Chapter 15
User-Defined Functions

Abstract Workflows often have sequences of common steps repeated within or across scripts. Functions allow programmers to group related steps of code, name them, and reuse them by calling them by name. This chapter discusses defining functions with required and optional parameters, returning values from functions, organizing code with functions, and managing variables inside and outside of functions.

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

- Define and call custom functions.
- Pass values into and out of functions.
- Write functions with optional arguments.
- Create functions which return multiple values.
- Describe when to employ a user-defined function.
- Explain why some functions have parameters and/or return values.

15.1 A Word About Function Words

By now, you are quite familiar with calling functions that are built-in (e.g., `float`, `len`, and `range`) or ones that are available when a module is imported (e.g., `os.path.dirname` or `arcpy.Describe`). As your own scripts become more complex, you will want to start writing your own functions. Functions provide a way to organize code so that it can easily be reused. As a quick preview, look at the following code, which defines a function named `printBird`:

```
def printBird():
    print """
     ,,,     ::.
    <*~)    ;;
    (  @}//
     ''
    """
```

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_15

The function definition starts with the keyword def (short for 'define'), followed by the function name, parentheses, and a colon. Function names follow the same rules as variable names (e.g., names can't be keywords, can't start with numbers, can't contain spaces, and so forth). Sometimes the parentheses are empty, sometimes not—more about that in a moment. The colon is required and signals the start of a block of code. The code block inside the definition is indented. These components are needed to define a function in Python. But this is just the definition. The function code block contains a set of related statements that are executed only when the function is called. Nothing is printed if you simply put the printBird function definition in a script and run the script. The function needs to be called to be executed. To call this function, use the name, followed by parentheses:

```
>>> printBird()

      ,,,    ::.
    <*~)    ;;
    (  @}//
     ''

>>>
```

The printBird example demonstrates how easy it is to define and call a simple custom function. Of course, user-defined functions can do much more than print punctuation art. This chapter provides more details about creating and using custom functions.

The term 'function' has several near synonyms in programming terminology—procedures, subroutines, and methods. The terms 'function', 'procedure', and 'subroutine' are used more or less interchangeably in Python. The term 'procedure' is used to indicate that the function does not explicitly return a value—more on returning values is coming up shortly. Other programming languages use the term 'subroutine' but this term is not used as frequently in Python. A 'method' is a special type of procedure associated with an object. The upcoming chapter on Python classes will reveal how custom methods are created and used. The term 'function' is used here since it is the most general term.

Your knowledge of using built-in functions is a good frame of reference for learning about custom functions. The vocabulary related to built-in functions also applies to custom functions:

- A code statement that invokes a custom function is referred to as a *function call.*
- Providing input for a custom function is referred to as *passing arguments* into the function.
- A term closely related to arguments, *parameters*, are the pieces of information that can be specified to a function.
- The *signature* of a custom function is a template for how to use the function and lists the required and optional parameters.
- Some custom functions come up with results from the actions they perform—they *return a value.*

Learning to write custom functions will deepen your understanding of these terms. Chapter 2 likened a function to a task assigned to a butler. You tell the butler your bidding and he goes off to make it happen. You don't have to concern yourself with the details of how he does it. You ask him to iron your shirt and he attends to the execution. You only need to know the task name (e.g., `ironApparel`). Functions organize code by grouping related statements together. Example 15.1 defines the `printArgs` function to loop through the script arguments and print them. The `printArgs` function relies on the script to import the `sys` module (outside the function). The import statement could also be placed inside the function (though Python style guidelines recommend importing modules at the start of a script instead of inside functions).

Example 15.1

```
# reportArgs.py
# Purpose: Print the script arguments.
import sys

def printArgs():
    '''Print user arguments.'''
    for index, arg in enumerate(sys.argv):
        print 'Argument {0}: {1}'.format(index, arg)

printArgs()
```

The last line of 'reportArgs.py' in Example 15.1 calls the function. The function must be defined before it is called; in other words, the call must come later in the script than the definition. Place a function call before the definition and a `NameError` exception is thrown.

15.1.1 How It Works

We'll use the three step buttons on the debugging toolbar (Figure 15.1) to explore how functions work. The "Step" button would be more aptly named 'Step in'. In this book, we'll refer to it as the "Step (In)" button. This button steps into a function that is being called. The 'Step over' button executes the function being called, but does not step inside. The 'Step out' button steps out of the current function. The black lines on the buttons represent lines of code with indentation and the arrows represent how the buttons control the debugging cursor.

Stepping through with the debugger shows that when Python encounters a function definition, it doesn't execute it; Rather, it stores the name of the function and only when it reaches a call to that function does it go inside the function code

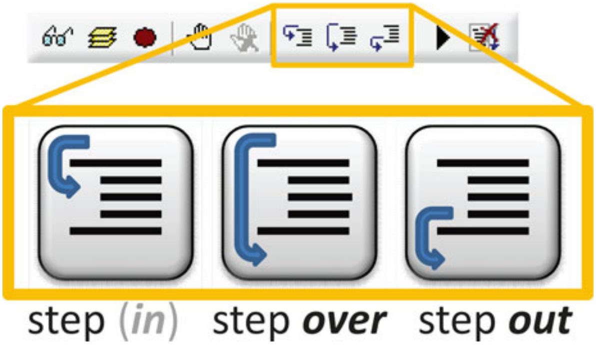
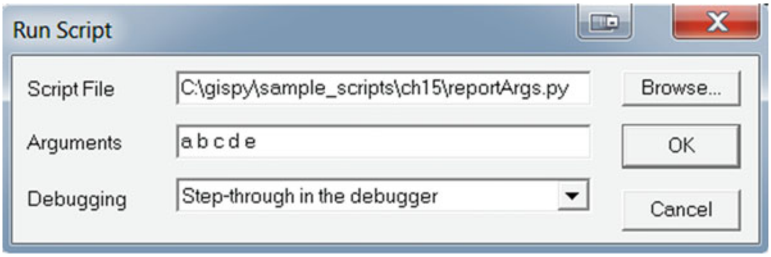


Figure 15.1 The three step buttons on the debug toolbar.

block and execute that code. To see this in action, run ‘reportArgs.py’ with the following steps:

1. Click the ‘Run’ button, enter arguments a, b, c, d, and e, and choose ‘Step-through in the debugger’ on the ‘Run Script’ form and click ‘Okay’ to start debugging.


Run Script

Script File	C:\gispy\sample_scripts\ch15\reportArgs.py	Browse...
Arguments	a b c d e	OK
Debugging	Step-through in the debugger	Cancel

2. Click the ‘Step (in)’ button two times to reach the function call. Notice the arrow ▷ jumps from the function `def` statement to the function call without entering the function definition.
3. Click the ‘Step (in)’ button one more time and the arrow jumps back to the function definition to start executing the function code.
4. This time, click the ‘Step over’ button to step through the `printArgs` function. (If you continue to use the ‘Step (in)’ button, execution will go inside the ‘winout.py’ script when the script reaches the print statement and you will have to click the ‘Step out’ button several times to return to ‘reportArgs.py’.)

5. Click the ‘Step over’ button several times until the first three arguments are printed.
6. Next click the ‘Step out’ button twice to step out of the function and return to the function call. Notice that the remaining two arguments are printed in the ‘Interactive Window’ because the remainder of the function statements were executed before exiting the function.
7. Click the ‘Step out’ button one more time to exit the script.

The ‘Step (in)’ button steps into functions in the code; the ‘Step over’ button bypasses the details. Run ‘reportArgs.py’ again in debug mode only selecting the ‘Step over’ button. Notice that the debugger doesn’t step through the execution of each line of code inside the function when the ‘Step over’ button is used. In contrast, the ‘Step (in)’ button demonstrates the flow more clearly—showing that the script does not run the function code block until it is called.

You may have noticed that the cursor jumped past the first line of code inside the function. This is not an executable line of code, but it serves a special purpose—it is a documentation string or docstring.

15.1.2 The Docstring

The string literal on the first line of code inside a function (such as, `'''Print user arguments.'''` in Example 15.1) is called a *docstring*. The docstring documents the function’s purpose. A one line docstring suffices for simple functions, but more than one line can be used if needed. Triple quotes are necessary for multi-line docstrings and style guidelines encourage triple quotes for single line docstrings to maintain consistency.

The docstrings not only remind the authors themselves of the function’s aim, but they facilitate sharing functions with others. Once the function definition is stored by Python (in other words, once the code has executed the `def` statement), the built-in `help` function can be used to print the docstring, as in the following code:

```
>>> help(printArgs)
Help on function printArgs in module__main__:
printArgs()
    Print user arguments.
```

The `printArgs` function prints arguments passed into the script by the user. Custom functions themselves can also be designed to take arguments, arguments that are passed to the function when it is called.

15.2 Custom Functions with Arguments

The butler can turn the lights off without further instructions, but if we want him to dim the lights, we should specify how low. Similarly, some functions require arguments while others don't. The `arcpy GetMessages` function, for example, automatically prints all the messages from the most recent tool call, but `GetMessage` requires a numerical index.

Example 15.2 consists of two similar functions, one with arguments (named `delNamedFCS)` and one without arguments (`delBuffFCS`). First, consider `delBuffFCS` which deletes each feature class in the current workspace whose name contains the word `Buff`. This is useful during script development. Suppose a geoprocessing script that buffers every shapefile listed in the workspace appends `Buff` to the input name. Each time the script runs, the output files from the previous run become input, doubling the number of input files and leading to output file names like 'firesBuffBuffBuff.shp'. Calling the `delBuffFCS` function restores the original workspace.

But suppose the script appends `Out` instead of `Buff` to each file name; Calling `delBuffFCS` won't help. The `delNamedFCS` function provides a more general solution, since it allows a string to be passed in to the function. This function deletes each feature class in the current workspace whose name contains the substring passed as an argument. The last line of Example 15.2 calls `delNamedFCS` with the argument `Out`. The `delString` variable in the definition list gets the value that is passed in to `delNamedFCS` when it is called. In this case the function deletes any file with the substring 'Out' in the name.

Example 15.2

```
# deleteFCS.py
# Purpose: Clear workspace of unwanted files.
# Usage: No arguments needed.
import arcpy

arcpy.env.workspace = 'C:/gispy/data/ch15/scratch'
def delBuffFCS():
    '''Delete feature classes with names containing "Buff".'''
    fcs = arcpy.ListFeatureClasses('*Buff*')
    for fc in fcs:
        arcpy.Delete_management(fc)
        print '{0} deleted.'.format(fc)

def delNamedFCS(delString):
    '''Delete feature classes with names containing delString.'''
    wildcard = '*{0}*'.format(delString)
    fcs = arcpy.ListFeatureClasses(wildcard)
    for fc in fcs:
        arcpy.Delete_management(fc)
        print '{0} deleted.'.format(fc)
delBuffFCS()
delNamedFCS('Out')
```

A custom function with arguments has the following general format:

```
def functionName (param1, param2, param3,...):
    '''Docstring describing the purpose.'''
    code statement(s)
```

The first line of the function definition (the one with the `def` statement) is the function signature. Each of the comma separated variable names inside the parentheses in the signature stands for a parameter. Parameters can be required or optional (more about that shortly). When the function is called, one value must be passed in for each required parameter in the signature and the values are separated by commas. The format for calling functions is as follows:

```
functionName(argument1, argument2, argument3,...)
```

Any number of arguments can be used in the signature. The following code defines a `batchBuffer` function which has four required parameters (workspace, featType, outSuffix, buffDistance):

```
# excerpt from batchBuff.py

def batchBuffer(workspace, featType, outSuffix, buffDistance):
    '''For a given workspace, buffer each
    feature class of a given feature type.'''
    arcpy.env.workspace = workspace
    fcs = arcpy.ListFeatureClasses('*', featType)
    for fc in fcs:
        fcParts = os.path.splitext(fc)
        outputName = fcParts[0] + outSuffix + fcParts[1]
        try:
            arcpy.Buffer_analysis(fc, outputName, buffDistance)
            print '{0} created.'.format(outputName)
        except:
            print 'Buffering {0} failed.'.format(fc)
```

To call the function, four arguments must be passed to the function. The `batchBuffer` function requires string arguments, which can be given as string literals or variables. The following code calls the function, with four string literal arguments:

```
>>> batchBuffer('C:/gispy/data/ch15', 'Polygon', 'Buff', '1 mile')
```

The parameters in the signature take on the values in the order in which they appear. The workspace variable takes the value 'C:/gispy/data/ch15', the `featType` variable takes the value `'Polygon'`, and so forth. Calling the function with less than the number of required parameters raises an exception:

```
>>> batchBuffer('C:/gispy/data/ch15', 'Polygon', 'Buff')
TypeError: batchBuffer() takes exactly 4 arguments (3 given)
```

The following code sets four string variables and then calls 'batchBuffer':

```
>>> wSpace = 'C:/gispy/data/ch15/tester.gdb'
>>> featureType = 'Point'
>>> outputSuffix = 'Ring'
>>> distance = '0.5 kilometers'
>>> batchBuffer(wSpace, featureType, outputSuffix, distance)
```

Notice that the argument variable names used in the function call are different from the parameter names in the signature (e.g., `eworkspace` and `wSpace`). This convention reduces confusion between function variables and non-function variables.

Note: Use names for variables inside function definitions that differ from those used outside the function definition.

The parameter data types are not specified in the signature, so it is important to use meaningful parameter names. Calling a function with the arguments of the wrong data type can lead to exception errors. For example, the 'batchBuffer' expects a string output suffix, not a numeric one. So the following code throws an exception:

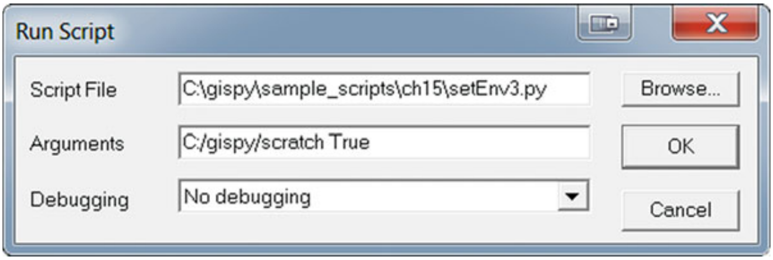
```
batchBuffer('C:/gispy/data/ch15', 'Polygon', 5, '1 mile')
```

The TypeError exception occurs when the function tries to concatenate a string and an integer (`fcParts[0] + outSuffix`).

15.2.1 Script Arguments vs. Functions Arguments

The term 'arguments' is used both in reference to scripts and functions. This can lead to some confusion, so we'll take a closer look.

- We refer to arguments passed into the script when it is run as *script arguments*. When the PythonWin IDE is used to run a script, these are passed in to the script via the 'Arguments' text box in the 'Run Scripts' dialog window.



Script File: C:\gispy\sample_scripts\ch15\setEnv3.py

Arguments: C:/gispy/scratch True

Debugging: No debugging

- The script arguments can then be retrieved inside the script using the `sys.argv` list or using the `GetParameterAsText` function in `arcpy` scripts.
- *Function arguments*, on the other hand, are those passed into a function when it is called in the script.

To clarify the difference, consider the three scripts in Example 15.3. 'setEnv1.py' takes no arguments whatsoever. It hard-codes the values for the environment settings. The `setEnviron1` function could be called repeatedly during a script to reset these properties after changes have been made. The second script, 'setEnv2.py' uses function arguments. Values are passed into the function to adjust these properties as needed. The third script, 'setEnv3.py', uses both script arguments and function arguments. The script arguments are used to set the `wSpace` and `overwrite` variables. The last line of this script passes these variables in to `setEnviron3` as function arguments.

Example 15.3

```
# setEnv1.py
import arcpy

def setEnviron1():
    arcpy.env.workspace = 'C:/gispy/data/ch15'
    arcpy.env.overwriteOutput = True
setEnviron1()
-------------------------
# setEnv2.py
import arcpy

def setEnviron2(workspace, overwriteVal):
    arcpy.env.workspace = workspace
    arcpy.env.overwriteOutput = overwriteVal

wSpace = 'C:/gispy/data/ch15/tester.gdb'
overwrite = False
setEnviron2(wSpace, overwrite)
-------------------------
```

```
# setEnv3.py
import arcpy, sys
wSpace = sys.argv[1]
overwrite = sys.argv[2]

def setEnviron3(workspace, overwriteVal):
    arcpy.env.workspace = workspace
    arcpy.env.overwriteOutput = overwriteVal
setEnviron3(wSpace, overwrite)
```

15.2.2 Optional Arguments

Many functions have optional parameters as well as required parameters. For example, the `arcpy` Buffer function requires three arguments (input features, output name, and buffer distance), but it also has four optional arguments that can be used to further specify the buffer behavior. Custom functions can be designed to handle optional parameters. Optional parameters are given a default value in the signature with the following format:

```
def funcName(reqP1,reqP2,…,optP1=defaultV1,optP2=defaultV2,…):
    '''Docstring…'''
    code statement(s)
```

Parameters that are given a default value in the signature are optional—if the caller does not pass in an argument for that parameter, the default value is used. The following function has one required and one optional parameter:

```
def setEnviron4(workspace, overwriteVal = True):
    arcpy.env.workspace = workspace
    arcpy.env.overwriteOutput = overwriteVal
```

The function can be called using one argument or two. Either one of the following statements can be used to call the `setEnviron4` function:

```
>>> setEnviron4('C:/gispy/data/ch15', False)
>>> setEnviron4('C:/gispy/data/ch15')
```

Both calls have the same effect, except the first one sets the output overwriting property to `False` and the second one sets it to `True`. A function can take any number of required and optional arguments, but required arguments must come before optional ones in the signature, since arguments are mapped to parameter positions simply by the order in which they are passed.

The function examples so far in this chapter have printed information, performed geoprocessing, and modified properties, but they haven't returned values. Another common purpose for functions is to calculate or derive values and return them to the caller. The next section discusses returning values in custom functions.

15.3 Returning Values

We've used functions that simply print information or modify the environment (e.g., the built-in `help` function or the `arcpy Delete` function); And we've used others that return values (e.g., the built-in `round` function or the `arcpy ListRasters` function). Now we'll discuss how to return values in custom functions. In terms of our butler metaphor, he can perform tasks that modify our environment (e.g., dim the lights) and he can also perform tasks that generate a tangible result (e.g., bring tea).

To create a custom function that returns a value to the caller, use a `return` statement. The value that follows the `return` keyword is returned to the caller and can be stored in a variable with an assignment statement. A function that returns a value has the following general format:

```
def functionName (param1, param2, param3,...):
    '''Docstring ...'''
    code statement(s)
    return valueToBeReturned
```

The format for calling functions that return values is as follows:

```
variableName = functionName(arguments1, argument2, argument3,...)
```

The following code defines a function that returns the list of field names for a given dataset:

```
# excerpt from fieldHandler.py
import arcpy

def getFieldNames(data):
    '''Get a list of field names.'''
    fields = arcpy.ListFields(data)
          fnames = [f.name for f in fields]
          return fnames
```

When the function is called, the function code is executed and the return statement sends information back to the caller. The following code calls the getFieldNames function with the full path of an input dataset as an argument:

```
>>> names = getFieldNames('C:/gispy/data/ch15/park.shp')
```

The function gets a list of the Field objects and derives the names and returns the resulting list. An assignment statement stores the return value in the names variable. The following code prints the result:

```
>>> names
[u'FID', u'Shape', u'COVER', u'RECNO']
```

Return values can be any data type. The following code defines a function that returns a Python Boolean value (True or False):

```
def fieldExists(data, name):
    '''Check if a given field name already exists.'''
    fieldNames = getFieldNames(data)
    isThere = name in fieldNames
    return isThere

>>> result = fieldExists('C:/gispy/data/ch15/park.shp', 'COVER')
>>> result
True
```

Custom functions can call other custom functions that are defined within the same module. Both getfieldNames and fieldExists are defined in the 'fieldHandler.py' script, so fieldExists can call getFieldNames. A custom function can also be called from another script, but the syntax is slightly different as discussed in the upcoming chapter on custom modules.

Whenever a return statement is reached within a function, the execution exits the function and goes back to where the function was called. This means that any lines of code that follow a return statement in the execution flow will not be executed. The 'countIntersection' function in Example 15.4 handles this incorrectly. The function calculates a temporary dataset, an intersection of features and then it calculates the number of features in the intersection file, and returns the value. Though the author may have intended to delete the temporary dataset before leaving the function, the code will never reach the Delete (Data Management) tool call because it occurs after the return statement.

Example 15.4

```
# oops.py
# Purpose: Count the number records in an intersections
#     between two datasets and delete the intersection file
#     (but the intersection output is not deleted since this line
#     of code is placed after the 'return' statement).
import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch15/tester.gdb'
arcpy.env.overwriteOutput = True

def countIntersection(dataList):
     '''Calculate the number of features in the intersection.'''
     tempData = 'intersectOut'
     arcpy.Intersect_analysis(dataList, tempData)
     res = arcpy.GetCount_management(tempData)
     print '{0} created.'.format(tempData)
     return int(res.getOutput(0))
     #uh-oh! The deletion is not going to happen.
     arcpy.Delete_management(tempData)
     print '{0} deleted.'.format(tempData)

inputData = ['schools','workzones']
count = countIntersection(inputData)
print 'There are {0} intersections.'.format(count)
```

The examples given so far have used the `return` keyword only once. It can be used more than once in a function, though it's not recommended. Example 15.5 imports the Python built-in `datetime` module to deal with Gregorian calendar dates. The call to this function passes in the birth date April, 20, 2003. This is converted to a `datetime` object. It then finds this year's birthday and compares the birthday to today's date. If the birthday has already occurred this year, it returns the difference between the years; Else, it takes an additional year off and returns this value. The `calculateAge` function uses two `return` statements, but it could easily be rewritten to only use one. Execution leaves the function when it reaches a `return` statement. When multiple `return` statements are used, there is more than one way to exit the function. Style guidelines discourage multiple `return` statements because, ideally there should only be one way to exit a function—making the code easier to interpret.

Example 15.5

```
# age.py
# Purpose: Calculate age.
import datetime

def calculateAge(yr, mo, day):
    '''Calculate age based on the given birth date.'''
    # Get datetime objects for birth date and today.
    born = datetime.date(yr, mo, day)
    today = datetime.date.today()
    # Get this year's birthday and handle leap year exceptions.
    try:
        birthday = born.replace(year=today.year)
    except ValueError:
        birthday = born.replace(year=today.year, day=born.day-1)
    # Return age.
    if birthday < today:
        return today.year - born.year
    else:
        return today.year - born.year - 1

print calculateAge(2012, 4, 20)
```

Usually, multiple `return` statements can be avoided by employing a variable within branches. For example, the last five lines of the `calculateAge` function can be replaced by the following lines of code which using only one `return` statement:

```
#Return age
if birthday < today:
        age = today.year - born.year
else:
        age = today.year - born.year - 1
return age
```

Both alternatives return the same value; However, the single return is preferred since the code can become difficult to follow in more complex functions when multiple exit locations are possible.

The examples given here store the return values in a variable and print the variable. Printing the return values helps with understanding what a custom function does, but it's important to note the difference between returning values and printing values within a function.

15.3.1 A Common Mistake: Where Did the `None` Come from?

In Python, all functions return something even if a `return` statement is not used. The `return` statement returns a value explicitly and should be used if the function is intended to return a value. However, if no `return` statement is used, the function returns `None`, which is a Python built-in constant that represents a null value. It's as if an implicit `return None` statement is added to the functions when no explicit one is used. The following script demonstrates a common mistake involving this phenomenon:

```
# except from returnVSprint.py
def positiveMinV1(numList):
    '''Find the minimum positive number in the list'''
    pos = []
    for val in numList:
        if val >= 0:
            pos.append(val)
    print min(pos)

theList = [8, 2.5, 0, 12, 5]
value = positiveMin(theList)
print value
```

This script prints the following output:

```
>>> 2.5
None
```

The function `positiveMinV1` correctly finds the minimum positive number in the list and prints it (2.5). But you may not have expected to see `None`. Can you spot the mistake? The function is called and the return value is being stored in the `value` variable. Since there is no explicit `return` statement, the function returns `None`, which is printed with the last line of code. Probably, the intention was not to print the value inside the function but rather to return the value as in the following code:

```
def positiveMinV2(numList):
    '''Find the minimum positive number in the list'''
    pos = []
    for val in numList:
        if val >= 0:
            pos.append(val)
    return min(pos)
```

> **Note** If a script prints a mysterious `None`, it often means that it's printing the return value of a function which does not contain an explicit `return` statement.

When authoring a function, determine if the purpose of the function is to return a value or to print information inside the function and design the function appropriately.

15.3.2 Returning Multiple Values

At times, you may want to return more than one value with a function. Suppose, for example, the function returns both the x and y coordinates of a point. Both x and y can be returned as separate values by using a comma to separate the values in the `return` statement. This returns a Python tuple. The following code defines a `midPoint` function which returns an x and a y value in the `return` statement:

```
# excerpt from returnMultVals.py

def midPoint(x1, y1, x2, y2):
    '''Calculate the midpoint of line segment (x1,y1), (x2,y2).'''
    xVal = (x1 + x2)/2.0
    yVal = (y1 + y2)/2.0
    return xVal, yVal
```

The calling function can use comma separated variable names to receive the return values, as in the following code which finds the midpoint of the line segment from (3,5) to (2,1):

```
>>> x, y = midPoint (3,5,2,1)
>>> x
3.0
>>> y
3.0
```

Alternatively, a single tuple variable can be used to store multiple returned values. Example 15.6 gets the current time before and after walking through the given subdirectory using the `arcpy.da.Walk` method. The count is found and printed for each file. Then the `diffTime` function is used to calculate the amount of time elapsed. The `diffTime` function returns a tuple of time components (weeks, days, hours, and so forth). The variable `t` is used to store the tuple when it is returned. Then a print statement indexes into the tuple to access the returned values.

Example 15.6

```
# walkCount.py
# Purpose: Walk and get the record count for
#    each file, where possible.
# Usage: inputdirectory
# Example input: C:/gispy/data/ch15
import arcpy, datetime, sys
mydir = sys.argv[1]
def diffTime(start, end):
    '''Calculate the difference between two datetime objects'''
    difference = end - start
    weeks, days = divmod(difference.days, 7)
    minutes, seconds = divmod(difference.seconds, 60)
    hours, minutes = divmod(minutes, 60)
    return weeks, days, hours, minutes, seconds

before = datetime.datetime.now()
for root, dirs, files in arcpy.da.Walk(mydir):
    for f in files:
        try:
            count = arcpy.GetCount_management(root + "/"+ f)
            print '{0}/{1}    Count = {2}'.format(root,f,count)
        except arcpy.ExecuteError:
            print arcpy.GetMessages()

after = datetime.datetime.now()

t = diffTime(before, after)

print 'Time elapsed: {0} weeks, {1} days,
{2}:{3}:{4}'.format(t[0],t[1],t[2],t[3],t[4])
```

When the number of return values is not known in advance, a list can be used to return the values, as shown in the `getFieldNames` function in sample script 'fieldHandler.py'.

15.4 When to Write Functions

To identify places where functions would be useful in your code, find related blocks of code that are repeated within scripts. For example, the following code uses several statements to print selected portions of the geoprocessing messages after two tools are called:

```
# scriptWithoutFunction.py
# Purpose: Call three tools (to find avg. nearest neighbor, intersection,
#          and get count) Print the results from avg. nearest neighbor
#          and get count without using a function.
import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch15/tester.gdb'
res = arcpy.AverageNearestNeighbor_stats('schools')
resList = res.getMessages().split('\n')
for message in resList:
    if '...' not in message and 'Time:' not in message:
        print message
```

```
arcpy.Intersect_analysis(['schools','workzones'],'intersectOutput')

res = arcpy.GetCount_management('intersectOutput')
resList = res.getMessages().split('\n')
for message in resList:
    if '...' not in message and 'Time:' not in message:
        print message
```

Example 15.7 moves the repeated code in 'scriptWithoutFunction.py' into a function named `reportResults`. The script then calls the function twice, clearing the clutter so that it's easier to tell at a glance the main activity occurring in the script—calling three geoprocessing tools, `AverageNearsestNeighbor`, `Intersect`, and `GetCount`.

It may only make sense to call a function such as `printArgs` (in Examples 15.1) one time in a script—there is only one set of arguments coming into the script, so it's unlikely to be useful to print them more than once. However, the function still bundles those related code statements together, so that the details of the operation appear as one statement within the main flow of the code and the reader can chose to ignore these details to focus on understanding the overall purpose of the function. A well-named function and its docstring can serve as a sufficient shorthand to signify its purpose.

> Functions are a good way to organize code, grouping related statements together, so that they can be called one or more times from within the same script or from other scripts.

Also, although a function such as `printArgs` may only be useful once in each script, it is likely to be useful in many scripts. Grouping the code into a function makes it easier to grab the related block and insert it into another script. In fact, in an upcoming chapter, we'll discuss how to call a function that's in another script, so that you can write a function once and call it from many scripts.

Example 15.7

```
# scriptWithFunction.py
# Purpose: Call three tools (to find avg. nearest neighbor,
#          intersection, and get count) Print the results from avg.
#          nearest neighbor and get count using a function.

import arcpy
arcpy.env.workspace = 'C:/gispy/data/ch15/tester.gdb'

def reportResults(resultObj):
    '''Print selected result messages.'''
    resList = resultObj.getMessages().split('\n')
    for message in resList:
        if '...' not in message and 'Time:' not in message:
            print message
```

```
res = arcpy.AverageNearestNeighbor_stats('schools')
reportResults(res)

arcpy.Intersect_analysis(['schools','workzones'], 'intersectOutput')
res = arcpy.GetCount_management('intersectOutput')
reportResults(res)
```

15.4.1 Where to Define Functions

In addition to the constraint of defining functions before calling them, programmers follow a few other patterns so that it's easy to locate function definitions in the script. The standard pattern groups function definitions together near the beginning of the script. Generally, they should not be dispersed throughout the code. Figure 15.2 shows an example of poor organization (on the left) and improved organization on the right. The header comments should be first, followed by any imports. Next, all the functions should be defined (usually in alphabetical order by the function names). The main processing activities and any calls to the function should follow the function definitions. Generally, functions should not be defined inside a loop.

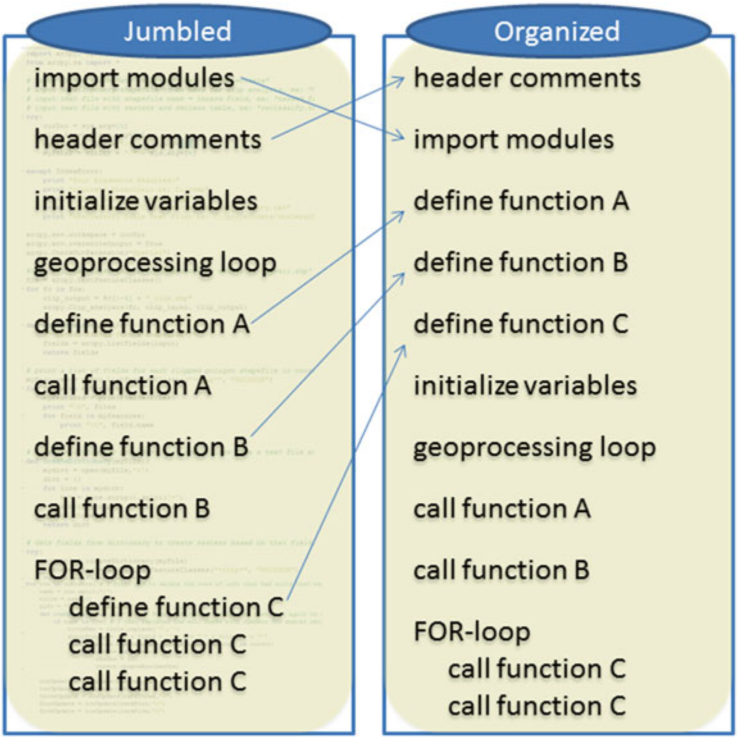


Figure 15.2 Example of poor code organization (on the left) and improved organization (on the right).

15.5 Variables Inside and Outside of Functions

When programming with functions, you need to be aware of several scenarios where mutable variables work differently from immutable ones. This section gives some examples and guidelines for avoiding confusion.

15.5.1 Mutable Arguments Can Change

Mutable and immutable variables behave somewhat differently when it come to functions. We previously discussed the concept of mutability in the context of lists and strings and how their methods work. Recall that indexing can be used to change the values of items in a mutable sequence such as a list:

```
>>> myList = ['a','b','c']
>>> myList[0] = 'z'
>>> myList
['z', 'b', 'c']
```

Also, the methods of a mutable object can modify the object itself:

```
>>> myList.sort()
['b', 'c', 'z']
```

The opposite is true of immutable data types such as strings:

```
>>> myStr = 'abc'
>>> myStr[0] = 'z'
Traceback (most recent call last):
File "<interactive input>", line 1, in ?
TypeError: object does not support item assignment
>>> myStr.replace('a','z')
'zbc' #This is the return result.
>>> myStr
'abc' #The string value is unchanged.
```

The mutability of a data type also effects how it behaves as an argument. Changes made to mutable function arguments within the function persist outside of the function; whereas, the opposite is true of immutable arguments. The following function modifies a list and an integer within the function:

```
def augment(myList, myInt):
        myList.append('some value')
        myInt = myInt + 1
```

The following code defines a list and a numeric variable and then calls the `augment` function:

```
>>> aList = ['first entry']
>>> num = 5
>>> print aList
['first entry']
>>> print num
5
>>> augment(aList, num)
```

After the function is called, the list is altered and the number is not:

```
>>> print aList
['first entry', 'some value']
>>> print num
5
```

The list is altered even if it is not returned to the caller. If this is an unintended result, you should use a deep copy which creates a new list object and then perform operations on the new list, as in the following example:

```
def augmentList(list1):
    list2 = list(list1)
    list2.append('some value')
    return list2
aList = ['first entry']
result = augmentList(aList)
```

In this case, the function preserves the original list and returns the modified list:

```
>>> print aList
['first entry']
>>> print result
['first entry', 'some value']
```

If you are familiar with the concepts of passing ‘by reference’ versus ‘by value’ from other programming languages, this is related to that. However, the ‘by value’ vs. ‘by reference’ discussion regarding Python leads to some confusing nuances about semantics. Instead, we chose to use the concept of mutable vs. immutable which avoids this confusion. Table 15.1 lists several familiar immutable and mutable data types. Sets and dictionaries will be covered in an upcoming chapter.

Table 15.1 Examples of mutable and immutable data types.

Immutable	Mutable
Numbers	Lists
Strings	Dictionaries
Tuples	Sets

15.5.2 Pass in Outside Variables

The first line of code in Example 15.8 assigns the value of 5 to the variable named `x`. This variable is created within the script but not within a class or another module—we'll refer to this as a *script level* variable. When a function uses a variable such as this, the variable should be passed into the script as an argument. The consequences of failing to do so are different depending on mutability. If the data type is immutable, the function can not alter the value of the variable. Attempting to do so throws an exception. In Example 15.8, the numerical variable, `x`, is defined at the beginning of the script, outside of the function definition. This numeric (immutable) variable can then be used anywhere in the script outside of the function. E.g., it is printed before the call to `addOne`. But `x` cannot be modified inside the definition of `addOne`. An `UnboundLocalError` occurs when the function attempts to assign a new value to `x`.

Example 15.8

```
# passVars.py
# Purpose: Demonstrate 'UnboundLocalError'.
# Usage: No script arguments needed.
x = 5
def addOne():
    x = x + 1
    print 'In here', x

print 'Out here', x
addOne()
```

Output in the Interactive Window:

```
>>> Out here 5
Traceback (most recent call last):
...
    x = x + 1
UnboundLocalError: local variable 'x' referenced before assignment
```

The problem can be corrected either by redefining the function so that it takes the value of `x` as an argument or by using a `global` statement inside the function. The second option is generally frowned upon because it can lead to bugs that are difficult uncover, so we won't discuss this option further.

The behavior shown in Example 15.8 protects programmers from changing the values of immutable variables unintentionally. However, if the data type is mutable, the function can alter the value (even if that was not the caller's intent). The following function finds the maximum number in a list, adds one to the number, and then appends this number to the list:

```
def appendNext():
    maxVal = max(myList)
    maxVal = maxVal + 1
    myList.append(maxVal)
myList = [1,2,3]
appendNext()
```

The new number has been added to the `myList` variable:

```
>>> myList
[1, 2, 3, 4]
```

Since no arguments are passed into the function, when the code (and function) becomes more complex, it may not be obvious that calling this function is going to change an outside variable. So best practice is to explicitly pass in script level variables to be used within functions.

> **Note** For consistency and transparency, don't rely on mutability—pass in script level variables to be used within functions as arguments. But remember that the script level value of a mutable argument can be altered by a function.

15.6 Key Terms

Custom functions
Default arguments
Docstrings
The `return` keyword
Script level variable

15.7 Exercises

1. **Test how it works:** Sample script 'circles.py' contains three custom functions and several calls to the functions. Run the code in debug mode following the given instructions and answer the related questions.

(a) Step through the code using the 'Step over' button twice to step over the import statement and then use only the 'Step (in)' button, counting the steps until you reach line 9. How many clicks were required to reach line 9?
(b) What is the value of 'mode' at this point?
(c) Continue stepping with the 'Step (in)' button until you step into the `return` statement. What is the name of the script that opens at this point? Explain how to quickly step out of this script and return to 'circles.py'.
(d) Run the script to completion with the 'Go' button and explain why the print statement on line 13 is not printed in the Interactive Window.
(e) Again, step through the code, this time only using the 'Step over' button until you exit the script. How many clicks were required to exit the script? Why is this number about three times less than the number of lines of executable code in the script?
(f) Which line of code printed the built-in constant, `None` and why?

2. **Use function-related terms:** Sample script 'circles.py' contains three custom functions and several calls to the functions. Identify the following code components for each function:

   (a) Function name.
   (b) Signature.
   (c) Docstring.
   (d) Required arguments.
   (e) Optional arguments.
   (f) Data type of the return value (string, int, list, Boolean, Constant, etc. or N/A if there is none).
   (g) Line(s) where the functions is called.

3. **functionPractice.py** Practice using function syntax by replacing the pseudocode in 'functionPractice.py' to define two functions and call them within a loop. The functions are described in parts a and b. The output prints information about each polygon feature class in the input workspace separated by fish punctuation art. Comments in the script demonstrate the sample output for a given input.

   (a) Function `printFish` should print a fish like this (no function arguments needed):

```
   '''
<')}>)={
    ``
```

   (b) Function `printDescription` should take one argument, the name of a GIS data file and then it should print information `Describe` object properties `name`, `dataType`, and `catalogPath`. A feature class named 'park' in 'tester.gdb' yields the following output:

```
Name: park
DataType: FeatureClass
CatalogPath: C:/gispy/data/ch15/tester.gdb\park
```

4. **boxes.py** Practice converting existing code to functions. Sample script 'boxes.py' contains several related blocks of code. Replace the existing code with three functions: `findPerimeter`, `isSquare`, and `createBoundingBoxes` and six function calls (two calls to each function). `findPerimeter` should require two arguments (length and width) and return the numeric perimeter of the box. `isSquare` should require two arguments (length and width) and return a Boolean value (`True` if the box is square and `False` otherwise). `createBoundingBoxes` should require two arguments, an input workspace and an output directory. It should set the `arcpy` workspace variable inside the function and then get the Polygon feature classes and the bounding box geoprocessing loop as in the given script but it should not return anything to the caller. Remember to group all the function definitions at the beginning of the script, just after the imports. Also, place the function calls so that the output is created in the same order as the given script.

5. **largePolys.py** Run the sample script 'largePolys.py' to observe how it works. Then modify the script so that it does NOT use a function. In other words, remove the `countLargePolygons` function definition and replace the function call with code inside the loop that achieves the same results.

6. **latLong.py** The sample script 'latLong.py' defines and calls a function (`dd2dms`) that converts numeric decimal degrees (dd) to string decimal/minutes/seconds (dms) format and a second function, `dms2dd` that does the converse. The script calls both the functions for two points, Point1 and Point 2 and prints the results. But there is an error in each function. Use the debugger to identify and repair the errors. Place a breakpoint on lines 21 and 31 and run to these points. Put variables `degs`, `mins`, `secs`, `dms`, and `dd` in the Watch window. Use code comments to record the errors you find in the script. The current output looks like this:

```
Point 1: (35.684072, -78.728027) -> (None, None)
Point 2: (-33 43 27.6234, 24 31 17.521484) -> (-32.2756601667,
24.5215337456)
```

When both the errors are corrected, the printed output should look like this:

```
Point 1: (35.684072, -78.728027) -> (35 41 2.6592, -78 43 40.8972)
Point 2: (-33 43 27.6234, 24 31 17.521484) -> (-33.7243398333,
24.5215337456)
```

7. **trapezoid.py** Sample script 'trapezoid.py' gives four measurements for three trapezoids, two base lengths (`b1` and `b2`), an altitude (`alt`), and an angle (`angle`). Add the following four functions to the script:

   (a) `calculateArea` should take three arguments, two base side lengths and the altitude, and it should return the area.

   (b) `isParallelogram` should take two arguments, the two base lengths. It should return `True` if the base lengths are the same and `False` otherwise.

(c) `isRectangle` should take three arguments, the two base lengths and a corner angle in degrees. It should return `True` if both the angle is 90 and it is a parallelogram (call `isParallelogram` to determine this). It should return `False` otherwise.

(d) `isSquare` should take four arguments, two base lengths, an altitude length, and an angle. It should return `True` if it is a rectangle (call `isRectangle` to determine this) and the base length equals the altitude.

Then call the functions for the three quadrilaterals given in the script and print the return values. The output should look like this:

```
Quad1: b1 = 4, b2 = 4, alt = 6, angle = 90
Area = 24.0
Is parallelogram? True
Is rectangle? True
Is square? False

Quad2: b1 = 5, b2 = 5, alt = 5, angle = 30
Area = 25.0
Is parallelogram? True
Is rectangle? False
Is square? False

Quad3: b1 = 8, b2 = 8, alt = 6, angle = 60
Area = 48.0
Is parallelogram? True
Is rectangle? False
Is square? False
```

8. **triangles.py** Write a script 'triangles.py' that takes three arguments, the lengths of three sides of a triangle. Then define the following three functions:

(a) `perimeter` takes three side lengths as argument and returns the perimeter length.

(b) `triangleType` takes three side lengths as arguments, determines if it is an equilateral triangle, an isosceles triangle, or neither. The result is returned.

(c) `area` takes three sides lengths as arguments and returns the area. Compute this as the square root of `(p*(p-a)*(p-b)*(p-c))`, where `p` is half the perimeter and `a`, `b`, and `c` are the side lengths. Use a `math` module function to compute the square root.

Call the functions and print the results as shown in these examples:

Example input1: 1 1 1

Example output1:

```
>>> Triangle sides: 1.0, 1.0, 1.0
Perimeter = 3.0
Type = Equilateral
Area = 0.433012701892
```

Example input2: 5 5 6

Example output2:

```
>>> Triangle sides: 5.0, 5.0, 6.0
Perimeter = 16.0
Type = Isosceles
Area = 12.0
```

Example input3: 3 4 5

Example output3:

```
>>> Triangle sides: 3.0, 4.0, 5.0
Perimeter = 12.0
Type = Neither equilateral nor isosceles
Area = 6.0
```

9. **segProc.py** Write a script 'segProc.py' that defines and calls a function named 'segLength' that computes the length of line segment between two points, `(x1,y1)` and `(x2,y2)`. Use the Pythagorean theorem for Euclidean distance. Design 'segProc.py' like 'setEnv3.py' in Example 15.3 to use both script arguments and function arguments. The script should take four arguments and place these in variables, `x1`, `y1`, `x2`, and `y2`. The function should also be defined to take four arguments, the `x` and `y` values of two points. Call the function using the script arguments. Use a `return` statement in the function to return the distance. Print the returned values, formatting output as shown in the example.

   Example script input: 3.2 1 6 8

   Example script print output:

   ```
   Segment (3.20,1.00) to (6.00,8.00) has length: 7.54
   ```

Tip The following code formats a floating point value to two decimal places:

```
>>> a = 4.33732984
>>> print 'The number {0:.2f}'.format(a)
The number 4.34
```

10. **fieldFunc.py** Write a script 'fieldProc.py'. The script should define a function named `printFields` that prints the name of the input file and the names of the fields in the input file. The script should take two arguments, a full path input file name and an output workspace. The function should take one argument, an input file name. Add code to the script to do the following: Call `printFields` on the script argument, the input file given by the user. Use the Copy (Data Management) tool to copy the file to the output directory. Add a FLOAT field named `AREA` to the new file. Finally, call `printFields` again on the new file.

Example script input: C:/gispy/data/ch15/park.shp C:/gispy/scratch/

Example script output:

```
>>> Fields in C:/gispy/data/ch15/park.shp:
FID
Shape
COVER
RECNO
C:/gispy/data/ch15/park.shp copied to C:/gispy/scratch/park.shp.
Fields in C:/gispy/scratch/park.shp:
FID
Shape
COVER
RECNO
AREA
```

11. **outNameProc.py** An output file name often needs to be dynamically created based on an input file name. For example, from input file 'C:/Data/NC.txt', we may want to create 'C:/Data/out/NCBuff.shp' and from input file 'C:/Data/VA.txt', we may want to create 'C:/Data/out/VABuff.shp', and so forth. We can use the same string and `os` module operations to generate each output name based on each input named.

    Sample script 'outNameProc.py' calls a function named `outName` which has yet to be defined. Add code to this script to define function `outName` that will create an output file name based on an input file name and three additional arguments. Specifically, the function should be defined to take one required argument and three optional arguments:

A required input file name (possibly containing the full file path)
An optional string to insert in the name just before the extension. Set the default value for this as `'Out'`.
An optional file extension. Set the default value for this to `'shp'`.
An optional output directory. Set the default for this to an empty string (`''`).

In the function, use `os.path` commands (`basename`, `splitext`, and `join`) to get the basename of the input file, strip the file extension (if any) from the input file name, append the string to be inserted, append the new file extension, and prepend the output workspace.

Example 1:

```
>>> name1 = outName('C:/Data/NC.txt', 'Points', 'dbf',
'C:/gispy/outData/')
>>> name1
'C:/gispy/outData/NCPoints.dbf'
```

Example 2:

```
>>> name2 = outName('C:/states//PA.prj')
>>> name2
'PAOut.shp'
```

Define the function (including a docstring) and compare the results against the output shown in the ### script comments. This script only needs to perform string manipulation, so no data files are needed for testing.

12. **centralPoint.py** Add code to the script ‘centralPoint.py’ so that it takes two arguments, a full path polygon shapefile name and an output directory. Then have the script perform several geoprocessing operations to investigate the central trend of the polygons. Specifically, find the centroid of each polygon using the Feature to Point (Data Management) tool, and then find the central feature of the polygon centroids using the Central Feature (Spatial Statistics) tool and the mean center of the polygon centroids using the Mean Center (Spatial Statistics) tool. Finally, find the distance between the central feature and the mean center using the Point Distance (Analysis) tool. Use the functions `getTime` and `timeDifference` given in the script to report the amount of time each of these geoprocessing tool call requires. When run without debugging, the computation times may all be so small that they are rounded to zero seconds. Step through with the debugger and to see non-zero times.

    Example script input: C:/gispy/data/ch15/park.shp C:/gispy/scratch/

    Example script output in *debug* mode:

```
[Dbg] >>> Time for FeatureToPoint to create parkPoints.shp:
0 weeks, 0 days, 0:0:2
[Dbg] >>> Time for CentralFeature to create parkCentral.shp:
0 weeks, 0 days, 0:0:3
[Dbg] >>> Time for MeanCenter to create parkMean.shp:
0 weeks, 0 days, 0:0:2
[Dbg] >>> Time for PointDistance to create parkMean2Central.dbf:
0 weeks, 0 days, 0:0:1
```

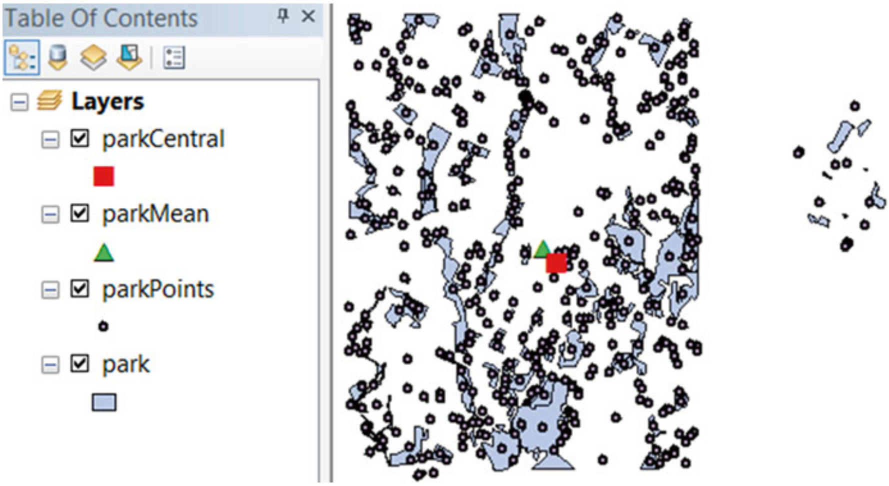


Table Of Contents
Layers
☑ parkCentral
🟥
☑ parkMean
▲
☑ parkPoints
●
☑ park
□