Chapter XIII

The Array Data Structure

Chapter XIII Topics

13.1 Introduction to Data Structures

13.2 Array Syntax

13.3 Array Commands

13.4 Random Arrays

13.5 The for..each Loop

13.1 Introduction to Data Structures

Before we discuss the *array data structure*, we need an idea of what *data structures* are in the first place. Early in the course you were introduced to *simple* data types like *integers*, *real numbers* and *Boolean* values. A simple data type variable is a location in memory that stores a single value that can be used by a computer program. Single values are practical for loop counter variables, maximum number of grades, the height of *Pikes Peak* and the number of medals won by the United States at the last Olympics.

Simple Data Type Definition

A *simple data type* is a data type that can only store one value.

Examples of *simple data types* are *integers*, *real numbers* and *Boolean* values.

Programs that handle passenger airline reservations, student college transcripts, employee payroll records and hospital patient information, require massive data storage. Such major storage requirements cannot be handled efficiently by thousands of simple data type variables, each storing a single value. You will need more sophisticated data types.

There are many situations where data needs to hold more than one value. Such a situation calls for using a *data structure*. So what is a data structure? Look at a building. Note that it is made up of smaller structures like rooms, halls, stairways, etc. A room is made up of walls, floors, ceilings, desks, chairs, etc.

Another example can be found with animals. Animals are organisms made up of *organ systems*. Each organ system is made up of *organs*. Organs are made up of *tissues*, and tissues are made up of *cells*. We could continue and work down to the molecular and atomic level, but for this analogy, assume that the *cell* is the simplest, lowest level. The whole point is that the structure, an organism in this case, is made up of other, smaller structures, until eventually you reach the smallest component.

These two examples are used to motivate the definition of a data structure. In computer science it really is the same idea. The only difference in structures is the nature of the smallest building block used to create the structure. In an animal organism it is a cell. In a building it may be a brick or a plank and in a computer science *data structure* it is a *simple data type*.

Data Structure Definition

A *data structure* is a data type that can store more than one value.

Examples of *data structures* are *arrays*, *records* and *files*.

The Array Data Structure

The *array* is the first historical data structure. This data structure became popular with the introduction of the first commercially, wide-used programming language, *FORTRAN*. FORTRAN, which means FORmula TRANslator, was designed for the scientific - number crunching - community. A data structure, like an array, was necessary for the storing of large quantities of numbers.

What does an array bring to mind? How about an array of flowers or an array of books, or an array of anything else? We think of an array as having multiple items - not a single item - and an array has the same type of items. We can have an array of integers, an array of real numbers, an array of characters, and an array of strings. An array can have any kind of element, as long as each element is the same data type. You will find the name *vector* used frequently for *one-dimensional* arrays and *matrix* for *two-dimensional* arrays.

Array Definition

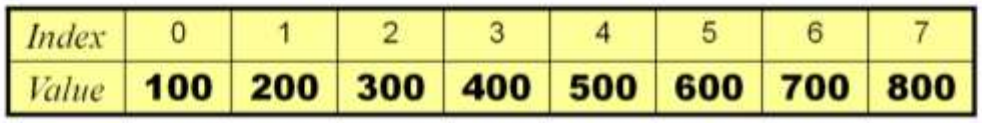
An *array* is a data structure with one or more elements
of the same type.

Every element of the array can be accessed directly.

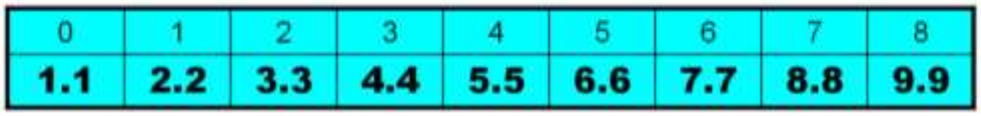
A one-dimensional array is frequently also called a *vector*.
A two-dimensional array is frequently also called a *matrix*.

Here are some examples of 1-D arrays (vectors) and 2-D arrays (matrixes):

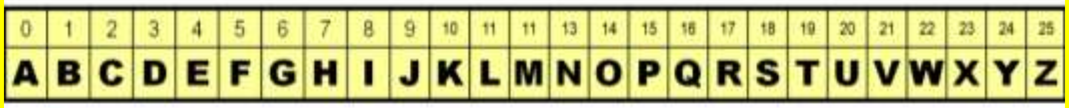
1-D Array / Vector Examples



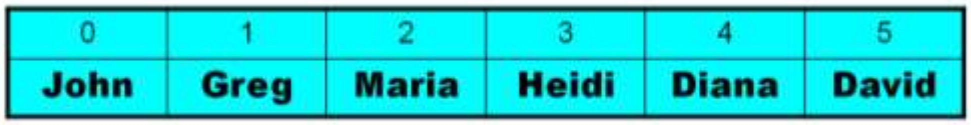
Index	0	1	2	3	4	5	6	7
Value	**100**	**200**	**300**	**400**	**500**	**600**	**700**	**800**



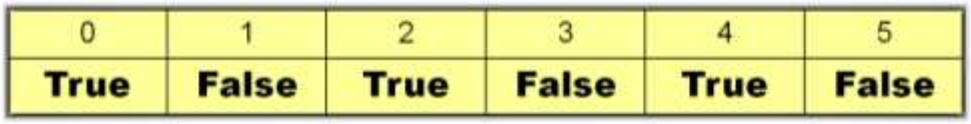
0	1	2	3	4	5	6	7	8
1.1	**2.2**	**3.3**	**4.4**	**5.5**	**6.6**	**7.7**	**8.8**	**9.9**



0	1	2	3	4	5	6	7	8	9	10	11	11	13	14	15	16	17	18	19	20	21	22	23	24	25
A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**

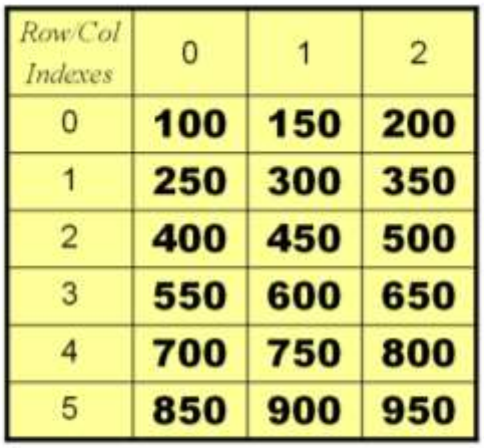


0	1	2	3	4	5
John	**Greg**	**Maria**	**Heidi**	**Diana**	**David**

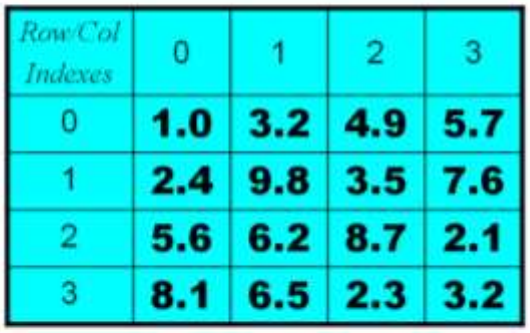


0	1	2	3	4	5
True	**False**	**True**	**False**	**True**	**False**

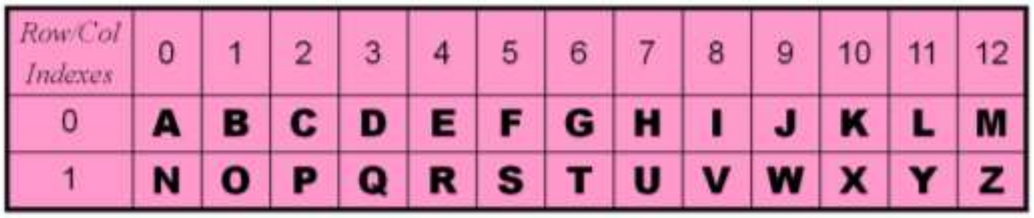
2-D Array / Matrix Examples



Row/Col Indexes	0	1	2
0	**100**	**150**	**200**
1	**250**	**300**	**350**
2	**400**	**450**	**500**
3	**550**	**600**	**650**
4	**700**	**750**	**800**
5	**850**	**900**	**950**



Row/Col Indexes	0	1	2	3
0	**1.0**	**3.2**	**4.9**	**5.7**
1	**2.4**	**9.8**	**3.5**	**7.6**
2	**5.6**	**6.2**	**8.7**	**2.1**
3	**8.1**	**6.5**	**2.3**	**3.2**



Row/Col Indexes	0	1	2	3	4	5	6	7	8	9	10	11	12
0	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
1	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**

The Record Data Structure

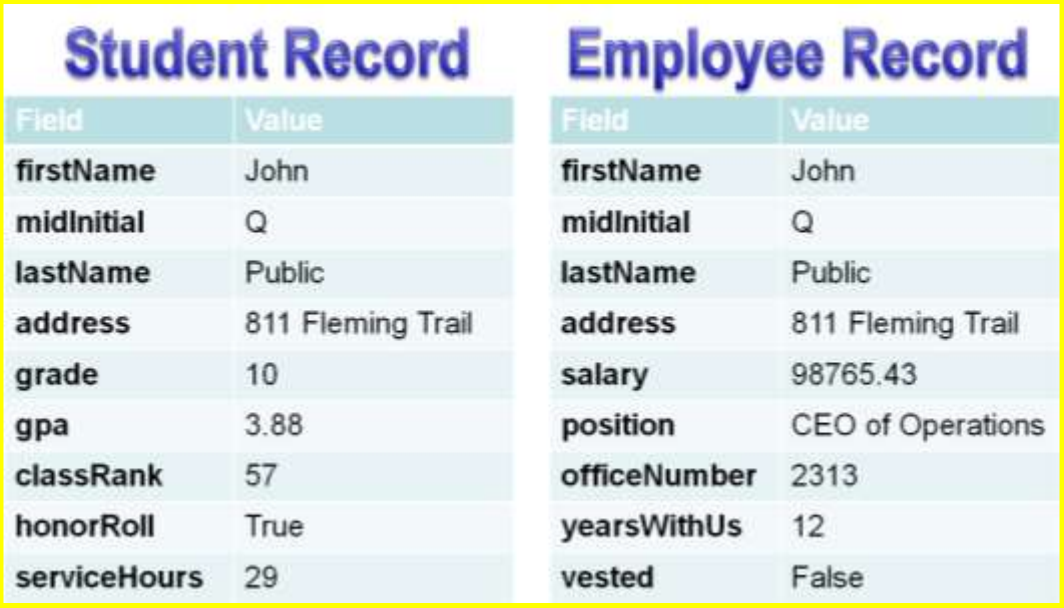
The business community was not very happy with the FORTRAN language and particularly with the data structure limitation of having an array and nothing else. In the business world, data is not of the same type. This is lovely in science and math where numbers rule the discipline, but in business it is another story.

Data storage in business requires storing names, addresses, birth dates, number of dependents, social security numbers, credit cards numbers, flight numbers, years worked, pay rates, credit balance available, etc. etc. etc. One solution was to create many different arrays. Each array specialized in one part of some business record. There was an array of names, an array of addresses, an array of pay rates and so on, which were called *parallel arrays*. This worked, but it was tedious.

A new programming language became popular, called COBOL (COmmon Business Oriented Language), which introduced the *record* data structure. What does the word *record* bring to mind? How about a student's record, an employee's record, a patient's record, a passenger's record? Each one of these records has the common thread of multiple information fields that can be of many different types. This type of data structure is precisely what the business world required. COBOL became a highly successful language (it helped that the Department of Defense adopted the language) and the record is now an integral part of programming.

Record Definition

A *record* is a data structure with one or more elements, called *fields*, of the same or different data types.



Student Record

Field	Value
firstName	John
midInitial	Q
lastName	Public
address	811 Fleming Trail
grade	10
gpa	3.88
classRank	57
honorRoll	True
serviceHours	29

Employee Record

Field	Value
firstName	John
midInitial	Q
lastName	Public
address	811 Fleming Trail
salary	98765.43
position	CEO of Operations
officeNumber	2313
yearsWithUs	12
vested	False

The File Data Structure

Programming languages have a convenient data structure that facilitates the transfer of data to and from external storage. The array and record may be lovely to store a complex set of data in the memory of a computer, but this data often needs to be used at some future date. The data needs to be stored in some permanent manner. The file data structure provides this ability.

File Definition

A *file* is an *internal* data structure - with an unspecified number of elements of the same type - assigned to an *external* file name. The file data structure allows transfer of data between internal and external storage.

Other Data Structures

The three data structures – array, record and file – introduced in this section are part of the Python programming language. These data types are ready to go and can be used with very little effort. Using built-in data structures is a good starting point in an introductory computer science course. There are many other types of data structures that the programmer can create for a wide variety of special purposes. The study and practice of these special user-created data structures is a major focus of some college-level computer science courses.

13.2 Array Syntax

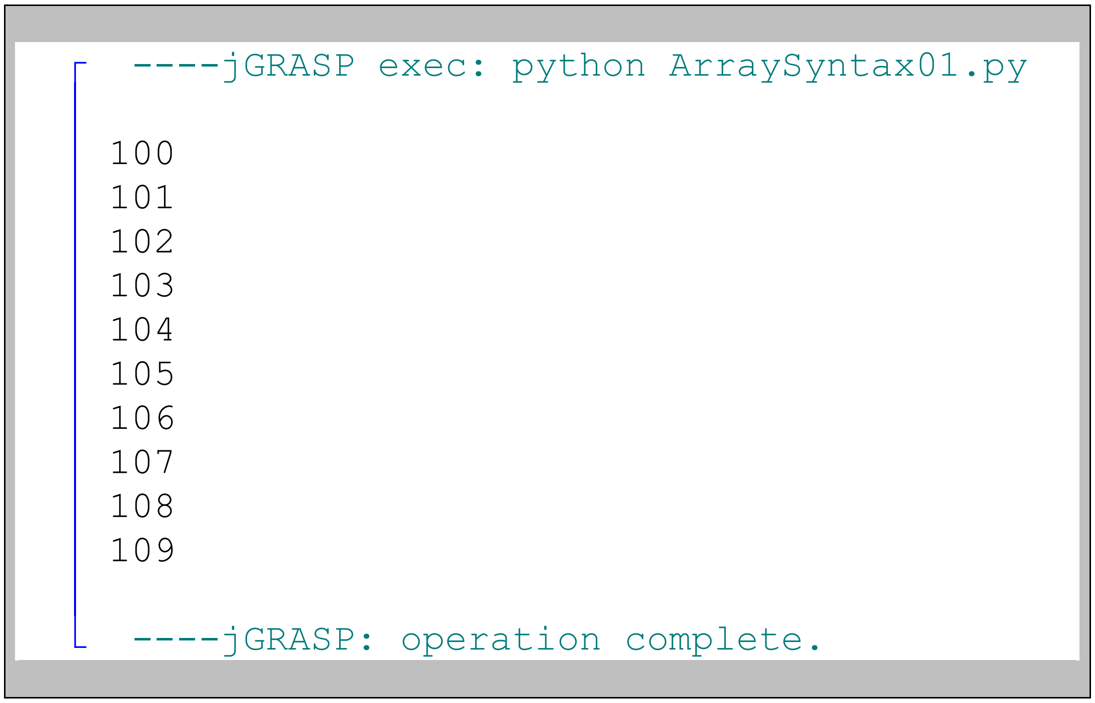
We are going to jump right in and look at our first Python program with arrays, which is also our first Python program using any data structure. Program **ArraySyntax01.py**, in Figure 13.1, demonstrates the necessary Python syntax to both create an array data structure and also access its individual array elements.

First, on line 7, we see that the **list** array is defined. Note here the clear difference between a *simple data type* and a *data structure*. If line 7 simply said **list = 100**, then list would be storing just 1 integer value, which would make it a simple data type. As we can see, this is not the case. **list** is not just storing 1 integer, it is storing an *array* of 10 integers.

Second, on lines 10 through 19, we see that each individual element – in this case, integer – of the array is accessed and displayed. This is made possible by using an *index*. An *index* is a number in [brackets] that follows the array identifier, in this case, **list**, to indicate which specific element of the array you wish to access. Think of an *array* as being a street in a neighborhood. The *index* would then be the numbers on each individual house. Without this number, there is no simple way to identify an individual house on that street.

Figure 13.1

```
 1 # ArraySyntax01.py
 2 # This program demonstrates the creation of an
 3 # array of integers in Python and demonstrates
 4 # how to access individual array elements.
 5
 6
 7 list = [100,101,102,103,104,105,106,107,108,109]
 8
 9 print()
10 print(list[0])
11 print(list[1])
12 print(list[2])
13 print(list[3])
14 print(list[4])
15 print(list[5])
16 print(list[6])
17 print(list[7])
18 print(list[8])
19 print(list[9])
```



```
----jGRASP exec: python ArraySyntax01.py

100
101
102
103
104
105
106
107
108
109

----jGRASP: operation complete.
```

There is something subtle that you may have missed. While there are 10 integers in the **list** array, did you notice that we never used index 10? Nowhere in the program is **list[10]** displayed. Did you also notice that we are displaying **list[0]**? Here is what must be understood. In many programming languages, including Python, the index of the first item of an array is **0**, not **1**. Since we started at **0**, this means the index of the tenth item in the array is **9**, not **10**. In general, we can say that if an array has **N** elements, the index of the last element will be **N-1**.

Array Index Note

Python arrays indicate individual elements with an index inside [brackets] following the array identifier, like **list[3]**

The array index is always an integer.

The index of the first item in the array is **0**.

In an array of **N** elements, the index of the last item is **N-1**.

Traversing Arrays

The title of this subsection may be a little confusing. What does it mean to "traverse" an array? If you "traverse a river" it means you are crossing it from one side to the other. Essentially, to *traverse* means "to go through". In the case of arrays, traversing means you go through the array, from the first element to the last element visiting every element in-between.

Technically, the previous program did traverse the array, but it did so in a very inefficient manner with 10 separate **print** commands. This is not practical; especially with larger arrays that stores hundreds or thousands of elements. If you look at the indexes used in the previous program, you see that they count from **0** to **9** in sequence. That sounds like something the **for** loop can do.

Program **ArraySyntax02.py**, in Figure 13.2, demonstrates how to use a **for** loop to traverse an array with far less code. Note how the loop counter, **k**, is used inside the [brackets]. The first time through the loop, when the value of **k** equals **0**, it will display **list[0]**. The second time through the loop, when **k** equals **1**, it will display **list[1]**. This continues all of the way through the last time through the loop, when **k** equals **9**, it will display **list[9]**. The output will be exactly the same as the first program.

Figure 13.2

```
# ArraySyntax02.py
# This program demonstrates a more efficient way
# to "traverse" an array by using a <for> loop.


list = [100,101,102,103,104,105,106,107,108,109]

print()
for k in range(10):
   print(list[k])

```

Program **ArraySyntax03.py**, in Figure 13.3, demonstrates that arrays can store more than just integers. First, there is an array called **letters** which stores 26 characters (which in Python are actually strings). Then there is an array of 10 real numbers called **gpas**, as in *grade point averages*. Finally, there is an array of 5 Boolean values called **booleans**. Note how all 3 arrays are created, traversed and displayed in the same manner as the previous program.

Figure 13.3

```python
# ArraySyntax03.py
# This program demonstrates an array of characters/strings,
# an array of real numbers and an array of Boolean values.


letters = ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z']
print()
for k in range(26):
   print(letters[k],end=" ")
print()

gpas = [3.9,2.75,2.1,1.65,4.0,3.25,2.45,0.95,3.88,3.75]
print()
for k in range(10):
   print(gpas[k],end=" ")
print()

booleans = [True,False,True,False,True]
print()
for k in range(5):
   print(booleans[k],end=" ")
print()

```

```
 ----jGRASP exec: python ArraySyntax03.py

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

3.9 2.75 2.1 1.65 4.0 3.25 2.45 0.95 3.88 3.75

True False True False True

 ----jGRASP: operation complete.
```

Program **ArraySyntax04.py**, in Figure 13.4, demonstrates another array of strings. This time, instead of mere letters, the array contains names. We can see that the array is created, traversed and displayed in the same manner as the previous 2 programs.

Figure 13.4

```
 1 # ArraySyntax04.py
 2 # This program demonstrates another string array.
 3 # This time names are stored.
 4 # NOTE: See what happens when you add or
 5 #       remove names from the <names> array.
 6
 7
 8 names = ["John","Greg","Maria","Heidi","Diana","David"]
 9 print()
10 for k in range(6):
11    print(names[k],end=" ")
12 print()
13
```

```
 ----jGRASP exec: python ArraySyntax04.py

John Greg Maria Heidi Diana David

 ----jGRASP: operation complete.
```

While the use of **for** loop does make the past 3 programs more efficient (as compared with the very first program in this chapter), there still is a problem in their lack of flexibility. Let us look just at the last program in Figure 13.4. Add some names to the array definition on line 8. Run the program again and see what happens. Those extra names are not displayed because the **for** loop is set up to only display the first **6** names in the array. Suppose you remove all of those extra names, and while you are at it, you also remove **"David"** as well, leaving only **5** names in the array. Now when you run the program, it will crash with a run-time error because you are trying to display the 6th name from an array that only has **5**.

Program **ArraySyntax05.py**, in Figure 13.5, shows one way to fix this problem by using the **len** function. **len** is a simple little function that returns the number of items in an array.

Figure 13.5

```python
# ArraySyntax05.py
# This program demonstrates a more flexible way to
# "traverse" an array by using the <len> function.
# NOTE: Now see what happens when you add or
#       remove names from the <names> array.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print("There are",len(names),"names in the array.")
print()

for k in range(len(names)):
   print(names[k], end = " ")
print()

```

```
 ----jGRASP exec: python ArraySyntax05.py

There are 6 names in the array.

John Greg Maria Heidi Diana David

 ----jGRASP: operation complete.
```

Now try the same experiment that you did with the first program. Add or remove as many names as you would like. The program still executes properly because the loop is being controlled by the actual *length* of the array, rather than a fixed number like **6**.

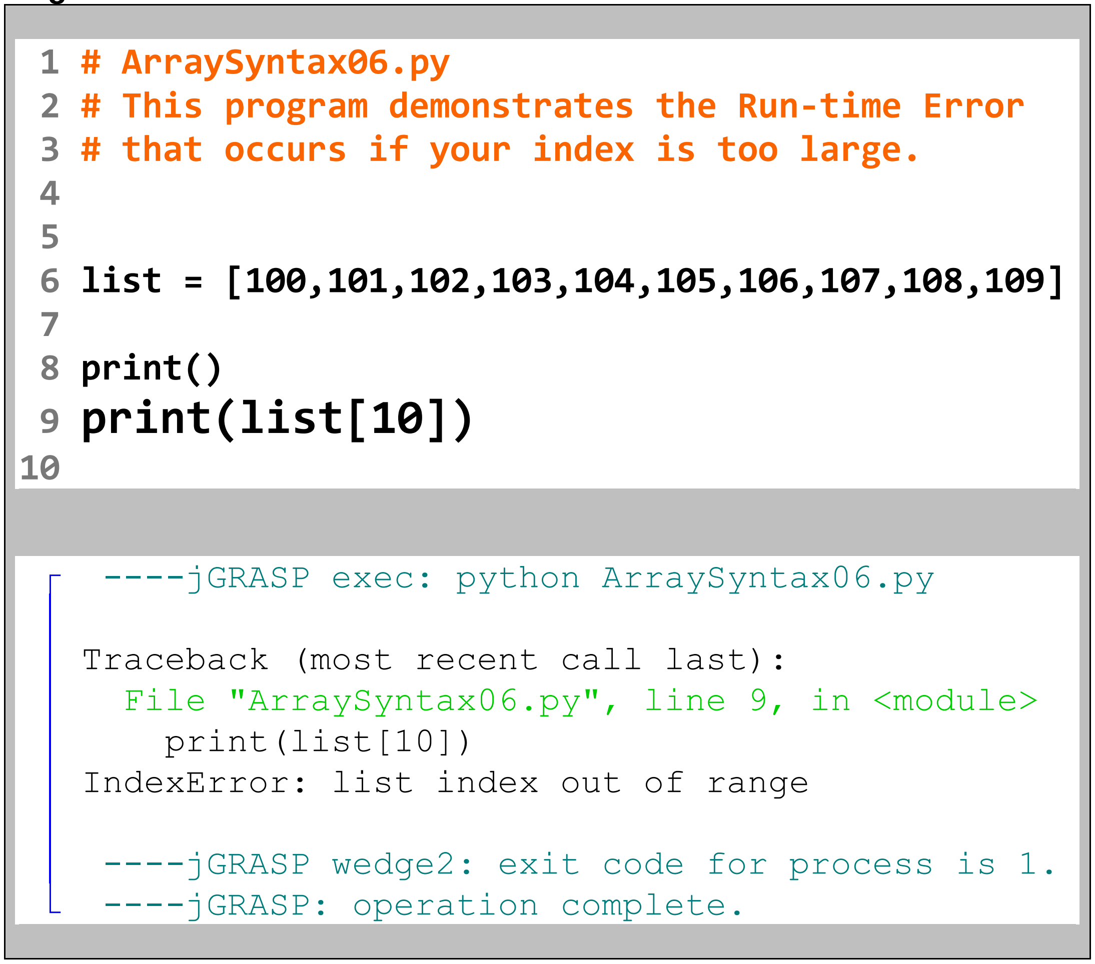
Program **ArraySyntax06.py**, in Figure 13.7 demonstrates the run-time error that you get when you use an index that is too large. In this case, we are using an index of **10**. Yes, there are 10 integers in the array, but remember that the first one is located at index **0**. This means the last one is located at index **9**. Index **10** is not valid. See the chart in Figure 13.6. The program crashes with the message: **IndexError: list index out of range**.

Figure 13.6

list Array

Index	0	1	2	3	4	5	6	7	8	9
Value	**100**	**101**	**102**	**103**	**104**	**105**	**106**	**107**	**108**	**109**

Figure 13.7



```python
# ArraySyntax06.py
# This program demonstrates the Run-time Error
# that occurs if your index is too large.


list = [100,101,102,103,104,105,106,107,108,109]

print()
print(list[10])

```

```
 ----jGRASP exec: python ArraySyntax06.py

Traceback (most recent call last):
  File "ArraySyntax06.py", line 9, in <module>
    print(list[10])
IndexError: list index out of range

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Program **ArraySyntax07.py**, in Figure 13.8 demonstrates a common mistake when traversing an array with a loop. This program uses the same array of 10 integers as the previous program. So, just like the previous program, the valid index range is **0** through **9**. In this program, the **while** loop counts from **1** to **10**. This causes 2 problems. First, since we are starting at index **1** instead of **0**, the program does not display the first number in the array. If you look at the output, you will see it starts with **101** instead of **100**. The second problem is more serious and occurs when the loop counter, **k**, reaches the value of **10**. Once this value is used as an index, the program crashes with the same error as the previous program.

Figure 13.8

```python
# ArraySyntax07.py
# This program demonstrates a common mistake when
# an array is traversed with a loop.  With 10 items
# in the array, the indexes range from 0 to 9, but
# the <while> loop erroneously counts from 1 to 10.
# This also causes a "list index out of range" error.


list = [100,101,102,103,104,105,106,107,108,109]

print()
k = 1
while k <= 10:
   print(list[k])
   k += 1

```

```
 ----jGRASP exec: python ArraySyntax07.py

101
102
103
104
105
106
107
```

```
108
109
Traceback (most recent call last):
  File "ArraySyntax07.py", line 14, in <module>
    print(list[k])
IndexError: list index out of range

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Sometimes, you may wish to quickly display the contents of an array without messing with the process of setting up a loop to traverse to the array. Here is where Python offers another shortcut. Program **ArraySyntax08.py**, in Figure 13.9 demonstrates that in Python, you can actually put the *array identifier* (name of the array) in a **print** statement. This is something that does not work in many other programming languages.

Figure 13.9

```python
# ArraySyntax08.py
# This program demonstrates that Python will allow
# you to display the entire contents of an array
# with a single <print> command.  The array is
# displayed in [brackets] separated by commas.

# NOTE: Not all languages allow you to display
#       the contents of an array in this manner.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)

```

```
 ----jGRASP exec: python ArraySyntax08.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

 ----jGRASP: operation complete.
```

When an Array is Not an Array

I am now going to say something that will blow your mind:

Python actually does not have arrays.

OK, that just seems crazy. Did we not just go through 8 programming examples of arrays in Python? To better understand what is going on, let us take a look at the next program example.

Program **ArraySyntax09.py**, in Figure 13.10 demonstrates that what we have been calling "arrays" in Python technically are not *arrays*. They are *lists*. So what is a *list*? Look at the **misc** list on line 10. See that in addition to 4 string values, this list also stores an integer value, a real number value and a Boolean value. This actually proves that **misc** is not an array because it is storing data of different data types.

Figure 13.10

```
 1 # ArraySyntax09.py
 2 # This program demonstrates that Python technically
 3 # does not have "Arrays".  Python has "Lists".
 4 # A list is an ordered sequence of elements.
 5 # An array is a list where all of the elements are
 6 # the same type. In spite of this, we will continue
 7 # to use Python lists as arrays.
 8
 9
10 misc = ["John","Greg","Maria","Heidi",7,3.14,True]
11
12 print()
13 print(misc)
14
```

```
 ----jGRASP exec: python ArraySyntax09.py

 ['John', 'Greg', 'Maria', 'Heidi', 7, 3.14, True]

 ----jGRASP: operation complete.
```

List Definition

A *list* is an ordered sequence of elements.

Alternate Array Definition

An *array* is a *list* where all of the elements are of the same data type.

Python Array/List Disclaimer

Python technically does not have *arrays*. Python has *lists*.

In spite of this, we will continue to use Python *lists* as *arrays* for the duration of this first year class.

Filling an Array with a Single Value

Sometimes, you need every element in an array to be exactly the same. Maybe you have an array of strings, and at the start of the program you need them all to be a blank space. Maybe you have an array of integers, and at the start of the program you need them all to be **0**. In the case of the zeros, you might be thinking, "Why do you not just do something like `list = [0,0,0,0,0,0]` ?" This is fine for 6 zeros or 10 zeros or maybe even 20 zeros. But what if our array needs to have 100 zeros or 1000 zeros? Then this becomes unacceptably tedious. We need something better.

Program **ArraySyntax10.py**, in Figure 13.11 demonstrates how to create an array of any size with the same element at every index. The secret is to first create an array with one single element and then you "multiply" it. An example should add some clarification. Look at the definition of **list1** on line 8. For **list1** I want an array of **20** integers, and I want all of them to be **7**. So I start with `[7]`, which is an array of one integer. Then I add the code `* 20`. In a sense, I am *multiplying* the array by **20**, so now I have an array of 20 sevens, instead of just one. This is similar to the way we use the plus (**+**) operator for *string contatenation*.

Figure 13.10

```python
# ArraySyntax10.py
# This program demonstrates how to create an array
# of a specific size filled with the same value.
# While most languages have an overloaded + operator,
# Python also has an overloaded * operator.


list1 = [7] * 20
list2 = [3.14] * 10
list3 = ['Q'] * 12
list4 = ["Hello!"] * 6
list5 = [True] * 10

print()
print(list1)
print()
print(list2)
print()
print(list3)
print()
print(list4)
print()
print(list5)

```

```
 ----jGRASP exec: python ArraySyntax10.py

[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]

[3.14, 3.14, 3.14, 3.14, 3.14, 3.14, 3.14, 3.14, 3.14, 3.14]

['Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q', 'Q']

['Hello!', 'Hello!', 'Hello!', 'Hello!', 'Hello!', 'Hello!']

[True, True, True, True, True, True, True, True, True, True]

 ----jGRASP: operation complete.
```

Combining Arrays

I just recently brought up the subject of *string concatenation*. If I can use the plus (**+**) operator to join together 2 strings, could I also use it to combine 2 arrays? That is the experiment performs by program **ArraySyntax11.py**, shown in Figure 13.12. If you look at lines 7 and 8, you see the definitions of 2 small arrays of integers. Line 10 has the command: **`listC = listA + listB`**. Then all 3 lists are displayed. The output shows that the experiment was successful. We can combine arrays with the plus (**+**) operator. This is essentially *array concatenation*.

Figure 13.12

```
 1 # ArraySyntax11.py
 2 # This program demonstrates that 2 arrays
 3 # can be combined to create a third, larger
 4 # array using the overloaded + operator.
 5 # This is very similar to "string concatenation".
 6
 7
 8 listA = [11,22,33,44,55]
 9 listB = [66,77,88,99]
10
11 listC = listA + listB
12
13 print()
14 print(listA)
15 print(listB)
16 print(listC)
17
```

```
 ----jGRASP exec: python ArraySyntax11.py

[11, 22, 33, 44, 55]
[66, 77, 88, 99]
[11, 22, 33, 44, 55, 66, 77, 88, 99]

 ----jGRASP: operation complete.
```

Program **ArraySyntax12.py**, in Figure 13.13, shows another example of *array concatenation*. This time the **+=** operator is used. When you look at the program and its output, you should see that the **+=** operator will join one array to the end of an existing array. This is essentially the same thing it does for strings.

Figure 13.13

```python
# ArraySyntax12.py
# This program demonstrates that the overloaded
# += operator behaves the same way with arrays
# as it does with strings.


list = [11,22,33,44,55]

print()
print(list)

list += [66,77,88,99]

print()
print(list)

```

```
 ----jGRASP exec: python ArraySyntax12.py

[11, 22, 33, 44, 55]

[11, 22, 33, 44, 55, 66, 77, 88, 99]

 ----jGRASP: operation complete.
```

Slicing

When working with arrays in Python, a *slice* refers to a piece of an array. Imaging you want a slice of birthday cake. To do this, you make one cut at the beginning of your slice and another cut at the end. When *slicing* arrays, you essentially do the same thing with `[`*start*`:`*stop*`]` syntax. Instead of a single index number in the array brackets, you will have two. The first specifies the beginning index of your slice. It would seem logic now for me to say the second index specifies the ending index of your slice. I wish that were true, but it is a tiny bit more complicated. The second index actually specifies the first index that is NOT part of the slice. It may help to look at the example of **`list[3:8]`** in figure 13.14. Note that while the slice starts on index **3**, it does not stop on index **8**. Instead, it stops BEFORE index **8** (meaning index **7**).

Figure 13.14

list Array

Index	0	1	2	3	4	5	6	7	8	9
Value	100	101	102	103	104	105	106	107	108	109

↑ **`list[3:8]`** *slice* ↑

Program **ArraySyntax13.py**, in Figure 13.15, demonstrates both the array and the slice from the example in Figure 13.14.

Figure 13.15

```python
# ArraySyntax13.py
# This program demonstrates how to display a
# "slice" of an array using the [start:stop]
# syntax.  The array is displayed starting on
# first index and stopping just BEFORE the second.


list = [100,101,102,103,104,105,106,107,108,109]

print()
print(list)

print()
print(list[3:8])

```

```
 ----jGRASP exec: python ArraySyntax13.py

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

[103, 104, 105, 106, 107]

 ----jGRASP: operation complete.
```

Program **ArraySyntax14.py**, in Figure 13.16, demonstrates that if you leave out the first number in a slice, your slice will simply begin at the beginning of the array. Note that this is exactly what you would get if the first number were **0**.

Figure 13.16

```
# ArraySyntax14.py
# This program demonstrates that if you leave
# out the first number in a slice, it will
# start at index 0 by default.


list = [100,101,102,103,104,105,106,107,108,109]

print()
print(list)

print()
print(list[0:8])

print()
print(list[:8])

```

```
 ----jGRASP exec: python ArraySyntax14.py

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

[100, 101, 102, 103, 104, 105, 106, 107]

[100, 101, 102, 103, 104, 105, 106, 107]

 ----jGRASP: operation complete.
```

Program **ArraySyntax15.py**, in Figure 13.17, demonstrates that if you leave out the second number in a slice, your slice will simply finish at the end of the array. It also shows something else. What if the second number were simply too big to still be in the array? You would probably expect to get some kind of "**list index out of range**" error message, like we did several programs ago. That is actually not what happens at all. Instead, the slice just stops at the end of the array. When you look at the output of the program, you should see that **`list[3:]`** and **`list[3:100]`** give the exact same result.

Figure 13.17

```
# ArraySyntax15.py
# This program demonstrates that if you leave
# out the second number in a slice, it will
# go all of the way to the end of the array.
# It also shows that it if the second number
# is way too large, you do not get an error.
# Instead, it also goes to the end of the array.


list = [100,101,102,103,104,105,106,107,108,109]

print()
print(list)

```

```python
print()
print(list[3:])

print()
print(list[3:100])

```

```
 ----jGRASP exec: python ArraySyntax15.py

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

[103, 104, 105, 106, 107, 108, 109]

[103, 104, 105, 106, 107, 108, 109]

 ----jGRASP: operation complete.
```

https://1001freedownloads.s3.amazonaws.com/vector/thumb/139446/johnny_automatic_slicing_bread.png

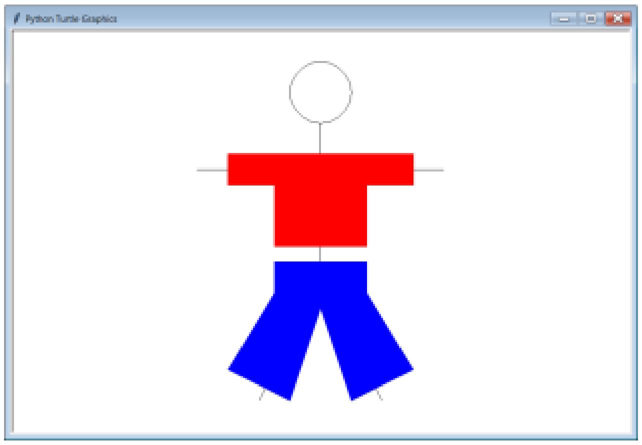
Hidden Arrays

It may surprise you to learn that long before we got to this chapter, you were already working with arrays. You may find this hard to believe, but there were a few occasions this year where arrays were used in the program examples. We just did not tell you they were "arrays" at the time.

Program **ArraySyntax16.py**, in Figure 13.18, is the first example of this. The originally was program **GraphicsLibrary20.py** from Chapter VI. Look at the **fillPolygon** commands on lines 19 and 21. Do you remember that with **drawPolygon** and **fillPoygon** you needed to put all of the parameters inside [brackets]? Well, the reason this is done is because you are actually passing an array as an argument. This is what gives these procedures the flexibility to draw polygons with any number of sides (provided there are at least 3).

Figure 13.18

```
 1 # ArraySyntax16.py
 2 # This program was originally program GraphicsLibrary20.py
 3 # from Chapter 6.  Look closely at the <fillPolygon> commands.
 4 # The reason the [brackets] are required is that we are actually
 5 # passing an "array of integers" to the <fillPolygon> procedure.
 6 # This means you have been using arrays for a while now.
 7
 8
 9 from Graphics import *
10
11 beginGrfx(1000,650)
12
13 drawCircle(500,100,50)
14 drawLine(500,150,500,400)
15 drawLine(500,400,400,600)
16 drawLine(500,400,600,600)
17 drawLine(300,225,700,225)
18 setColor("blue")
19 fillPolygon([425,375,425,425,350,550,450,600,
500,450,550,600,650,550,575,425,575,375])
20 setColor("red")
21 fillPolygon([350,200,650,200,650,250,575,250,
575,350,425,350,425,250,350,250])
22
23 endGrfx()
```



13.3 Array Commands

There are several commands, some procedures and some functions, that can be used to help with array processing. The first was actually already shown in the previous section. Program **ArrayCommands01.py**, in Figure 13.19, essentially repeats program **ArraySyntax05.py** which introduced the **len** function. Remember, this function will return the number of items in an array.

Figure 13.19

```
# ArrayCommands01.py
# This program reviews the <len> function which was
# the first array comamnd shown in this chapter.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print("There are",len(names),"names in the array.")
print()

for k in range(len(names)):
   print(names[k], end = " ")
print()
```

```
 ----jGRASP exec: python ArrayCommands01.py

There are 6 names in the array.

John Greg Maria Heidi Diana David

 ----jGRASP: operation complete.
```

In computer science, you hear the term “append” from time to time. What does “append” mean. Well, you have probably seen books with an “appendix”. This is an extra chapter that was added to the end of the book. To *append* means to *add something to the end*. In the case of arrays, it means to add a new element to the end of an existing array. Program **ArrayCommands02.py**, in Figure 13.20, demonstrates the **append** command. At the beginning of the program, the original array of **names** is displayed. Then **"Mike"** is *appended* and the list is displayed a second time. Note that **'Mike'** has now been added to the end of the array. Then **"Kisha"** is *appended* and the list is displayed a third time. Note that **'Kisha'** has now been added to the end of the array.

Figure 13.20

```python
# ArrayCommands02.py
# This program demonstrates the <append> command
# which will add a new item to the end of an array.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)

names.append("Mike")

print()
print(names)

names.append("Kisha")

print()
print(names)
```

```
 ----jGRASP exec: python ArrayCommands02.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David', 'Mike']

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David', 'Mike', 'Kisha']

 ----jGRASP: operation complete.
```

The **append** command is great when adding to an existing array, but what do you do if you need to build an array from scratch? The answer is you still use **append**. Keep in mind that **append** just requires an existing array. It is OK if this array is empty at the beginning. Program **ArrayCommands03.py**, in Figure 13.21, starts by creating an *empty array* with the command `numbers = []` on line 10. The `[]` indicates an empty array. Even though this array is empty, it still *exists* and is even displayed. Now individual items can be appended, which is done in the **for** loop. After that is finished, the array is displayed again.

Figure 13.21

```
 1 # ArrayCommands03.py
 2 # This program uses the <append> command to build an
 3 # array from scratch.  The empty brackets on line 9
 4 # indicate that <numbers> starts out as an empty array,
 5 # which is displayed.  Then, in the <for> loop, several
 6 # numbers are "appended" and the array is displayed again.
 7
 8
 9 numbers = []
10
11 print()
12 print(numbers)
13
14 for k in range(10):
15    numbers.append(100 + k)
16
17 print()
18 print(numbers)
19
```

```
 ----jGRASP exec: python ArrayCommands03.py

[]

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

 ----jGRASP: operation complete.
```

Sometime, you want to add a new element to an existing array, but you do not want to add it to the end. Maybe, you have a sorted array, and you want to *insert* the new item into its proper position. Program **ArrayCommands04.py**, in Figure 13.22, demonstrates how to use the **insert** command to do just that. This program starts off with the same **names** array from program **ArrayCommands02.py**. It also adds the same 2 names, "**Mike**" and "**Kisha**". Notice that this time, the named are not *appended* (added to the end), they are *inserted* at specific index locations. As before, the array is displayed 3 times. The first is at the beginning of the program. The other 2 times are after each insertion.

Figure 13.22

```python
# ArrayCommands04.py
# This program demonstrates the <insert> command which will
# "insert" a new item in the array at a particular index.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)

names.insert(4,"Mike")

print()
print(names)

names.insert(2,"Kisha")

print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands04.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

['John', 'Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['John', 'Greg', 'Kisha', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

 ----jGRASP: operation complete.
```

The whole reason for storing something is so you can retrieve it later. Frequently, we need to find the location of an item in an array. Program **ArrayCommands05.py**, in Figure 13.23, demonstrates how to do this with the **index** function. This function returns the location or *index* of an item. When you execute the program, you see that **"Diana"** is at **index 4** and **"John"** is at **index 0**.

Figure 13.23

```python
# ArrayCommands05.py
# This program demonstrates the <index> function
# which will return the "index" of a particular
# item in the array.


names = ["John","Greg","Maria","Heidi","Diana","Kisha"]

print()
print(names)

print("\nDiana is located at index",names.index("Diana"))

print("\nJohn is located at index",names.index("John"))


```

```
 ----jGRASP exec: python ArrayCommands05.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'Kisha']

Diana is located at index 4

John is located at index 0

 ----jGRASP: operation complete.
```

It should make sense that if we can insert item into an array, we can also remove items from an array. Program **ArrayCommands06.py**, in Figure 13.24, demonstrates how to do this with the **remove** command. This time, the **names** array starts with 8 names. Then **"Mike"** and **"John"** are removed from the array.

Figure 13.24

```
# ArrayCommands06.py
# This program demonstrated the <remove> command
# which will remove a specific item from the array.


names = ["John","Greg","Kisha","Maria","Heidi","Mike","Diana","David"]

print()
print(names)

names.remove("Mike")

print()
print(names)

names.remove("John")

print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands06.py

['John', 'Greg', 'Kisha', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['John', 'Greg', 'Kisha', 'Maria', 'Heidi', 'Diana', 'David']

['Greg', 'Kisha', 'Maria', 'Heidi', 'Diana', 'David']

 ----jGRASP: operation complete.
```

Questions:
Suppose an item exists in an array more than once.
What would happen if you try to **remove** it?
What would happen if you try to locate it with **index**?

Program **ArrayCommands07.py**, in Figure 13.25, experiments with this concept. Note that **"David"** is in the array twice, both at index **4** and at index **6**. When I use **index** to locate **"David"**, it returns index **4**, because that it the first time **"David"** shows up in the array. This is the same **"David"** that gets **remove**d from the array on line 12, for the same reason. After the first **"David"** is removed, the remaining **"David"** can be located at **index 5**.

Figure 13.25

```python
# ArrayCommands07.py
# This program demonstrates that if the same item is in the
# array more than once, the <index> command will only find
# the "index" of the first occurrence and the <remove> command
# will only "remove" the first occurrence from the array.


names = ["John","Greg","Maria","Heidi","David","Diana","David"]

print()
print(names)

print("\nDavid is located at index",names.index("David"))
names.remove("David")

print()
print(names)

print("\nDavid is located at index",names.index("David"))

```

```
 ----jGRASP exec: python ArrayCommands07.py

['John', 'Greg', 'Maria', 'Heidi', 'David', 'Diana', 'David']

David is located at index 4

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

David is located at index 5

 ----jGRASP: operation complete.
```

Question:
What happens if you attempt to remove an item from an array that does not exist?

Program **ArrayCommands08.py**, in Figure 13.26, experiments with this concept. This time, the **names** array starts with 6 names, none of which are **"Mike"**. Then the program attempts to remove **"Mike"** from the array. The result is a *run-time error* that states **ValueError: list.remove(x): x not in list**.

Figure 13.26

```python
# ArrayCommands08.py
# This program demonstrates that attempting to
# "remove" an item that is not in the array will
# cause the program to crash.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)

names.remove("Mike")

print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands08.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']
Traceback (most recent call last):
  File "ArrayCommands08.py", line 12, in <module>
    names.remove("Mike")
ValueError: list.remove(x): x not in list

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Well, this is a definite problem. Before we figure out a solution, let us look at a similar problem by asking another, very similar, question:

What happens if you attempt to <u>locate</u> an item from an array that does not exist?

Program **ArrayCommands09.py**, in Figure 13.27, experiments with this concept. It uses the same exact array from the previous program – an array that still does not contain **"Mike"**. When the program tries to find the index of **"Mike"** in the array, we get another *run-time error*. This time it states **ValueError: 'Mike' is not in list**. This is very similar to the run-time error from the previous program.

Figure 13.27

```
# ArrayCommands09.py
# This program demonstrates that attempting to find
# the "index" of an item that is not in the array will
# cause the program to crash.


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)

print("\nMike is located at index",names.index("Mike"))

```

```
 ----jGRASP exec: python ArrayCommands09.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']
Traceback (most recent call last):
  File "ArrayCommands09.py", line 12, in <module>
    print("\nMike is located at index",names.index("Mike"))
ValueError: 'Mike' is not in list

 ----jGRASP wedge2: exit code for process is 1.
 ----jGRASP: operation complete.
```

Now, locating and removing items from arrays are two very common tasks in array processing. If attempting to locate/remove a non-existent element from an array crashes the program, then we need a way to detect if an item is in the array before we attempt to use either the **index** or **remove** command.

Program **ArrayCommands10.py**, shown in Figure 13.28, demonstrates the **in** operator and how you can use it check is a particular value exists "in" the array. If the program confirms that an item is "in" the array, its location will be found with **index**; otherwise a simple message states the item is not in the array without needing to crash the program.

Figure 13.28

```
# ArrayCommands10.py
# The program uses the <in> operator to check if a particular item
# is "in" the array before attempting to find its "index".
# This prevents the program from crashing.
```

```


names = ["John","Greg","Maria","Heidi","Diana","David"]

print()
print(names)
print()

if "Heidi" in names:
   print("'Heidi' is in the list at index",names.index("Heidi"))
else:
   print("'Heidi' is not in the list.")
print()

if "Leon" in names:
   print("'Leon' is in the list at index",names.index("Leon"))
else:
   print("'Leon' is not in the list.")

```

```
 ----jGRASP exec: python ArrayCommands10.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

'Heidi' is in the list at index 3

'Leon' is not in the list.

 ----jGRASP: operation complete.
```

Program **ArrayCommands11.py**, shown in Figure 13.29, demonstrates that the same **in** operator can also be used to check is a particular value exists "in" the array before you **remove** it. It also shows that the item that is being removed can actually be entered by the user. There are two outputs shown. In the first, "**Heidi**" is entered and then removed from the list. This is confirms when the list is shown again. In the second, "**Leon**" is entered and we are informed he is not in the list.

Figure 13.29

```python
# ArrayCommands11.py
# The program uses the <in> operator to check if a particular item
# is "in" the array before attempting to "remove" it.
# This prevents the program from crashing.


names = ["John","Greg","Maria","Heidi","Diana","David"]
print()
print(names)
print()

name = input("Who do you wish to remove from the list?  -->  ")
print()

if name in names:
   names.remove(name)
   print(name,"has been removed from the list.")
   print()
   print(names)
else:
   print(name,"was not in the list.")
```

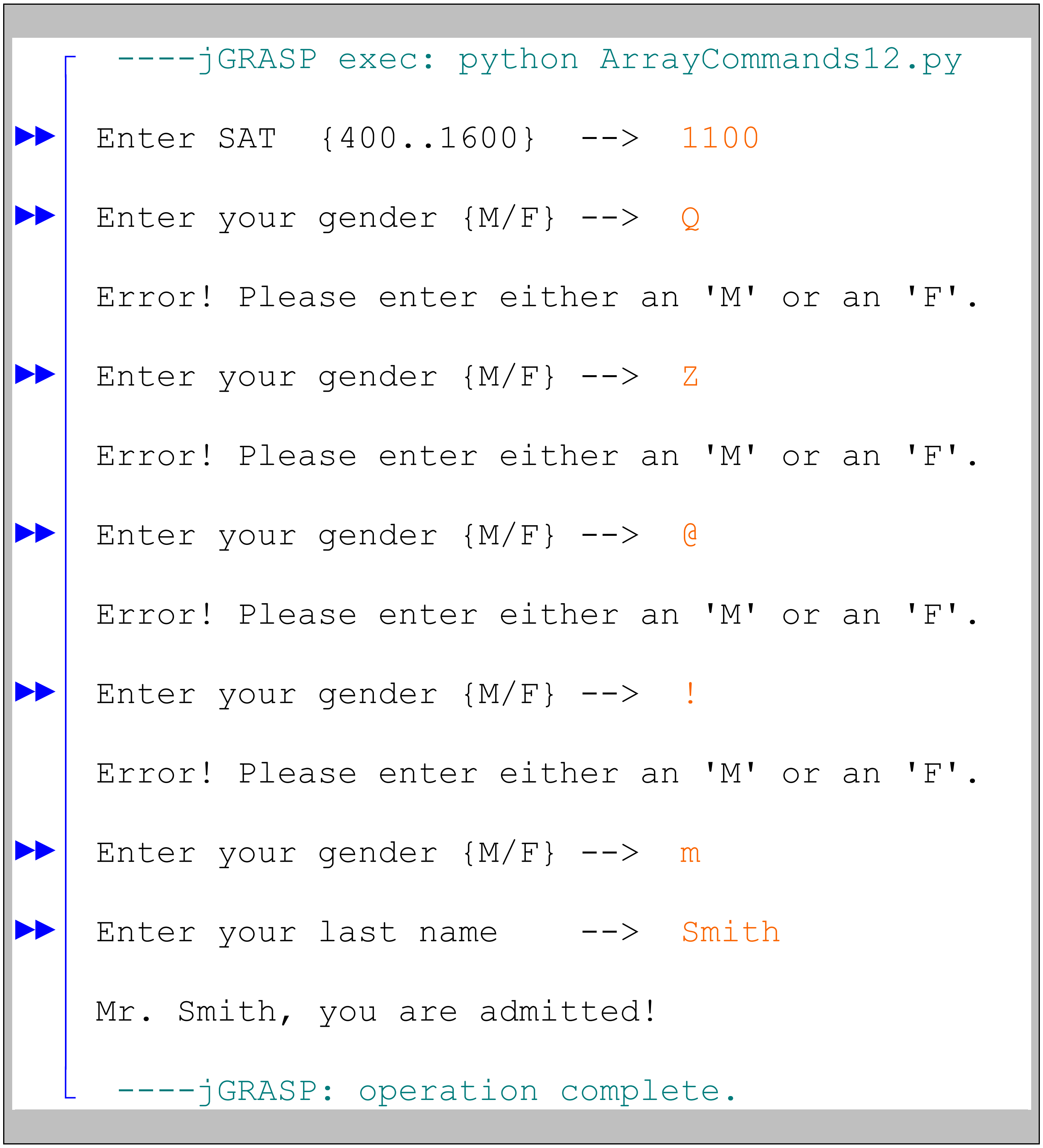
```
 ----jGRASP exec: python ArrayCommands11.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

Who do you wish to remove from the list?  -->  Heidi

Heidi has been removed from the list.

['John', 'Greg', 'Maria', 'Diana', 'David']

 ----jGRASP: operation complete.
```

```
 ----jGRASP exec: python ArrayCommands11.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David']

Who do you wish to remove from the list?  -->  Leon

Leon was not in the list.

 ----jGRASP: operation complete.
```

Program **ArrayCommands12.py**, in Figure 13.30, shows another, recent example of a “Hidden Array". This one is from Chapter XI and was originally program **InputProtection04.py**. On line 21 we have the statement **`genderOK = gender in ['M','m','F','f']`**. Without this shortcut, you would have to type out **`genderOK = gender == 'M' or gender == 'm' or gender == 'F' or gender == 'f'`**. The point here is **`['M','m','F','f']`** is an array! If **gender** equals any of the items in that array, **genderOK** will be **True**.

Figure 13.28

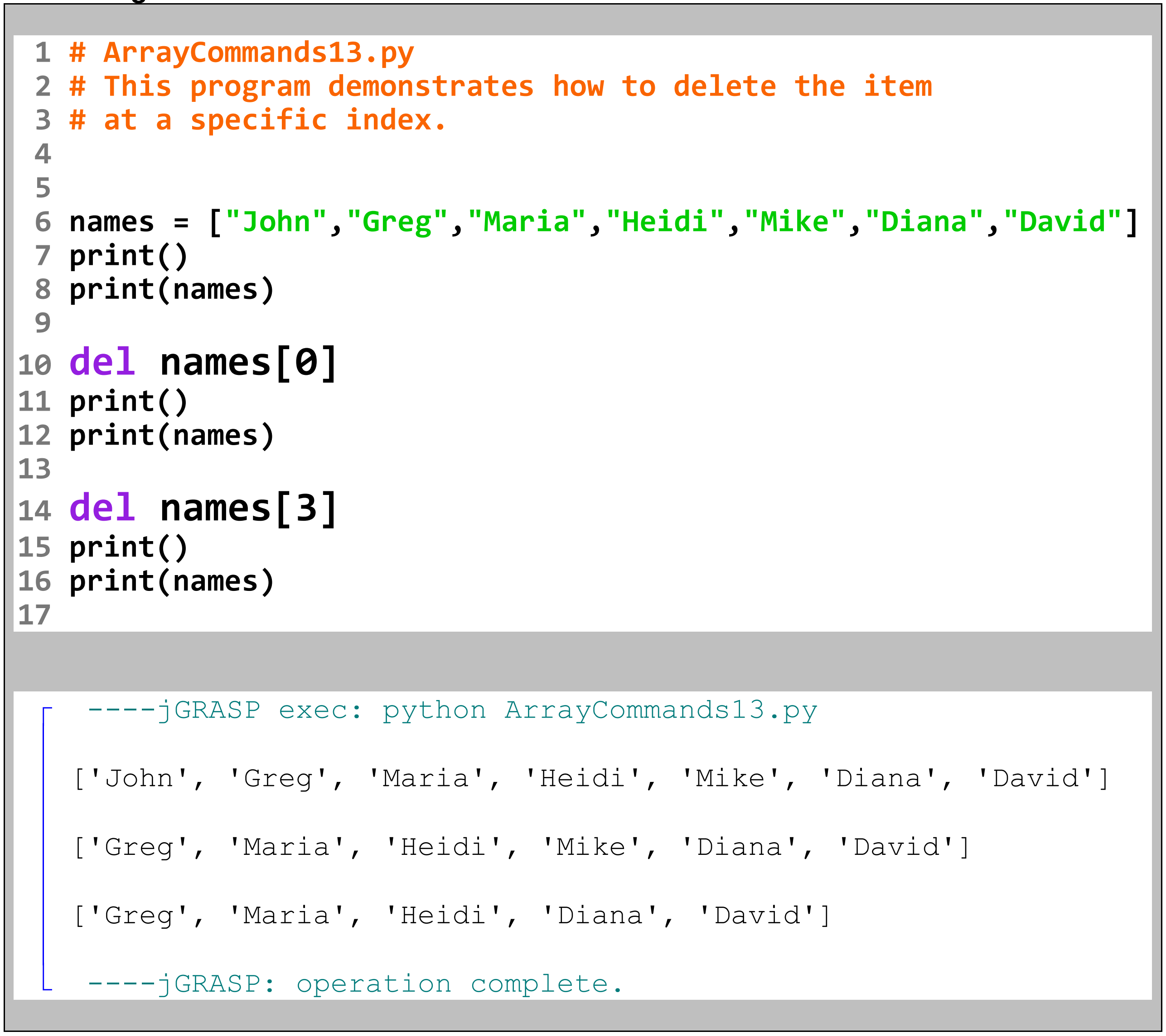
```
 1 # ArrayCommands12.py
 2 # This program was originally program InputProtection04.py from
 3 # Chapter 11.  The second <while> loop shows that an array can
 4 # be used to make a long compound condition less complicated.
 5 # Without it, the command would read:
 6 # genderOK = gender == 'M' or gender == 'm' or gender == 'F' or gender == 'f'
 7
 8
 9 sat = 0
10 satOK = False
11 while not satOK:
12    sat = eval(input("\nEnter SAT  {400..1600}  -->  "))
13    satOK = 400 <= sat <= 1600
14    if not satOK:
15       print("\nError! Please enter a number between 400 & 1600.")
16
17 gender = ''
18 genderOK = False
19 while not genderOK:
20    gender = input("\nEnter your gender {M/F} -->  ")
21    genderOK = gender in ['M','m','F','f'] # This is an array.
22    if not genderOK:
23       print("\nError! Please enter either an 'M' or an 'F'.")
24
25 lastName = input("\nEnter your last name    -->  ")
26 print()
27
28 if gender == 'M' or gender == 'm':
29    print("Mr.",lastName,end = ", ")
30 if gender == 'F' or gender == 'f':
31    print("Ms.",lastName,end = ", ")
32
33 if sat >= 1100:
34    print("you are admitted!")
35 else:
36    print("you are not admitted.")
```



```
 ----jGRASP exec: python ArrayCommands12.py
Enter SAT  {400..1600}  -->  1100
Enter your gender {M/F} -->  Q
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  Z
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  @
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  !
Error! Please enter either an 'M' or an 'F'.
Enter your gender {M/F} -->  m
Enter your last name    -->  Smith
Mr. Smith, you are admitted!
 ----jGRASP: operation complete.
```

When it comes to removing elements from an array, there are actually 3 different options. You have already seen the **remove** command. The second option is the **del** command. So what is the difference? As you have seen, the **remove** command allows you to "remove" elements based on their value. The **del** command allows you to **del**ete items based on their index. This is demonstrated by program **ArrayCommands13.py**, shown in Figure 13.31.

Figure 13.31



```
# ArrayCommands13.py
# This program demonstrates how to delete the item
# at a specific index.


names = ["John","Greg","Maria","Heidi","Mike","Diana","David"]
print()
print(names)

del names[0]
print()
print(names)

del names[3]
print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands13.py

['John', 'Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['Greg', 'Maria', 'Heidi', 'Diana', 'David']

 ----jGRASP: operation complete.
```

We have seen 2 commands so far (**remove** and **del**) which allow us to remove/delete items from an array. There is one more. Program **ArrayCommands14.py**, in Figure 13.32, demonstrates the **pop** command. While it has a different syntax, it is similar to **del** because it deletes based on the index.

Figure 13.32

```python
# ArrayCommands14.py
# This program demonstrates another way to delete the item
# at a specific index.


names = ["John","Greg","Maria","Heidi","Mike","Diana","David"]
print()
print(names)

names.pop(0)
print()
print(names)

names.pop(3)
print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands14.py

['John', 'Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

['Greg', 'Maria', 'Heidi', 'Diana', 'David']

 ----jGRASP: operation complete.
```

You may wonder why the **pop** command exists if it does the same thing **del**. It is true that programming languages sometimes have some redundancy in them. Back in Chapter VI you learned the **pow** function does the exact same thing as the ****** operator. While there are several examples of redundancy in Python, this is actually not one of them. The previous two program examples only made it look like **pop** and **del** do the exact same thing. The next two program examples will show you how they are different because both commands can do things the other command cannot.

Program **ArrayCommands15.py**, in Figure 13.33, demonstrates a couple special features of the **pop** command. The first thing we notice is that **pop** is actually a function. By that I mean it "returns a value". The value **pop** returns is the same value that was just "popped" from the array. There is another feature. Suppose you wish to delete the last item in an array, but you do not know its index or its value. The **pop** command can do this simply by not using an argument. Remember, neither of these features works with **del**.

Figure 13.33

```python
# ArrayCommands15.py
# This program demonstrates that <pop> actually
# returns the removed item.  <del> does not do this.

names = ["John","Greg","Maria","Heidi","Mike","Diana","David"]
print()
print(names)

print()
print(names.pop(0))
print(names)

print()
print(names.pop(3))
print(names)

print()
print(names.pop())  # Removes the last element
print(names)
```

```
 ----jGRASP exec: python ArrayCommands15.py

['John', 'Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

John
['Greg', 'Maria', 'Heidi', 'Mike', 'Diana', 'David']

Mike
['Greg', 'Maria', 'Heidi', 'Diana', 'David']

David
['Greg', 'Maria', 'Heidi', 'Diana']

 ----jGRASP: operation complete.
```

Program **ArrayCommands16.py**, in Figure 13.34, demonstrates **del**'s special feature. Specifically, the **del** command can delete more than one item at a time. It can delete an entire *slice* of items. You just need to use the same slice **[***start***:***stop***]** syntax. Remember, the first number specifies the beginning index of your slice. The second index actually specifies the first index that is NOT part of the slice. So, when deleting the **names[2:5]** slice from the **names** array, it is the items from indexes **2** through **4** (not **5**) that get deleted. Remember, this feature does not work with **pop**.

Figure 13.34

```
# ArrayCommands16.py
# This program shows that <del> allows you to delete
# an entire "slice" from an array.  Remember, you need
# to specify the starting and stopping index.  In this
# program, the [2:5] specifies deleting the names from
# indexes 2-4 (not 5).  This cannot be done with <pop>.


names = ["John","Greg","Maria","Heidi","Mike","Kisha","Diana","David"]

print()
print(names)

del names[2:5]

print()
print(names)

```

```
 ----jGRASP exec: python ArrayCommands16.py

['John', 'Greg', 'Maria', 'Heidi', 'Mike', 'Kisha', 'Diana', 'David']

['John', 'Greg', 'Kisha', 'Diana', 'David']

 ----jGRASP: operation complete.
```

Program **ArrayCommands17.py**, in Figure 13.35, is actually very similar to program **ArraySyntax12.py**. The earlier program used the **+=** operator to join or concatenate one array to the end of an existing array. Now with program **ArrayCommands15.py**, we see that this can also be done with the **extend** command.

Figure 13.35

```
# ArrayCommands17.py
# This program is very similar to ArraySyntax12.py
# It shows that the <extend> command does the same
# thing as the += operator.


list = [11,22,33,44,55]

print()
print(list)

list.extend([66,77,88,99])

print()
print(list)

```

```
 ----jGRASP exec: python ArrayCommands17.py

[11, 22, 33, 44, 55]

[11, 22, 33, 44, 55, 66, 77, 88, 99]

 ----jGRASP: operation complete.
```

NOTE: Both of these commands do the exact same thing:

`list += [66,77,88,99]` and `list.extend([66,77,88,99])`

Sorting Arrays

Any array of information is easier to work with if it is sorted. Sometime when you build a array, you take special care to **insert** every array item into its proper position. Other times, arrays are built simply by appending one item after another, regardless of their value. The result is an array of seemingly random information. When this happens, it is useful if we can *sort* the array. Program **ArrayCommands18.py**, in Figure 13.36, has 3 arrays: **names**, **ages** and **gpas**. **names** is an array of strings; **ages** is an array of integers; and **gpas** is an array of real numbers. At the start of the program, none of the arrays are sorted. On lines 19 through 21, the **sort** command is used with each array. To show that the **sort** command did in fact work, the arrays are displayed both before and after the sorting.

Figure 13.36

```
 1 # ArrayCommands18.py
 2 # This program demonstrates the <sort> command which
 3 # will "sort" an array of strings alphabetically and
 4 # "sort" an array of numbers in ascending order.
 5
 6
 7 names = ["John","Greg","Maria","Heidi","Diana","David","Mike","Kisha"]
 8 ages  = [23,76,13,45,54,18,3,103,37,89,63]
 9 gpas  = [3.9,2.75,2.1,1.65,4.0,3.25,2.45,0.95,3.88,3.75]
10
11 print()
12 print(names)
13 print()
14 print(ages)
15 print()
16 print(gpas)
17 print()
18
19 names.sort()
20 ages.sort()
21 gpas.sort()
22
23 print()
24 print(names)
25 print()
26 print(ages)
27 print()
28 print(gpas)
```

```
 ----jGRASP exec: python ArrayCommands18.py

['John', 'Greg', 'Maria', 'Heidi', 'Diana', 'David', 'Mike', 'Kisha']

[23, 76, 13, 45, 54, 18, 3, 103, 37, 89, 63]

[3.9, 2.75, 2.1, 1.65, 4.0, 3.25, 2.45, 0.95, 3.88, 3.75]


['David', 'Diana', 'Greg', 'Heidi', 'John', 'Kisha', 'Maria', 'Mike']

[3, 13, 18, 23, 37, 45, 54, 63, 76, 89, 103]

[0.95, 1.65, 2.1, 2.45, 2.75, 3.25, 3.75, 3.88, 3.9, 4.0]

 ----jGRASP: operation complete.
```

While the previous program worked, there is one minor issue with the **gpas** array. When you are trying to determine the valedictorian and the salutatorian, you want to see the highest grade point averages at the beginning of the array, not the end. It is true that the **sort** command only sorts in *ascending* order, but once the array is sorted, we can use the **reverse** command to "reverse" the order of the list. This is demonstrated by program **ArrayCommands19.py**, shown in Figure 13.36. Note that the **gpas** array is displayed 3 times. First the original unsorted list is displayed, followed by the sorted (ascending order) list, followed by the reversed (descending order) list.

Figure 13.37

```
# ArrayCommands19.py
# This program demonstrates how to sort an array of
# numbers in descending order.  It is done in 2 steps.
# First you <sort> the list; then you <reverse> it.


gpas = [3.9,2.75,2.1,1.65,4.0,3.25,2.45,0.95,3.88,3.75]

print()
print(gpas)

gpas.sort()

```

```python
print()
print(gpas)

gpas.reverse()

print()
print(gpas)
```

```
 ----jGRASP exec: python ArrayCommands19.py

[3.9, 2.75, 2.1, 1.65, 4.0, 3.25, 2.45, 0.95, 3.88, 3.75]

[0.95, 1.65, 2.1, 2.45, 2.75, 3.25, 3.75, 3.88, 3.9, 4.0]

[4.0, 3.9, 3.88, 3.75, 3.25, 2.75, 2.45, 2.1, 1.65, 0.95]

 ----jGRASP: operation complete.
```

In addition to **sort**, Python also has a **sorted** command. While both **sort** and **sorted** "sort" array items, they are a couple important differences. First, the **sorted** command does not actually sort the array. Instead, **sorted** is a function that returns a sorted copy of the array, leaving the original array unchanged. Second, **sorted** has an option to sort in reverse order.

Program **ArrayCommands20.py**, in Figure 13.38, demonstrates the **sorted** function. We start off with the same **gpas** array from the previous program, only this time it is called **gpas1**. Then a second array, **gpas2**, is created as a sorted copy of **gpas1**. Then a third array, **gpas3**, is created as a **reverse** sorted copy of **gpas1**. Finally, the three arrays are displayed. The output starts with descending array of **gpas3**. Then we see the ascending array of **gpas2** and finally the unsorted **gpas1** array. The arrays were intentionally shown in this order to prove that the **sorted** function did not affect the original **gpas1** array at all.

Figure 13.38

```python
# ArrayCommands20.py
# This program demonstrates the <sorted> function.
# While similar to the <sort> command, there are 2
# important differences:
# 1. <sorted> does not actually sort the array.
#    Instead, it returns a sorted copy of the array.
# 2. <sorted> has an option to sort in reverse order.
```

```


gpas1 = [3.9,2.75,2.1,1.65,4.0,3.25,2.45,0.95,3.88,3.75]

gpas2 = sorted(gpas1)

gpas3 = sorted(gpas1, reverse=True)

print()
print(gpas3)
print(gpas2)
print(gpas1)
```

```
 ----jGRASP exec: python ArrayCommands20.py

[4.0, 3.9, 3.88, 3.75, 3.25, 2.75, 2.45, 2.1, 1.65, 0.95]
[0.95, 1.65, 2.1, 2.45, 2.75, 3.25, 3.75, 3.88, 3.9, 4.0]
[3.9, 2.75, 2.1, 1.65, 4.0, 3.25, 2.45, 0.95, 3.88, 3.75]

 ----jGRASP: operation complete.
```

There is one potential issue when you sort arrays. This one deals with arrays of strings and is demonstrated by program **ArrayCommands21.py**, shown in Figure 13.39. This program has an **animals** array which stores the names of 8 different animals. Both *sort* commands are used and the sorted array is displayed twice.

Figure 13.39

```
# ArrayCommands21.py
# This program demonstrates a problem that happens when you try
# to sort strings that contain both CAPITAL and lowercase letters.
# Capital letters have numeric code values from 65-90.
# Lowercase letters have numeric code values from 97-122.
# If lowercase letters are sorted together with capital letters,
# the capitals will come first because of the smaller code values.


animals = ["hippo","monkey","ZEBRA","TOUCAN","lion",
           "GIRAFFE","cheetah","ELEPHANT"]

print()
print(sorted(animals))
animals.sort()
print(animals)
```

```
 ----jGRASP exec: python ArrayCommands21.py

['ELEPHANT', 'GIRAFFE', 'TOUCAN', 'ZEBRA', 'cheetah', 'hippo', 'lion', 'monkey']
['ELEPHANT', 'GIRAFFE', 'TOUCAN', 'ZEBRA', 'cheetah', 'hippo', 'lion', 'monkey']

 ----jGRASP: operation complete.
```

This output probably looks a bit strange. How is this a "sorted array"? Since when does **"ZEBRA"** come before **"cheetah"**? Do keep in mind that sorting with strings is based on the numerical values of the characters. This can have some interesting results. Capital letters have numeric code values that range from 65 (uppercase 'A') to 90 (uppercase 'Z'). Lowercase letters have numeric code values that range from 97 (lowercase 'a') to 122 (lowercase 'z'). This means lowercase **'a'** has a larger numerical code than capital **'Z'**, which means that in a mixed *uppercase/lowercase* sorting environment, you will find that the **Z** will be placed before the **a**. Fixing this issue requires converting all of the string to either all uppercase letters or all lowercase letters. This is something that we will explorer in the next chapter.

https://azpng.com/png/2019/06/13/animal-clipart-free-library-herd-of-animals-in-the.png

13.4 Random Arrays

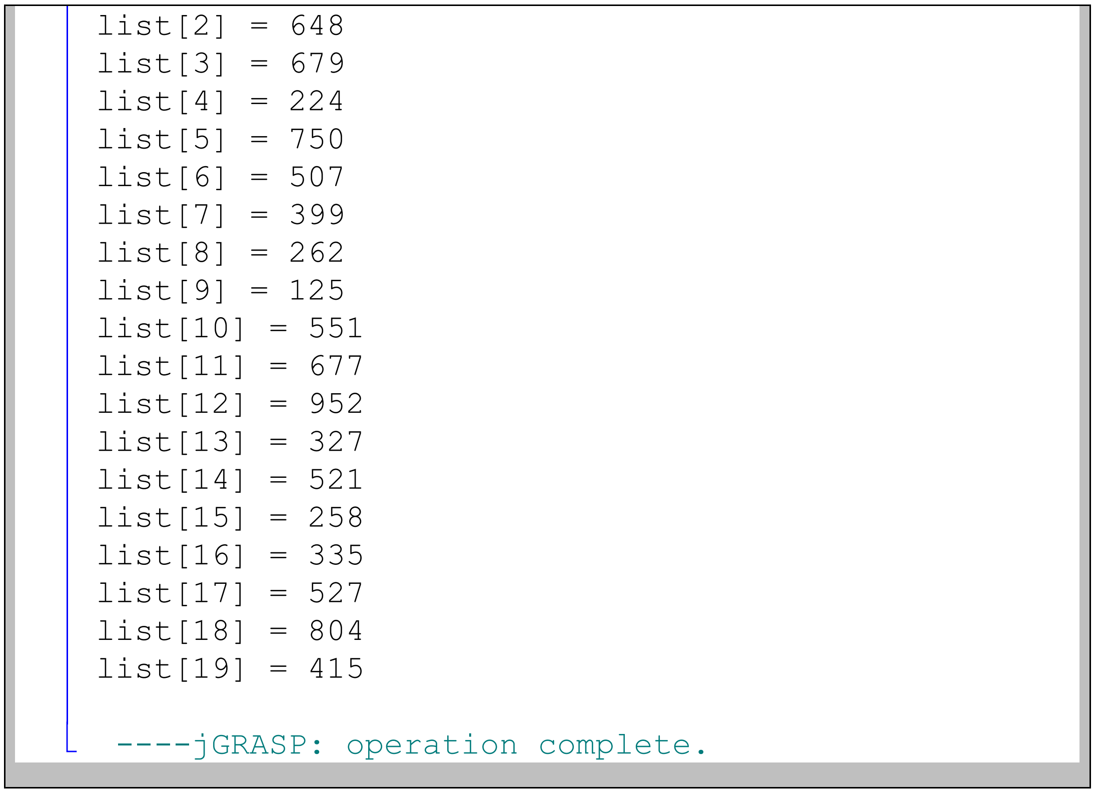
Hard coding array elements during array definitions can be tedious. This is especially tedious if there are many arrays values. Entering array elements during program execution also becomes quite annoying when there are many numbers to be added. Frequently, in the process of learning computer science it is important to process a large quantity of data, and such data in many cases does not need to be any specific values. In other words, random data is just fine.

Program **RandomArrays01.py**, in Figure 13.40, creates 20 random 3-digit integers. Once created, each random number is immediately appended to the **list** array. When you run this program, you will not get the exact same set of integers that I am showing in my output. Remember, the numbers are random. However, they will all still be in the same range from **100** to **999**.

Figure 13.40

```
# RandomArrays01.py
# This program fills an array with random
# 3-digit integers and then displays the
# entire contents of the array.


from random import randint


list = []
for k in range(20):
   list.append(randint(100,999))

print()
for k in range(20):
   print("list["+str(k)+"] =",list[k])

```

```
 ----jGRASP exec: python RandomArrays01.py

list[0] = 215
list[1] = 337
```



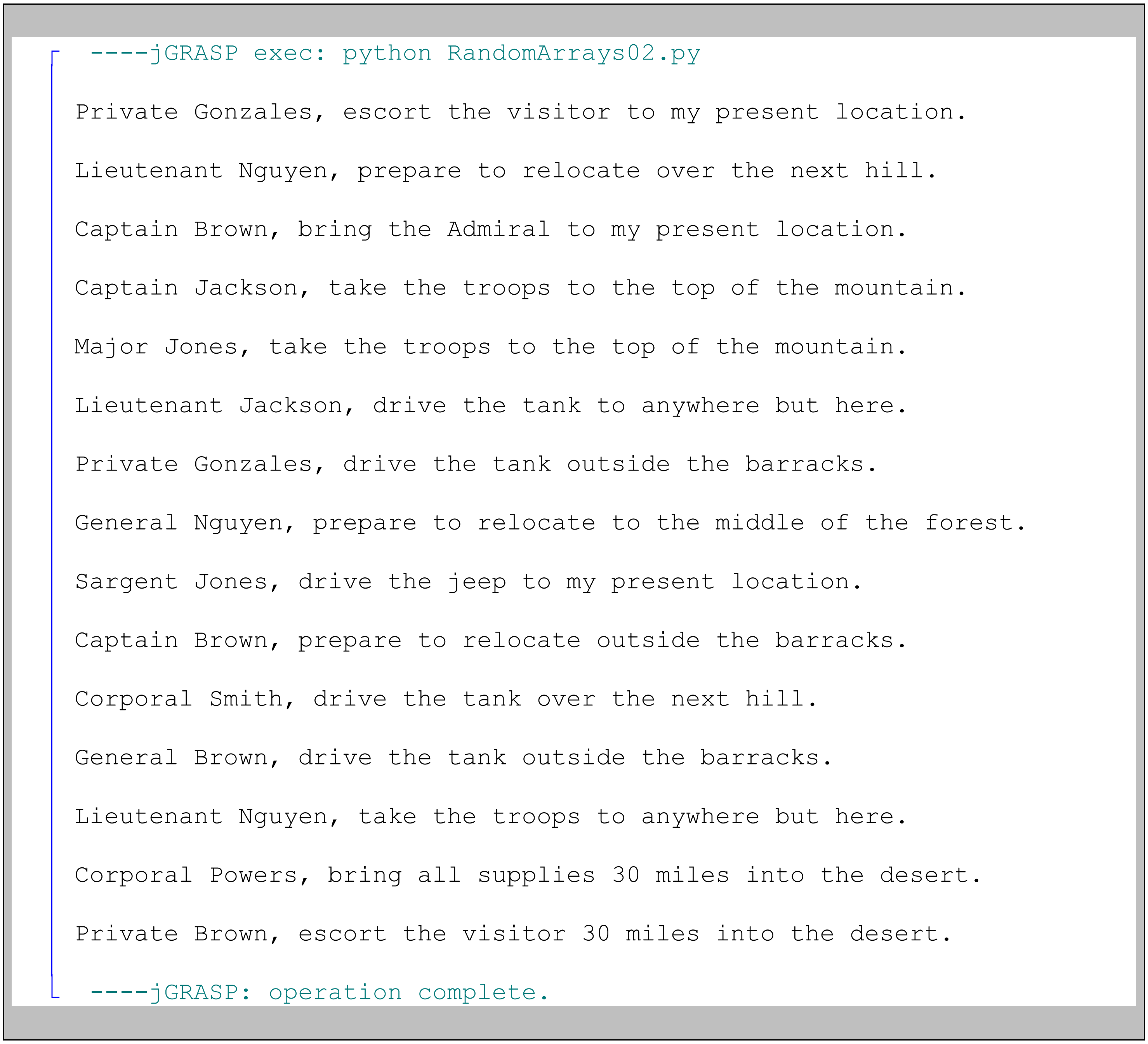
```
list[2] = 648
list[3] = 679
list[4] = 224
list[5] = 750
list[6] = 507
list[7] = 399
list[8] = 262
list[9] = 125
list[10] = 551
list[11] = 677
list[12] = 952
list[13] = 327
list[14] = 521
list[15] = 258
list[16] = 335
list[17] = 527
list[18] = 804
list[19] = 415

 ----jGRASP: operation complete.
```

In terms of "Random Arrays" I think the following is a much better example. Several years ago I saw an advertisement for a solder action figure. This was not a cheap piece of plastic found in a bucket of 100 soldiers. This 12 inch action figure was fairly sophisticated. It had a built-in voice synthesizer so that it could speak. The commercial claimed the action figure could speak over half a million different commands. You might be thinking that somewhere in the world, there was some poor guy that had to type 500,000+ different sentences for this action figure. But that is not how it worked at all. In the previous program you saw an array that was filled with random data. What if the data is not random at all, but instead the <u>index</u> is random. This would allow you to access the information randomly.

This is what is done in program **RandomArrays02.py**, shown in Figure 13.41. This program does not have the capability to generate 500,000+ different commands, but it can generate exactly 2401. Here is how it works. There are 4 different arrays. One has 7 **ranks**; one has 7 **names**; one has 7 **actions** and the last one has 7 **locations**. 4 random indexes are chosen, one for each array. This allows me to create a sentence with a random rank, followed by a random name, followed by a random action followed by a random location. With 7 items in each array there are 7 * 7 * 7 * 7 or 2401 different sentences possible. The action figure from several years ago did something very similar, but it had about 27 items in each array. This means 27 * 27 * 27 * 27 = 531,441 different sentences possible.

Figure 13.41

```python
# RandomArrays02.py
# This program will display 15 random sentences.
# With 7 different ranks, 7 different people,
# 7 different actions and 7 different locations,
# there are more than 2400 different sentences possible.


from random import randint


ranks = ["Private","Corporal","Sargent","Lieutenant","Captain","Major","General"]

names = ["Smith","Gonzales","Brown","Jackson","Powers","Jones","Nguyen"]

actions = ["drive the tank","drive the jeep","take the troops",
           "bring all supplies","escort the visitor","prepare to relocate",
           "bring the Admiral"]

locations = ["over the next hill", "to the top of the mountain",
             "outside the barracks", "30 miles into the desert",
             "to the middle of the forest", "to my present location",
             "to anywhere but here"]


for k in range(15):
   randomRank     = randint(0,len(ranks)-1)
   randomPerson   = randint(0,len(names)-1)
   randomAction   = randint(0,len(actions)-1)
   randomLocation = randint(0,len(locations)-1)

   sentence = ranks[randomRank] + " " + names[randomPerson] + ", " + \
              actions[randomAction] + " " + locations[randomLocation] + "."

   print("\n" + sentence)

```



```
 ----jGRASP exec: python RandomArrays02.py

Private Gonzales, escort the visitor to my present location.

Lieutenant Nguyen, prepare to relocate over the next hill.

Captain Brown, bring the Admiral to my present location.

Captain Jackson, take the troops to the top of the mountain.

Major Jones, take the troops to the top of the mountain.

Lieutenant Jackson, drive the tank to anywhere but here.

Private Gonzales, drive the tank outside the barracks.

General Nguyen, prepare to relocate to the middle of the forest.

Sargent Jones, drive the jeep to my present location.

Captain Brown, prepare to relocate outside the barracks.

Corporal Smith, drive the tank over the next hill.

General Brown, drive the tank outside the barracks.

Lieutenant Nguyen, take the troops to anywhere but here.

Corporal Powers, bring all supplies 30 miles into the desert.

Private Brown, escort the visitor 30 miles into the desert.

 ----jGRASP: operation complete.
```

13.5 The for..each Loop

There is a variation of the **for** loop tht is sometimes called the **for..each** loop. To understand how it work, let us look at program **forEach01.py**, shown in Figure 13.42. This program actually has 2 identical arrays, **listA** and **listB**. **listA** is traversed with a "traditional" **for** loop. **listB** is traversed with the **for..each** loop. When we look at the output, it seems that both loop yield the exact same result.

Figure 13.42

```python
# forEach01.py
# This program compares 2 "flexible" ways to
# "traverse" an array.  This first reviews the
# used of the <len> function.  The second uses
# the <for..each> loop which some people prefer.
# For displaying the contents of an array, both
# ways work just fine.


listA = [100,101,102,103,104,105,106,107,108,109]
print()
for k in range(len(listA)):
   print(listA[k], end = " ")
print()


listB = [100,101,102,103,104,105,106,107,108,109]
print()
for number in listB:
   print(number, end = " ")
print()

```

```
 ----jGRASP exec: python forEach01.py

100 101 102 103 104 105 106 107 108 109

100 101 102 103 104 105 106 107 108 109

 ----jGRASP: operation complete.
```

Let us take a closer look:

Traditional **for** Loop	The **for..each** Loop

`for k in range (len(listA)):`
`   print(listA[k], end = " ")`
	`for number in listB:`
	`   print(number, end = " ")`

The "traditional" **for** loop is what we are used to. We have a *loop control variable*, **k** in this case, that counts from **0** to 1 less than the **range** value. Since the **len**gth of the **listA** array is **10**, the loop counter counts from **0** to **9**.

The **for..each** loop may be a little confusing at first. The first thing that many people notice is that the word "each" does not appear anywhere. This is true. Note also that the **for..each** loop manages to access individual elements in the array without using an index. This is also true. The reason this is called "the **for..each** loop" is the *item variable*, **number** in this case, will take on value of "each" item in the array. Essentially, you can read this **for..each** loop as

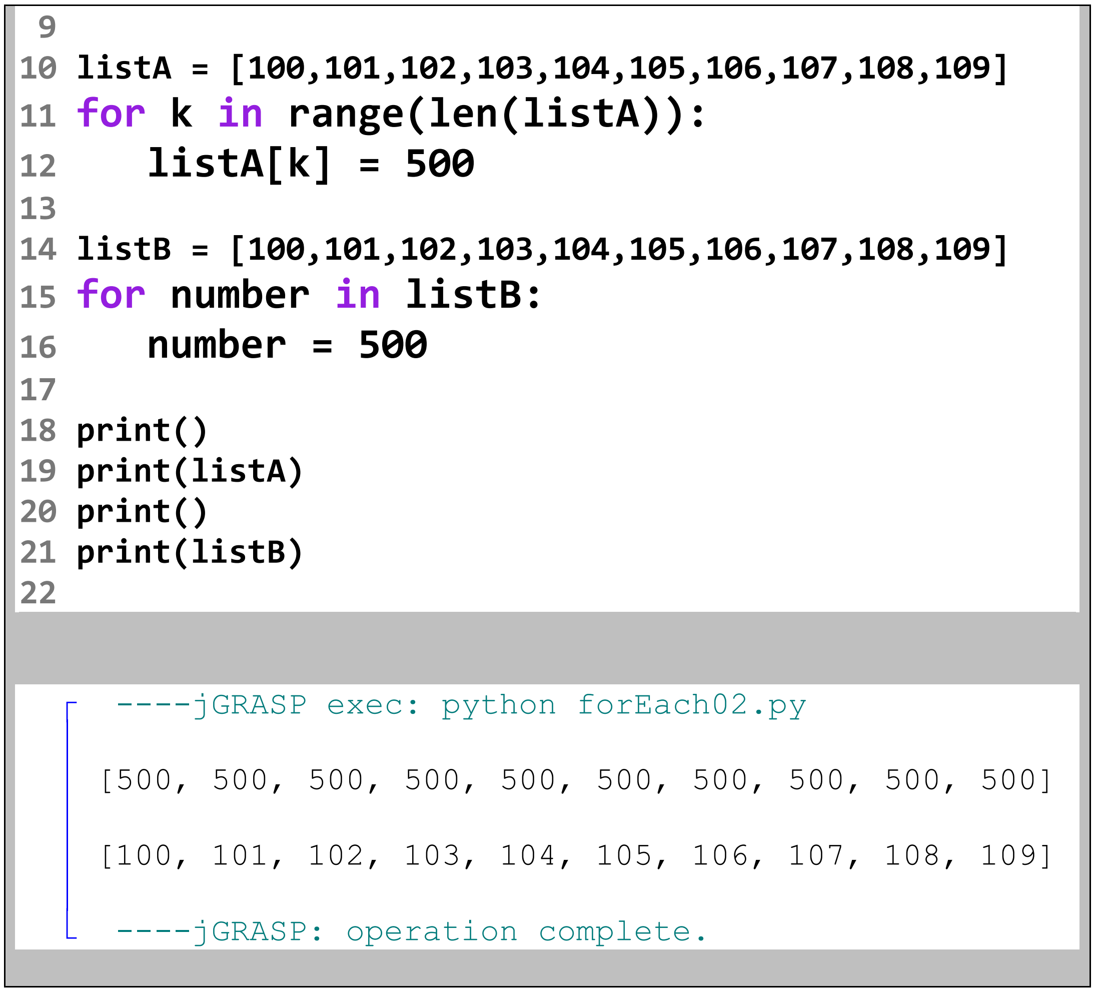
"For each and every **number** in the **listB** array, display the **number**."

The **for..each** loop was created to help make traversing arrays (and other data structures) more readable, and to help eliminate the common OBOB (**O***ff* **B***y* **O***ne* **B***ug*) that occurs when people forget that the first index of the array is **0**. Some people prefer the **for..each** loop over the traditional **for** loop. Some do not. In this particular example, it seems that both loops work just as well.

Program **forEach02.py**, in Figure 13.43, does another comparison between the 2 **for** loops. This time, we are not displaying the value of each item as the arrays are traversed. Instead, we are trying to changes their values. Specifically, we are trying to change the value of every number in both arrays to **500**. This seems simple enough; however, notice what happens when you run the program.

Figure 13.43

```
# forEach02.py
# This program does another comparison between
# the <for> loop and the <for..each> loop.
# This time, the mission is to change all
# of the numbers in both arrays to 500.
# The output shows that the <for..each> loop is
# not able to change the contents of an array.

```



```python

listA = [100,101,102,103,104,105,106,107,108,109]
for k in range(len(listA)):
    listA[k] = 500

listB = [100,101,102,103,104,105,106,107,108,109]
for number in listB:
    number = 500

print()
print(listA)
print()
print(listB)

```

```
 ----jGRASP exec: python forEach02.py

[500, 500, 500, 500, 500, 500, 500, 500, 500, 500]

[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]

 ----jGRASP: operation complete.
```

So what happened. The traditional **for** loop worked. What not the **for..each** loop? To answer this question, you need to truly understand how the **for..each** loop works. In this program, **number** is the *item variable* which takes in the value of every number or item in the **listB** array, one after the other. In other words, **number** receives a COPY of every number or item in the **listB** array. Altering the value of **number** only alters a copy of the information. It has no effect on any or the original items in the array itself.

So, the **for..each** loop is fine if you want to traverse an array and retrieve the values of the individual array elements, but not if you want to change them.

It that is? Well, for now, yes. We can have an entire chapter just on the **for..each** loop and how it is used with different data structures, but that would go way beyond the scope of this first year course. You may wonder then why I even mentioned the **for..each** loop. There are 2 reasons. First, I will actually use it in some of the program examples in the next few chapters. Second, in spite of its limitations, a lot of people prefer it over the traditional **for** loop for certain programs. So I wanted you to at least have some "exposure" to it.