Chapter 3
Basic Data Types: Numbers and Strings

Abstract All Python objects having a data type. Built-in Python data types, such as integers, floating point values, and strings are the building blocks for GIS scripts. This chapter uses GIS examples to discuss Python numeric data types, mathematical operators, string data types, and string operations and methods.

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

- Perform mathematical operations on numeric data types.
- Differentiate between integer and floating point number division.
- Determine the data type of a variable.
- Index into, slice, and concatenate strings.
- Find the length of a string and check if a substring is contained in a string.
- Replace substrings, modify text case in strings, split strings, and join items into a single string.
- Differentiate between string variables and string literals.
- Locate online help for the specialized functions associated with strings.
- Create strings that represent the location of data.
- Format strings and numbers for printing.

3.1 Numbers

Python has four numeric data types: int, long, float, and complex. Examples in this book mainly use `int` (signed integers) and `float` (floating point numbers). With dynamic typing, the variable type is decided by the assignment statement which gives it a value. Float values have a decimal point; Integer values don't. In the following example, `x` is an integer and `y` is a float:

```
>>> x = 2
>>> y = 2.0
>>> type(x)
```

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_3

Table 3.1 Numerical operators.

Operation	Operator	Example	Result
Addition	+	7+2	9
Subtraction	-	7-2	5
Multiplication	*	7*2	14
Division	/	7 / 2	3
Exponentiation	**	7**2	9
Modulus division	%	7 % 2	1

```
<type 'int'>
>>> type(y)
<type 'float'>
```

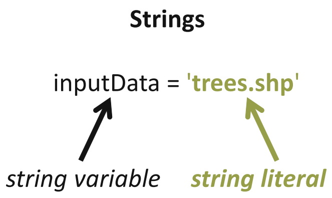
By default, PythonWin uses cyan-blue to display numeric values (as well as output in the Interactive Window). Table 3.1 shows the operators symbols, +, −, *, **, and / for addition, subtraction, multiplication, exponentiation, and division. Mathematical order of operations is preserved. `2 + 3 * 4` gives 14 not 20, though `2 + (3 * 4)` is a clearer way to write it. Be aware that Python integer division and floating point division give different results. If both division operands are integers, the result will be *integer division* with the fractional part truncated; in other words, it always rounds down. If at least one of the operands are float, the result retains higher precision.

```
>>> 8/3
2
>>> 8.0/3
2.6666666666666665
```

In Python versions 3.0 and higher, this behavior is changed so that 8/3 gives the same result as 8.0/3.

3.2 What Is a String?

By default, PythonWin uses olive-green text to display string literals. A *string literal* is a set of characters surrounded by quotation marks. A variable assigned a string literal value is called a *string variable*. The difference between these two terms is important, but both of these items are sometimes referred to simply as 'strings'. The characters inside the quotes in the string literal are interpreted literally. In GIS scripts, we use string literals for things like the name of a workspace, input file, or output file. Meanwhile, the characters that make up a string variable are simply a name for the program to use to represent its value. Variable names should not be surrounded by quotes. The operations and methods described in the next section apply to both string literals and string variables. Both are objects of data type `str` (for string).



The type of quotes used to create a Python string literal is flexible, as long as the opening and closing quotes match. Here are two variations:

```
>>> inputData = 'trees.shp'
>>> inputData
'trees.shp'
>>> inputData = "trees.shp"
>>> inputData
'trees.shp'
```

There is a subtle difference between printing strings with and without the `print` function in the Interactive Window; the former removes the quotation marks entirely and the latter uses single quotation marks.

```
>>> print inputData
trees.shp
>>> inputData
'trees.shp'
```

String literals in single or double quotes cannot span more than one line. A string literal with no closing quote raises a `SyntaxError` as shown in the traceback message:

```
>>> output = 'a b c
Traceback ( File "<interactive input>", line 1
output = "a b c
              ^
SyntaxError: EOL while scanning string literal
```

Triple quotes can be used to create a string that spans more than one line:

```
>>> output = """a b c
... d e f"""

>>> print output
a b c
d e f
```

The triple quotes store the carriage return as part of the string. In some cases we don't want a carriage return stored in the string. For example, we might need to store a long file path name. The path won't be interpreted correctly if a carriage return is embedded in it. But very wide lines of Python code are awkward to read. As a rule of thumb, Python lines of code should be less than 100 characters wide, so the reader doesn't need to scroll right. A *line continuation character* (`'\'`), a backslash embedded in a string at the end of a line, allows a string to be written on more than one line, while preserving the single line spacing in the string literal value. In other words, though it is assigned on multiple lines, it will be printed as a single line variable. Place the character at the end of the line and the string literal can be continued on the next line, but the string value will not contain the carriage return:

```
>>> output = 'a b c \
d e f \
g h i'

>>> print output
a b c d e f g h i
```

Numerical characters surrounded by quotation marks are considered to be strings literals by Python.

```
>>> FID = 145
>>> type(FID)
<type 'int'>
>>> countyNum = '145'
>>> type(countyNum)
<type 'str'>
```

If the variable is a string type, you can perform string operations on it, as described in the next section.

3.3 String Operations

GIS Python programming requires frequent string manipulation. To deal with file names, field names, and so forth, you'll need to be familiar with finding the length of a string, indexing into a string, concatenating, slicing, and checking for a substring in a string. Examples of each operation follow.

3.3.1 Find the Length of Strings

The built-in `len` function finds the length of a string literal:

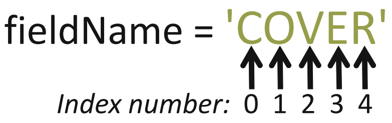
```
>>> len('trees.shp')
9
```

Or the length of the value held by a string variable:

```
>>> data = 'trees.shp'
>>> len(data)
9
```

3.3.2 *Indexing into Strings*

Each character in a string has a numbered position called an *index*. The numbering starts with zero, in other words Python uses *zero-based indexing*. From left to right, the indices are 0, 1, 2, and so forth.



Indexing into a string means pointing to an individual character in a string using its index number. To index into a string variable, use square brackets after the variable name and place an index number inside the brackets. The general format for indexing looks like this:

```
variableName[index_number]
```

This example assigns a value to a string variable and then indexes the first character in the string value:

```
>>> fieldName = 'COVER'
>>> fieldName[0]
'C'
```

Since indexing is zero-based, the last valid index is the length of the string minus one. Attempting to using an invalid index number results in an `IndexError`:

```
>>> len(fieldName)
5
>>> fieldName[5]
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
IndexError: string index out of range
>>> fieldName[4]
'R'
```

Negative indices count one-based from the right. This can be useful for getting the last character without checking the string length.

```
>>> fieldName[-1]
'R'
```

It is not possible to change the value of an individual character of a string with indexing. Attempting to do so results in a `TypeError`, as shown in the following code:

```
>>> fieldName[0] = 'D'
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

This occurs because Python strings are *immutable*. The word ‘immutable’ is a synonym for unchangeable. Of course, we can change the value of a string variable by assigning an entirely new string to it, as in the example below, but in this context, immutability refers to this specific quality of not being able to change individual parts of an existing object. To change the value of `fieldName` from `COVER` to `DOVER`, you need to use a string literal:

```
>>> fieldName = 'DOVER'
```

Or, you can use the string `replace` method discussed in Section 3.4:

```
>>> fieldName = fieldName.replace('C','D')
```

3.3.3 Slice Strings

Indexing can be thought of as getting a substring which is only one character long. To get more than one character, use slicing instead of indexing. Slicing gets a substring (a slice) of a string. Slicing uses square brackets with a colon inside. A number on either side of the colon indicates the starting and ending index of the desired slice. The left number is inclusive, the right is exclusive.

```
>>> fieldName = 'COVER'
>>> fieldName[1:3]
'OV'
```

The letter `O` is the index 1 character in the string `'COVER'` and the letter `E` is the index 3 character, so the slice starts at the letter `O` and ends just before the letter `E`.

If the left slice number is omitted, the slice starts at the beginning of the word. If the right slice number is omitted, the slice ends at the end of the word.

```
>>> fieldName[:3]
'COV'
>>> fieldName[1:]
'OVER'
```

We often want to use the *base name* of a file, the part without the extension, to build another file name, so that we can create output names based on input names. Counting from the right with a negative index can be used to find the base name of a file which has a three digit file extension. Omitting the left index starts from the beginning of the string. Using `-4` as the right index removes the last four characters—the three digit file extension and the period—leaving the base name of the file.

```
>>> inputData = 'trees.shp'
>>> baseName = inputData[:-4] # Remove the file extension.
>>> baseName
'trees'
```

This approach assumes you know the file extension length. A more general solution using the os module is discussed in Chapter 7.

3.3.4 Concatenate Strings

Concatenation glues together a pair of strings. You use the same sign for addition, but it acts differently for strings. The plus sign performs addition on numeric values and concatenation on strings.

```
>>> 5 + 6 # adding two numbers together
11
>>> '5' + '6' # concatenating two strings
'56'

>>> rasterName = 'NorthEast'
>>> route = 'ATrain'
>>> output = rasterName + route
>>> output
'NorthEastATrain'br />
```

Both of the variables being concatenated must be string types or you'll get a `TypeError`. For example, using the plus sign between a numeric variable and a string variable causes an error. Python doesn't know whether to perform addition or concatenation.

```
>>> i = 1
>>> rasterName = 'NorthEast'
>>> output = rasterName + i
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

The `TypeError` says that a string cannot be concatenated with an integer object. Solve this problem by using the built-in `str` function which returns a string data type version of an input value. Type conversion is referred to as *casting*. To combine a string with a numeric value, cast the number to a string. Then the number is treated as a string data type for the concatenation operation:

```
>>> i = 145
>>> str(i)
'145'
>>> rasterName = 'NorthEast'
>>> output = rasterName + str(i)
>>> output
'NorthEast145'
```

We often use concatenation with slicing to create an output name based on an input file name.

```
>>> inputData = 'trees.shp'
>>> baseName = inputData[:-4]
>>> baseName
'trees'
>>> outputData = baseName + '_buffer.shp'
>>> outputData
'trees_buffer.shp'
```

3.3.5 Check for Substring Membership

The `in` keyword enables you to check if a string contains a substring. Python documentation refers to this as checking for 'membership' in a string. Suppose you want to check if a file is a buffer output or not, and you have named each buffer output file so that the name contains the string `buff`.

```
>>> substring = 'buff'
>>> substring in outputData
True
>>> substring in inputData
False
```

Table 3.2 Sequence operations on strings.

```
>>> exampleString = 'tuzigoot'
```

Operation	**Sample code**	**Return value**
Length	`len(exampleString)`	`8`
Indexing	`exampleString[2]`	`'z'`
Slicing	`exampleString[:-4]`	`'tuzi'`
Concatenation	`exampleString+exampleString`	`'tuzigoottuzigoot'`
Membership	`'ample' in exampleString`	`False`

These string operations are powerful when combined with batch processing (discussed in Chapters 10 and 11). Table 3.2 summarizes these common string operations. As we'll discuss shortly, they can be applied to other data types as well. Strings and other data types that have a collection of items are referred to as *sequence* data types. The characters in the string are the individual items in the sequence. The operations in Table 3.2 can applied to any of the sequence data types in Python.

3.4 More Things with Strings (a.k.a. String Methods)

Along with the operations described above, processing GIS data often requires additional string manipulation. For example, you may need to replace the special characters in a field name, you may need to change a file name to all lower case, or you may need to check the ending of a file name. For operations of this sort, Python has built-in string functions called string methods. *String methods* are functions associated particularly with strings; they perform actions on strings. Calling a string method has a similar format to calling a built-in function, but in calling a string method, you also have to specify the string object—using dot notation. *Dot notation* for calling a method uses both the object name and the method name, separated by a dot. The general format for dot notation looks like this:

```
object.method(argument1, argument2, argument3,...)
```

'Object' and 'method' are object-oriented programming (OOP) terms and 'dot notation' is an OOP specialized syntax.

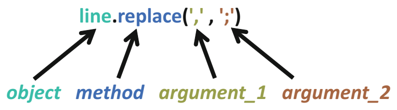
- Everything in Python is an *object*. For example, numbers, strings, functions, constants, and exceptions are all object*s*. Other programming languages use this term more narrowly to refer to data types which have associated functions and attributes. In Python, most objects have accompanying functions and attributes, which are also referred to as methods and properties. As soon as a variable is assigned a value, it is a string object which has string methods.
- A *method i*s a function that performs some action on the object. Methods are simply functions that are referred to as 'methods' because they are performed

on an object. The terms ‘calling methods’, ‘passing arguments’, and ‘returning values’ apply to methods in the same way they apply to functions. The example below calls the `replace` method. The variable, `line`, is assigned a string value. Then the dot notation is used to call the string method named `replace`. This example returns the string with commas replaced by semicolons:

```
>>> line = '238998,NPS,NERO,Northeast'
>>> line.replace(',' , ';')
'238998;NPS;NERO;Northeast'
```

- *Dot notation* is an object-oriented programming convention used on string objects (and other types of objects) to access methods and properties, that are specially designed to work with those objects.

The image below points out the components in the `replace` method example. The dot notation links the object method to the object. The object is the variable named `line`. The method (`replace`) takes two string arguments. These arguments specify the string to replace (a comma) and the replacement (a semicolon).

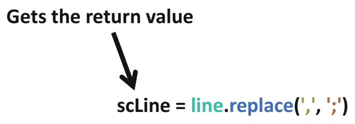


String methods need to be used in assignment statements. Strings are immutable, so string methods do not change the value of the string object they are called on; instead, they return the modified value.

For example, the `replace` call does not alter the value of the variable, `line`:

```
>>> line = '238998,NPS,NERO,Northeast'
>>> line.replace(',' , ';')
>>> line
'238998,NPS,NERO,Northeast'
```

An assignment statement can be used to store the return value of the method. Call the `replace` method on the right-hand side of an assignment statement like this:



```
>>> line = '238998,NPS,NERO,Northeast'
>>> semicolonLine = line.replace(',' , ';')
>>> semicolonLine
'238998;NPS;NERO;Northeast'
```

To alter the original variable called `line`, use it as the variable being assigned. In other words, you can put it on both sides of the equals sign:

```
>>> line = line.replace(',' , ';')
>>> line
'238998;NPS;NERO;Northeast'
```

Not all methods require arguments; However, even when you don't pass any arguments, you must use the parentheses. For example, to change the letters of a string to all uppercase letters, use the upper method like this:

```
>>> name = 'Delaware Water Gap'
>>> name = name.upper()
>>> print name
DELAWARE WATER GAP
```

The `split` and `join` methods are used in many GIS scripts. These methods involve another Python data type called a `list` (Python lists, the main topic of Chapter 4, are containers for holding sets of items). The `split` method returns a list of the words in the string, separated by the argument. In the example below, the `split` method looks for each occurrence of the forward slash (/) in the string object and splits the string in those positions. The resulting list has five items, since the string has four slashes.

```
>>> path = 'C:/gispy/data/ch03/xy1.txt'
>>> path.split('/')
['C:', 'gispy', 'data', 'ch03', 'xy1.txt']
```

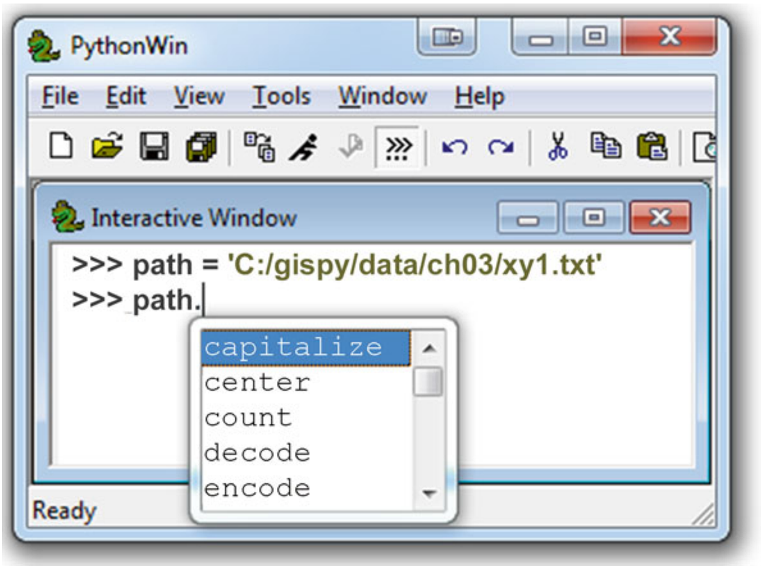
The `split` and `join` methods have inverse functionality. The `split` method takes a single string and splits it into a list of strings, based on some delimiter. The `join` method takes a list of strings and joins them with into a single string. The string object value is placed between the items. For example, `elephant` is placed between `1`, `2`, and `3` here:

```
>>> numList = ['1', '2', '3']
>>> animal = 'elephant'
>>> animal.join(numList)
'1elephant2elephant3'
```

In the following example, the `join` method is performed on a string literal object, semicolon (`;`). The method inserts semicolons between the items and returns the resulting string:

```
>>> pathList = ['C:', 'gispy', 'data', 'ch03', 'xy1.txt']
>>> ';'.join(pathList)
'C:;gispy;data;ch03;xy1.txt'
```

IDEs make it easy to browse for methods of an object by bringing up a list of choices when you type the object name followed by a dot. For example, if you create a string variable named `path` and then type `path` in the PythonWin Interactive Window, a *context menu* appears with a list of string methods. This menu of choices is referred to as a context menu because the menu choices update dynamically based on the context in which it is generated. You can scroll and select a method by clicking on your choice.



```
>>> path = 'C:/gispy/data/ch03/xy1.txt'
>>> path.
```

In PythonWin, context menus appear in the Interactive Window or after the code in a script has been executed (after the script has been run). In PyScripter, the context menus appear as soon as a variable has been defined within a script. If your choice doesn't appear in the context menu, check for spelling errors. Try using the context menu to bring up the `endswith` method for a string variable in the PythonWin Interactive Window. As you start typing the method name, the list box rolls to that name and you can use the 'Tab' key to complete the selection. The `endswith` method checks if the ending matches the argument and returns `True` or `False`:

```
>>> path = 'C:/gispy/data/ch03/xy1.txt'
>>> path.endswith('.shp')
False
>>> path.endswith('.txt')
True
```

The examples here demonstrated a few of the string methods. In fact there are many more, including `capitalize`, `center`, `count`, `decode`, `encode`, `endswith`, `expandtabs`, find, `index`, `isalnum`, `isalpha`, `isdigit`, `islower`, `isspace`, `istitle`, `isupper`, `join`, `ljust`, `lower`, `lstrip`, `partition`, `replace`, `rfind`, `rindex`, `rjust`, `rpartition`, `rsplit`, `restrip`, `split`, `splitlines`, `startswith`, `strip`, `swapcase`, `title`, `translate`, `upper`, and `zfill`. Help documentation and a comprehensive list of string methods is available online (search for 'Python String Methods'). String method names are often intuitive. Testing them in the Interactive Window helps to clarify their functionality.

3.5 File Paths and Raw Strings

When dealing with file paths, you will encounter strings literals containing escape sequences. *Escape sequences* are sequences of characters that have special meaning. In string literals, the backslash (\) is used as an *escape character* to encode special characters. The backslash acts as a line continuation character when placed at the end of a line in a string literal as described in Section 3.2. But when a backslash is followed immediately by a character in a string literal, the backslash along with the character that follows it are called an *escape sequence* and the backslash is interpreted as a signal that the next character is to be given a special interpretation. For example, the string literals `'\n'` and `'\t'` are escape sequences that encode 'new line' and 'tab'. New line and tab characters are used to control the space around the visible characters. They are referred to as *whitespace characters*, because the characters themselves are not visible when printed.

White space escape sequences	
`\n`	new line
`\t`	tab

When printed without the built-in `print` function in the Interactive Window, whitespace escape sequences are printed as entered:

```
>>> 'X\tY\tValue\n\n16\t255\t6.3'
'X\tY\tValue\n\n16\t255\t6.3'
```

When printed with the built-in `print` function, whitespace escape sequences are interpreted to modify the formatting:

```
>>> print 'X\tY\tValue\n\n16\t255\t6.3'
X     Y     Value

16    255    6.3
```

The `strip` method can be used to strip unwanted leading or trailing whitespace from a string. This is often called for when processing files. The following example prints a string before and after using the `strip` method:

```
>>> dataRecord = ' \n\t\tX\tY\tZ\tM\t'
>>> print dataRecord

                X     Y     Z     M
>>> dataRecord = dataRecord.strip()
>>> dataRecord
'X\tY\tZ\tM'
>>> print dataRecord
X     Y     Z     M
>>>
```

Escape sequences can lead to unintended consequences with file paths that contain backslashes. In this example, the `t` in terrain and the `n` in `neuse` are replaced by whitespace when the string is printed:

```
>>> dataPath = 'C:\terrain\neuse_river'
>>> dataPath
'C:\terrain\neuse_river'
>>> print dataPath
C:      errain
euse_river
```

Here are three options for avoiding this problem:

1. Use a forward slash instead of a backward slash. The forward slash is not an escape character, but is interpreted correctly as a separator in file paths. This book uses forward slashes (southwest to northeast) instead of backward slashes, as in this example:

   ```
   >>> dataPath = 'C:/terrain/neuse_river'
   >>> print dataPath
   C:/terrain/neuse_river
   ```

2. Double the backslashes. This works because the first slash is an escape character that tells the code to interpret the second slash literally as a slash.

   ```
   >>> dataPath = 'C:\\terrain\\neuse_river'
   >>> print dataPath
   C:\terrain\neuse_river
   ```

3. Use *raw strings*. When you first come across a string literal preceded by a lowercase `r`, you might guess that it's a typo. But in fact, placing an `r` just before

a string literal creates a raw string. Python uses the raw value of a raw string. In other words, it disregards escape sequences, as in the following example:

```
>>> dataPath = r'C:\terrain\neuse_river'
>>> print dataPath
C:\terrain\neuse_river
```

3.6 Unicode Strings

When you start using ArcGIS functionality in Chapter 6, you will begin to see a lowercase `u` preceding strings that are returned by GIS methods. The `u` stands for unicode string. A *unicode string* is a specially encoded string that can represent thousands of characters, so that non-English characters, such as the Hindi alphabet can be represented. A unicode string is created by prepending a `u` to a string literal, as shown here:

```
>>> dataFile = u'counties.shp'
>>> dataFile
u'counties.shp'
>>> type(dataFile)
<type 'unicode'>
```

The difference between `'str'` and `'unicode'` string data types in Python lies in the way that the strings are encoded. The default encoding for Python ‘str’ strings is based on the American Standard Code for Information Interchange (ASCII). Because ASCII encodings were designed to encode English language characters, they can only represent hundreds of characters; whereas, the more recently developed unicode technique can encode thousands. Because of its capability to encode non-English languages, software programs, including the ArcGIS Python interface, have begun to use unicode encodings more often.

You don’t need to know exactly how unicode or ASCII strings encoding works. You just need to know that in your GIS scripts, you can handle `'unicode'` strings just like `'str'` strings. They have the same string operations and methods. The following examples demonstrate using a few string methods and operations on the unicode variable `dataFile`:

```
>>> dataFile.endswith('.shp') # Does the string end with '.shp'?
True
>>> dataFile.startswith('co') # Does the string start with 'co'?
True
>>> dataFile.count('s') # How many times does 's' occur in the string?
2
```

The output, from methods and operations that return strings, is unicode when the input object is unicode:

```
>>> dataFile.upper() # Return an all caps. string.
u'COUNTIES.SHP'
>>> dataFile[5] # Index the 6th character in the string.
u'i'
>>> dataFile + dataFile # Concatenate two strings.
u'counties.shpcounties.shp'
```

Just as the quotation marks are not printed by the built-in `print` function, the unicode `u` is not printed when you use the built-in `print` function to print a unicode string:

```
>>> print dataFile
counties.shp
```

3.7 Printing Strings and Numbers

The built-in `print` function is used frequently in scripting, so we'll show a few examples of how it can be used. As mentioned earlier, the `print` function does not use parentheses around the arguments (though this changes in Python 3.0 in which the parentheses become required). The arguments are the expressions to be printed. We often want to print multiple expressions within the same print statement and these expressions need to be linked together so that the print statement uses them all. Here we demonstrate three approaches to linking expressions to be printed:

Commas. When commas are placed between variables in a `print` expression, the variable values are printed separated by a space. The print statement inserts a space where each comma occurs between printed items:

```
>>> dataFile = 'counties.shp'
>>> FID = 862
>>> print dataFile, FID
counties.shp 862
```

The expression can be a combination of comma separated string literals, numbers, and variables of assorted data types:

```
>>> print 'The first FID in', dataFile, 'is', FID, '!'
The first FID in counties.shp is 862 !
```

Concatenation. The spacing created by the commas may be undesirable in some situations. For example, we usually don't want a space before punctuation.

Concatenation can be used to avoid this problem, though concatenation introduces its own complications. Recall that concatenation uses a plus sign to join strings. If we replace each comma in the above expression with a plus sign, we get a `TypeError` because one piece of the expression is not a string:

```
>>> print 'The first FID in' + dataFile + 'is' + FID + '!'
Traceback (most recent call last):
File "<interactive input>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

`FID` is an integer type, so it must be cast to string. This works, but the spacing isn't correct since the plus signs are not replaced by spaces:

```
>>> print 'The first FID in' + dataFile + 'is' + str(FID) + '!'
The first FID incounties.shpis862!
```

When using concatenation statements to print strings, you may have to tailor the spacing, by adding spaces in the string literals:

```
>>> print 'The first FID in ' + dataFile + ' is ' + str(FID) + '!'
The first FID in counties.shp is 862!
```

String formatting. The string `format` method provides an alternative that uses place-holders to allow you to lay out a string with the spacing you want and handle different variable types without casting. The `format` method is performed on a string literal object, which contains place-holders for variables and gets the variables as a list of arguments. Place-holders are inserted into string literals as numbers within curly brackets (`{0}`, `{1}`, `{2}`... and so forth). The numbers refer to the zero-based index of the arguments in the order they are listed. The string values of arguments are substituted for the place-holders. The method returns a string with these substitutions. This example calls the `format` method in a print statement:

```
>>> print 'The first FID in {0} is {1}!'.format(dataFile, FID)
The first FID in counties.shp is 862!
```

The `format` method uses the dot notation. Here's a detailed breakdown of our example:

- `'The first FID in {0} is {1}!'` is the object.
- `format` is the method.
- `dataFile` and `FID` are the arguments. The string value of `dataFile` is substituted for `{0}`. The string value of `FID` is substituted for `{1}` because dataFile appears first in the argument list and FID appears second.
- The return value is printed.

Triple quotes can be used in combination with the `format` method to create multi-line strings:

```
>>> print '''X     Y     Value
... -------------------
... {0}     {1}     {2}'''.format(16, 255, 6.3)
X     Y     Value
-------------------
16    255    6.3
```

3.8 Key Terms

`int` data type	Casting
`float` data type	The `in` keyword
Integer division	Dot notation
`str` data type	Objects
String literal	Methods
String variable	Context menus
Line continuation	Whitespace characters
Zero-based indexing	Escape sequences
Built-in `len` function	Raw strings
Slicing	Unicode strings
Concatenating	String formatting

3.9 Exercises

1. The Python statements on the left use string operations involving the variable `happyCow`. Match the Python statement with its output. All answers MUST BE one of the letters A through I. The string variable called `happyCow` is assigned as follows:

```
>>> happyCow = 'meadows.shp'
```

Python statement	**Output** (notice there are nine letters)
1. `happyCow[0]`	
2. `happyCow[0:5] + happyCow[-4:]`	A. `IndexError`
3. `len(happyCow)`	B. `'meado.shp'`
4. `happyCow[0:5]`	C. `'meado'`
5. `happyCow[-4:]`	D. `True`
6. `happyCow[11]`	E. `False`
7. `happyCow[:5]`	F. `11`
8. `happyCow in "5meadows.shp"`	G. `'w'`
9. `happyCow[5]`	H. `'.shp'`
10. `'W' in happyCow`	I. `'m'`

2. These Python statements use string methods and operations involving the variable `LCS_ID`. Determine if each Python statement is true or false. The double equals signs return true if the two sides are equal. The `!=` signs return true if the two sides are not equal. The string variable called `LCS_ID` is assigned as follows: `LCS_ID = '0017238'`

   (a) `'17' in LCS_ID`
   (b) `LCS_ID.isdigit()`
   (c) `LCS_ID.lstrip('0') == '17238'`
   (d) `LCS_ID.zfill(10) == '10101010'`
   (e) `LCS_ID + '10' == 17248`
   (f) `LCS_ID[6] == '3'`
   (g) `len(LCS_ID) == 7`
   (h) `LCS_ID[0:7] == '0017238'`
   (i) `int(LCS_ID) + 10 == 17248`
   (j) `LCS_ID != 17238`

3. The Python statements on the left use string methods and operations involving the variable `state`. Match the Python statement with its output. All answers MUST BE one of the letters A through L. The string variable called `state` is assigned as follows:

```
state = 'missiSSippi'
```

Python statement	Output
1. `state.count('i')`	A. `'Mississippi'`
2. `state.capitalize()`	B. `'miRRiRRippi'`
3. `state.endswith('ippi')`	C. `['m', 'ss', 'SS', 'pp', '']`
4. `state.find('i')`	D. `'m'`
5. `';'.join([state,state])`	E. `'MISSISSIPPI'`
6. `state.lower().replace('ss','RR')`	F. `'missiSSippi;missiSSippi'`
7. `state.split('i')`	G. `True`
8. `state.upper()`	H. `'i'`
9. `state[7:]`	I. `4`
10. `state[1]`	J. `1`
11. `state[0:1]`	K. `False`
12. `'Miss' in state`	L. `'ippi'`

4. Test your understanding of this chapter's 'Key terms' by matching the Python statement with a term or phrase that describes it. The four variables used in the matching have been assigned as follows:

```
>>> dataDir = 'C:/data'
>>> data = 'bird_sightings'
>>> count = 500
>>> n = 3
```

Python statement	Output
1. `count/n`	A. Casting
2. `'nest'`	B. Indexing
3. `r'count\n'`	C. Slicing
4. `u'hatchling'`	D. Finding the length of a string
5. `len(data)`	E. Concatenating
6. `str(count)`	F. Line continuation
7. `dataDir + '/' + data`	G. String literal
8. `data[0:n]`	H. String method that is the inverse of the `join` method
9. `data[n]`	I. Escape sequence
10. `'bird' in data`	J. Integer division
11. `'{0} records'.format(count)`	K. Raw string
12. `data.split('_')`	L. Unicode string
13. `'Bird data \` `Wing span'`	M. String formatting

5. **printPractice.py** Modify sample script 'printPractice.py', so that it prints the same statement four times, but using four distinct techniques—hard-coding, commas, concatenation, and string formatting. The hard-coding is already done. The other three techniques have been started but they contain mistakes. These techniques should each use all three of the provided variables. Once the mistakes have been corrected, run the script to make sure that it prints the statement identically four times:

```
Found 12 lights in the 5 mi. buffer and 20 intersections.
Found 12 lights in the 5 mi. buffer and 20 intersections.
Found 12 lights in the 5 mi. buffer and 20 intersections.
Found 12 lights in the 5 mi. buffer and 20 intersections.
```

6. This problem deals with date time stamps that have the form MM/DD/YYYY HH:MM:SSxM (where xM is AM for morning and PM otherwise). Part (a) creates a date-time variable which is used in all other parts. Solutions should work for other values of this variable. Write one or more lines of code to achieve the outcome described in each part.
   (a) Use an assignment statement to set a variable named `dt` to the following string literal: '07/28/2055 05:25:33PM'.
   (b) Use the `in` keyword in a code statement to print `True` if the value of `dt` is in the morning (and `False` otherwise).
   (c) Use slicing to extract the month from `dt`.
   (d) Use the `split` method twice and then use indexing to extract the hour from `dt`.
   (e) Use the `split` method twice and then use indexing to extract the year from `dt`.

7. Write one or more lines of code to achieve the outcome described in each part.

   (a) Often geoprocessing involves sorting data files based on their names. For example, you may want to move all the files from Region 1 to a separate directory. Write a line of code to check if a variable named `filename` contains a substring that matches the value of a variable named `code`. The examples below show how the line of code could be used.

   ```
   >>> # Sample input 1:
   >>> filename = 'Forest361Region1_rec.shp'
   >>> code = 'Region1'
   >>> # Insert line of code here.
   True
   >>> # Sample input 2:
   >>> filename = 'Marsh12Region4.shp'
   >>> code = 'Region1'
   >>> # Insert line of code here.
   False
   ```

   (b) Write a line of code that uses the `rstrip` method and an assignment statement to remove the trailing whitespace from the string variable named `data`. The variable contains a line of tab delimited North Carolina county forestry data. The example below shows how the line of code could be used.

   ```
   >>> # Sample input:
   >>> data = 'Site:\tNortheast Prong\tDARE\t01\t\n\n'
   >>> data
   'Site:\tNortheast Prong\tDARE\t01\t\n\n'
   >>> # Insert line of code here.
   >>> data
   'Site:\tNortheast Prong\tDARE\t01'
   ```

   (c) Suppose that we are editing data and saving the results in another file. We want to append the string `'_edited'` to the input base name to create the output name. For example, if the input file is named `'countiesNC.txt'`, the output file should be named `'countiesNC_edited.txt'`. If the input file is named `'riversWVA.txt'`, the output file should be named `'riversWVA_edited.txt'`, and so forth. Write a line of code that uses slicing, concatenation, and a variable named `input` to assign a value to a variable named `output`. The example below shows how the line of code could be used.

```
>>> # Sample input:
>>> inputName = 'counties.shp'
>>> # Insert line of code here.
>>> print outputName
counties_edited.shp
```

(d) Write a line of code using the string `format` method to print the desired output. Pass the given variables as arguments. The output from the print statements should appear exactly as shown in the output below.

```
>>> eagleNests = 2
>>> forestAcreage = 10
>>> campsites = 5
>>> # Insert line of code here.
There are 2 nests and 5 campsites within 10 acres.
```

(e) Write a line of code using the string `format` method to print the desired output.

Pass the given variables as arguments. The output from the print statements should appear exactly as shown in the output below.

```
>>> bufferInput = 'crimeSites'
>>> num = 32
>>> # Insert line of code here.
crimeSites.shp buffered. Results saved in C:/buff32.shp
```