitives, functions, and user-defined objects. The following code sample displays the type of an integer and a string:

```
var1 = 123
var2 = 456.78
print("type var1: ",type(var1))
print("type var2: ",type(var2))
```

The output of the preceding code block is here:

```
type var1:  <type 'int'>
type var2:  <type 'float'>
```

3.26 Summary

This chapter showed you how to work with various Python data types. In particular, you learned about tuples, sets, and dictionaries. Next you learned how to work with lists and how to use list-related operations to extract sublists. You also learned how to use Python data types in order to define tree-like structures of data.

CHAPTER 4

INTRODUCTION TO NUMPY AND PANDAS

- What is NumPy?
- What Are NumPy Arrays?
- Working with Loops
- Appending Elements to Arrays (1)
- Appending Elements to Arrays (2)
- Multiply Lists and Arrays
- Doubling the Elements in a List
- Lists and Exponents
- Arrays and Exponents
- Math Operations and Arrays
- Working with "-1" Subranges with Vectors
- Working with "_1" Subranges with Arrays
- Other Useful NumPy Methods
- Arrays and Vector Operations
- NumPy and Dot Products (1)
- NumPy and Dot Products (2)
- NumPy and the "Norm" of Vectors
- NumPy and Other Operations
- NumPy and the reshape() Method

The first half of this chapter starts with a quick introduction to the Python NumPy package, followed by a quick introduction to Pandas and some of its useful features. The Pandas package for Python provides a rich and powerful set of APIs for managing datasets. These APIs are very useful for machine learning and deep learning tasks that involve dynamically "slicing and dicing" subsets of datasets.

The first section contains examples of working arrays in NumPy, and contrasts some of the APIs for lists with the same APIs for arrays. In addition, you will see how easy it is to compute the exponent-related values (square, cube, and so forth) of elements in an array.

The second section introduces subranges, which are very useful (and frequently used) for extracting portions of datasets in machine learning

tasks. In particular, you will see code samples that handle negative (-1) subranges for vectors as well as for arrays, because they are interpreted one way for vectors and a different way for arrays.

The third part of this chapter delves into other NumPy methods, including the reshape() method, which is extremely useful (and very common) when working with images files: some TensorFlow APIs require converting a 2D array of (R,G,B) values into a corresponding one-dimensional vector.

The fourth part of this chapter briefly describes Pandas and some of its useful features. This section contains code samples that illustrate some nice features of DataFrames and a brief discussion of series, which are two of the main features of Pandas. The second part of this chapter discusses various types of DataFrames that you can create, such as numeric and Boolean DataFrames. In addition, you will see examples of creating DataFrames with NumPy functions and random numbers.

The fifth section of this chapter shows you how to manipulate the contents of DataFrames with various operations. In particular, you will also see code samples that illustrate how to create Pandas DataFrames from CSV (Comma Separated Values) files, Excel spreadsheets, and data that is retrieved from a URL. The third section of this chapter gives you an overview of important data cleaning tasks that you can perform with Pandas APIs.

4.1 What is NumPy?

NumPy is a Python module that provides many convenience methods and also better performance. NumPy provides a core library for scientific computing in Python, with performant multidimensional arrays and good vectorized math functions, along with support for linear algebra and random numbers.

NumPy is modeled after MatLab, with support for lists, arrays, and so forth. NumPy is easier to use than Matlab, and it's very common in TensorFlow code as well as Python code.

4.1.1 Useful NumPy Features

The NumPy package provides the *ndarray* object that encapsulates *multi*dimensional arrays of homogeneous data types. Many ndarray operations are performed in compiled code in order to improve performance.

Keep in mind the following important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size, whereas Python lists can expand dynamically. Whenever you modify the size of an *ndarray*, a new array is created and the original array is deleted.
- NumPy arrays are homogeneous, which means that the elements in a NumPy array must all have the same data type. Except for NumPy arrays of objects, the elements in NumPy arrays of any other data type must have the same size in memory.
- NumPy arrays support more efficient execution (and require less code) of various types of operations on large numbers of data.
- Many scientific Python-based packages rely on NumPy arrays, and knowledge of NumPy arrays is becoming increasingly important.
- Now that you have a general idea about NumPy, let's delve into some examples that illustrate how to work with NumPy arrays, which is the topic of the next section.

4.2 What are NumPy Arrays?

An *array* is a set of consecutive memory locations used to store data. Each item in the array is called an *element*. The number of elements in an array is called the *dimension* of the array. A typical array declaration is shown here:

```
arr1 = np.array([1,2,3,4,5])
```

The preceding code snippet declares `arr1` as an array of five elements, which you can access via `arr1[0]` through `arr1[4]`. Notice that the first element has an index value of 0, the second element has an index value of 1, and so forth. Thus, if you declare an array of 100 elements, then the 100th element has index value of 99.

Note: The first position in a NumPy array has index 0.

NumPy treats arrays as vectors. Math operations are performed element-by-element. Remember the following difference: "doubling" an array *multiplies* each element by 2, whereas "doubling" a list *appends* a list to itself.

Listing 4.1 displays the contents of `nparray1.py` that illustrates some operations on a `NumPy` array.

Listing 4.1: nparray1.py

```
import numpy as np

list1 = [1,2,3,4,5]
print(list1)

arr1  = np.array([1,2,3,4,5])
print(arr1)

list2 = [(1,2,3),(4,5,6)]
print(list2)

arr2  = np.array([(1,2,3),(4,5,6)])
print(arr2)
```

Listing 4.1 defines the variables `list1` and `list2` (which are Python lists), as well as the variables `arr1` and `arr2` (which are arrays), and prints their values. The output from launching Listing 4.1 is here:

```
[1, 2, 3, 4, 5]
[1 2 3 4 5]
[(1, 2, 3), (4, 5, 6)]
[[1 2 3]
 [4 5 6]]
```

As you can see, Python lists and arrays are very easy to define, and now we're ready to look at some loop operations for lists and arrays.

4.3 Working with Loops

Listing 4.2 displays the contents of `loop1.py` that illustrates how to iterate through the elements of a `NumPy` array and a Python list.

Listing 4.2: loop1.py

```
import numpy as np

list = [1,2,3]
arr1 = np.array([1,2,3])

for e in list:
  print(e)
```

```
for e in arr1:
  print(e)

list1 = [1,2,3,4,5]
```

Listing 4.2 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The next portion of Listing 4.2 contains two loops, each of which iterates through the elements in `list` and `arr1`. As you can see, the syntax is identical in both loops. The output from launching Listing 4.2 is here:

```
1
2
3
1
2
3
```

4.4 Appending Elements to Arrays (1)

Listing 4.3 displays the contents of `append1.py` that illustrates how to append elements to a `NumPy` array and a Python list.

Listing 4.3: append1.py

```
import numpy as np

arr1 = np.array([1,2,3])

# these do not work:
#arr1.append(4)
#arr1 = arr1 + [5]

arr1 = np.append(arr1,4)
arr1 = np.append(arr1,[5])

for e in arr1:
  print(e)

arr2 = arr1 + arr1

for e in arr2:
  print(e)
```

Listing 4.3 initializes the variable `list`, which is a Python list, and also the variable `arr1`, which is a `NumPy` array. The output from launching Listing 4.3 is here:

```
1
2
3
4
5
2
4
6
8
10
```

4.5 Appending Elements to Arrays (2)

Listing 4.4 displays the contents of `append2.py` that illustrates how to append elements to a `NumPy` array and a Python list.

Listing 4.4: append2.py

```
import numpy as np

arr1 = np.array([1,2,3])
arr1 = np.append(arr1,4)

for e in arr1:
  print(e)

arr1 = np.array([1,2,3])
arr1 = np.append(arr1,4)

arr2 = arr1 + arr1

for e in arr2:
  print(e)
```

Listing 4.4 initializes the variable `arr1`, which is a `NumPy` array. Notice that `NumPy` arrays do not have an "append" method: this method is available through `NumPy` itself. Another important difference between Python lists and `NumPy` arrays: the "+" operator *concatenates* Python lists, whereas this operator *doubles* the elements in a `NumPy` array. The output from launching Listing 4.4 is here:

```
1
2
3
4
```