Chapter 1

Using Python in labeling and field calculations

Introduction

As you begin working with Python as a programming language and start incorporating Python scripts into ArcGIS, you will find that there are many places where Python code can be used. This use may be as a small code snippet as demonstrated in this chapter or in fully developed programs as you will see in later chapters. For these first tutorials, take extra time to research the various Python and ArcGIS components and the structure of the code. As the projects become more complex, you will appreciate understanding the basics of this type of programming.

Tutorial 1-1 Python introduction and formatting labels

Python code can be used in places other than fully developed scripts. The Label Expression dialog box in ArcGIS allows you to insert code to control labels on your map.

Learning objectives

- Basics of Python
- Text formatting
- Variable manipulation

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "What is Python?"
- "Building label expressions"

Introduction

Python is a powerful scripting and programming tool, but you need to know the basic rules of the game before you start playing. This tutorial presents a summary of the components most commonly used in ArcGIS. You can reference the Python documentation online at http://www.python.org and other Python reference books, such as *Python Scripting for ArcGIS* by Paul A. Zandbergen (Esri Press, 2013), for full descriptions and more advanced tools. Also, research the ArcGIS-related tool you will be using in ArcGIS for Desktop Help, where you will find descriptions of the tools and code samples that can be used to better understand the tool's usage.

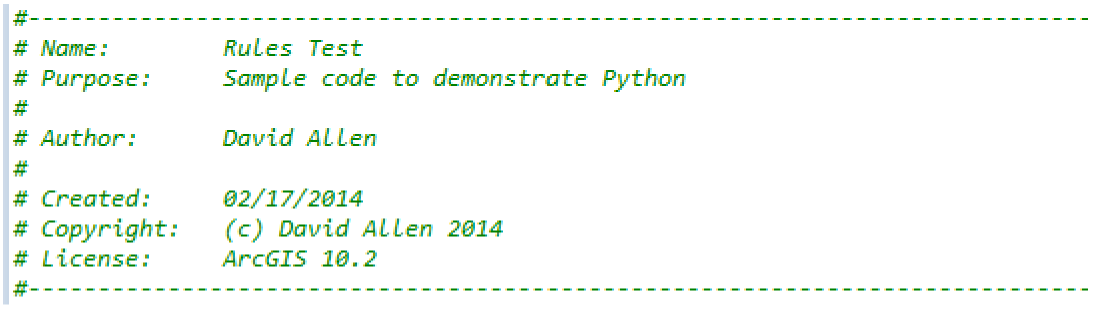
Here are some basic rules for Python:

- Python code runs in a linear fashion—from top to bottom.
- Python includes variables, which can contain a variety of data types, including numbers, strings, lists, tuples, and objects (with properties).
- Variable types (e.g., numeric, string, list, date) do not need to be declared—Python determines the variable type based on the input.
- Variable names are case sensitive—"myFeatureClass" is not the same as "myfeatureclass."
- Either single or double quotation marks can be used when creating string-type variables—the Python code interpreter does not care, so "myFeatureClass" is the same as 'myFeatureClass.'
- Indentation in Python is important. Indenting is a way to group tools and operations into a set of code within your script, such as a code block associated with an if or while statement. Indentations are typically two spaces or four spaces; you can use tabs, but do not mix tabs and spaces.

The next few steps will let you practice some of these rules before you tackle the first tutorial.

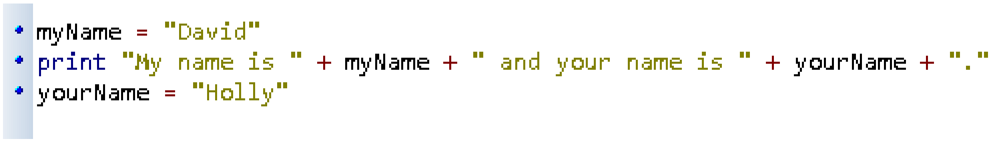
1. **Open your integrated development environment (IDE), and start a new script.**

The example shown in the graphic is a modified PyScripter template for ArcGIS that includes the name of the script, the author, a script description, the date of creation, and the license level that this script might require. Information on setting up this template in PyScripter is found in appendix A. Note that these lines are preceded by a hashtag, which denotes them as comments and not code that can be run.



```
#-------------------------------------------------------------------------------
# Name:        Rules Test
# Purpose:     Sample code to demonstrate Python
#
# Author:      David Allen
#
# Created:     02/17/2014
# Copyright:   (c) David Allen 2014
# License:     ArcGIS 10.2
#-------------------------------------------------------------------------------
```

2. **Type the code as shown:**



```
myName = "David"
print "My name is " + myName + " and your name is " + yourName + "."
yourName = "Holly"
```

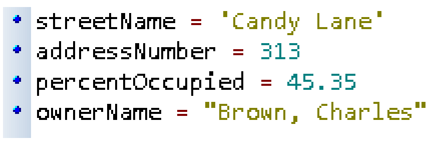
This code creates a couple of variables and prints them to the IDE code window. Note that the variable is created simply by using the equals sign, and the various parts are brought together in the print statements using the plus sign. This is called *concatenation*, which basically creates one line of text out of all the components.

If you were to try and run this code, you'd get an error. Why? Python runs these lines in order, from top to bottom. The print statement is run before the second variable is defined.

3. **Change the order of the statements so that they will run correctly. Save the script for future reference, and then run the script.**

Remember that Python runs from top to bottom, so the lines of code must be in the correct order.

4. **Type the code shown in the graphic to use different types of formatting to create four variables:**



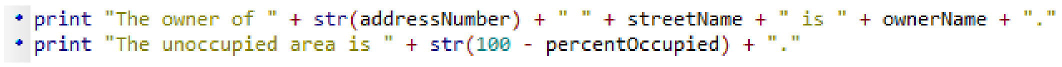
```
streetName = 'Candy Lane'
addressNumber = 313
percentOccupied = 45.35
ownerName = "Brown, Charles"
```

Note the format of the variable names. The names are descriptive of what they contain, start with lowercase letters, and use uppercase letters to distinguish words within the name. This is called *camel case*. Although this format is not required, it is standard in the ArcGIS for Desktop Help sample code.

Two of the example variables shown are strings (text), and each of them uses a different style of quotation marks. Both styles can be used, and both are considered regular strings. Note that the numbers are also different. One has decimal points, and one does not, but both are still interpreted in Python as numeric. The IDE shows these lines in different colors so that you can tell the number variables from the string variables.

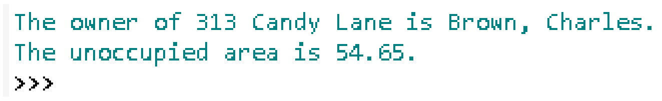
Expressions can be used to concatenate the strings and to perform math on the numbers. The graphic in step 2 shows an example of concatenating string variables with the plus sign. Numbers can be concatenated into these types of sentences as well, but numbers must first be converted to strings using the string formatting method, .str(). Because the IDE colored the numbers differently, you can easily tell when a conversion is necessary.

5. Type the code shown in the graphic, and then run your script to see the results.



```
print "The owner of " + str(addressNumber) + " " + streetName + " is " + ownerName + "."
print "The unoccupied area is " + str(100 - percentOccupied) + "."
```

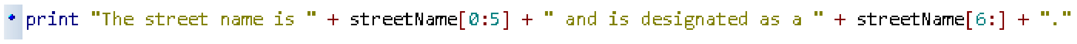
The strings are concatenated together, and the number is added once it is converted to a string. Note that in the second line, there is some math occurring inside the string conversion function. This calculation is fine as long as the result is converted to a string. Also, pay attention to where the extra spaces are added to the text to make the sentence appear correct when printed.



```
The owner of 313 Candy Lane is Brown, Charles.
The unoccupied area is 54.65.
>>>
```

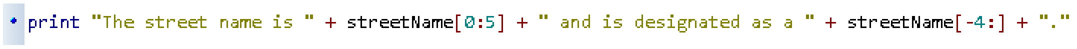
It is also possible to slice characters from a string variable. Each character in the variable is automatically assigned an index number, starting at the left with zero (0). You can count over to the characters you want, and then slice those characters from the string. To slice characters, add square brackets at the end of the variable ([]), and then inside the brackets, add the starting index number, a colon, and the ending index number.

6. Type the print statement shown in the graphic to slice only the street name from the streetName variable.



```
print "The street name is " + streetName[0:5] + " and is designated as a " + streetName[6:] + "."
```

The first slice gets all the characters starting at index number 0 over to but not including index number 5. The second slice gets all the characters starting at index number 6 and over to the end. The indexes can also be counted with negative numbers from the end of the string. The word *Lane* could also have been sliced using this statement, which gets all the characters starting four back from the end and proceeding to the end, as shown:

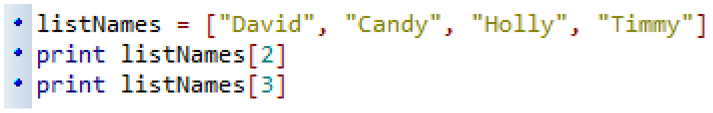


```
print "The street name is " + streetName[0:5] + " and is designated as a " + streetName[-4:] + "."
```

A common use of the negative slice is to remove the file extension from the end of a file name. The example in the previous graphic, using -4, would remove .shp from the end of a file name, regardless of its length.

Another type of variable is a list. List variables can contain many values and are used extensively in ArcGIS to hold lists of feature classes, file names, and workspaces. The individual values within the string are accessed by using an index number. Each value is given an index number, starting at 0. For example, a variable with eight list items would have index numbers from 0 to 7.

7. **Type the code shown in the graphic to assign a list to a variable, and then print one of the values using its index number.**



```
listNames = ["David", "Candy", "Holly", "Timmy"]
print listNames[2]
print listNames[3]
```

Remember that the first value is given index number 0, so this code will print the names Holly and Timmy.

These are simple examples of creating and manipulating variables. More complex manipulations follow in other sections of the book. Try some of these things in ArcMap.

8. **Close the IDE you've been using and save the file for future reference.**

These text manipulations can be used in various parts of ArcGIS, and this tutorial examines using them in a labeling expression. The interface you use for labeling features in a map layout will recognize Python script and allow you to do formatting and make on-the-fly changes to the text that you may be using from a layer attribute. This feature may save you a lot of time when the attribute values are not formatted exactly as you like or when your data has to meet a standard that is different from other datasets you may be using.

Scenario

A standard ArcMap layout is used by your department to review the owners of properties in the fictitious City of Oleander, Texas. The formatting is not the best, but it is functional for your internal use. Recently, the city manager asked you to make more of this type of data available through online maps, and the formatting is not appropriate. You should explore some ways to use Python scripting to make the text more presentable.

Data

The data is the parcel and ownership data for the City of Oleander, Texas, a small community in the Oleander/Fort Worth Metroplex (O/FW). A map document is set up to display data for each property from the owner name field.

SCRIPTING TECHNIQUES

The Label Expression dialog box has a Python function built in to make it easier to use the code. A function is basically a set of code that can receive one or more values and return values back to the program that called it. In the case of the label expression, the built-in function is named FindLabel() (which you should not change). This function appears on a line starting with *def*, meaning that it defines the function. The function may also have one or more field names at the end appearing in parentheses. These field names are passed to the code of the function so that you can use and manipulate their values. Finally, a return statement returns a value to the program that called the function, which in this case is the label expression. For labels, make sure to return a single value.

The Label Expression dialog box lets you use a variety of Python text formatting methods, including the following:

- .capitalize()—makes the first character of the string a capital letter
- .find(X)—finds the specified character in the string
- .isdigit()—returns the value of True if the variable is alphanumeric
- .lower()—makes the entire string lowercase
- .lstrip()—strips any spaces from the left end of the string
- .replace(X,Y)—replaces one character with another in the string
- .rstrip()—strips any spaces from the right end of the string
- .strip()—strips spaces from the start and end of a string
- .title()—capitalizes the first character of every word in a string
- .upper()—makes the entire string uppercase

Check your Python reference books for more string and numeric handlers for variables. Some are obscure, but you never know when they will be useful.

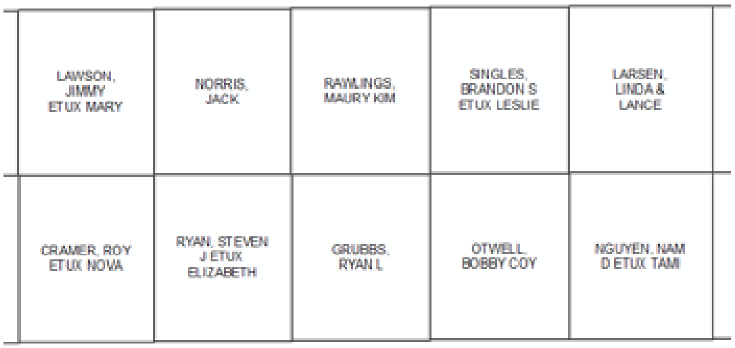
Use Python in the Label Expression dialog box

1. **Using the information provided, write the sequence of the calculation in nontechnical language, as shown in the following list. Then use this sequence to decide the formal structure of the code. The goal is to use Python string-handling techniques to dress up the labels on this map.**

   - **Change the text to read as a regular name (first, then last).**
   - **Add an ampersand (&) if a couple owns the house.**
   - **Change the text to upper- and lowercase rather than all uppercase.**

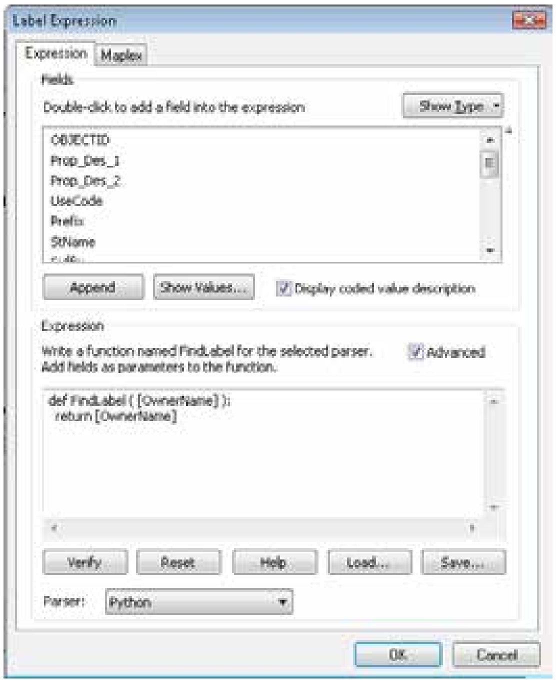
   This informal description of the processes your code will complete is the pseudo code. Even simple pseudo code is useful, and as you tackle more complex projects, your pseudo code will reflect this complexity.

2. **Start ArcMap, and open the map document Tutorial 1-1 from the location where you installed the book's sample data and exercises.**



The map in the graphic shows the Elm Fork subdivision with a label on each property that displays the owner's name. Although this format is acceptable, the map would look better if the text were formatted to show the first name followed by the last name. The current format of the data is "last name–comma–space–first name(s)." This data can be sliced into separate pieces to show it as "first name(s)–space–last name." This slice involves using the .find() method on the string, which returns the index number of the character you will search for, and then using that location in a slice operation. If you can find the location of the comma, everything before the comma is the last name, and everything two characters past the comma (remember the space) is the first name(s).

3. **Open the properties of the Elm Fork Addition layer, and click the Labels tab. Then click the Expression button. It shows the expression as [OwnerName]. At the bottom of the Expression box, click the Parser arrow, select Python, and then select the Advanced check box in the middle of the dialog box on the right side.**



Double-click to add a field into the expression

OBJECTID
Prop_Des_1
Prop_Des_2
UseCode
Prefix
StName

☑ Display coded value description

Expression

Write a function named FindLabel for the selected parser.
Add fields as parameters to the function.

☑ Advanced

```
def FindLabel ( [OwnerName] ):
  return [OwnerName]
```

Parser: Python

This step automatically enters the first few lines of Python code to define the FindLabel() function. At the end of this line is a colon (:). All the code that is to be evaluated as part of this function will appear below this line and be indented, ending with a return statement that sends a value back to the FindLabel() function and ultimately onto the map. The standard indentation in Python is four spaces, but the code here uses only two. Next, create a new line of code, indent the code two spaces, and use the .find() method to determine the index number of the comma in each value.

4. **Place your cursor after the colon, and press Enter. Add two spaces, and then type the following:**

   rawName = [OwnerName]

   It is not always necessary to put the field name into a variable, but it will make the process easier to understand.

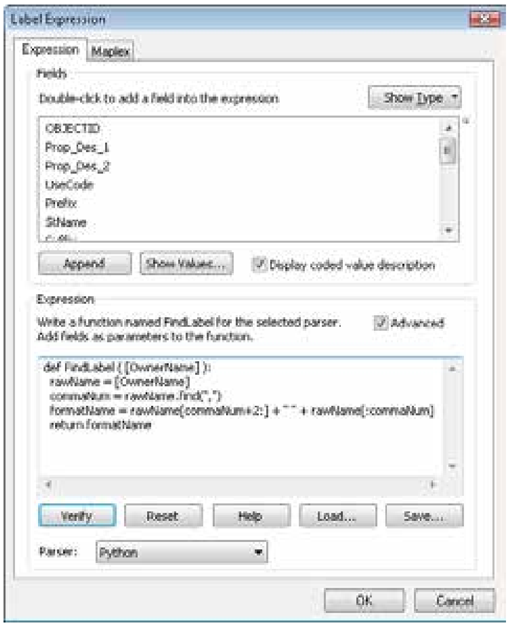
5. **Press Enter, add two spaces, and type the following:**

   commaNum = rawName.find(“,”)

   The value of commaNum is now equal to the index number of the comma. See if you can write the code to format the string correctly, and store it in a variable named **formatName**. Here is what needs to happen:

   Slice out everything starting at two characters past the comma over to the end of the string, add a space, and add the slice of characters from the start of the string over to but not including the comma.

6. **After the line of code, press Enter, and add two spaces. Type the code to perform the field formatting. Then replace the expression [OwnerName] in the last line with** formatName**, as shown. Click OK, and close the layer properties to see the result.**



Write a function named FindLabel for the selected parser.
Add fields as parameters to the function.

```
def FindLabel ( [OwnerName] ):
  rawName = [OwnerName]
  commaNum = rawName.find(",")
  formatName = rawName[commaNum+2:] + " " + rawName[:commaNum]
  return formatName
```

Parser: Python

Notice how you can do math in the slice command. A few more things still need to be corrected. The ownership files use the Latin abbreviation ETUX (et ux) in front of a spouse's name. Replace that with an ampersand using the .replace() method.

7. **Open the layer properties, and modify the expression by adding the .replace() method at the end of the rawName statement, as shown:**

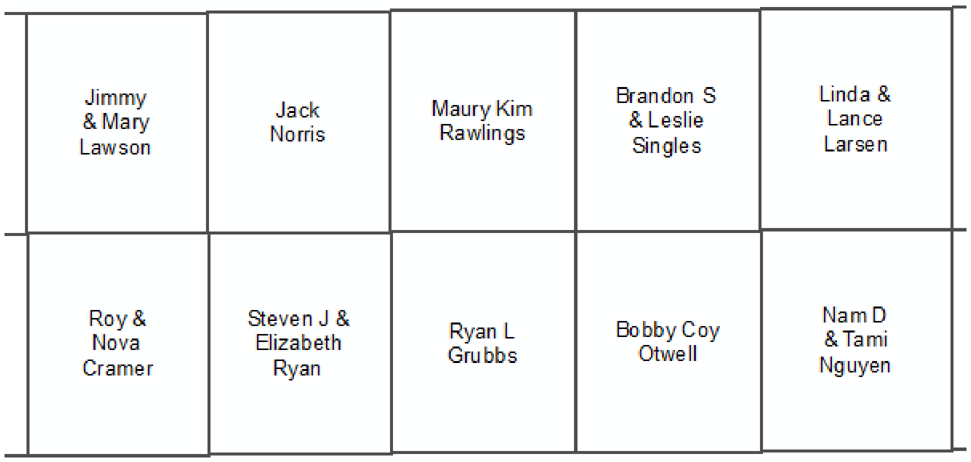
```
rawName = [OwnerName].replace("ETUX","&")
```

8. **Add the .title() method at the end of the return statement to make the names appear in upper- and lowercase, as shown:**

```
return formatName.title()
```

The resulting labels show the first and last names in upper- and lowercase, along with ampersands in place of the combined wording ETUX.

9. **When your code matches the graphic, click the Save button, and save the code you have written to a calculation (.cal) file for later reference. Then click OK to run the script and perform the calculation.**



10. **Close the map document.**

Exercise 1-1

Open the map document Exercise 1-1, a map of the subdivisions in Oleander with their names displayed. The city manager needs to show this map to another city and wants you to dress it up a little. First, the names should not have .PDF at the end, and second, the names should be in upper- and lowercase with the first letter of each word capitalized.

Write the steps needed to accomplish this task, including the Python code to use for the labeling. Then apply the code to the labeling expression in the map to reformat the text.

Tutorial 1-1 review

This code performed a lot of string handling with different variables. Note how you are able to find specific characters in a variable, store the index number, and use that number to slice the string in different ways. There may be times when this gets tricky, such as when an address number has an apartment or unit number that is a letter character (e.g., 304 A Pine St.). Looking for the first space would not give you the correct address number. You could look for the instance where a character stands alone—in other words, any single character with a space on both sides. To give a more complicated example, what if you had single-letter street names, such as D Street or P Street? The stand-alone character is the street name and not part of the address number. To make it even more confusing, Galveston, Texas, has half streets, such as P½ Street. This situation may require some complex Python coding to solve.

Study questions

1. Will finding the first space always identify the address number? Could you find all the spaces? Write some code to find the first stand-alone character in a string (a single character with a space on both sides).
2. Can strings and numbers be combined in Python? Write code to concatenate a text string, such as "Miles to go are," with a numeric variable, such as 102.
3. Give examples of when you might use the .upper(), .title(), and .lower() methods.

Tutorial 1-2 Decision making in the Label Expression dialog box

Labels can be simple displays of data from an attribute table, or with some Python coding, labels can be used to show values that are not in the table.

Learning objectives

- Defining functions
- Using if-elif-else logic for decision making
- Changing the label display text

Preparation

Research the following topic in ArcGIS for Desktop Help:

- "Using If-then-else logic for branching"

Introduction

In tutorial 1-1, you learned how to use Python code to control labels in a map document using various text-handling techniques. Those techniques involved simply arranging and reformatting the existing values of the fields used for labeling. You can also create labels that use entirely different text strings from the fields—it all depends on what you send back to the label expression with the return command.

In this tutorial, you will apply decision-making techniques to the labels in ArcMap. You will be able to pull the value out of the field and use it to determine what the label should contain.

Scenario

A file has been provided by the planning office of Oleander. This file needs a map to display at City Council meetings that shows the general zoning categories of Oleander. You have a good dataset for this, but the zoning is shown as code rather than a text description. You must use some decision-making tasks to turn the codes into the appropriate labels.

Data

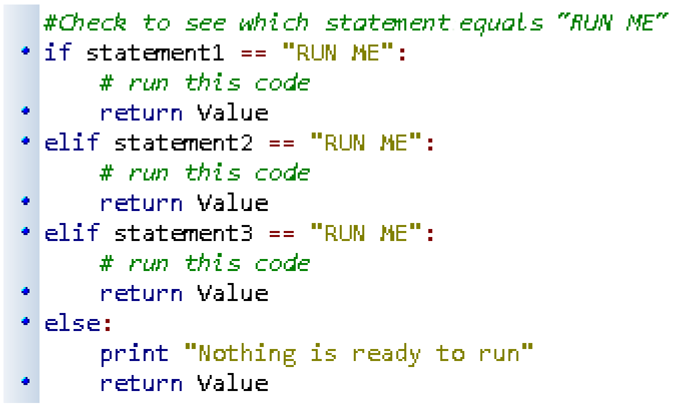
You are provided with the zoning map for the City of Oleander containing a layer with the zoning district polygons named General Zoning Districts. This layer has a field named Code, which contains a coded value for zoning. The codes are as follows:

RES = Residential
MF = Multi-Family
SPEC = Special District
C = Commercial
I = Industrial

SCRIPTING TECHNIQUES

You will need to build a condition statement using the if statement. This book covers if statements in more detail in later chapters, but for now, here is a quick introduction.

Every programming language has some version of the if statement to perform branching based on a condition being true, and Python is no exception, providing one of the easiest if formats. The first if statement tests a condition that can evaluate to either True or False, along with a set of code to run if the statement is true. This statement can be followed by any number of elif statements to run if the condition is tested to be false—and the elif condition statements are all evaluated in sequence until one is true. Finally, an else statement is provided with no condition to be met and is run if all the other conditions are false. A basic decision-making statement looks like this:



```
#Check to see which statement equals "RUN ME"
if statement1 == "RUN ME":
    # run this code
    return Value
elif statement2 == "RUN ME":
    # run this code
    return Value
elif statement3 == "RUN ME":
    # run this code
    return Value
else:
    print "Nothing is ready to run"
    return Value
```

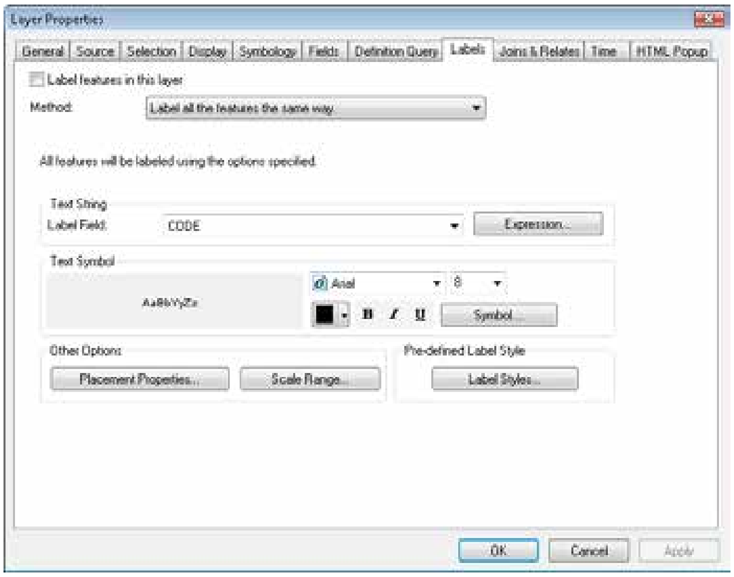
Note that the evaluator is a double equals sign (check your Python reference for other evaluators such as the greater than symbol or the lesser than symbol), and the code for each if, elif, or else statement is kept indented until the next statement. This format designates which lines of code to run if the statement evaluates to True. The return statement tells the code what to send back to ArcMap—in this case, the text for the label.

Use if-elif-else statements in the Label Expression dialog box

1. **Write the pseudo code for this project:**

- Get the code value from the field.
- Determine what text to use for each code value.
- Return the text string to the label expression.

2. Start ArcMap, and open the map document Tutorial 1-2. Right-click the General Zoning Districts layer, and open the properties. If necessary, click the Labels tab as shown, and set the Label field to CODE.



Label features in this layer

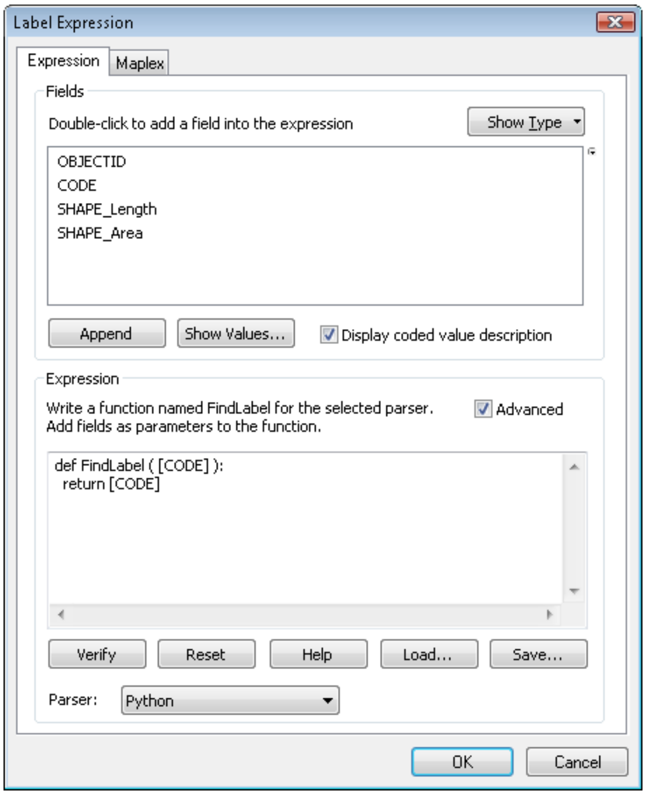
Method: Label all the features the same way.

All features will be labeled using the options specified.

Label Field: CODE

Arial 8

3. Click the Expression button. Do you remember how to set this to accept Python code? (Hint: set the Parser to Python and click Advanced.)



Double-click to add a field into the expression

OBJECTID
CODE
SHAPE_Length
SHAPE_Area

[✓] Display coded value description

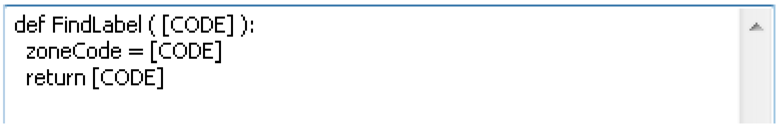
Write a function named FindLabel for the selected parser.
Add fields as parameters to the function.

[✓] Advanced

```
def FindLabel ( [CODE] ):
  return [CODE]
```

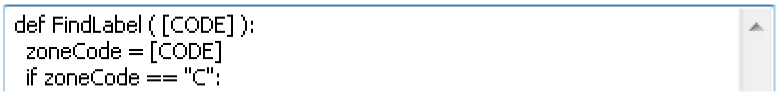
Parser: Python

4. The framework is ready for the code to be entered. First set a variable to equal the zoning code that is brought into the script by the function. Add two spaces before the variable name to keep the indentation, as shown:



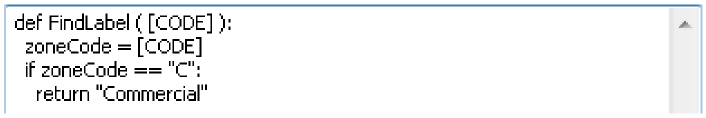
```
def FindLabel ( [CODE] ):
  zoneCode = [CODE]
  return [CODE]
```

5. Construct the if statement. Remove the line "return [CODE]" because this will be replaced within the if statement. Add a new line, indent two spaces, and add the if statement, remembering to add the colon at the end of the line, as shown:



```
def FindLabel ( [CODE] ):
  zoneCode = [CODE]
  if zoneCode == "C":
```

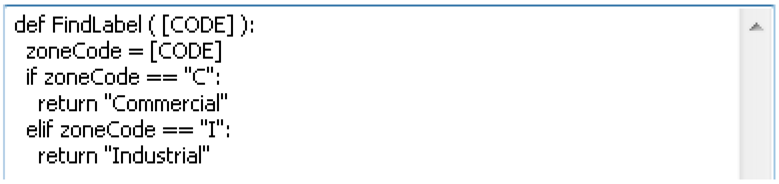
6. Add another line, and indent four spaces. Everything that keeps the four-space indentation will be evaluated when this if statement evaluates to True. Add a return statement to tell ArcMap what to use as a label, as shown:



```
def FindLabel ( [CODE] ):
  zoneCode = [CODE]
  if zoneCode == "C":
    return "Commercial"
```

Next, add code to handle what happens when the if statement is not true—you will set up the next label. Python makes checking multiple conditions easy with the use of an elif statement. Remember that this statement also gets a colon at the end of the line, and everything after the colon that is kept to a four-space indentation will run if the condition is true.

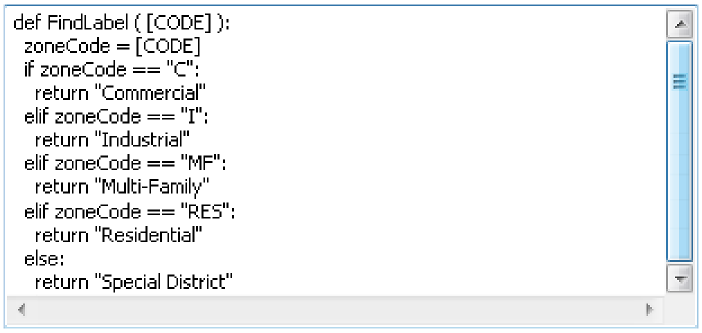
7. Add another line, indent two spaces, and type the elif statement; then add another line with a four-space indentation and the return statement.



```
def FindLabel ( [CODE] ):
  zoneCode = [CODE]
  if zoneCode == "C":
    return "Commercial"
  elif zoneCode == "I":
    return "Industrial"
```

Your turn

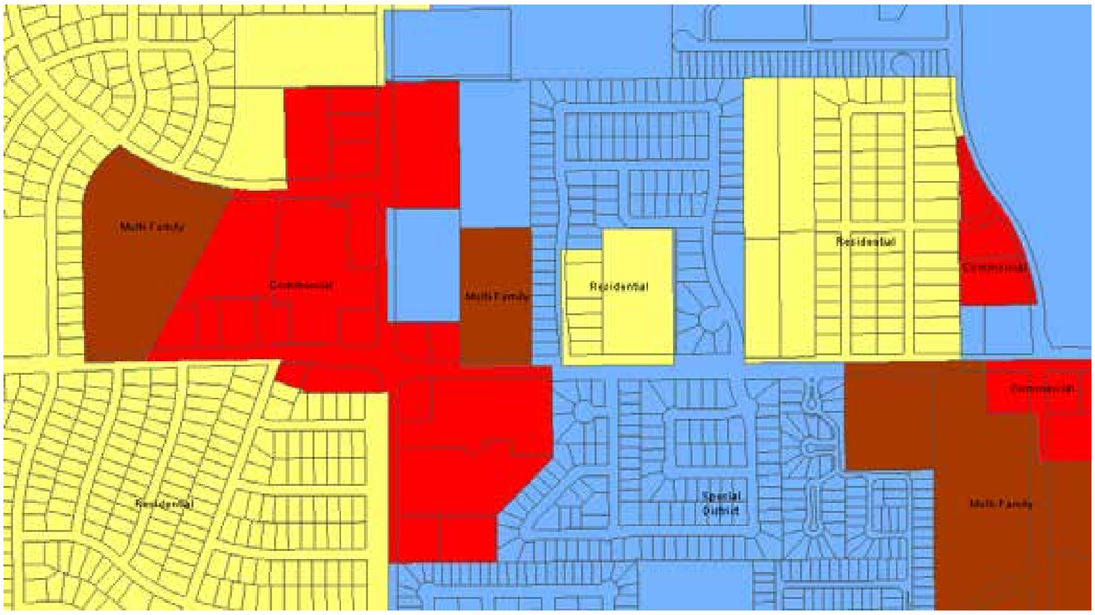
Add the rest of the code to test each of the remaining conditions. Remember that the last line can be an else statement with no condition. Try it from your own notes first before looking at the code in the next image. (**Hint:** *make sure to watch your case on variables; use the == evaluator; put a colon at the end of each line with an if, elif, or else statement but not after the return statement; and watch your indentations carefully).*



```
def FindLabel ( [CODE] ):
 zoneCode = [CODE]
 if zoneCode == "C":
  return "Commercial"
 elif zoneCode == "I":
  return "Industrial"
 elif zoneCode == "MF":
  return "Multi-Family"
 elif zoneCode == "RES":
  return "Residential"
 else:
  return "Special District"
```

8. **Click OK and then OK to run the code and create the labels.**

If you have an issue, go back over the code, and pay particular attention to the things noted in the “Your turn” hint.



You will see that your code has turned the zoning codes into a more descriptive label. It is important to remember that these labels are generated on the fly and used for this map only; these labels are not stored anywhere in the attribute table.

Exercise 1-2

The other map turned out so well that the planner wants a similar map for the comprehensive land development data. Open the map document Exercise 1-2, and use Python code in the label expression to turn the following USE_CODE values into descriptive text:

A1 = Single Family
B1 = Multi-Family
F1 = Commercial
F2 = Industrial
PRK = Parks
ROW = Right-of-Way
TX10 = Special Texas District

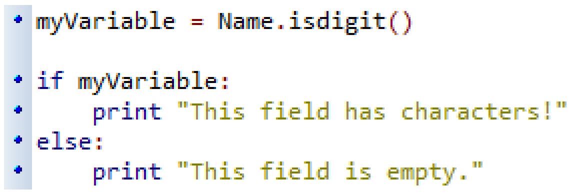
Write your code before attempting to type the expression in the Label Expression dialog box.

Tutorial 1-2 review

The if statement in Python is one of the easiest to use in the programming world. For any set of conditions, continue adding elif statements until you have handled all the possible conditions. It is good practice to put an else statement after the last elif statement just to handle a situation that is not the norm or that you might have forgotten about or to handle the last known condition without having to use a condition statement.

If statements can also be nested, but this is only done to test two entirely different sets of conditions. For instance, your first if statement might be to identify the land-use code, and you nest an if statement within that statement to determine whether the particular parcel fits within a certain range of acreage. However, you would not need to use nesting to test for more land-use conditions as you would with other programming languages. These conditions could easily be handled with additional elif statements, one for each additional land use.

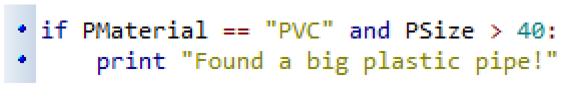
Another interesting note with if statements is that the condition need only return a value of True or False. If you are testing a variable that equates to True or False, you need only put the variable as the condition for the if statement. For instance, the .isDigit() function returns the values of True for a variable containing alphanumeric characters and False if the variable is empty. If you retrieved a variable named Name that contained characters, the code myVariable = Name.isdigit() would set myVariable to either True or False, depending on whether the field had an alphanumeric value. This situation would make the if statement look like this:



```
myVariable = Name.isdigit()

if myVariable:
    print "This field has characters!"
else:
    print "This field is empty."
```

The if condition statements can also test for more than one variable at a time, just like a query statement in ArcMap. The only trick is that there must be a combination of values that equate to True, or the code will never run. For instance, if you needed to find all the 40-inch or larger PVC pipe, the code would look like this:



```
if PMaterial == "PVC" and PSize > 40:
    print "Found a big plastic pipe!"
```

It does not matter that one condition tested a string-type value and the other tested a numeric value. Be careful with the condition though, because an incorrect AND or OR can send your code in an unwanted direction. If you are unfamiliar with the AND and OR handlers, check them out in ArcGIS for Desktop Help, and try practicing the statement as a definition query in ArcMap.

Study questions

1. When would you nest if statements, and when would you rely on elif statements? Write the code to find parcels that have a land-use code of A1, and note whether the acreage is less than two acres, from two to five acres, or more than five acres. Add additional code to find parcels with a land-use code of B1, and perform the same test for area.
2. How would you format a complex condition statement? Write the code to find employees over 50 who are retired if their age is stored in a field named *currAge*, and the field noting retirement status is a true/false field named *statusRetired*.
3. Where can you find more Python methods that deal with string and numeric variables? Write code to find all the employees with a last name containing more than 10 characters if their last names are stored in a field named *LastName* (or else it will not fit on the new engraved name tags).

Tutorial 1-3 Using Python in the Field Calculator

Snippets of Python code can be used in the Field Calculator to perform complex functions, including any of the text formatting commands shown in the Label Expression dialog box.

Learning objectives

- Text formatting with Python
- Using Python code blocks
- Concatenating text values

Preparation

Research the following topics in ArcGIS for Desktop Help:

- "Calculate field examples"
- "Fundamentals of field calculations"

Introduction

The first two tutorials show how to use Python code to alter the labeling in an ArcMap map document. This labeling made the maps look great, but remember that the changes occurred only in the labels and were not stored anywhere permanently.

To make a more permanent change to your data, you can use Python to calculate the values in fields. You would normally use the Field Calculator and a simple expression to set a field value, but there are limits to what you can accomplish with the simple expressions that are allowed. By using Python code in the Field Calculator, you can store the results for future use by yourself and others.

Scenario

You received some data from the Fire Department, and it would like you to format the addresses so that its analyst can geocode them and make a simple presentation map. To do the geocoding, the address needs to be in a single field, and right now that information is parsed out into five fields. You must write an expression in the Field Calculator to create a single address with the components in the right sequence, and without extra spaces.

Data

You are provided a file named FireRuns2010 with the calls for service data from the Oleander Fire Department. The file contains a variety of data about the type of call, the time it was received and dispatched, and which unit responded. The file also contains address information that has been split into five fields:

addNum = address number
stPrefix = street prefix
stName = street name
stType = street type
suffDir = street suffix direction

SCRIPTING TECHNIQUES

Using Python code in the Field Calculator is a little trickier than the labels expression because ArcMap does not automatically create the code for the function. You need to set the parser language to Python, and then select the Show Codeblock check box. This will expose two empty boxes where you will type your code.

The lower box, called the Expression box, will look familiar. This is the box that you normally work with for simple calculations and that you used to hold your label formatting code. Instead of using a regular statement, you will have the expression call a function from the code box described below. The syntax is to name the expression, which can be anything you like, but common practice is to begin the name with a lowercase *fn* to denote a function. Then in parentheses, add all the fields from the attribute table that you will be using in your code. It does not matter how many you use, and they will be separated by commas. You can add them by double-clicking the field name in the Fields list.

All the code work is done in the Pre-Logic Script Code box, or code box. The first line defines the function you named in the Expression box. It then has a set of parentheses and contains a variable name for each of the fields that you are sending to it. For instance, if your statement in the Expression box has five fields coming in, the function in the Pre-Logic Script Code box must set up five variables to accept them. The function ends with a colon, and all the code that follows must be indented. There is no automatic indenting here, so you must manually add spaces for indenting and then keep track of them.

Next, type all your Python code to process the data and perform the calculation, with the last line being the return statement. This code sends a value back to the Expression box, which is saved to the field. As with the label expressions, the use of if statements may result in several return statements.

Debugging this code can be problematic, but it is best to start by checking the indent levels. Then go back over the syntax of the code.

Use Python in the Field Calculator

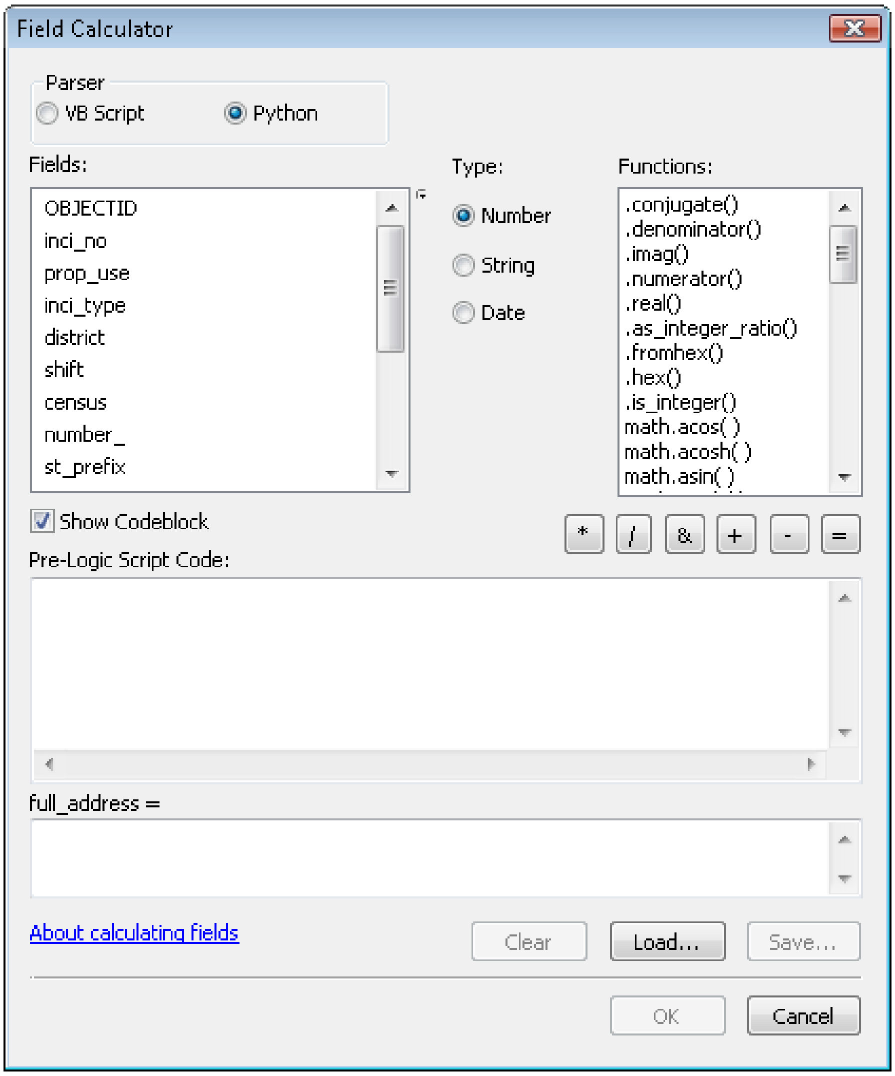
1. **Write pseudo code to describe the process for reformatting the field value and storing the result:**
    - Concatenate the values for address number, street prefix, street name, street type, and street suffix direction into a single value.
    - Store this value in the empty field full_address.

2. **Start ArcMap, and open the map document Tutorial 1-3. If necessary, switch to the List By Source view in the table of contents. Open the FireRuns2010 table.**

3. **Right-click the field full_address, and click Field Calculator. A warning about editing outside an edit session appears. Click Yes to continue.**

It can be dangerous to edit outside an edit session in ArcMap because there is no undo option. If you make a mistake, it is permanent. In this case, you are populating an empty field, so there is no risk of destroying any critical data by calculating the field.

4. **In the Parser box, click Python. Below the Fields list, select the Show Codeblock check box.**


Field Calculator

Parser: VB Script [✓ Python]

Fields:

OBJECTID
inci_no
prop_use
inci_type
district
shift
census
number_
st_prefix

Type: [✓ Number] String Date

Functions:

.conjugate()
.denominator()
.imag()
.numerator()
.real()
.as_integer_ratio()
.fromhex()
.hex()
.is_integer()
math.acos( )
math.acosh( )
math.asin( )

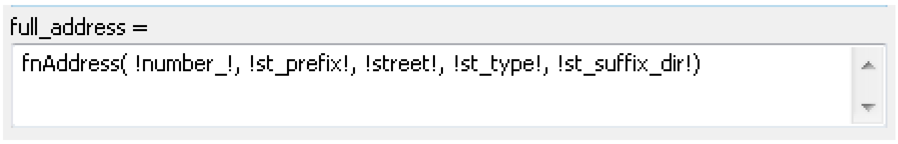
[✓] Show Codeblock

Pre-Logic Script Code:

full_address =

In the Expression box, you will create a name for the function, such as fnAddress, and write the Python code to perform your operation. The name here can be anything you like, but standard convention is to start functions with a lowercase *fn* to identify their type and add a descriptive name in uppercase.

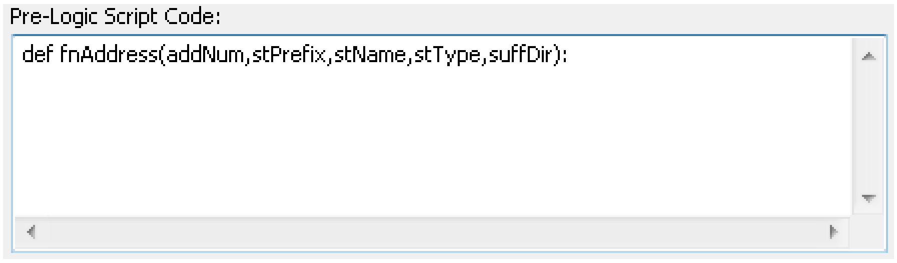
5. **In the Expression box under “full_address =,” type the line shown in the graphic. Type the function name, and double-click the field names to enter them. Be sure to add commas between the field names.**



```
full_address =
fnAddress( !number_!, !st_prefix!, !street!, !st_type!, !st_suffix_dir!)
```

Double-clicking the field names to add them to the function will automatically place the proper characters around the field name. These may be quotation marks, exclamation points, or square brackets, among others, depending on the data source. It is hard to know exactly which characters will be needed, so using the double-click method ensures that the correct characters are used.

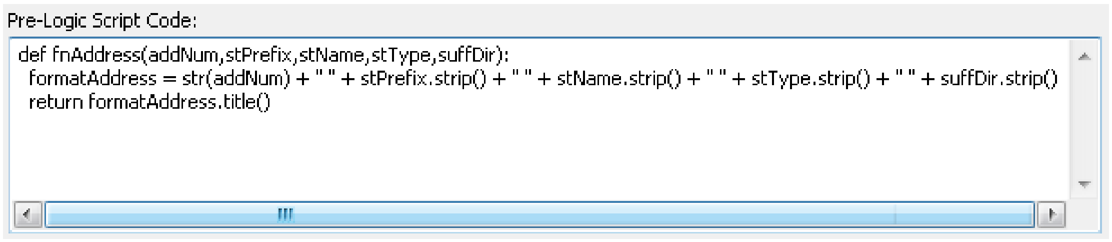
6. **In the Pre-Logic Script Code box, type the following code:**



```
Pre-Logic Script Code:
def fnAddress(addNum,stPrefix,stName,stType,suffDir):
```

This defines a function named fnAddress and accepts each of the fields listed in the Expression box. **Note:** This is the format for all code that you will ever write in the Field Calculator. The Label Expression dialog box names a function, followed by all the fields that will be used in the script. Then the Pre-Logic Script Code defines the function beginning with *def* and accepts each of the fields into a Python variable that you name.

7. **Write the Python code to format and concatenate the fields into a single string. Remember to account for a space between the values. When you have the code entered, click OK to perform the calculation. If you have difficulties, check your code against the graphic, but try writing the code on your own first.**



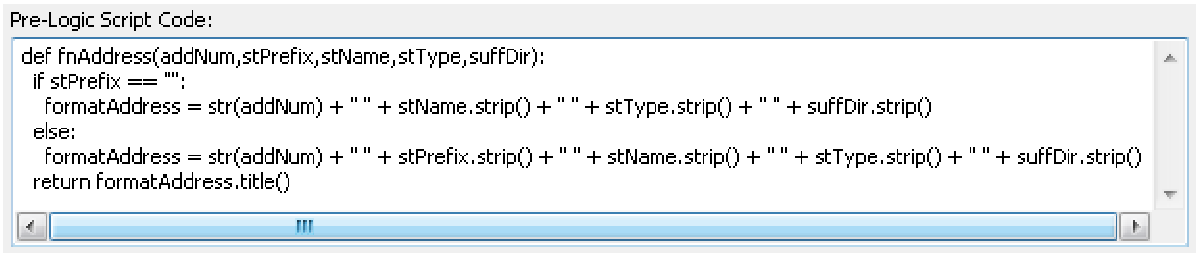
```
Pre-Logic Script Code:
def fnAddress(addNum,stPrefix,stName,stType,suffDir):
  formatAddress = str(addNum) + " " + stPrefix.strip() + " " + stName.strip() + " " + stType.strip() + " " + suffDir.strip()
  return formatAddress.title()
```

This worked pretty well, but there is still a problem. If the stPrefix string is empty, an additional space is added to the output string. Also, extra spaces are added at the end of the output string if suffDir is empty. Using what you know about if statements, try writing the code that will test for this field being empty, and then handle the case of what to do if it is empty. It should have this logic (add this to your pseudo code):

- If the field stPrefix is empty, write the concatenation without this field included.
- Add a command at the end of the output to strip off blank spaces.
- If the field stPrefix is not empty, write the same concatenation as before with the command added to strip blank spaces from the end of the string.

You should also investigate other Python formatting tools to remove any blank spaces from the values and perhaps to control the capitalization.

8. **Right-click the full_address field, and open the Field Calculator dialog box. Type the code you wrote—making sure to use the correct indentations with the if statements. (Hint: indent two spaces for every command after the def statement and four spaces for every command to run with the if statement.) Click OK to test it—the completed code looks like this:**



Pre-Logic Script Code:

```
def fnAddress(addNum,stPrefix,stName,stType,suffDir):
  if stPrefix == "":
    formatAddress = str(addNum) + " " + stName.strip() + " " + stType.strip() + " " + suffDir.strip()
  else:
    formatAddress = str(addNum) + " " + stPrefix.strip() + " " + stName.strip() + " " + stType.strip() + " " + suffDir.strip()
  return formatAddress.title()
```

Note the use of indentations to distinguish the commands for the different parts of the if statement. This example also includes the .strip() and .title() functions to help with the text formatting.

You can see that complex scripts can be developed for use in the Field Calculator. The format for any script you write will be the same: name a function in the Expression box along with the fields you will be using in the script, define a function in the Pre-Logic Script Code box with a Python variable for each field named in the function, and write the code to do your processing.

Exercise 1-3

The chief would like to export this data into another program for analysis, but there must be a field describing which station responded. The field district has a number that designates the station code, but the chief would like it in the format “Station 1” instead of the code.

Add a text field to the table FireRuns2010, and name it **Station.** Then write a script in the Field Calculator that will populate the field with the appropriate text:

151 = “Oleander Station 1”
551 = “Oleander Station 1”
152 = “Oleander Station 2”
552 = “Oleander Station 2”
153 = “Oleander Station 3”
553 = “Oleander Station 3”

Anything else should be made equal to “Outside Station.”

As a bonus, add the shift code at the end of each value. For instance, for 551 shift B, the output would read “Oleander Station 1 – Shift B.”

Tutorial 1-3 review

As you can see, the code in the Field Calculator can get complex. The trick is to maintain the indentations because the code block box does not handle indent levels automatically, as a good IDE would. It is sometimes good practice to build these statements in your IDE using some preset dummy data variables to test the syntax and set the indentations correctly. Then you can copy and paste the code to the Field Calculator.

The same rules that apply to if statements in the Label Expression dialog box also apply to if statements in the Field Calculator dialog box. Follow these rules carefully, and test any condition statements to make sure that they will not send your code out of control and that they have an instance that equates to True.

In both the Field Calculator and Label Expression dialog boxes, you are required to manage your own indent levels. Because of this, it is good practice to test your code in an IDE first for syntax and indentations before placing it in the Field Calculator. It is also advisable to calculate values into new, empty fields rather than to calculate values over existing values. If something is wrong in your code, you will destroy the original data values. These values would be impossible to recover if you were calculating outside an edit session.

This tutorial includes condition statements using string values. If the fields or values being compared do not match in case (uppercase, lowercase, or a combination thereof), the values may not equate to True, even when they are the same except for case. In this instance, you can use one of the string formatting functions to force the case to match before performing the testing.

Study questions

1. Could these same scripts be used to display labels on a temporary basis rather than storing the output string in a field? Write a code example to complete the exercise as a label statement (if you think it can be done).
2. What other string formatting statements are available for use in Field Calculator scripts? Write code to compare name fields from two different tables if the first is *Name* and stores a value in upper- and lowercase letters and the second is *EmpName* and stores a value in all uppercase letters.
3. Besides using double equals signs (==) for “is equal to,” what other evaluators are available in Python? Write code to find pipe sizes starting at 8 inches and going up to 12 inches.

Tutorial 1-4 Decision making in the Field Calculator

Complex operations in the Field Calculator can include advanced Python math functions and if-elif-else logic.

Learning objectives

- Writing a Python code block
- Using if-elif-else logic
- Text formatting

Preparation

Research the following topic in ArcGIS for Desktop Help:

- "Using if-then-else logic for branching"

Introduction

In tutorial 1-3, you learned how to perform various text formatting techniques in the Field Calculator. The Field Calculator allows you to perform a variety of math functions as well. Python has many math expressions built in, such as adding, subtracting, multiplying, dividing, exponentials, and square roots, but Python also has a math module that can be imported to add higher-level math operators. This scenario uses simple math operators, but you can explore other Python code references for more options.

Scenario

The city engineer is preparing to do a flow rate study on the sewer system data. It is a gravity flow system, and the flow rate in gallons per minute (gpm) of wastewater must be calculated for each pipe size. In addition, she wants you to include a drag coefficient for the different pipe materials because some types of pipe are not flowing at the optimum rate due to friction or buildup in the pipes. Because you do not know the slope of the pipes, calculate the flow rate assuming a 1 percent grade, and she can calculate a more accurate flow rate when better slope data is acquired.

Your script must find the pipe size, get the flow rate for that size, and multiply the flow rate by the drag coefficient of the pipe material. The flow rates are as follows:

4 in. = 30 gpm
6 in. = 70 gpm
8 in. = 175 gpm
10 in. = 280 gpm
12 in. = 410 gpm

The drag coefficients for the different materials are as follows:

Ductile iron	= 0.82
Reinforced concrete	= 0.88
Vitrified clay	= 0.92
High-density polyethylene	= 0.97
Polyvinyl chloride	= 0.97

Notice that even the best pipe, with a coefficient of 0.97, does not allow for wastewater to flow at the maximum rate. Because the pipe must maintain an air gap to allow the wastewater to flow freely, the maximum theoretical flow rate is never achieved.

Data

The data is the sewer utility data for the City of Oleander. The pipeline database already has an empty field in which you will calculate the flow rate. In addition, there is a field containing the pipe size and another one containing a description of the pipe material. A definition query has been applied to limit the pipe sizes to less than 14 inches to save time typing a lot of code. In reality, this process would be done on the entire dataset.

SCRIPTING TECHNIQUES

The first few tutorials used various Python controls to manipulate strings, but there are just as many available for mathematical functions. The common controls are addition (+), subtraction (-), multiplication (*), and division (/), but a variety of more complex functions exist, such as exponentials and square roots. A double-asterisk (**) operator is used for an exponential, so 2 to the power of 4 would be 2**4. Seven squared would be 7**2 (seven to the second power).

Python also has a separate math module that can be referenced from your scripts to do more scientific calculations, but these calculations are not addressed here.

Make decisions in the Field Calculator

1. **Try writing your pseudo code for this project on your own before referring to the description shown:**

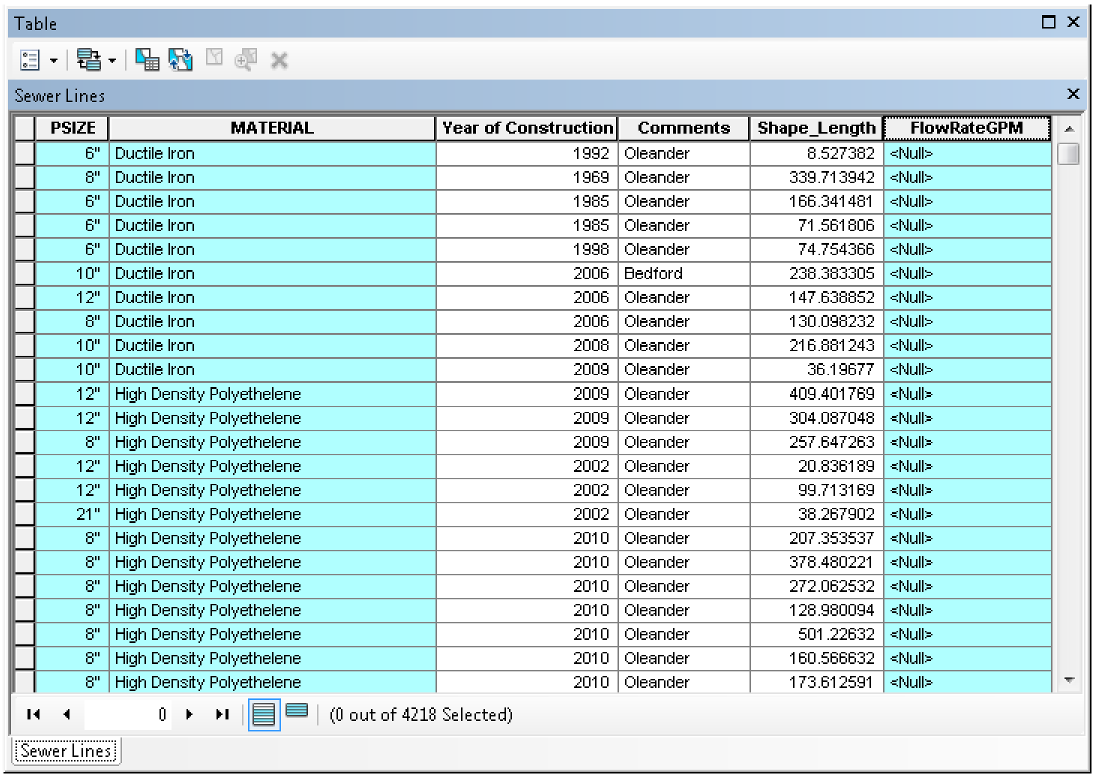
```
# Get the pipe size and pipe material from the database
# Use the pipe size to determine the correct flow rate
# Find out the pipe material
# Perform the calculation
```

This pseudo code shows a general outline of the process.

2. Think about each of these steps, and determine what type of Python scripting you may have to write to accomplish the goal, and add that description to the pseudo code. You can be more specific in this step because the next step requires writing the actual code.

```
# Get the pipe size and pipe material from the database.
#   1. Define a function to bring the fields PSIZE and MATERIAL
#      into the Field Calculator.
#      Create two variables to hold these values.
# Use the pipe size to determine the correct flow rate.
#   2. Use an if statement to determine the pipe size.
#      There will be an if, then an elif for each pipe size in the list.
# Find the pipe material.
#   3. Nest an if statement in the pipe size if statement to set the
#      drag coefficient.
#      There will be an if, then an elif for each material type.
# Perform the calculation.
#   4. Perform the calculation, and store the result in a variable.
#      Use a return statement to send the results back to the function
#      and set the field value.
```

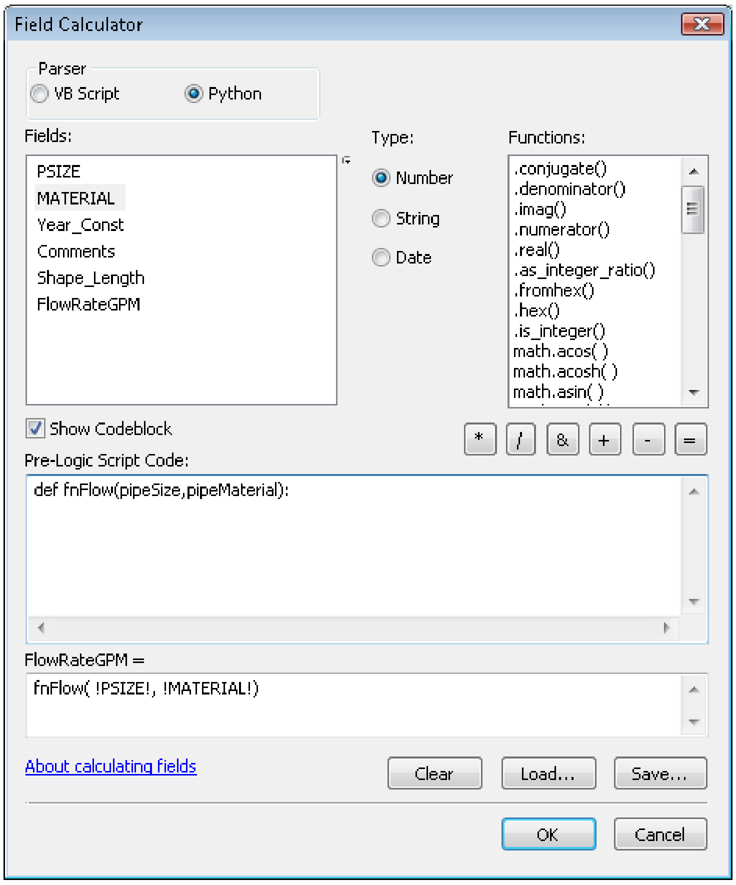
3. Start ArcMap, and open the map document Tutorial 1-4. This is the sewer utility data for Oleander, zoomed in on a small area. Open the attribute table for the Sewer Lines layer, and note the fields in the table that you will be using in this process.



PSIZE	MATERIAL	Year of Construction	Comments	Shape_Length	FlowRateGPM
6"	Ductile Iron	1992	Oleander	8.527382	<Null>
8"	Ductile Iron	1969	Oleander	339.713942	<Null>
6"	Ductile Iron	1985	Oleander	166.341481	<Null>
6"	Ductile Iron	1985	Oleander	71.561806	<Null>
6"	Ductile Iron	1998	Oleander	74.754366	<Null>
10"	Ductile Iron	2006	Bedford	238.383305	<Null>
12"	Ductile Iron	2006	Oleander	147.638852	<Null>
8"	Ductile Iron	2006	Oleander	130.098232	<Null>
10"	Ductile Iron	2008	Oleander	216.881243	<Null>
10"	Ductile Iron	2009	Oleander	36.19677	<Null>
12"	High Density Polyethelene	2009	Oleander	409.401769	<Null>
12"	High Density Polyethelene	2009	Oleander	304.087048	<Null>
8"	High Density Polyethelene	2009	Oleander	257.647263	<Null>
12"	High Density Polyethelene	2002	Oleander	20.836189	<Null>
12"	High Density Polyethelene	2002	Oleander	99.713169	<Null>
21"	High Density Polyethelene	2002	Oleander	38.267902	<Null>
8"	High Density Polyethelene	2010	Oleander	207.353537	<Null>
8"	High Density Polyethelene	2010	Oleander	378.480221	<Null>
8"	High Density Polyethelene	2010	Oleander	272.062532	<Null>
8"	High Density Polyethelene	2010	Oleander	128.980094	<Null>
8"	High Density Polyethelene	2010	Oleander	501.22632	<Null>
8"	High Density Polyethelene	2010	Oleander	160.566632	<Null>
8"	High Density Polyethelene	2010	Oleander	173.612591	<Null>

0 (0 out of 4218 Selected)

4. **Right-click the FlowRateGPM field, and open the Field Calculator dialog box. Click the settings to allow for the entry of Python code, and define a function to accept Python code as described in step 1 of your pseudo code. Refer to tutorial 1-3 for a reminder of how to do this. The function should pass the PSIZE and MATERIAL fields to the code block (in each step, try out your own code before referring to the graphics).**



Parser: [✓ Python]
Type: [✓ Number]
Show Codeblock: ✓

Pre-Logic Script Code:

```
def fnFlow(pipeSize,pipeMaterial):
```

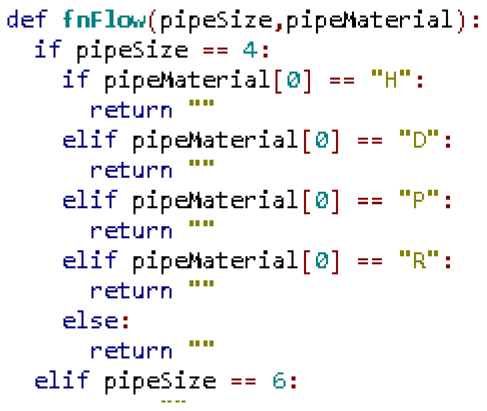
FlowRateGPM =

```
fnFlow( !PSIZE!, !MATERIAL!)
```

5. **Construct the if statement as outlined in step 2 of your pseudo code. Lay out the entire structure to determine pipe size before thinking about the steps to find the pipe material. Remember to control your indentations. The return statements return a blank value and are here just as placeholders.** Note: **The displays of code shown in the graphic are from an IDE for legibility. You may want to develop this code in an IDE or text editor, and copy it to the Field Calculator when you are done.**

```
def fnFlow(pipeSize,pipeMaterial):
  if pipeSize == 4:
    return ""
  elif pipeSize == 6:
    return ""
  elif pipeSize == 8:
    return ""
  elif pipeSize == 10:
    return ""
  elif pipeSize == 12:
    return ""
```

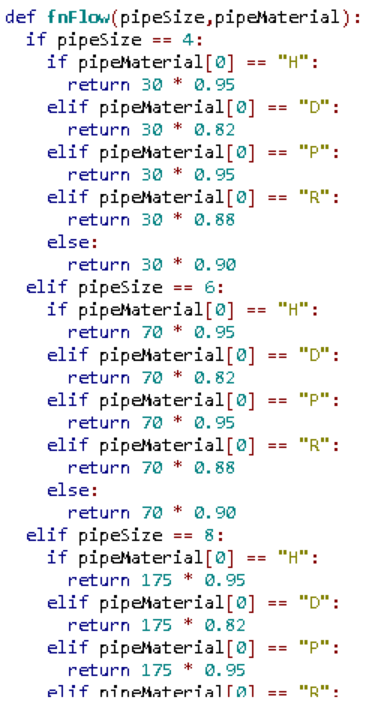
6. **Move on to step 3 of the pseudo code, and add the nested if statement to determine material type. The if and elif statements are indented two spaces from the pipe size condition statement.**



```
def fnFlow(pipeSize,pipeMaterial):
  if pipeSize == 4:
    if pipeMaterial[0] == "H":
      return ""
    elif pipeMaterial[0] == "D":
      return ""
    elif pipeMaterial[0] == "P":
      return ""
    elif pipeMaterial[0] == "R":
      return ""
    else:
      return ""
  elif pipeSize == 6:
```

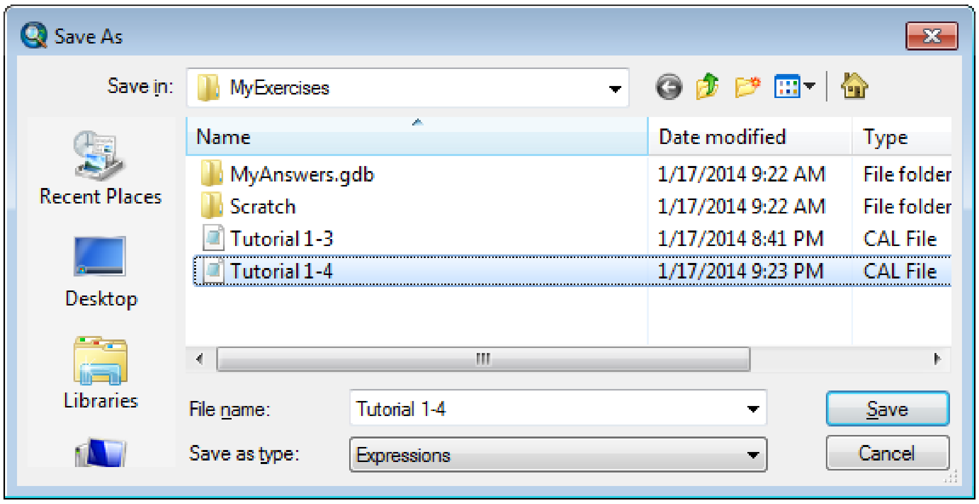
Note the use of the slice function to get the first letter of each description in the if statement. This function keeps you from having to type the entire description, but make sure that when you use this function you are producing a unique value for each if statement. Although only one set of condition statements for pipe materials is shown, these statements need to occur for each pipe size.

7. **Finally, you must construct the calculation, and add it for each condition.** Note: **only part of the final script is shown in the graphic.**



```
def fnFlow(pipeSize,pipeMaterial):
  if pipeSize == 4:
    if pipeMaterial[0] == "H":
      return 30 * 0.95
    elif pipeMaterial[0] == "D":
      return 30 * 0.82
    elif pipeMaterial[0] == "P":
      return 30 * 0.95
    elif pipeMaterial[0] == "R":
      return 30 * 0.88
    else:
      return 30 * 0.90
  elif pipeSize == 6:
    if pipeMaterial[0] == "H":
      return 70 * 0.95
    elif pipeMaterial[0] == "D":
      return 70 * 0.82
    elif pipeMaterial[0] == "P":
      return 70 * 0.95
    elif pipeMaterial[0] == "R":
      return 70 * 0.88
    else:
      return 70 * 0.90
  elif pipeSize == 8:
    if pipeMaterial[0] == "H":
      return 175 * 0.95
    elif pipeMaterial[0] == "D":
      return 175 * 0.82
    elif pipeMaterial[0] == "P":
      return 175 * 0.95
    elif pipeMaterial[0] == "R":
```

8. **Once you have the entire script in the Field Calculator code box, click Save, and save to your MyExercises folder.**



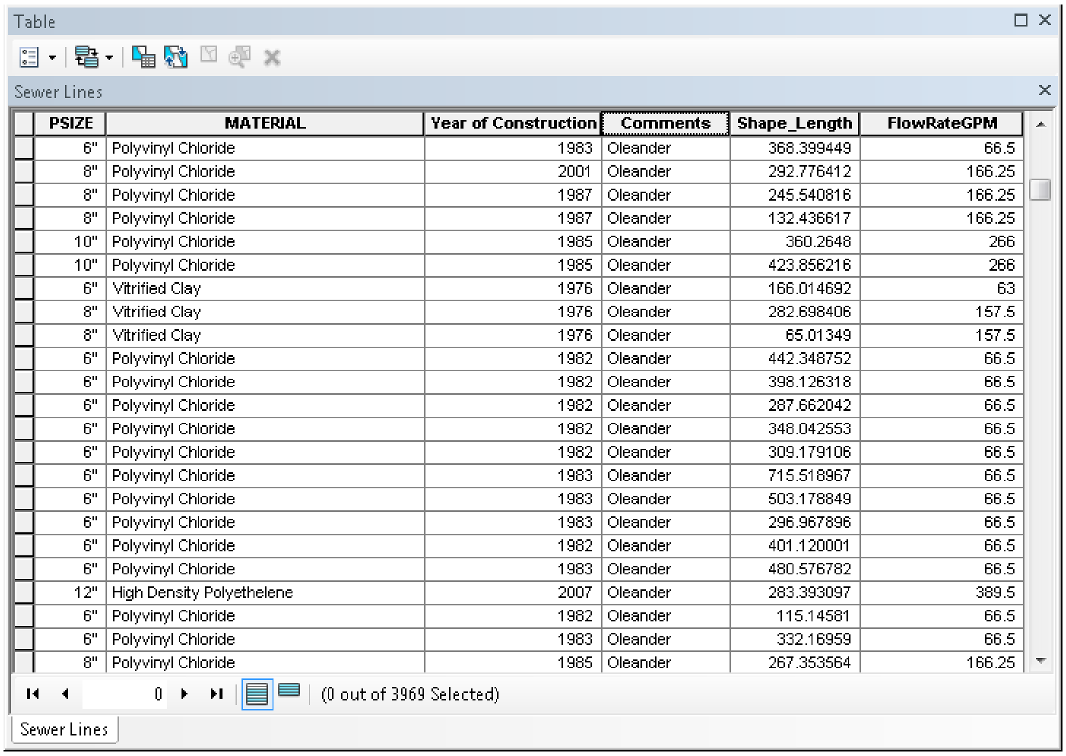
Save in: MyExercises

Name	Date modified	Type
MyAnswers.gdb	1/17/2014 9:22 AM	File folder
Scratch	1/17/2014 9:22 AM	File folder
Tutorial 1-3	1/17/2014 8:41 PM	CAL File
Tutorial 1-4	1/17/2014 9:23 PM	CAL File

File name: Tutorial 1-4

Save as type: Expressions

9. **Click OK to run the code, and see the results calculated into the field. (Hint: make sure Python is still selected as the parser in the Field Calculator dialog box—it sometimes resets to the default of VB Script.)**



PSIZE	MATERIAL	Year of Construction	Comments	Shape_Length	FlowRateGPM
6"	Polyvinyl Chloride	1983	Oleander	368.399449	66.5
8"	Polyvinyl Chloride	2001	Oleander	292.776412	166.25
8"	Polyvinyl Chloride	1987	Oleander	245.540816	166.25
8"	Polyvinyl Chloride	1987	Oleander	132.436617	166.25
10"	Polyvinyl Chloride	1985	Oleander	360.2648	266
10"	Polyvinyl Chloride	1985	Oleander	423.856216	266
6"	Vitrified Clay	1976	Oleander	166.014692	63
8"	Vitrified Clay	1976	Oleander	282.698406	157.5
8"	Vitrified Clay	1976	Oleander	65.01349	157.5
6"	Polyvinyl Chloride	1982	Oleander	442.348752	66.5
6"	Polyvinyl Chloride	1982	Oleander	398.126318	66.5
6"	Polyvinyl Chloride	1982	Oleander	287.662042	66.5
6"	Polyvinyl Chloride	1982	Oleander	348.042553	66.5
6"	Polyvinyl Chloride	1982	Oleander	309.179106	66.5
6"	Polyvinyl Chloride	1983	Oleander	715.518967	66.5
6"	Polyvinyl Chloride	1983	Oleander	503.178849	66.5
6"	Polyvinyl Chloride	1983	Oleander	296.967896	66.5
6"	Polyvinyl Chloride	1982	Oleander	401.120001	66.5
6"	Polyvinyl Chloride	1983	Oleander	480.576782	66.5
12"	High Density Polyethelene	2007	Oleander	283.393097	389.5
6"	Polyvinyl Chloride	1982	Oleander	115.14581	66.5
6"	Polyvinyl Chloride	1983	Oleander	332.16959	66.5
8"	Polyvinyl Chloride	1985	Oleander	267.353564	166.25

0 (0 out of 3969 Selected)

If your code is not running correctly, double-check the variable names, the indentations, the colons, and so on. For long, complex scripts like this one, you can do them in your IDE, which checks syntax as you go, and then copy and paste the final script to the Field Calculator code box.

Exercise 1-4

The city engineer has a similar project using the water line data. Open the map document Exercise 1-4 and look at the attribute table for the Distribution Laterals layer. Use the fields PSIZE, PTYPE (for material), and Shape_Length to calculate the desired use factor.

The formula is PSIZE * Shape_Length * Material Coefficient, using the material coefficients from the following table:

	For pipes 6 in. or less	For pipes 8 in. or larger
Asbestos concrete	80	95
Cast iron	60	75
Polyvinyl chloride	90	105

Write pseudo code to determine the process. Then use the Field Calculator to complete the process, and place the answer in the DiamLengthPressure field.

Tutorial 1-4 review

Working in the Field Calculator is unique because it lets you do a series of checks and calculations on a per-feature or per-row basis. Each feature is evaluated individually. To do this in stand-alone Python scripts, you use a *cursor*, which you will learn about later, so the Field Calculator is like an automatic cursor.

There are limitations to the Field Calculator, however. Although you can bring a number of field values from the current feature class into the script, you can only deal with one output field at a time—and for that matter, only one feature class or one table at a time. In a full Python script, you can control any number of field values, feature classes, or tables simultaneously and send output to other fields, other feature classes and tables, or even files outside ArcGIS.

Study questions

1. If you have a feature class with property values (Tot_Value) and a separate feature class with lot size (Acreage), could you write a script to calculate the value per acre using the Field Calculator? Write the script (if you think you can do it).
2. You need to calculate a property drainage coefficient based on lot size (Acreage), land use (Use_Code), an impervious area (Imperv_Area), and soil type (Dirt_Type). If all these fields are in the same feature class, can they all be used in the Field Calculator? Write the code to bring all these fields into the Field Calculator dialog box (if you think you can do it).
3. Name three things to watch for when nesting if statements.

Tutorial 1-5 Working with Python date formats

Dates are a complex item in any programming language, but Python makes using them easy. Basic formatting techniques are used to extract and manipulate data information.

Learning objectives

- Using Python date directives
- Working with date information
- Building complex Python objects

Preparation

Research the following topic in ArcGIS for Desktop Help:

- "Fundamentals of date fields"

Introduction

As you have seen in tutorials 1-3 and 1-4, complex calculations can be made in the Field Calculator using Python directives and if condition statements. One type of calculation that causes concern among programmers is performing date calculations because the fields that hold the dates are not standard fields, and they are not structured on a base 10 calculation like common numbers. A date field is a special type of field that can contain the day, month, year, and time of day in a variety of formats, which means that a standard Python variable is not able to contain the data. Instead, you must use a Python date object. In using the date object, make sure to specify the date components as they are brought into the object so that they can be retrieved as needed.

You also must pay attention to the time field. Merely subtracting the time values will not produce the desired results. The hours, minutes, and seconds must be calculated separately, and at the same time, you must also be aware of the scenario covering multiple days.

Scenario

The fire chief has provided you with the calls for service data for the past year and wants you to calculate the elapsed time between the dispatch time and the time that the vehicle arrived on the scene in decimal minutes. Any call that exceeds five minutes will need to be investigated. Although this sounds simple, it can be one of the more complex functions to perform.

Data

The calls for service data has the fields dispatched and arrived, which represent the time the vehicle left the station and the time it arrived on the scene.

SCRIPTING TECHNIQUES

Remember that variables typically hold a single value, but objects can hold multiple values. The list objects that you used earlier are a good example of objects with multiple values. The key to working with objects is to understand the format of what the object holds and how to access it. As the book progresses, you will see more examples of objects and how to research their structure.

Date objects can actually hold both date and time, or date only or time only, so particular care must be taken to assess the values in the object before deciding how to process them. Once this is done, identify each of the components using a format code called a *date directive*. The following list of directives will help you decide how to format the date object:

- %a = abbreviated weekday name
- %A = full weekday name
- %b = abbreviated month name
- %B = full month name
- %c = preferred date and time representation
- %C = century number (the year divided by 100, range 00 to 99)
- %d = day of the month (01 to 31)
- %D = same as %m/%d/%y
- %g = like %G, but without the century
- %G = four-digit year corresponding to the ISO week number (see %V)
- %h = same as %b
- %H = hour, using a 24-hour clock (00 to 23)
- %I = hour, using a 12-hour clock (01 to 12)
- %j = day of the year (001 to 366)
- %m = month (01 to 12)
- %M = minute
- %n = add a new line
- %p = either a.m. or p.m., according to the given time value
- %r = time in a.m. and p.m. notation
- %R = time in 24-hour notation
- %S = second
- %t = Tab character

- %T = current time, equal to %H:%M:%S
- %u = weekday as a number (1 to 7), where Monday = 1 (**Warning:** in the Sun Solaris operating system, Sunday = 1.)
- %U = week number of the current year, starting with the first Sunday as the first day of the first week
- %V = the ISO 8601 week number of the current year (01 to 53), where week one is the first week that has at least four days in the current year, and with Monday as the first day of the week
- %W = week number of the current year, starting with the first Monday as the first day of the first week
- %w = day of the week as a decimal, where Sunday = 0
- %x = preferred date representation without the time
- %X = preferred time representation without the date
- %y = year without a century (range 00 to 99)
- %Y = year including the century
- %Z or %z = time zone or name or abbreviation
- %% = a literal % character

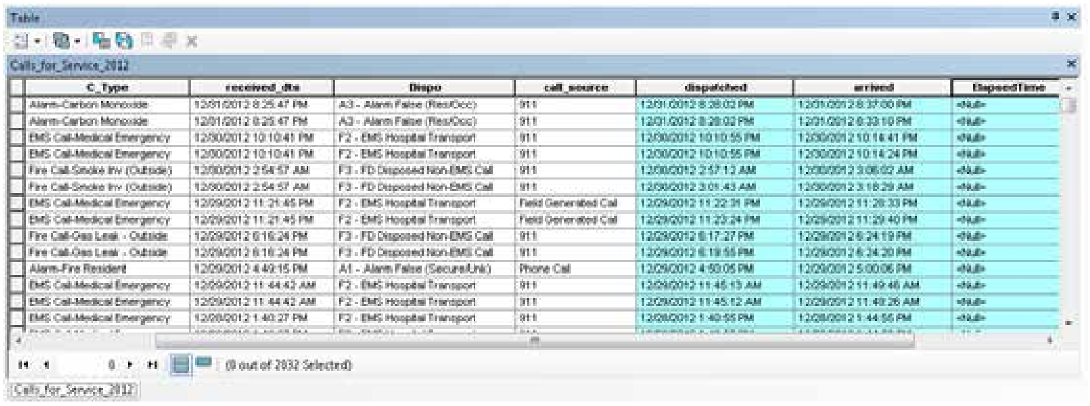
You must use the Python DateTime module to correctly use the date object. This is a standard Python module that contains specialized functions and methods for working specifically with date information. The syntax is to add "from datetime import datetime" at the beginning of your code. This syntax brings in the date and time functions you will use in the calculations. For example, the date string "10/25/2012" would use the format string "%m/%d/%Y".

The DateTime module also includes special functions to perform math on dates. Simple subtraction of dates using the standard Python math functions could produce incorrect results. Imagine a call for service that started at 11:58 a.m. and ended four minutes later at 12:02 p.m. Performing a simple subtraction of these values would produce a negative number. The same would be true of a call that occurred just before midnight and ran into the next day. But once the values in the fields are put into date objects, subtracting them produces a time delta object. The function total_seconds() can be used to extract a value into a numeric variable, and dividing this by 60 converts the value to minutes.

Handling time is also tricky. Seconds and minutes are on a base 60 system, with hours being on a base 12 system, or a base 24 system for military time. You determine the base when you format the Python object. The directives handle almost any combination, but make sure you match them to the data carefully. This will ensure success with your code.

Work with the Python DateTime module

1. Open the map document Tutorial 1-5. In the table of contents, click the List By Source button, and open the table Calls_for_service_2012. Note the fields that you are working with.



C_Type	received_dts	Dispo	call_source	dispatched	arrived	ElapsedTime
Alarm-Carbon Monoxide	12/31/2012 8:25:47 PM	A3 - Alarm False (Res/Occ)	911	12/31/2012 8:28:02 PM	12/31/2012 8:37:00 PM	<Null>
Alarm-Carbon Monoxide	12/01/2012 8:25:47 PM	A3 - Alarm False (Res/Occ)	911	12/31/2012 8:28:02 PM	12/31/2012 8:33:10 PM	<Null>
EMS Call-Medical Emergency	12/30/2012 10:10:41 PM	F2 - EMS Hospital Transport	911	12/30/2012 10:10:55 PM	12/30/2012 10:14:41 PM	<Null>
EMS Call-Medical Emergency	12/30/2012 10:10:41 PM	F2 - EMS Hospital Transport	911	12/30/2012 10:10:55 PM	12/30/2012 10:14:24 PM	<Null>
Fire Call-Smoke Inv (Outside)	12/30/2012 2:54:57 AM	F3 - FD Disposed Non-EMS Call	911	12/30/2012 2:57:12 AM	12/30/2012 3:06:02 AM	<Null>
Fire Call-Smoke Inv (Outside)	12/30/2012 2:54:57 AM	F3 - FD Disposed Non-EMS Call	911	12/30/2012 3:01:43 AM	12/30/2012 3:18:29 AM	<Null>
EMS Call-Medical Emergency	12/29/2012 11:21:45 PM	F2 - EMS Hospital Transport	Field Generated Call	12/29/2012 11:22:31 PM	12/29/2012 11:28:33 PM	<Null>
EMS Call-Medical Emergency	12/29/2012 11:21:45 PM	F2 - EMS Hospital Transport	Field Generated Call	12/29/2012 11:23:24 PM	12/29/2012 11:29:40 PM	<Null>
Fire Call-Gas Leak - Outside	12/29/2012 6:16:24 PM	F3 - FD Disposed Non-EMS Call	911	12/29/2012 6:17:27 PM	12/29/2012 6:24:19 PM	<Null>
Fire Call-Gas Leak - Outside	12/29/2012 6:16:24 PM	F3 - FD Disposed Non-EMS Call	911	12/29/2012 6:19:55 PM	12/29/2012 6:24:20 PM	<Null>
Alarm-Fire Resident	12/29/2012 4:49:15 PM	A1 - Alarm False (Secure/Unk)	Phone Call	12/29/2012 4:50:05 PM	12/29/2012 5:00:06 PM	<Null>
EMS Call-Medical Emergency	12/29/2012 11:44:42 AM	F2 - EMS Hospital Transport	911	12/29/2012 11:45:13 AM	12/29/2012 11:49:46 AM	<Null>
EMS Call-Medical Emergency	12/29/2012 11:44:42 AM	F2 - EMS Hospital Transport	911	12/29/2012 11:45:12 AM	12/29/2012 11:49:26 AM	<Null>
EMS Call-Medical Emergency	12/28/2012 1:40:27 PM	F2 - EMS Hospital Transport	911	12/28/2012 1:40:55 PM	12/28/2012 1:44:56 PM	<Null>

Notice the format of the date. The fields have the following structure:

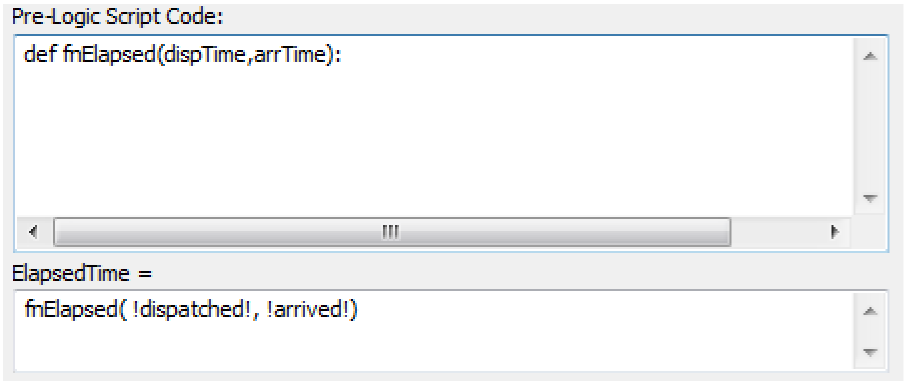
- Month shown as two digits followed by a slash
- Day of the month shown as two digits followed by a slash
- Year shown as four digits followed by a space
- Hour in base 12 shown as one or two digits followed by a colon
- Minutes in base 60 shown as two digits followed by a colon
- Seconds in base 60 shown as two digits followed by a space
- AM and PM shown as two uppercase characters to determine morning or afternoon, respectively

The key to working with dates is to select the correct formatting string during the object assignment. Use the function .strptime() with the syntax strptime(date, format) where date is the date string you are bringing in from ArcMap, and format is the Python directive to identify each component of the date.

2. Write the general and detailed pseudo code necessary to calculate the elapsed time. Include the date string formatting directives to accept the data from the calls for service table. Use the preceding list in "Scripting techniques" to determine which directives to use. (As usual, try to write the complete pseudo code for each step before comparing your results with the graphics).

```
# Get the date fields from ArcMap.
#   1. Define a function to bring the fields "dispatched" and "arrived"
#      into the Field Calculator.
#      Create two variables to hold these values.
# Format the strings into Python date objects.
#   2. Select the correct directives for the date.
#      %m/%d/%Y %I:%M:%S %p
# Subtract the dates.
#   3. Subtract the formatted date objects to create a time delta object.
# Output the results in decimal minutes.
#   4. Use the total_seconds() function, and divide the results by 60.
#      Use round() to round the results to two decimal places.
```

3. **Right-click the ElapsedTime field, and open the Field Calculator dialog box. Enter the code to set up the function, and bring in the necessary fields.**


Pre-Logic Script Code:

```
def fnElapsed(dispTime,arrTime):
```

ElapsedTime =

```
fnElapsed( !dispatched!, !arrived!)
```

4. **Add the code to format the date string. Import the Python date-handling method, and then create a Python object with the correct date format, as shown:**

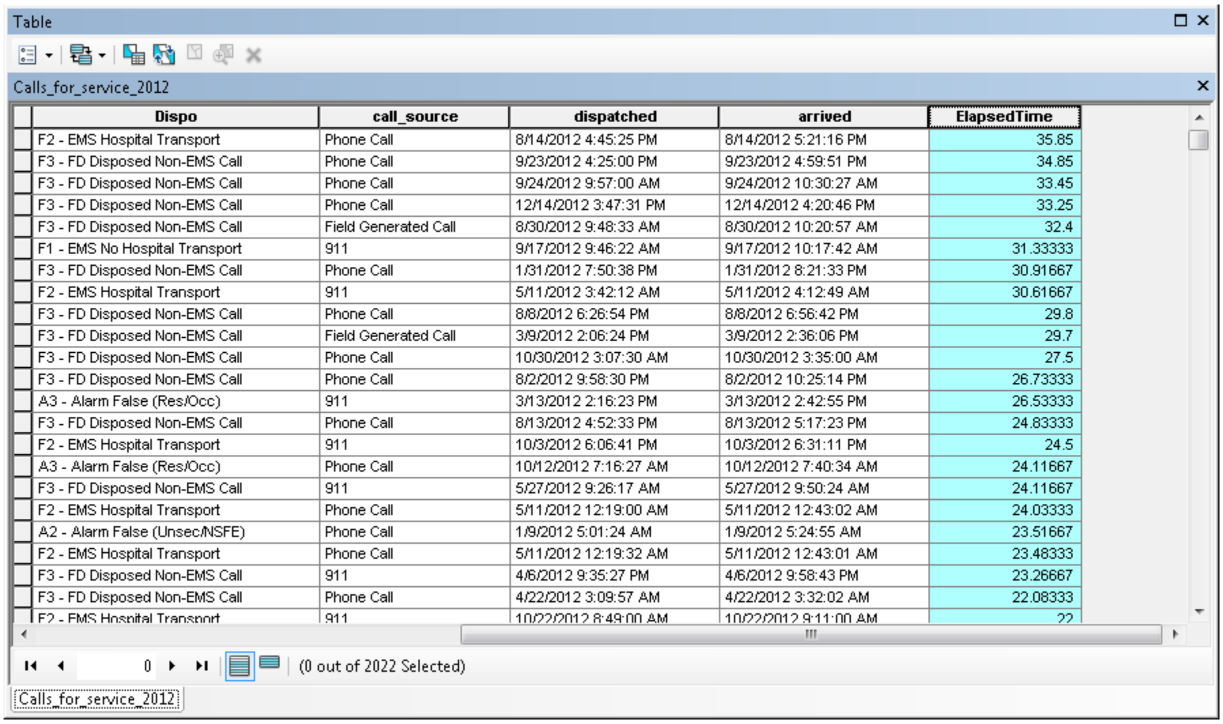
```
def fnElapsed(dispTime,arrTime):
  from datetime import datetime
  dateDispatch = datetime.strptime(dispTime,"%m/%e/%Y %I:%M:%S %p")
  dateArrive = datetime.strptime(arrTime,"%m/%e/%Y %I:%M:%S %p")
```

The date subtraction is next. The calculation must accommodate the possibility of a time that bridges the day or the morning to afternoon break. Subtracting the two date objects will result in a time delta object, which is designed to automatically accommodate the time changes. The time delta object holds the elapsed time in seconds and is retrieved using the total_seconds() function. When divided by 60, the result is the total elapsed time in decimal minutes.

5. **Write your version of the code to perform the date subtraction and compare it to this:**

```
def fnElapsed(dispTime,arrTime):
  from datetime import datetime
  dateDispatch = datetime.strptime(dispTime,"%m/%d/%Y %I:%M:%S %p")
  dateArrive = datetime.strptime(arrTime,"%m/%d/%Y %I:%M:%S %p")
  timeDiff = dateArrive - dateDispatch
  elapMin = timeDiff.total_seconds()/60
  return round(elapMin,2)
```

6. **Add the return statement, and send the results back to the table. Click OK to see the results. By sorting the field in descending order, you can quickly see which calls exceeded the required response time.**



Table

Calls_for_service_2012

Dispo	call_source	dispatched	arrived	ElapsedTime
F2 - EMS Hospital Transport	Phone Call	8/14/2012 4:45:25 PM	8/14/2012 5:21:16 PM	35.85
F3 - FD Disposed Non-EMS Call	Phone Call	9/23/2012 4:25:00 PM	9/23/2012 4:59:51 PM	34.85
F3 - FD Disposed Non-EMS Call	Phone Call	9/24/2012 9:57:00 AM	9/24/2012 10:30:27 AM	33.45
F3 - FD Disposed Non-EMS Call	Phone Call	12/14/2012 3:47:31 PM	12/14/2012 4:20:46 PM	33.25
F3 - FD Disposed Non-EMS Call	Field Generated Call	8/30/2012 9:48:33 AM	8/30/2012 10:20:57 AM	32.4
F1 - EMS No Hospital Transport	911	9/17/2012 9:46:22 AM	9/17/2012 10:17:42 AM	31.33333
F3 - FD Disposed Non-EMS Call	Phone Call	1/31/2012 7:50:38 PM	1/31/2012 8:21:33 PM	30.91667
F2 - EMS Hospital Transport	911	5/11/2012 3:42:12 AM	5/11/2012 4:12:49 AM	30.61667
F3 - FD Disposed Non-EMS Call	Phone Call	8/8/2012 6:26:54 PM	8/8/2012 6:56:42 PM	29.8
F3 - FD Disposed Non-EMS Call	Field Generated Call	3/9/2012 2:06:24 PM	3/9/2012 2:36:06 PM	29.7
F3 - FD Disposed Non-EMS Call	Phone Call	10/30/2012 3:07:30 AM	10/30/2012 3:35:00 AM	27.5
F3 - FD Disposed Non-EMS Call	Phone Call	8/2/2012 9:58:30 PM	8/2/2012 10:25:14 PM	26.73333
A3 - Alarm False (Res/Occ)	911	3/13/2012 2:16:23 PM	3/13/2012 2:42:55 PM	26.53333
F3 - FD Disposed Non-EMS Call	Phone Call	8/13/2012 4:52:33 PM	8/13/2012 5:17:23 PM	24.83333
F2 - EMS Hospital Transport	911	10/3/2012 6:06:41 PM	10/3/2012 6:31:11 PM	24.5
A3 - Alarm False (Res/Occ)	Phone Call	10/12/2012 7:16:27 AM	10/12/2012 7:40:34 AM	24.11667
F3 - FD Disposed Non-EMS Call	911	5/27/2012 9:26:17 AM	5/27/2012 9:50:24 AM	24.11667
F2 - EMS Hospital Transport	Phone Call	5/11/2012 12:19:00 AM	5/11/2012 12:43:02 AM	24.03333
A2 - Alarm False (Unsec/NSFE)	Phone Call	1/9/2012 5:01:24 AM	1/9/2012 5:24:55 AM	23.51667
F2 - EMS Hospital Transport	Phone Call	5/11/2012 12:19:32 AM	5/11/2012 12:43:01 AM	23.48333
F3 - FD Disposed Non-EMS Call	911	4/6/2012 9:35:27 PM	4/6/2012 9:58:43 PM	23.26667
F3 - FD Disposed Non-EMS Call	Phone Call	4/22/2012 3:09:57 AM	4/22/2012 3:32:02 AM	22.08333
F2 - EMS Hospital Transport	911	10/22/2012 8:49:00 AM	10/22/2012 9:11:00 AM	22

0 (0 out of 2022 Selected)

Calls_for_service_2012

Dates can be problematic, but this example should help you understand how to do a variety of date calculations. The key is to use the correct directive when formatting the Python date object.

Exercise 1-5

Open the map document Exercise 1-5. Perform the same elapsed-time calculations on the Calls_for_service_2010 data. Notice that the date and time data are in separate fields.

Write the pseudo code first to determine which directives are necessary to create the Python date objects.

Write the code in the Field Calculator, and store the results in the ElapsedTime field.

Tutorial 1-5 review

The examples in this tutorial cover how to properly format dates as input and place them into variables. Then these variables were used for calculations. The key to working with dates is to know the date format of the input value. Some date fields contain the full range of information, including date and time. Other fields may have only the date, and yet others may have only the time.

The date directives listed at the start of this tutorial also handle the formatting of output dates. For instance, you could format a date into a Python date object and print the corresponding day of the week using %u or the corresponding week of the year using %U or %W (depending on whether you start your week on Sunday or Monday).

Study questions

1. Research the date handlers in your Python reference book, and list the function to get today's date.
2. What format directives would you use on this date string: 31 12 2013 23:59:59 (New Year's Eve in London)?
3. What format directive would you use to give the full weekday name?