Chapter 9
Decision-Making and Describing Data

Abstract

Scripts routinely need to perform different operations based on some deciding criteria. The decision may be very simple, ('if the field doesn't exist, add the field') or it may be more complex ('if the shapefile has point geometry, compute a buffer, but if it has polygon geometry, perform an intersection'). Chapter 8 introduced 'decision-making' structures. These are often referred to as 'branching' structures, because the workflow diagram branches where the decision occurs. For example, the 'Yellowstone Hiker's Flowchart' branches in three places: 'Do I see an animal?', 'Is it a bear?' and 'Is it a killer rabbit?'. Pseudocode uses `IF`, `THEN`, `ELSE IF`, `ELSE`, and `ENDIF` key words to express decision-making workflows. This chapter presents the Python syntax, conditional expressions, ArcGIS tools that make selections, the `arcpy Describe` method, handling optional input, and creating directories.

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_9

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

- Implement `IF`, `ELSE IF`, and `ELSE` structures in Python.
- Explain when to use only an `IF` block, when to use an `ELSE IF` block, and when to use an `ELSE` block.
- Specify decision-making conditions with comparison, logical, and membership operators.
- Select data within a table using SQL comparison and logical operators.
- Design syntactically and logically sound compound conditional expressions.
- Identify code testing cases for branching.
- Use data properties to make decisions.
- Handle optional user input.
- Safely create output directories.

"That one's past its 'Sell-By' date."

Decision-making is expressed with conditional constructs. If some condition is true, some action is taken. E.g., if the animal is a bear, ring a bell. In Python, conditional constructs begin with the `if` keyword. This keyword, followed by a condition, followed by a block of indented code, make up the simplest conditional construct. The syntax looks like this:

```
if condition:
    code statement(s)
```

If the condition is true, the indented block of code is executed. Otherwise, it is skipped. A condition is checked and an action is taken (or not), based on that condition. The semicolon and the indentation are required. At least one indented code statement must follow the line that ends in a colon (`:`). The conditional construct is the first instance of a compound code block discussed so far in this book. We'll introduce a number of additional compound code blocks in upcoming chapters. All compound code blocks in Python require a colon at the end of the first line and indentation in the line that follows.

Example 9.1

Python	Pseudocode

`if speciesCount < 500:`
`  print 'Endangered species'`
`print speciesFID`
	`IF species count < 500 THEN`
	`  PRINT 'Endangered species'`
	`ENDIF`
	`PRINT species FID`

The code in Example 9.1 only prints 'Endangered species' if the count is low, but prints the species ID for *every* species. The species ID print statement is outside of the `IF` block since it is dedented. Examples 9.1–9.4 show Python on the left and Pseudocode on the right. Can you spot the four differences between the Python code and pseucode? Python uses a colon at the end of the first line as opposed to a THEN keyword. Python keywords are lower case, pseudocode keywords are uppercase. Python uses the `elif` keyword as opposed to the pseudocode `ELSE IF` keywords. Pseudocode uses `ENDIF` to enclose the `IF` block statements; Python uses indentation instead.

Python IDEs automatically move the cursor to an indented position when you start a compound code block. If you notice that the code is not automatically indenting, it means you forgot to type the colon. When using the Interactive (or Interpreter) Window for compound code blocks, IDEs print ellipses (…) to indicate that you're inside a compound code block. To end a compound code block in the Interactive Window, you need to press the 'Enter' key twice.

```
>>> speciesCount = 250
>>> if speciesCount < 500:
...     print 'Endangered species'
...
Endangered species
```

Example 9.1 has no contingency plan. If the species count is 500 or higher, no special action is taken. It uses only one decision-making code block. When you want to execute two different actions depending on if the condition is true or false, add an `ELSE` block after the IF block. Example 9.2 has two decision-making code

blocks. It uses `if` and `else` to handle the two possible outcomes (valid or not). Either way it prints a phrase, either confirming that the area is valid or warning that it's not.

Example 9.2: Check for valid polygon areas.

```
if area > 0:
    print 'The area is valid.'
else:
    print 'The area is invalid.'
```

```
IF polygon area > 0 THEN
    PRINT The area is valid
ELSE
    PRINT The area is invalid
ENDIF
```

To handle more complex decisions, where you want to look at multiple conditions, you can use multiple decision-making blocks. Suppose you're making plans for the weekend. If the weather is sunny and calm, you'll swim in the ocean; if the weather is not sunny and calm, but sunny and windy, you'll fly a kite instead. If the weather is cold and clear, you'll ice-skate, and in all other conditions, you'll stay home and write some Python geoprocessing code. The pseudocode for your weekend plans would be as follows:

```
Check the weather forecast
IF weather is sunny and calm THEN
    Swim in ocean.
ELSE IF weather is sunny and windy THEN
    Fly a kite.
ELSE IF weather is cold and clear THEN
    Go ice-skating.
ELSE
    Write GIS Python code.
ENDIF
```

To make the decision like a computer, you would read from top to bottom and check the weather against each of these weather conditions until you reach one that matches. Then you would do the activity that was planned for that weather. You wouldn't even consider any activities in branches below where you found the match. If none of the weather conditional match, you still have an activity planned. In rain or snowy conditions, for example, you will write Python code! The pseudocode uses `ELSE IF` to check the alternative weather conditions and `ELSE` to catch all other weather. Python keywords `elif` and `else` can optionally follow an `if` code block, to specify more than one action. A colon and indentation are required for `elif` and `else` since these are block code statements (see Examples 9.3 and 9.4). A decision-making code block can use any number of `elif` keywords, but it can use at most one `else` keyword. Using one or more `elif` allows you to specify multiple conditions. An `ELSE` block is used for default actions and must come last, if it is used. Since `else` is used as a catch-all, it is not followed by a condition, only a colon.

9.1 Conditional Expressions

The condition following `if` or `elif` keywords is a Boolean expression. Expressions that are being used for their true or false value are referred to as *Boolean expressions*. In this book, the keywords `if` or `elif` followed by a Boolean expression is referred to as a *conditional expression*. A Boolean expression can be evaluated for its truth value using the built-in `bool` function, which returns a value of `True` or `False`. Any object in Python can be evaluated in this way. For example, `0` is evaluated as `False`, but `5` is evaluated as `True`:

```
>>> bool(0)
False
>>> bool(5)
True
```

The majority of object values evaluate as true. Table 9.1 shows seven false Python objects. Two built-in constants (`False` and `None`) evaluate as false. The number zero evaluates as false, but all other numbers evaluate as true. Empty strings, tuples, and lists evaluate as false, but all other values of these data types evaluate as true. An empty string literal value has two quotation marks (single or double) with nothing to separate them (not even a space). An empty list is just a pair of square brackets (spacing doesn't matter). Knowing this short list of false objects will help you build well-structured conditional expressions.

The conditional expressions in Example 9.1 (`speciesCount < 500`) and Example 9.2 (`area > 0`) compare variables to numeric values, using less than and greater than signs. These signs are examples of *comparison operators*. Operators like these are often needed to construct conditional expressions. Comparison, logical, and membership operators are discussed next.

Comparison, logical, and membership operators are collectively referred to as Boolean operators. Table 9.2 shows examples of these types of Boolean operators.

Table 9.1 False Python constants and data values.

Python value	Description
`False`	Built-in Python constant
`None`	Built-in Python constant
`0`	The number zero
`''`	An empty string. (No spaces between the quotes)
`()`	An empty tuple
`[]`	An empty list
`{}`	An empty dictionary

Table 9.2 Boolean operators.

Python comparison operators	
(x and y are objects such as numbers or strings)	
`x < y`	`# x is less than y`
`x > y`	`# x is greater than y`
`x <= y`	`# x is less than or equal to y`
`x >= y`	`# x is greater than or equal to y`
`x == y`	`# x is equal to y`
`x != y`	`# x is not equal to y`
Python logical operators	
(x and y are objects, often Boolean expressions)	
`x` **`and`** `y`	True if both `x` and `y` are true.
`x` **`or`** `y`	True if either `x` or y or both are true.
`not` `x`	True if `x` is false.
Membership operators	
(x is any object and y is a sequence type object)	
`x` **`in`** `y`	True if `x` is in `y`.
`x` **`not in`** `y`	True if `x` is not in `y`.

9.1.1 Comparison Operators

Comparison operators use the familiar < and > symbols from mathematics for 'less than' and 'greater than'. 'Less than or equal to' and 'greater than or equal to' are represented by the <= and >= operators respectively. `x` and `y` being compared in Table 9.2 can be any type of object. These operators reference an ordering assigned to objects. Numbers are less than characters. Capital string characters are less than lower case and letters are ordered based on position in the alphabet.

```
>>> 100 < 'A' < 'a' < 'b'
True
```

Comparison operators compare individual characters in strings from left to right until a smaller character is reached.

```
>>> 'aa' < 'ab'
True
```

Numerical ordering applies to numbers, but not to strings of numeric characters. Comparison operators compare numeric characters in strings from left to right until a smaller numerical character is reached. The following example shows that the number `128` is evaluated as greater than `15`, but the string `'128'` is evaluated as

less than the string `'15'` because the second character of `'128'` is lower than the second character `'15'` (i.e., `'2'<'5'`):

```
>>> 128 > 15
True
>>> '128' > '15'
False
```

Equality is tested with the double equals sign (the == operator). Inequality (‘not equal to’) is tested with the != operator. Example 9.3 checks the equality of a pair of strings.

Example 9.3: Print the ID numbers of highways and rivers.

```
if classType == 'major highway':
    print 'Highway--', FID
elif classType == 'river':
    print 'River--', FID
```

```
IF class type is highway THEN
    PRINT Highway ID number
ELSE IF class type is river THEN
    PRINT River ID number
ENDIF
```

Example 9.3 only prints an `FID` if the class type is highway or river, but Example 9.4 will always print an `FID`, since an `ELSE` block was added.

Example 9.4: Print the ID number for *all* class types.

```
if classType == 'highway':
    print 'Highway--', FID
elif classType == 'river':
    print 'River--', FID
else:
    print 'Other--', FID
```

```
IF class type is highway THEN
    PRINT Highway ID number
ELSE IF class type is river THEN
    PRINT River ID number
ELSE
    PRINT other class type ID number
ENDIF
```

Note Cast numerical script arguments to numbers when using them in numerical comparisons. Without casting, comparisons may not work as expected:

```
>>> 25 < '20'
True
```

9.1.2 Equality vs. Assignment

One common mistake is using = when == is intended. The single equals sign (=) is an assignment operator, not a comparison operator. To set x equal to five, use `x = 5`. But to compare x to five, use `x == 5`. The double equals tests for equality. Attempting to perform an assignment in a conditional expression results in an error:

```
>>> if x = 5:
Traceback ( File "<interactive input>", line 1
if x = 5:
     ^
SyntaxError: invalid syntax
```

9.1.3 Logical Operators

Logical operator keywords `and`, `or`, and `not` are used to encode logical conditions. For example, they can check if a condition is not true, or if two conditions are true, or if one condition is true and another isn't, or if either of two conditions are true, and so forth. The following code prints the file name if it has an '.shp' or '.txt' extension:

```
>>> fileName = 'park.shp'
>>> if fileName.endswith('.shp') or fileName.endswith('.txt'):
...     print fileName
...
park.shp
```

The following code prints file names that do not have a '.csv' extension:

```
>>> if not fileName.endswith('.csv'):
...     print fileName
...
park.shp
```

Expressions that use `and` and `or` are called *compound conditional expressions*, because they combine two or more conditions. The expressions on either side of the keywords `and` and `or` are evaluated independently. Sometimes this doesn't correspond to how we would normally formulate a condition in natural language. For example, suppose we want to print the species name only when the species is salmon or tuna. This might erroneously get translated to Python like this:

```
>>> species = trout'
>>> if species == 'salmon' or 'catfish':
...     print species
...
trout
```

This code fails because of how it evaluates the conditions: It checks if the species is `'salmon'`. Then it checks if `'catfish'` is true—which it is, since it's not an empty string. No value of `species` will make this statement false, since the string literal `'catfish'` is always true. A complete comparison expression is needed on both sides, as in the following code:

```
>>> if species == 'salmon' or species == 'catfish:
...     print species
...
```

9.1.4 Membership Operators

Membership operators can also be harnessed to form conditional expressions. Logical operators can be stringed together to test more than two conditions, as in the following example:

```
if classType == 'major highway' or classType == 'river' or \
   classType == 'stream' or classType == 'bridge':
    print classType, '--',FID
else:
    print 'Other--', FID
```

However, this is an ideal situation for using the `in` keyword instead. This example shows a more elegant solution to check if the class is in one of the special categories:

```
specialTypes = ['highway', 'river', 'stream', 'bridge']
if classType in specialTypes:
    print classType, '--',FID
else:
    print 'Other--', FID
```

Conversely, it can be used to identify only those items that are not in the list:

```
specialTypes = ['highway', 'river', 'stream', 'bridge']
if classType not in specialTypes:
    print classType, '--', FID
```

9.2 ArcGIS Tools That Make Selections

Many ArcGIS tools and `arcpy` functions have a `where_clause` parameter that can be used to make a selection of features, such as selecting the features in 'park.shp' which have a cover type of 'orch' (for orchard). For example, the third parameter of the Select (Analysis) tool shown here is named `where_clause`.

Syntax

Select_analysis (in_features, out_feature_class, {where_clause})

Parameter	Explanation	Data Type
in_features	The input feature class or layer from which features are selected.	Feature Layer
out_feature_class	The output feature class to be created. If no expression is used, it contains all input features.	Feature Class
where_clause (Optional)	An SQL expression used to select a subset of features. The syntax for the expression differs slightly depending on the data source. For	SQL Expression

These expressions should not be confused with the Python expressions used for calculating values in some tools, such as the Calculate Field (Data Management) tool (discussed in Section 6.3.2). `where_clause` parameters are expressions that are either true of false and they must be specified as SQL expressions, as opposed to Python expressions. SQL is a specialized programming language for managing databases. The term *where-clause* refers to the SQL keyword 'WHERE' which is used to make selections in SQL syntax. The syntax for SQL in ArcGIS may differ slightly from other versions of SQL with which you are familiar. Like Python conditional expressions, SQL where-clauses use comparison and logical operators. <, <=, >, and >= are used in the same way, but SQL uses a single equals sign (=) to check equality, unlike Python which reserves the single equals sign for assignment. Inequality is represented by <> in SQL, whereas Python uses `!=`. The same logical operators can be used in SQL but they are usually uppercased in SQL (`AND`, `OR`, and `NOT`); Whereas, in Python, they must be lowercase. Table 9.3 can be used as a quick reference for these operators.

The syntax for `where_clause` ArcGIS tool parameters is slightly tricky because this is not stand alone SQL, but rather, SQL embedded in Python. The `where_clause` tool parameter is specified as a Python string, so it has to be surrounded by quotation marks like this:

```
>>> whereClause1 = 'RECNO >= 400'
>>> type(whereClause1)
<type 'str'>
```

Table 9.3 Python vs. SQL operators.

Python	SQL
Comparison operators	
<	<
>	>
<=	<=
>=	>=
==	=
!=	<>
Logical operators	
and	AND
or	OR
not	NOT

For these examples, consider the shapefile 'park.shp' which has a numeric field named `'RECNO'` and a text field named `'COVER'`. `whereClause1` specifies the values of the `'RECNO'` field greater than or equal to 400. The field name is specified on the left of the comparison operator (conventionally uppercased, though this is not case sensitive). To the right of the comparison operator the field value is specified. The entire phrase is surrounded by quotation marks to make it a Python string. Example 9.5 shows this clause being used in a tool call. It is passed as the third parameter into the Select (Analysis) tool to specify the selection to be made. The output file will contain only those records with `'RECNO'` field values greater than or equal to 400.

Example 9.5: Using a hard-coded where-clause.

```
# where_clause1.py
# Purpose: Select features with high reclassification numbers.
# Usage: No arguments needed.

import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch09'
inputFile = 'park.shp'
whereClause = 'RECNO >= 400'
arcpy.Select_analysis(inputFile, 'C:/gispy/scratch/out.shp', whereClause)
```

Compound clauses can be made with logical operators. Just like in Python, a complete expression must be used on either side of the logical operator. For example, this clause specifies the `'RECNO'` field values between 400 and 410:

```
>>> whereClause2 = 'RECNO > 400 AND RECNO < 410'
```

In contrast, `'RECNO > 400 AND < 410'` is not a valid SQL expression and will cause an error.

For text fields, an extra set of quotation marks need to be used around the field value. Two distinct types of quotation marks are used (single for the interior quotes,

double for the exterior quotes), so that Python matches them correctly. For example, the following clause could be used to select the records where the `'COVER'` field has the value `'orch'`:

```
>>> whereClause3 = "COVER = 'orch'"
```

If single quotation marks are used throughout the phrase, a syntax error occurs:

```
>>> whereClauseBogus = 'COVER = 'orch''
Traceback ( File "<interactive input>", line 1
whereClauseBogus = 'COVER = 'orch''
                                ^
SyntaxError: invalid syntax
>>>
```

Python, reads the quotation marks from left to right and assumes that the second quotation mark it reaches (the one just before `orch`) matches the first one, leaving the remainder of the characters outside the string.

In summary, the format for simple `where_clause` expressions is one of the following:

```
"NUMERIC_FIELD_NAME comparison_operator numeric_value"
"TEXT_FIELD_NAME comparison_operator 'text_value'"
```

In some applications the field name or the field value to be used in the expression is not determined beforehand, so it can't be hard-coded. The `where_clause` parameter is a Python string, so we can use the string `format` method to build the clause with variables. The following code uses a variable field value to create `whereClause4` (whose value is identical to `whereClause3`):

```
>>> fieldValue = 'orch'
>>> whereClause4 = "COVER = '{0}'".format(fieldValue)
>>> whereClause4
"COVER = 'orch'"
```

The `format` method substitutes the arguments into the string for the placeholders. Notice the single quotation marks are still used around the placeholder `{0}` so that they are in place when the replacement is made. The same method could be used for a variable field name or for both a variable field name and value, as in the following example:

```
>>> fieldName = 'COVER'
>>> fieldValue = 'orch'
>>> whereClause5 = "{0} = '{1}'".format(fieldName, fieldValue)
>>> whereClause5
"COVER = 'orch'"
```

Example 9.6 uses a string formatted clause to extract values from a raster. The field name and value are passed into the script by the user. If the script is run with the example arguments, 'COUNT' and '6000', the output raster only contains cells where the value of the 'COUNT' field is greater than 6000. In Example 9.6, the selected values are saved in a new raster; `where_clause` parameters can also be used to select records for operations on a subset of the records as discussed next.

Example 9.6: Using a where clause with variables.

```
# where_clause2.py
# Purpose: Extract raster features by attributes based on
#          user input.
# Usage: fieldName fieldValue
# Example input: COUNT 6000

import arcpy, sys

arcpy.env.workspace = 'C:/gispy/data/ch09'
arcpy.CheckOutExtension('Spatial')

inputRast = 'getty_rast'

fieldName = sys.argv[1]
fieldValue = sys.argv[2]

whereClause = '{0} > {1}'.format(fieldName, fieldValue)

outputRast = arcpy.sa.ExtractByAttributes(inputRast, whereClause)
outputRast.save('C:/gispy/scratch/attextract')
del outputRast
```

9.2.1 Select by Attributes and Temporary Feature Layers

The Select Layer By Attribute (Data Management) tool is another tool that uses a `where_clause` parameter. This tool requires a little bit of explanation for use in Python because both the input and output are temporary layers not feature classes such as shapefiles or geodatabase feature classes (Section 6.6 discussed the temporary layers in the context of the Make XY Event Layer (Data Management) tool). Performing a selection on a feature class and saving the selection as a feature class requires making a feature layer, selecting, and saving the selection, as in the following steps:

Step 1. The 'Make Feature Layer' tool can be used to create a temporary layer from a feature class. The following code creates a temporary feature layer:

```
>>> arcpy.env.workspace = 'C:/gispy/data/ch09/tester.gdb'
>>> tmp = 'tmpLayer'
>>> arcpy.MakeFeatureLayer_management('park', tmp)
```

Step 2. The following code prepares a compound `where_clause` variable to find the small wooded plots (those with `area < 20000`) with 'woods' land cover and performs the selection:

```
>>> maxArea = 20000
>>> coverType = 'woods'
>>> whereClause = "Shape_area < {0} AND \
COVER ='{1}'".format(maxArea, coverType)
>>> arcpy.SelectLayerByAttribute_management(tmp,
                                            'NEW_SELECTION',
                                            whereClause)
<Result 'tmpLayer'>
```

Step 3. Finally, the following code saves the temporary layer to a feature class:

```
>>> output = 'smallWoods'
>>> arcpy.CopyFeatures_management(tmp, output)
<Result 'C:/gispy/data/ch09\\smallWoods'>
```

9.3 Getting a Description of the Data

For geoprocessing, we often make decisions based on data properties. The `arcpy` function named `Describe` allows you to access properties of a data object. It takes one argument, the name of an Esri data element or geoprocessing object. Examples of valid arguments include a Feature Class name, a Layer name, a Raster Dataset name, and so forth. The following code calls the `Describe` function with the Raster Dataset named `'getty_rast'` as an argument:

```
>>> import arcpy
>>> arcpy.env.workspace = 'C:/gispy/data/ch09'
>>> rastFile = 'getty_rast'
>>> desc = arcpy.Describe(rastFile)
```

The `Describe` function returns a `Describe` object, which is stored in the variable named `desc`:

```
>>> desc
<geoprocessing describe data object object at 0x0092D2D0>
```

9.3.1 `Describe` Object Properties

The `Describe` object has a set of properties with information about the data. These properties fall into two categories:

- Universal properties—any valid input has all of these properties.
- Specialized properties—these properties depend on the data type of the input.

Whenever you create a `Describe` object, it has values for the set of universal properties that are common to any object being described. These are things like `baseName`, `dataType`, and `extension`, to name just a few. To access these properties, use dot notation with the `Describe` object. In this example, we print a few of the universal properties:

```
>>> desc.dataType
u'RasterDataset'
>>> desc.baseName
u'getty_rast'
>>> desc.extension
u''
>>> desc.catalogPath
u'C:/gispy/data/ch09/getty_rast'
```

The additional, specialized properties depend on the data type of the argument. For example, when the argument is a `RasterDataset` type, like 'getty_rast', there are a set of `RasterDataset` Properties, such as `bandCount`, `compressionType`, and `format`. Again, access these with dot notation:

```
>>> desc.CatalogPath
u'C:/gispy/data/ch09/getty_rast'
>>> desc.bandCount
1
>>> desc.compressionType
u'RLE'
>>> desc.format
u'GRID'
```

The `Describe` object `format` property should not be confused with the string method by the same name. The `format` property refers to the image format (GRID, JPEG, PNG, etc.)

Many of the properties return string values, but some return `arcpy` objects. These objects may have their own set of methods and properties, which can be accessed with dot notation. For example, the `extent` property returns an `Extent` object with minimum and maximum values for x, y, z, and m, as well as other

properties. The following code assigns an `Extent` object to a variable named `bounds`, then uses the variable to find the maximum x value:

```
>>> bounds = desc.extent
>>> bounds
<Extent object at 0x3338c50[0x125b4d70]>
>>> bounds.XMax
2167608.390378157
```

9.3.2 Lists of Properties

The ArcGIS Resources site lists the universal and specialized properties. The 'Describe object Properties' page lists the universal properties; any valid input has all of these properties. The 'Describe (arcpy)' page has a list of the specialized data types, each linked to a list of properties.

Many of the special data types have access to a few general types. The `Dataset` properties and `Table` properties are available to many types of `Describe` objects. For example, the 'Raster Dataset Properties' page says 'Dataset Properties also supported.' This means Raster Dataset type `Describe` objects also have all the `Dataset` properties, such as, `extent` and `spatialReference`.

Figure 9.1 depicts a portion of the `Describe` functionality. The `Describe` function is one of the `arcpy` functions. It returns a `Describe` object. The box highlighted in Figure 9.1 lists a few of the `Describe` object properties. The box (labeled "describe data objects") represents the `Describe` object and lists a few of its universal properties. The darker boxes inside this one represent some of the specialized property groups. For example, `FeatureClass` type objects have six properties, plus they have access to `Dataset` and `Table` properties and `RasterDataset` type objects have five properties, plus they have access to Dataset and Raster Band properties. Only a few of the properties are listed in the boxes as a reminder of the many available properties. The ArcGIS Resources documentation contains the complete property lists for each data type.

9.3.3 Using Specialized Properties

Care must be taken when using the specialized properties. If you try to use the wrong type of property, an error will occur. The following code attempts to use the `format` property but it fails because 'park.shp' is a ShapeFile data type so it doesn't have this specialized property:

```
>>> fcFile = 'C:/gispy/data/ch09/park.shp'
>>> desc2 = arcpy.Describe(fcFile)
>>> desc2.dataType
```

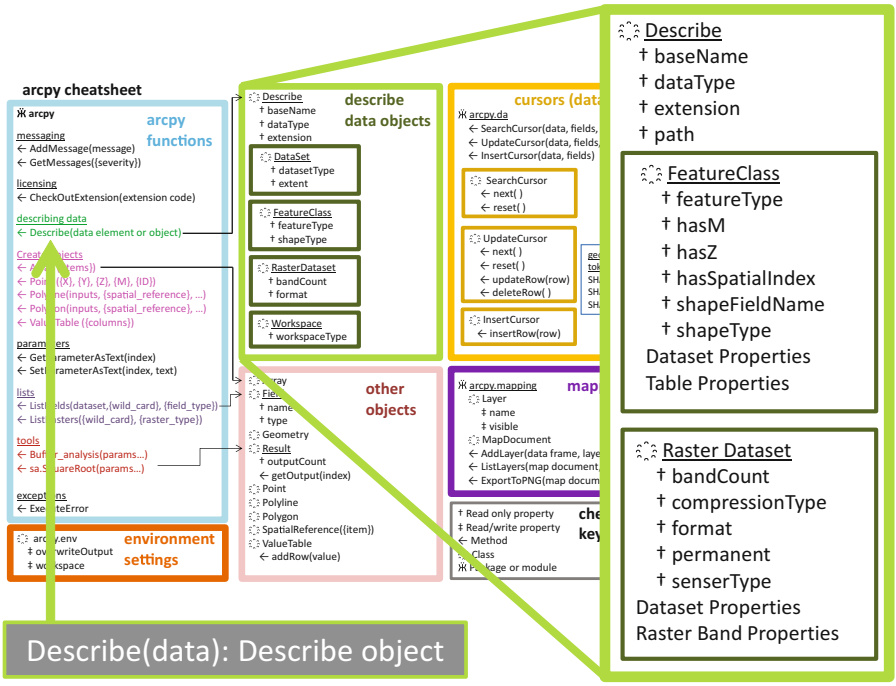

Figure 9.1 The `arcpy Describe` function returns a `Describe` object. These boxes show a subset of the `Describe` object properties.

```
u'ShapeFile'
>>> desc2.format
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
AttributeError: DescribeData: Method format does not
exist
```

For this reason, the specialized properties are usually used inside conditional constructs which check the `dataType` of the object before using these properties. The `dataType` property returns a string value, an alias that `arcpy` uses for the given type of data. Table 9.4 lists the `dataType` values for ten sample datasets. Others can be found in the ArcGIS Resources pages for the data types, linked to the 'Describe (arcpy)' page. Example 9.7 checks the value of the `dataType` property instead of just using the format property on any `Describe` object. If the second argument is 'getty.tif', the script prints 'Image format: TIFF'. If the input is 'park. shp', the script prints nothing.

Table 9.4 `Describe` object `dataType` values for sample data.

Data description	Sample data	dataType value
Shapefile '.dbf' table	park.dbf	'ShapeFile'
Independent '.dbf' table	site1.dbf	'DbaseTable'
Directory	C:/gispy/data/ch09	'Folder'
Geodatabase feature class	tester.gdb/data1	'FeatureClass'
Geodatabase raster	tester.gdb/aspect	'RasterDataset'
Layer file	park.lyr	'Layer'
Raster Dataset	getty_rast	'RasterDataset'
Shapefile	park.shp	'ShapeFile'
Text File	xy1.txt	'TextFile'
Workspace	tester.gdb	'Workspace'

Example 9.7: Using a Describe properties inside a conditional block.

```
# describeRaster.py
# Purpose: Report the format of raster input file.
# Usage: workspace, raster_dataset
# Example: C:/gispy/data/ch09 getty.tif

import arcpy
arcpy.env.workspace = arcpy.GetParameterAsText(0)
data = arcpy.GetParameterAsText(1)
desc = arcpy.Describe(data)
if desc.dataType == 'RasterDataset':
    print 'Image format: {0}'.format(desc.format)
```

9.3.4 Compound vs. Nested Conditions

Sometimes you may want to check more than one condition before calling a geoprocessing tool. For example, to use the Smooth Line (Cartography) tool, you can check the `dataType` and the `shapeType`. You can do this with a compound statement, but any specialized conditions (such as `shapeType`) must be placed after the more general conditions (such as `dataType`) because the conditions in a compound statement are evaluated left to right. This code causes an error:

```
>>> desc = arcpy.Describe('C:/gispy/data/ch09/getty.tif')
>>> desc.dataType
u'RasterDataset'
>>> if desc.shapeType == 'Polyline' and \
dsc.dataType in ['FeatureClass', 'Shapefile']:
```

```
...     print 'Smooth line'
...
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
AttributeError: DescribeData: Method shapeType does not exist
```

How could you reorder the compound condition to correct this problem? In Example 9.8, 'smoothLineCompound.py' correctly uses a compound statement to only call the Smooth Line tool for `'FeatureClass'` or `'ShapeFile'` types with a `shapeType` of `'Polyline'`.

Example 9.8: Using a Describe properties inside a conditional block.

```
# smoothLineCompound.py
# Usage: workspace, features_with_line_geometry
# Example 1: C:/gispy/data/ch09 trails.shp
# Example 2: C:/gispy/data/ch09 park.shp

import arcpy

arcpy.env.overwriteOutput = True arcpy.env.workspace =
arcpy.GetParameterAsText(0) data =
arcpy.GetParameterAsText(1)
outFile = 'C:/gispy/scratch/smoothOut'
desc = arcpy.Describe(data)
if desc.dataType in ['FeatureClass' ,'ShapeFile'] and \
                    desc.shapeType == 'Polyline':
    result = arcpy.SmoothLine_cartography(data, outFile,
                                          'BEZIER_INTERPOLATION')
    print 'Smooth line created {0}.'.format(result.getOutput(0))
```

Compound conditions work well as long as exactly the same behavior is desired for both conditions. However, suppose we want to tailor warning messages depending on which condition fails. Then we need to use nested conditional constructs. The pseudocode for this would look like this:

```
# GET input
IF input data feature class or shapefile THEN
    IF the shape type is line THEN
        CALL the smooth line tool.
    ELSE
        Warn the user about shape type requirements.
    ENDIF
ELSE
    Warn the user about the data type requirements.
ENDIF
```

The nested conditions allow the script to use one ELSE block for each condition. Example 9.9 shows the corresponding script.

Example 9.9: Using Describe properties inside a nested conditional block.

```
# smoothLineNested.py
# Usage: workspace, features_with_line_geometry
# Example 1: C:/gispy/data/ch09 trails.shp
# Example 2: C:/gispy/data/ch09 park.shp
# Example 3: C:/gispy/data/ch09 getty.tif

import arcpy arcpy.env.overwriteOutput = True

arcpy.env.workspace =arcpy.GetParameterAsText(0)

data = arcpy.GetParameterAsText(1)
outFile = 'C:/gispy/scratch/output'
desc = arcpy.Describe(data)

if desc.dataType in ['FeatureClass' ,'ShapeFile']:
    if desc.shapeType == 'Polyline':
        result = arcpy.SmoothLine_cartography(data, outFile,
                                              'BEZIER_INTERPOLATION')
        print 'Smooth line created {0}.'.format(result.getOutput(0))
    else:
        print 'Warning: shape type is {0}. Smooth Line only works
            on Polyline shape types. '.format( desc.shapeType)
else:
    print "Warning: Input data type must be 'FeatureClass' or 'ShapeFile'."
    print 'Input dataType:', desc.dataType
```

9.3.5 Testing Conditions

Scripts that have conditional constructs, should be tested with input that fulfill each of the conditions they are built to handle. For example, 'smoothLineNested.py', in Example 9.9, considers three outcomes. We can test for each of these by varying the input.

1. Input data type is FeatureClass or Shapefile and shape type is Polyline.

   Input: C:/gispy/data/ch09 trails.shp

   Output:

   ```
   Smooth line created C:\gispy\scratch\output.shp.
   ```

2. Input data type is FeatureClass or Shapefile but shape type is not Polyline.

   Input: C:/gispy/data/ch09 park.shp

   Output:
   ```
   Warning: shape type is Polygon. SmoothLine only works on Polyline
   shape types.
   ```

3. Input data type is neither FeatureClass nor Shapefile.

   Input: C:/gispy/data/ch09 getty.tif

   Output:
   ```
   Warning: Input data type must be 'FeatureClass' or 'ShapeFile'.
   Input dataType: RasterDataset
   ```

9.4 Required and Optional Script Input

Conditional constructs can also be used to handle optional script arguments. The script must somehow check whether or not optional arguments have been given and set a default value if necessary. For example, the workflow represented by the following pseudocode optionally takes one argument, a directory path, and lists the files in the given directory or the files in the script directory:

```
IF argument found THEN
    SET working directory to argument.
ELSE
    SET working directory to script directory.
ENDIF
List files in working directory.
```

This can be implemented using the `sys` module `argv` list, as in Example 9.10. Since `sys.argv` is a Python list, if a script uses an index that is not available, an `IndexError` will be reported:

```
>>> import sys
>>> sys.argv[1]
Traceback (most recent call last):
    File "<interactive input>", line 1, in <module>
IndexError: list index out of range
```

To avoid this, you can check the length of the list before trying an index for an optional argument. Recall that the first item in this list (`sys.argv[0]`) is the script file name:

```
>>> sys.argv[0]
'C:\\gispy\\sample_scripts\\ch09\\scriptPathOptionalv1.py'
```

This means that the length of the list is the number of user arguments plus one. Therefore, if the user passes in one argument, the list length is 2.

Example 9.10

```
# scriptPathOptionalv1.py
# Purpose: List the files in the given directory or
#       the current directory.
# Usage: {directory_path}
import sys, os

if len(sys.argv) == 2:
    workingDir = sys.argv[1]
else:
    # Get the script location
    scriptPath = os.path.abspath(__file__)
    workingDir = os.path.dirname(scriptPath)

print '{0} contains the following files:'.format(workingDir)
print os.listdir(workingDir)
```

Alternatively, this can be implemented with the `GetParameterAsText` function, as in Example 9.11. The default value returned by this function is an empty string (`''`). Since an empty string is considered false, and all other strings are considered true, we can use the value of `arcpy.GetParameterAsText(0)` as the conditional expression. Both versions of this script work with or without an argument.

Example 9.11

```
# scriptPathOptionalv2.py
# Purpose: List the files in the given directory or
#        the current directory.
# Usage: {directory_path}
import arcpy, os

if arcpy.GetParameterAsText(0):
    workingDir = arcpy.GetParameterAsText(0)
else:
    # Get the script location
    scriptPath = os.path.abspath(__file__)
```

```
    workingDir = os.path.dirname(scriptPath)
print '{0} contains the following files:'.format(workingDir)
print os.listdir(workingDir)
```

Conditional constructs can also be used to enforce required arguments. The km/mile converter in Example 9.12 requires two arguments, a numerical distance and a distance unit. All arguments are passed into `sys.argv` as string types. The first argument is a numerical value, so we need to use the built-in `float` function to get the numerical value (cast the value). The unit is converted to all lower case when it's tested (using `unit.lower()`), so that the input is not case sensitive.

Example 9.12: Simple distance converter.

```
# distanceConvertv1.py
# Purpose: Converts km to miles and vice versa.
# Usage: numerical_distance, unit_of_measure
# Example: 5 km

import sys

dist = float(sys.argv[1]) # Cast string to float
unit = sys.argv[2]

mileList = ['mi','mi.','mile','miles']

if unit.lower() in mileList:
    output = dist*1.6
    print '{0} {1} is equivalent to {2} kilometers(s).'.format(
                                            dist, unit, output)
else:
    output = dist*.62
    print '{0} {1} is equivalent to {2} mile(s).'.format(
                                        dist, unit, output)
```

If either argument is missing, the script will throw an `IndexError` exception. Since the user might not understand this, it could be helpful to instead print a message about how the script is designed to be run. Example 9.13 adds this information and also makes the second argument optional. If no user arguments are given, the script warns the user and exits. A `sys` module function named `exit` is used to exit the script. If one user argument is given, the script sets a default distance unit, otherwise the unit is set to the second argument. The remainder of the script is the same as version 1.

Example 9.13: Distance converter with input checking.

```
# distanceConvertv2.py
# Purpose: Converts km to miles and vice versa.
# Usage: numerical_distance, {unit_of_measure}
# Example: 5 km
```

```
import sys

numArgs = len(sys.argv)

# If no user arguments are given, exit the script and warn the user.
if numArgs == 1:
    print 'Usage: numeric_distance {distance_unit (mi or km)}'
    print 'Example: 5 km'
    sys.exit(0) # exit the script

# If only one user argument is given, set the unit to miles.
if numArgs < 3:
    unit = 'miles'
    print '''Warning: No distance unit provided.
    Assuming input is in miles.'''
else:
    # Get the unit provided by the user
    unit = sys.argv[2]

# Get the numeric distance (cast string to float).
dist = float(sys.argv[1])

# Perform conversion.
mileList = ['mi','mi.','mile','miles']

if unit.lower() in mileList:
    output=dist*1.6
    print '{0} {1} is equivalent to {2} kilometers(s).'.format(
                                            dist, unit, output)
else:
    output = dist*.62
    print '{0} {1} is equivalent to {2} mile(s).'.format(
                                        dist, unit, output)
```

When handling several user input cases, it's a good idea to test an example of each input. Table 9.5 shows test cases for 'distanceConvert2.py'. The last test case (2.2 bananas) gives an example of how the script could be made more robust. This improvement is left as an exercise.

Table 9.5 Five test cases for Example 9.13.

Input	Output
(No arguments)	Usage: numeric_distance {distance_unit (mi or km)} Example: 5 km
2.3	Warning: No distance unit provided. Assuming input is in miles 2.3 miles is equivalent to 3.68 kilometers
5 km	5.0 km is equivalent to 3.1 miles
10 MI	10.0 MI is equivalent to 16.0 kilometers
2.2 bananas	2.2 bananas is equivalent to 1.364 mile(s).

9.5 Creating Output Directories

Sometimes we want to store output files in a different workspace than the input file workspace. For example, to create a backup copy of files using the same name in another directory or to group a batch of output files in a directory on their own, we need to send the output to another directory. In some cases, the output directory may exist; in other cases, the script might create it for us. If this isn't handled carefully, the script could try to write output in a directory that doesn't exist or it could try to create a directory that already exists. Either of these would cause the script to fail. `os` module functions can be used to avoid these problems. The `os.path.exists` function checks if a file or directory exists and the `os.mkdir` creates a new directory as shown in the following code:

```
>>> myDir = 'C:/gispy/data/ch09/happyGoat'
>>> os.path.exists(myDir)
False
>>> os.mkdir(myDir)
>>> os.path.exists(myDir)
True
>>> os.mkdir(myDir)
Traceback (most recent call last):
File "<interactive input>", line 1, in <module>
WindowsError: [Error 183] Cannot create a file when that file already
exists: 'C:/gispy/data/ch09/happyGoat'
```

Calling `os.mkdir` on an existing directory causes an error. To avoid this error, use these two functions together in a conditional construct and the `not` logical operator as follows:

```
if not os.path.exists(myDir):
    os.mkdir(myDir)
```

If the directory exists, it won't call `os.mkdir`. Place these lines of code in the script before the output directory is used, as shown in Example 9.14. This example uses `arcpy.env.workspace` to set the location of the file to be copied. But it uses a full path file name for the output file. The call to `os.path.join` creates this full path name by joining the output directory path with the name of the file being copied. In this way, a duplicate of the original with the same name is created in the backup directory.

The `os` module commands work for directories, but not necessarily for some specialized Esri structures. `arcpy` has equivalent commands for these structures which can be used in the same way. The following code uses `arcpy` commands to create a file geodatabase if it doesn't already exist:

```
>>> outPath = 'C:/gispy/data/ch09/'
>>> gdbName = 'happyHorse.gdb'
```

```
>>> if not arcpy.Exists(outPath + gdbName):
...     arcpy.CreateFileGDB_management(outPath, gdbName)
...
<Result 'C:/gispy/data/ch09\\happyHorse.gdb'>
```

The Create File GDB (Data Management) tool requires the geodatabase path and geodatabase name as two separate parameters; Otherwise, the approach is the same.

Example 9.14

```
# copyFilev2.py
# Purpose: Copy a file.
# Usage: source_directory destination_directory file_to_backup
# Example: C:/gispy/data/ch09/ C:/gispy/scratch/backup park.shp
#          The example works even if the C:/gispy/scratch/backup
#          directory doesn't exist yet.

import arcpy, os

arcpy.env.workspace = arcpy.GetParameterAsText(0)
outputDir = arcpy.GetParameterAsText(1)
fileToCopy = arcpy.GetParameterAsText(2)

if not os.path.exists( outputDir ):
    os.mkdir( outputDir )

outputFile = os.path.join(outputDir, fileToCopy)
arcpy.Copy_management( fileToCopy, outputFile )

print 'source =', os.path.join(arcpy.env.workspace, fileToCopy)
print 'destination =', outputFile
```

Printed output:

```
source = C:/gispy/data/ch09\park.shp
destination = C:/gispy/scratch/backup\park.shp
```

9.6 Key Terms

`if`, `elif`, `else`, Python keywords
Conditional constructs
Boolean expressions
Conditional expressions
Compound conditional expressions
The `os.path.exists` function
The `os.mkdir` function
The `CreateFileGDB` tool

9.7 Exercises

General Instructions: The `Describe` method, along with other geoprocessing tools can be useful for performing batch processing on a geodatabase or a directory. The next discussion covers batch processing. For now, we can perform some fundamental actions that can be extended with batch processing.

1. **conditionalSound.py** Inspect the sample script 'conditionalSound.py' shown below. Then predict what sounds will play with each of these input conditions:

   (a) No arguments.
   (b) C:/gispy/data/ch09/park.shp
   (c) C:/gispy/data/ch09/tree.gif
   (d) C:/gispy/data/ch09/jack.jpg
   (e) C:/gispy/data/ch09/xy1.txt

   Write down your answers and then verify them by running the sample script 'conditionalSound.py'. Did you get them all correct? Explain any mistakes you made.

conditionalSounds.py

```
import winsound, os, sys, time, arcpy

scriptPath =  os.path.abspath(__file__)
mydir = os.path.dirname(scriptPath) + '/'

def playSound(soundfile):
    # Function to play the input sound file using the winsound module.
    winsound.PlaySound(soundfile, winsound.SND_FILENAME|winsound.SND_ASYNC)
    # Wait 1.5 seconds
    time.sleep( 1.5 )

# If sys.argv list length is one, no argument was passed.
if len(sys.argv) < 2:
    playSound( mydir + 'wah_wah_wah.wav' )

else:
    fileName = sys.argv[1]

    # Get the describe object
    desc = arcpy.Describe(fileName)
    dataType = desc.dataType

    if dataType == 'RasterDataset':
        playSound( mydir + 'haha.wav' )

        if desc.Format == 'GIF':
            playSound( mydir + 'pukpukpuk.wav' )

        else:
            playSound( mydir + 'doh.wav' )

    elif dataType == 'ShapeFile':
        playSound( mydir + 'oh.wav' )


playSound( mydir + 'yeehaa.wav' )
```

2. For each of the sample scripts named in bold, modify the script as described in each part while maintaining the same functionality as the original version of the script.

   (a) **cond1.py** Use `if`, `elif`, and `else` keywords, to replace the sequential if statements.
   (b) **cond2.py** Remove the unnecessary conditional branch.
   (c) **cond3.py** Rewrite the nested conditional as a compound conditional.

3. **box.py** Write a script that determines if a point is inside of the box bounded by the points $(0,0)$ and $(1,1)$. The script should take two required arguments, an x and a y coordinate. Hint: Remember that script arguments are strings.

Input	Output
0.2 0.4	(0.2, 0.4) is in the box.
–0.1 0.5	(–0.1, 0.5) is outside the box.
0 0	(0.0, 0.0) is in the box.

4. **ski.py** Season pass fees for a North Carolina ski resort are as listed in the table below. Write a script that takes the age of the skier as input (one required argument) and prints the season pass fee as output. Use conditional `if`, `elif`, and `else` keywords in this simple script. Hint: Remember that script arguments are strings.

Age range	Season pass fee ($)
Ages 6 and under	30
Ages 7–18	319
Ages 19–29	429
Adult (ages 30 plus)	549

5. **describeData.py** Write a script to take one required argument, the full path file name of a dataset as input. Using conditional constructs along with the `Describe` object, the script should determine if the `dataType` is `ShapeFile` or `RasterDataset`. In the first case, it should print the `shapeType`; in the second, it should print the format. If it is neither of these, it should print the `dataType`. Example input and resulting output:

Input	Output
C:/gispy/data/ch09/park.shp	Polygon
C:/gispy/data/ch09/tree.gif	GIF
C:/gispy/data/ch09/xy1.txt	TextFile

6. **gridToPoly.py** Given a full path file name of a GRID raster, sample script 'gridToPoly.py' converts GRID format raster datasets to polygon feature classes. It also handles three other user input scenarios. Give four input condition that will test each branch of the code. Use sample data in the 'C:/gispy/data/ch09' directory and fill in the input-output table to list the input and the corresponding printed output:

Input	Output

7. **tableSelect.py** Practice using an ArcGIS tool that makes a selection. The park cover types in 'park.shp' include 'woods', 'orch', and 'others'. Fire risk experts would like a table including only those polygons with a cover type of 'woods'. Write a script to perform a Table Select that will generate this table and name the output table 'wooded.dbf'.

8. **currency.py** Write a script that converts between US dollars and Euros. (Use a rate of 1 US = 0.7 E for this exercise.) The script should take one numerical required argument and one optional argument (the currency, E or US). If the user gives two arguments, perform the conversion. If the user gives only one argument, a number, assume the number is given in US dollars, warn the user, and perform the conversion. If the user gives no arguments, print a statement explaining how to run it. Examples of test cases and resulting output are shown below:

Input	Output
45.32 E	45.32 Euros is equivalent to 64.74 US Dollars
55	Since you didn't specify a currency, I'm assuming US to Euros. 55 US Dollars is equivalent to 38.5 Euros
100 US	100.00 US Dollars is equivalent to 70.00 Euros
(No arguments)	Usage: number {currency (US or E)} Example: 100 US

9. **temperatureConvert.py** Write a temperature conversion script to convert between Celsius and Fahrenheit, using the following equations for conversion:

   $F = 1.8 * C + 32$

   $C = (0.56) * (F - 32)$

   The script should take one numerical required argument and one optional argument (the scale, F or C). If the user gives two arguments, perform the conversion. If the user gives only one argument, a number, assume the number is given in Fahrenheit, warn the user, and perform the conversion. If the user provides no arguments, print a statement explaining how to run it. Several test cases and resulting output are shown below:

Input	Output
32 F	32 Fahrenheit is equivalent to 0.0 Celsius
100 C	100 Celsius is equivalent to 212.0 Fahrenheit
55	Since you didn't specify a scale, I'm assuming F to C. 55 Fahrenheit is equivalent to 12.88 Celsius
(No arguments)	Usage: number {unit (C or F)} Example: 32 F

10. **distanceConvertv3.py** Sample script, 'distanceConvertv2.py', only tests if the input lower-cased unit is one of the four strings in the miles list (['mi', 'mi.', 'mile', 'miles']). Any other unit is assumed to stand for kilometers. Input of '2.2 bananas' gives output of '2.2 bananas is equivalent to 1.364 mile(s)'. Modify the script so that if the lowercased unit is not in the miles list, nor the kilometers list (['km', 'km.', 'kilometer', 'kilometers']), the script prints a warning and does not perform a conversion. Example input and resulting output:

Input	Output
(No arguments)	Usage: numeric_distance {distance_unit (mi or km)} Example: 5 km
2.3	Warning: No distance unit provided. Assuming input is in miles 2.3 miles is equivalent to 3.68 kilometers
5 km	5.0 km is equivalent to 3.1 miles
10 MI	10.0 MI is equivalent to 16.0 kilometers
2.2 bananas	Warning: unit must be either miles or kilometers

11. **compactBranch.py** Write a script that takes one argument, a full path dataset file name, as input. If the `dataType` is 'Workspace' and the workspaceType is 'LocalDatabase', it compacts the file, and reports the file size before and after compacting. Otherwise, it warns the user that it could not perform the compact operation. Example input and resulting output:

Input	Output
C:/gispy/data/ch09/cities.mdb	File size BEFORE compact: 1552384 File size AFTER compact: 397312
C:/gispy/data/ch09/xy1.txt	Input data must be a personal or file geodatabase. Could not compact: xy1.txt

12. **bufferBranch.py** Write a script to take a required full path file name argument and an optional buffer distance argument. Perform Buffer (Analysis) on the file only if it is a Shapefile type (If the filename ends with '.shp', assume it is a valid shapefile). If it's not a shapefile, warn the user that it will not buffer the file. Use the input buffer distance, if it is given; otherwise, use a default buffer distance of 1 mile. Place the output in the same directory as the input, and report the output file name. Example input and resulting output:

Input	Output
C:/gispy/data/ch09/park.shp	No buffer distance given. Used default buffer distance of 1 mile Buffer output: C:/gispy/scratch/parkBuffer.shp
C:/gispy/data/ch09/xy1.txt "8000 meters"	Input data format must be shapefile. Could not buffer input file C:/gispy/scratch/xy1.txt
C:/gispy/data/ch09/park.shp "200 Meters"	Buffer distance: 200 Meters Buffer output: C:/gispy/scratch/parkBuff.shp