Chapter 4
Basic Data Types: Lists and Tuples

Abstract GIS scripting frequently involves manipulating collections of items, such as files or data records. For example, you might have multiple tabular data files with special characters in the fields preventing direct import into ArcGIS. A script to replace the characters can use lists to hold the file names and field names. Then the files and fields can be batch processed. Built-in Python data types, such as lists, are useful for solving this kind of GIS problem. This chapter presents the Python list data type, list operations and methods, the range function, mutability, and tuples. The chapter concludes with a debugging walk-through to demonstrate syntax checking and tracebacks.

Chapter Objectives
After reading this chapter, you'll be able to do the following:

- Create Python lists.
- Index into, slice, and concatenate lists.
- Find the length of a list and check if an item is in a list.
- Append items to a list.
- Locate online help for the list methods.
- Create a list of numbers automatically.
- Differentiate between in-place methods and methods that return a value.
- Create and index tuples.
- Check script syntax.
- Interpret traceback error messages.

4.1 Lists

A Python *list* is a data type that holds a collection of items. The items in a list are surrounded by square brackets and separated by commas. The syntax for a list assignment statement looks like this:

```
listVariable = [item1, item2, item3,...]
```


L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_4

Here is an example in which `fields` is a Python list of field names from a comma separated value ('.csv') file containing wildfire data:

```
>>> fields = ['FireId', 'Org', 'Reg-State', 'FireType']
>>> fields
['FireId', 'Org', 'Reg-State', 'FireType']
>>> type(fields)
<type 'list'>
```

All of the items in the `fields` list are strings, but a list can hold items of varying data types. For example, a list can contain numbers, strings and other lists. The `stateData` list assigned here contains a string, a list, and an integer:

```
>>> stateData = ['Florida', ['Alabama', 'Georgia'], 18809888]
```

This list contains both numeric and string items:

```
>>> exampleList = [10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']
```

You can also create an empty list by assigning empty square brackets to a variable, as in this example:

```
>>> dataList = []
```

4.1.1 Sequence Operations on Lists

Lists, like strings are one of the sequence data types in Python. The sequence operations discussed in the context of string data types also apply to lists; the length of a list can be found, lists can be indexed, sliced, and concatenated, and you can check if an item is a member of a list. Table 4.1 shows sample code and output for each of these operations.

Table 4.1 Sequence operations on lists.

`>>> exampleList = [10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']`

Operation	Sample code	Return value
Length	`len(exampleList)`	`7`
Indexing	`exampleList[6]`	`'cusp'`
Slicing	`exampleList[2:4]`	`[1.5, 'b']`
Concatenation	`exampleList + exampleList`	`[10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp', 10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']`
Membership	`'prune' in exampleList`	`False`

The operations work on list items just like they work on string characters; however, lists are mutable. This means that indexing can also be used to *change* the value of an item in the list. Recall that strings are immutable and trying to change an indexed character in a string returns an error. In the following example, a list is created, then on the second line, an indexed list item is modified. The last line in the example shows that the first item in the list has been changed:

```
>>> exampleList = [10000, 'a', 1.5, 'b', 'banana', 'c', 'cusp']
>>> exampleList[0] = 'prune' # modifying the first item in the list.
>>> exampleList
['prune', 'a', 1.5, 'b', 'banana', 'c', 'cusp']
```

4.1.2 List Methods

Also like strings, list objects have a specific set of methods associated with them including `append`, `extend`, `insert`, `remove`, `pop`, `index`, `count`, `sort`, and `reverse`. For a complete description of each, search online for Guido Van Rossum's Python Tutorial for the 'More on lists' section. Like string methods, list methods use the dot notation:

```
object.method(arguments1, argument2, argument3,...)
```

There is one notable difference between list and string methods; it relates to mutability. Many list methods are in-place methods. *In-place methods* are those methods which change the object that calls them. Whereas, other methods do not alter the object that calls them; instead, they return a value. In-place list methods such as `append`, `extend`, `reverse`, and `sort` do not use an assignment statement to propagate the new list. Instead, they modify the existing list. The example below creates a `fireIDs` list and appends a new ID to the end of the list using the `append` method:

```
>>> # Initialize the list with 4 IDs.
>>> fireIDs = ['238998', '239131', '239135', '239400']
>>> newID = '239413'
>>> fireIDs.append(newID) # Changing the list in-place.
>>> fireIDs
['238998', '239131', '239135', '239400', '239413']
>>> # New value was appended to the end of the list.
```

Since `append` is an in-place method, the original list is modified, in-place. In contrast, the list method named 'count' is not an in-place method. It does not alter the

list, but returns a value instead. The following example creates a `fireTypes` list and determines the number of code 11 fires that had occurred, using the `count` method:

```
>>> fireTypes = [16, 13, 16, 6, 17, 16, 6, 11, 11, 12, 14, 13, 11]
>>> countResults = fireTypes.count(11)
>>> print countResults
3 # The list contains three elevens.
>>> fireTypes # Note that the list contents are unchanged:
[16, 13, 16, 6, 17, 16, 6, 11, 11, 12, 14, 13, 11]
```

4.1.3 The Built-in `range` Function

The built-in `range` function is a convenient way to generate numeric lists, which can be useful for batch processing tasks. The `range` function takes one to three numeric arguments and returns a list of numbers. If you pass in one numeric argument, `n`, it returns a Python list containing the integers `0` through `n - 1`, as in the following example:

```
>>> range(9)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

By using a second argument, you can modify the lower bound:

```
>>> range(5,9)
[5, 6, 7, 8]
```

By using a third argument, you can change the step size:

```
>>> range(0,9,2)
[0, 2, 4, 6, 8]
```

The `range` function is used again in a later chapter to create numeric lists for looping.

4.1.4 Copying a List

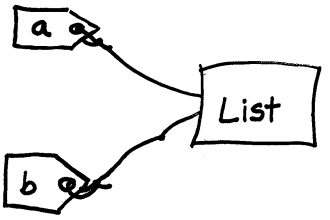
As we've just discussed, in-place methods like `reverse` and `sort` alter the order of the list. For example, here the `reverse` method reverses the order of the `fireIDs` list:

```
>>> fireIDs = ['238998', '239131', '239135', '239400']
>>> fireIDs.reverse()
```

```
>>> fireIDs
['239400', '239135', '239131', '238998']
```

In some cases, it may be necessary to keep a copy of the list in its original order. Python variables can be thought of as tags attached to objects. Because of this, there are two types of copy operations for objects: shallow copies and deep copies. A *shallow copy* attaches two tags to the same list object. A *deep copy* attaches each tag to a separate list object. The assignment statement shown in the following example doesn't work as you might intuitively expect. Python list `a` is copied to list `b` but both the lists are reversed when `a` is reversed. This is a shallow copy:

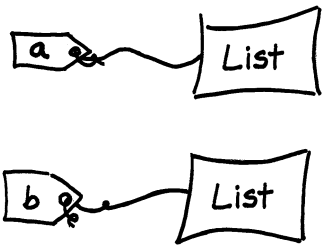
```
>>> a = range(1,11)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> b = a # "shallow copy" list a
>>> b
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a.reverse() # reverse list a
>>> a
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> b
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1] # list b is also reversed
```



Both tags `a` and `b` are pointing to the same object. Instead, we need to create a second object via a deep copy to retain a copy of the list in its original order. To do this we need to create a new list and attach the `b` tag to it. The built-in `list` function constructs a new list object based on a list given as an argument. This built-in function is used to create a deep copy in the next example. `list(a)` constructs a new `list` object and this is assigned to `b`. In the end, `a` is reversed, but `b` retains the original order:

```
>>> a = range(1,11)
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> b = list(a) # "deep copy" list a
>>> b
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a.reverse() # reverse list a
>>> a
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> b
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] # list b is not reversed
```



4.2 Tuples

A *tuple* is another Python data type for holding a collection of items. A tuple is a set of comma separated items, but round brackets (parentheses) are used instead of square brackets. When a tuple is printed, it is surrounded by parentheses. A tuple can be created with or without parentheses around the items:

```
>>> t = "abc", 456, 'wxyz'
>>> type(t)
<type 'tuple'>
>>> t
('abc', 456, 'wxyz')
```

```
>>> t2 =(4.5, 7, 0.3)
>>> type(t2)
<type 'tuple'>
>>> t2
(4.5, 7, 0.3)
```

Like strings, tuple items can be indexed but they are not mutable. It is not possible to assign individual items of a tuple with indexing:

```
>>> t[0]
'abc'
>>> t[1]
456
>>> t2[0]
4.5
>>> t2[0] = 5
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuples are used in situations where immutability is desirable. For example, they are often used for (x, y, z) point coordinates or for holding the set of items in a data record.

4.3 Syntax Check and Tracebacks

Most of the examples in the book until this point have used the Interactive Window to explain the basic Python components. In upcoming chapters, we will begin to write complete scripts. Before we do so, it will be useful to become familiar with syntax checking in PythonWin. Chapter 2 discussed examples of errors, tracebacks, and exceptions thrown by interactive input. Now we'll compare interactive input error handling to script window error handling.

The Interactive Window automatically gives you immediate feedback in the form of a traceback message when you enter code that contains a syntax error, because it evaluates (runs) the code when you enter it. For example, a traceback message is printed, because of the following code, in which the `print` keyword is misspelled as `pirnt`:

```
>>> pirnt test
Traceback (File "<interactive input>", line 1
pirnt test
         ^
SyntaxError: invalid syntax
```

When you enter code in a PythonWin script window, you don't automatically receive immediate feedback as you're writing, but you can trigger syntax checking when you're ready by clicking a button. The PythonWin syntax checker is similar to a word processing spelling checker. The 'check' button, the one with the checkmark icon (), on the PythonWin standard toolbar checks the syntax without executing the code. It finds syntax errors, such as incorrect indentation, misspelling of keywords, and missing punctuation.

The feedback panel in the lower left corner of PythonWin shows the result of the check. If an error is detected, the feedback panel says "Failed to check—syntax error—invalid syntax" as shown in Figure 4.1. The cursor jumps to a position near to the first syntax error it detects in the script, usually the same line or the line following the erroneous line. To try this yourself,

1. Create a new script with one line of code that says the following: pirnt test
2. Initially, the 'Check' button is disabled (grayed out ). You can't check the syntax until the script has been saved at least once. Save the script in 'C:\gispy\scratch' as script 'test.py'.
3. Click in the 'test.py' script window to make sure your window focus is there and not in the Interactive Window.

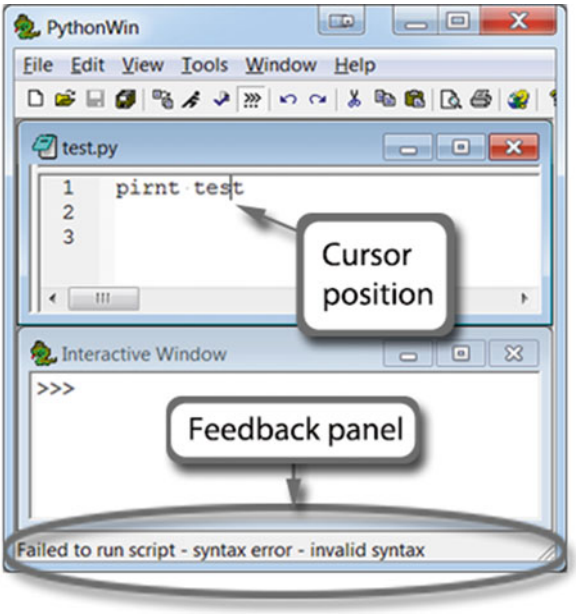


Figure 4.1 Syntax checker finds an error.

4. Now click the 'Check' button  , which became enabled when the script was saved.
5. Observe the feedback panel and the cursor (see Figure 4.1). The feedback panel says `Failed to check—syntax error—invalid syntax` and the cursor has jumped to the erroneous line. It can't detect exactly what's wrong, so it gives this generic message and moves the cursor near to where it detects an error.
6. Next click the 'Run' button  to see if it will run. The feedback shows a syntax error message—"Failed to *run script…*" (Figure 4.1). If you overlook the feedback panel, it may seem as if the program is not responding to the 'Run' button click. Check the feedback panel whenever you run a script.
7. The script will not run until the syntax errors are repaired; Repair the syntax error—change 'pirnt' to 'print'.
8. Click the 'Check' button again.
9. The feedback bar reports success: "Python and TabNanny successfully checked the file". The TabNanny, a script that checks for inconsistent indentation, is run as part of the syntax checking process.
10. A successful syntax check does not necessarily mean that the code is free of errors. Click the 'Run' button now that the syntax has been corrected. The feedback panel

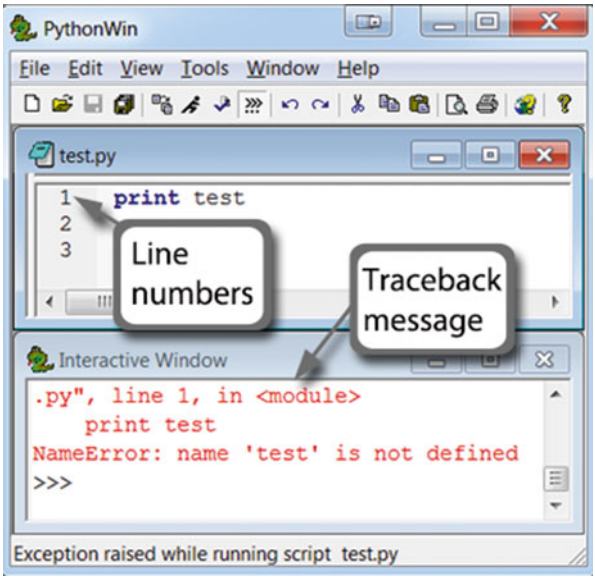


Figure 4.2 Exception raised.

reports that an exception was raised. This time a traceback error is printed in the Interactive Window (Figure 4.2) because the error was only detected when the script ran, not during the syntax checking phase.

The syntax check can only detect errors that violate certain syntax rules. Some errors can only be detected when Python attempts to run the code and then an error message appears in the feedback panel when it reaches a problem. When Python tries to print `test`, it throws an exception and prints a traceback message. A script traceback differs slightly from interactive input traceback because the message traces the origin of the error.

Entering the code from 'test.py' in the Interactive Window prints the following traceback:

```
>>> print test
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
NameError: name 'test' is not defined
```

Whereas, the traceback message in Figure 4.2 thrown by running 'test.py' reads as follows:

```
>>> Traceback (most recent call last):
  File "C:\Python27\ArcGIS10.3\Lib\site-
packages\pythonwin\pywin\framework\scriptutils.py", line 326, in
RunScript
    exec codeObject in __main__.__dict__
  File "C:\gispy\scratch\test.py", line 1, in <module>
    print test
NameError: name 'test' is not defined
```

In many cases, the most pertinent information appears near the end of the traceback. The last line names the exception (`NameError`) and describes the error (name `test` is not defined). The last line of both the interactive traceback and the script window traceback are identical, but the preceding lines indicate the location of the error. In one case, this is `<interactive input>`.

```
  File "<interactive input>", line 1, in <module>
```

In the other case, the line of code is printed beneath the location of the line of the script which caused the error (line 1 of test.py).

```
  File "C:\gispy\scratch\test.py", line 1, in <module>
    print test
```

Python keeps track of a call stack. The call stack contains a list of files (and functions) that are called as Python runs. When 'text.py' is run from PythonWin, there are two items on the call stack, the 'scriptutils.py' and 'test.py' files. The 'scriptutils.py' file is called by PythonWin when you select 'Run'. This script, in turn, prompts the 'test.py' script to run. This is why the traceback lists 'scriptutils.py' and 'test.py' in this order. Can you tell which line of 'scriptutils.py' triggered 'test.py' to run? These lines show the line of code that was triggered in scriptutils.py:

```
  File "C:\Python27\ArcGIS10.3\Lib\site-
packages\pythonwin\pywin\framework\scriptutils.py", line 326, in
RunScript
    exec codeObject in __main__.__dict__
```

We can ignore this part of the traceback. The only part of the stack that is interesting for our purposes is the line of code which we wrote that caused the error. This is why we focus mainly on the last few lines of the message. Traceback messages print the script line number where the error occurred. The examples in the upcoming

chapters of this book usually report only a portion of tracebacks. Turn on IDE line numbers to help with traceback interpretation.

To show line numbers in PythonWin:
Tools > Options…> Editor tab > Set "Line Numbers" Margin Width >= 30 pixels.

To show line numbers in PyScripter:
Tools > Options > Editor Options > Display > Gutter: check 'Visible' and 'Show line numbers'.

These examples show that the Interactive Window prints a traceback message immediately when you enter erroneous code because the code is evaluated immediately; whereas, script window code is only checked when you use the syntax check button or try to run the code. If a syntax error is found in a script, the feedback bar reports an error and the cursor moves near to the first detected error. Some errors are not detectable by the syntax check. These errors stop the script from running and print traceback messages in the Interactive Window. The traceback, read from bottom to top, names the exception, describes the error, and prints the erroneous line and the line number in the script where it was found. In summary, checking syntax, showing line numbers, and reading tracebacks will help you build working scripts as you progress through the upcoming chapters.

Note Always check the feedback panel and the Interactive Window when you run a script in PythonWin.

4.4 Key Terms

`List` data type in-place method
Mutable vs. immutable
Shallow copy vs. deep copy
Built-in `list` function
Sequence operations (length, indexing, concatenation, slicing, membership)
`Range` built-in function
List methods
`tuple` data type

4.5 Exercises

1. Practice interpreting exceptions and diagnosing errors as you follow the given steps. Then answer the follow-up questions.
   (a) Locate 'addWithErrors.py' in the Chapter 2 sample_scripts directory. Open it in an IDE and save it as 'addWithOutErrors.py'. This script is like 'add_version2.py' from Chapter 2, but several errors have been introduced.
   (b) Run the syntax check.
   The feedback panel says: "Failed to check—syntax error—invalid syntax" and the cursor jumps to line 7. In the steps below, you'll see that there are several errors, but the code is evaluated from top to bottom, so the cursor jumps to the first error Python detects. Do you see a problem on that line?
   (c) The keyword 'import' is misspelled. Replace 'iport' with 'import'.
   (d) Run the syntax check.
   The feedback panel says: "Failed to check - syntax error - invalid syntax" and now the cursor jumps to line 13. (Now line 13 is the first place an error is detected).
   (e) But there's nothing wrong with line 13! Can you find the error on the previous line? The closing parenthesis is missing on line 12. The cursor jumps to line 13 because Python detected an assignment statement before the parentheses were closed. Until it reached line 13, it didn't detect that there was anything wrong. Add ')' to the end of line 12. So that it looks like this:

```
b = int(sys.argv[2])
```

   (f) Run the syntax check.
   The feedback panel says: "Failed to check—syntax error—invalid syntax" and the cursor jumps to line 14. Do you see the syntax error on line 14?
   (g) Line 14 is missing a quotation mark on the string. Add a quotation mark (") after the `print` keyword as follows (and note the changes in the color of the text): `"The sum is {0}.".format(c)`
   (h) Run the syntax check.
   The feedback panel says: "Python and TabNanny successfully checked the file". All the detectable syntax errors have been corrected, so the script can be run.
   (i) Run the script with no arguments by clicking the 'Run' button. The following traceback message is printed:

```
  File "C:\gispy\scratch\addWithOutErrors.py", line 10, in
<module>
    a = sys.argv[1]
IndexError: list index out of range
```

   (j) The traceback message says an `IndexError` occurred on line 10 of 'addWithOutErrors.py'. In this case, the traceback is generated because we

didn't provide any user arguments. Index 1 is out of range, because no script arguments were used.

(k) Run the code with arguments 3 and 8. The following traceback message is printed:

```
  File "C:\gispy\scratch\addWithOutErrors.py", line 13, in
<module>
c = a + b
    TypeError: cannot concatenate 'str' and 'int' objects
```

(l) The traceback message says a `TypeError` occurred on line 13 of the script. It says it cannot concatenate a string object with an integer object. When you get a type error, it's useful to check the type of the variables, using the built-in type function, to see if they are the data type that you expect them to be. Check the data types of 'a' and 'b' in the Interactive Window using the built-in `type` function:

```
>>> type(a)
<type 'str'>
>>> type(b)
<type 'int'>
```

(m) `a` is a string because arguments are strings unless they are cast to another type. `b` is cast to integer. Cast `a` to integer too by changing line 10 to look like this:

```
a = int(sys.argv[1])
```

(n) Run the script again. The feedback bar says "Script File 'C:\gispy\scratch\addWithoutErrors.py' returned exit code 0" and the following prints in the Interactive Window.

```
>>> The sum is 11.
```

'Exit code 0' is good news. It means the script ran to completion without throwing any exceptions.

Follow-up questions:

(i) What does the cursor do when you use the syntax checker and a syntax error occurs? Specifically, where does it go?

(ii) How does PythonWin display a quotation mark that's missing its partner (the mark on the other end of the string literal)?

(iii) When you use your mouse to select a parenthesis that has a matching partner, how does PythonWin display the parentheses?

(iv) When you use your mouse to select a parenthesis that does not have a matching partner, how does PythonWin display the parenthesis?

2. The Python statements on the left use operations involving the list type variable `places`. Match the Python statement with its output. All answers MUST BE one of the letters A through I. If you think the answer is not there, you're on the wrong track! The list variable named `places` is assigned as follows:

```
places = ['roads', 'cities', 'states']
```

Python statement	**Output** (notice there are nine letters)
1. `places[0]`	A. `3`
2. `places[0:2] + places[-3:]`	B. `'roads'`
3. `places[0][0]`	C. `IndexError`
4. `len(places)`	D. `['roads', 'cities']`
5. `places[0:5]`	E. `['roads', 'cities', 'states']`
6. `places[-4:]`	F. `['roads', 'cities', 'roads', 'cities', 'states']`
7. `places[11]`	G. `False`
8. `places[:2]`	H. `'r'`
9. `'towns' in places`	I. `'cities'`
10. `places[1]`	

3. The Python statements on the left use list methods and operations (and one string method) involving the list variable, `census`. Match the Python statement with the resulting value of the `census` variable. Consider these as individual statements, not cumulative. In other words, reset the value of `census` to its original value between each command. All answers MUST BE one of the letters A through G. **Notice that the question asks for matching the Python statement with the resulting value of the census variable and not matching the function with the value it returns. If you think the answer is not there, you're on the wrong track!**

   The list variable called `census` is assigned as follows:

```
census = ['4', '3', '79', '1', '66', '9', '1']
```

Python statement	**Resulting value of census** (there are only 7 letters)
1. `len(census)`	A. `[2,'4','3','79','1','66','9','1']`
2. `census.insert(0, 2)`	B. `['1','1','3','4','66','79','9']`
3. `census.append(2)`	C. `['4','3','79','66','9','1']`
4. `census.remove('1')`	D. `['4','3','79','1','66','9','1']`
5. `census = '0'.join(census)`	E. `'403079010660901'`
6. `census.pop(3)`	F. `['1','9','66','1','79','3','4']`
7. `census.count('1')`	G. `['4','3','79','1','66','9','1',2]`
8. `census.sort()`	
9. `census.reverse()`	

4. Answer these three questions related to the difference between in-place methods and methods that return a value.

   (a) What is the value of the variable `census` after all of these statements are executed?

```
>>> census = [4, 3, 79, 1, 66, 9, 1]
>>> census.sort()
>>> census.pop()
>>> census.reverse()
```

```
(1) [4, 3, 79, 1, 66, 9, 1]
(2) [79, 66, 9, 4, 3, 1, 1]
(3) [79, 66, 9, 4, 3, 1]
(4) [66, 9, 4, 3, 1, 1]
```

   (b) What is the value of the variable `happySheep` after ALL of these statements are executed? Hint: if you think the correct answer is not there, you're on the wrong track.

```
>>> happySheep = 'knoll.shp'
>>> happySheep.replace('ll', 'ck')
>>> happySheep.upper()
>>> happySheep[:4]
```

```
(1) 'Knock.shp'
(2) 'knoll.shp'
(3) 'KNO'
(4) 'KNOCK.SHP'
```

(c) Use an example from parts a and b to explain what an in-place method does, as opposed to a method that only returns a value.

5. Select the line or lines of code that set `listA` to `[0, 10, 20, 30, 40]`.

(a) `listA = range(10, 50, 10)`
(b) `listA = range(0, 40, 10)`
(c) `listA = range(0, 50, 10)`
(d) `listA = range(0, 40)`
(e) `listA = range(40, 0, -10)`
(f) `listA = range(0, 41, 10)`

6. The Northeastern region fire data is a text file containing a set of fields followed by records containing entries for those fields. The following lines of code use Python lists to store the field names and the first data record:

```
>>> fields = ['FireId', 'Org', 'Reg-State', 'FireType']
>>> record = ['238998', 'NPS', 'Northeast', '11']
```

Select the line(s) of code that use indexing to print only the value of the Reg-State field in the given record.

(a) `print fields[3]`
(b) `print record[1:3]`
(c) `print record[2]`
(d) `print fields + records`
(e) `print record[-2]`

7. The following code uses list methods and operations to find the minimum value of a list. Write a line of code that does the same thing but uses a Python built-in function (with no dot notation).

```
>>> fireTypes = [8, 4, 2, 5, 7]
>>> fireTypes.sort() # Sort the numeric list
>>> print fireTypes[0] # Use indexing to print the minimum
2
```

8. The following Interactive Window code samples use square braces in three different ways. Use chapter key terms from Section 4.4 to describe what each code statement is doing.

(a) `>>> theList = ['spam', 'eggs', 'sausage', 'bacon', 'spam']`
(b) `>>> theList[1]`
`'eggs'`

(c)
```
>>> theList[:3]
['spam', 'eggs', 'sausage']
```

9. Answer the following questions about the traceback message shown below:

   (a) What is the name of the exception?
   (b) What is the name of the script containing the error?
   (c) Which line of the script contains the error?
   (d) Explain why this error might have occurred.

```
Traceback (most recent call last):
  File "C:\Python27\ArcGIS10.2\Lib\site-
packages\pythonwin\pywin\framework\scriptutils.py",
line 326, in RunScript
    exec codeObject in __main__.__dict__
  File "C:\gispy\sample_scripts\ch04\County.py",
    line 23, in <module> District = sys.argv[5]
IndexError: list index out of range
```

10. **noMoreErrors.py** The sample script named 'noMoreErrors.py' currently contains five errors. Modify the script to remove all errors. As you remove errors, note whether the error resulted in a feedback bar message or a traceback message and note the message. Modify the last five lines of the script to report the error messages you encountered. The first two of these have been completed for you already. When the script is repaired, the first portion of the output will look like this:

```
NoMoreErrors.py
b
Mucca.gdb
Dyer20
DYER
#1. FEEDBACK BAR: Failed to check - syntax error - EOL while
   scanning
string literal
#2. TRACEBACK: IndexError: string index out of range
```

Error messages for #3–5 will also be printed, when the script is complete, but they are omitted here.