Chapter 2
Creating Python functions and classes

2.1 Introduction

This chapter describes how to create custom functions in Python that can be called from elsewhere in the same script or from another script. Functions are organized into modules, and modules can be organized into a Python package. ArcPy itself is a collection of modules organized into a package. By creating custom functions, you can organize your code into logical parts and reuse frequently needed procedures. This chapter also describes how to create custom classes in Python, which makes it easier to group together functions and variables.

Custom functions and classes are important for writing longer and more complex scripts. They allow you to better organize your code as well as reuse important elements of your code. A good understanding of functions and classes is also important because they are used frequently in other chapters in the book. Many example scripts and tools published by others also contain custom functions and classes.

2.2 Functions and modules

Before getting into creating custom functions, a quick review of functions is in order. *Functions* are blocks of code that perform a specific task. Python includes many built-in functions, such as `help()`, `int()`, `print()`, and `str()`. Most functions require one or more arguments, which serve as the input for the function.

Using a function is referred to as *calling* the function. When you call a function, you supply it with arguments. Consider the `print()` function:

```
name = "Paul"
print(name)
```

The result is

```
Paul
```

In this example, the argument of the `print()` function is a variable, and this variable has a value. The `print()` function outputs the value to the console.

The general syntax of a function is:

```
<function>(<arguments>)
```

In this syntax, `<function>` stands for the name of the function, followed by the parameters of the function in parentheses. Function arguments are also called *parameters*, and these terms are often used interchangeably.

Python has several dozen built-in functions. For a complete list of built-in functions, see https://docs.python.org/3/library/functions.html. You will use several built-in functions in a typical Python script.

You can also import additional functionality from other modules. A *module* is like an extension that can be imported into Python to extend its capabilities. Typically, a module consists of several specialized functions. Modules are imported using a special statement called `import`. The general syntax for the import statement is

```
import <module>
```

Once you import a module in a script, all functions in that module are available to use in that script. Consider the `random` module, for example. You can import this module to access several different functions. The following code generates a random number from 1 to 99 using the `randrange()` function of the `random` module.

```
import random
random_number = random.randrange(1, 100)
print(random_number)
```

The code to generate a random number has already been written and is shared with the Python user community. This code can be used freely by anyone who needs it. The `random` module contains several different functions, and many of them are closely related. Whenever your script needs a random number, you don't have to write the code yourself. You can import the `random` module and use any of its functions.

One of the most widely used modules is the `os` module, which includes several functions related to the operating system. For example, the `os.mkdir()` function creates a new folder in the current working directory, as follows:

```
import os
os.mkdir("test")
```

The general syntax to use a function from a module that is not one of the built-in functions is as follows:

```
import <module>
<module>.<function>(<arguments>)
```

In other words, you first must import the module using `import<module>`, and then reference the module when calling the function using `<module>.<function>.<parameters>`.

When writing scripts for ArcGIS Pro, you can use the ArcPy package to access the functionality of ArcGIS Pro within a script. ArcPy is referred to as a *package* because it consists of several modules, functions, and classes, but to work with ArcPy, you import it just like a module. That is why most geoprocessing scripts start off as follows:

```
import arcpy
```

Once you import ArcPy, you can use one of its many functions. For example, the `arcpy.Exists()` function determines whether a dataset exists and returns a Boolean value of `True` or `False`. The following code determines whether a shapefile exists:

```
import arcpy
print(arcpy.Exists("C:/Data/streams.shp"))
```

This code follows the regular Python syntax `<module>.<function>(<parameters>)`, where `arcpy` is the module, and `Exists()` is the function, even though ArcPy is technically considered a package.

ArcPy includes several modules, including the data access module `arcpy.da`. This module is used for describing data, performing editing tasks, and following database workflows. The `da.Describe()` function determines the type of dataset, as well as several properties of the dataset. For example, the following code determines the geometry shape type of a shapefile:

```
import arcpy
desc = arcpy.da.Describe("C:/Data/streams.shp")
print(desc["shapeType"])
```

For a polyline shapefile, the result is `Polyline`.

The general syntax for using a function of an ArcPy module is

```
arcpy.<module>.<function>(<arguments>)
```

In the preceding example code, `Describe()` is a function of the `arcpy.da` module.

When referring to a function, it is important to refer to the module that it is part of. For example, ArcPy also includes a `Describe()` function. So both `arcpy.Describe()` and `arcpy.daDescribe()` are valid functions, but they work in different ways.

Now that you've reviewed the use of functions and modules, the next section introduces creating your own custom functions.

2.3 Creating functions

In addition to using existing functions, you can create your own *custom functions* that can be called from within the same script or from other scripts. Once you write your own functions, you can reuse them whenever needed. This capability makes code more efficient because there is no need to write code for the same task over and over.

Python functions are defined using the `def` keyword. The `def` statement contains the name of the function, followed by any arguments in parentheses. The syntax of the `def` statement is

```
def <functionname>(<arguments>):
```

There is a colon at the end of the statement, and the code following a `def` statement is indented the same as any block of code. This indented block of code is the *function definition*.

For example, consider the script helloworld.py as follows:

```
def printmessage():
    print("Hello world")
```

In this example, the function `printmessage()` has no arguments, but many functions use parameters to pass values. Elsewhere in the same script, you can call this function directly, as follows:

```
printmessage()
```

The complete script is as follows:

```
def printmessage():
    print("Hello world")
printmessage()
```

When the script runs, the function definition is not executed. In other words, the line of code starting with `def` and the block of code that follows don't do anything. In the third line of code, the function is called, and then it is executed. The result of the script is

```
Hello world
```

This is a simple example, but it illustrates the basic structure of a custom function. Typically, functions are more elaborate. Consider the following example: you want to create a list of the names of all the fields in a table or feature class. There is no function in ArcPy that does this.

However, the `ListFields()` function allows you to create a list of the fields in a table, and you can then use a `for` loop to iterate over the items in the list to get the names of the fields. The list of names can be stored in a list object. The code is as follows:

```
import arcpy
arcpy.env.workspace = "C:/Data"
fields = arcpy.ListFields("streams.shp")
namelist = []
for field in fields:
    namelist.append(field.name)
```

Now, say you anticipate that you will be using these lines of code often—in the same script or other scripts. You can simply copy the lines of code, paste them where they are needed, and make any necessary changes. For example, you will need to replace the argument `"streams.shp"` with the feature class or table of interest.

Instead of copying and pasting the entire code, you can define a custom function to carry out the same steps. First, you must give the function a name—for example, `listfieldnames()`. The following code defines the function:

```
def listfieldnames():
```

You can now call the function from elsewhere in the script by name. In this example, when calling the function, you want to *pass* a value to the function—that is, the name of a table or feature class. To make this possible, the function must include an argument to receive these values. The argument must be included in the definition of the function, as follows:

```
def listfieldnames(table):
```

Following the `def` statement is an indented block of code that contains what the function does. This block of code is identical to the previous lines of code, but now the *hard-coded* value of the feature class is replaced by the argument of the function, as follows:

```
def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
```

Notice how there are no hard-coded values left in the function. The lack of hard coding is typical for custom functions because you want a function to be reusable in other scripts.

The last thing needed is a way for the function to pass values back, also referred to as *returning* values. Returning values ensures that the function not only creates the list of names, but also returns the list so it can be used by any code that calls the function. This is accomplished using a `return` statement. The completed description of the function is as follows:

```
def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
    return namelist
```

Once a function is defined, it can be called directly from within the same script, as follows:

```
fieldnames = listfieldnames("C:/Data/hospitals.shp")
```

Running the code returns a list of the field names in a table or feature class using the function previously defined. Notice that the new function `listfieldnames()` can be called directly because it is defined in the same script.

One important aspect is the placement of the function definition relative to the code that calls the function. The custom function can be called only after it is defined. The correct organization of the code is as follows:

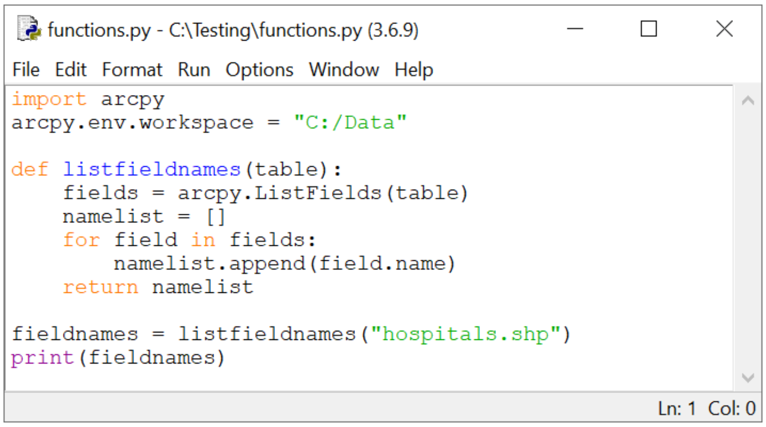
```
import arcpy
arcpy.env.workspace = "C:/Data"
def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
    return namelist
fieldnames = listfieldnames("hospitals.shp")
print(fieldnames)
```

If the function is called before the function definition, the following error is returned:

```
NameError: name 'listfieldnames' is not defined
```

Complex scripts with several functions therefore often start with defining several functions (and classes), followed by the code that calls these functions later in the script.

In addition, it is common to add empty lines around the blocks of code that define functions to improve readability, as shown in the following figure.


functions.py - C:\Testing\functions.py (3.6.9)
File Edit Format Run Options Window Help
Ln: 1 Col: 0

```
import arcpy
arcpy.env.workspace = "C:/Data"

def listfieldnames(table):
    fields = arcpy.ListFields(table)
    namelist = []
    for field in fields:
        namelist.append(field.name)
    return namelist

fieldnames = listfieldnames("hospitals.shp")
print(fieldnames)
```

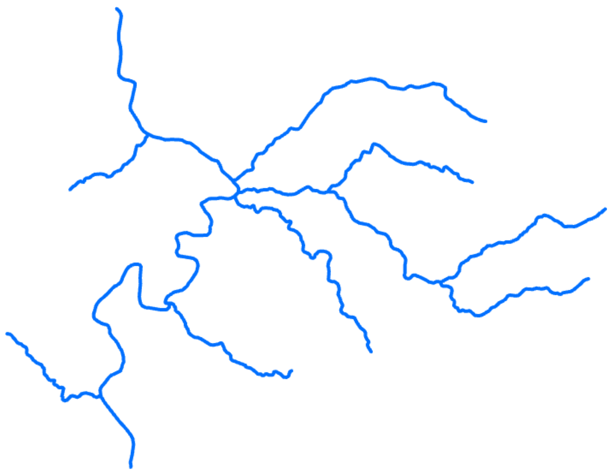
The example function uses an argument, called `table`, which makes it possible to pass a value to the function. A function can use more than one argument, and arguments can be made optional. The arguments should be ordered so that the required ones are listed first, followed by the optional ones. Arguments are made optional by specifying default values.

Custom functions can be used for many other tasks, including working with geometry objects. Next, an example script is explained, and then it will be converted to a custom function. The example script calculates the sinuosity index for each polyline feature representing a river segment. Sinuosity, in this context, is defined as the length of the polyline representing the river segment divided by the straight-line distance between the first and last vertex of the polyline. Segments that are relatively straight have a sinuosity index of close to 1, whereas meandering segments have higher values, up to 1.5 or 2. The calculation can be accomplished by using properties of a `Polyline` object—i.e., `length`, `firstPoint`, and `lastPoint`. The script to print the sinuosity index for every polyline feature in a feature class is as follows:

```
import arcpy
import math
arcpy.env.workspace = "C:/Data/Hydro.gdb"
fc = "streams"
with arcpy.da.SearchCursor(fc, ["OID@", "SHAPE@"]) as cursor:
    for row in cursor:
        oid = row[0]
        shape = row[1]
        channel = shape.length
        deltaX = shape.firstPoint.X - shape.lastPoint.X
        deltaY = shape.firstPoint.Y - shape.lastPoint.Y
        valley = math.sqrt(pow(deltaX, 2) + pow(deltaY, 2))
        si = round(channel / valley, 3)
        print(f"Stream ID {oid} has a sinuosity index of {si}")
```

A brief explanation of how the script works is in order. A search cursor is used to obtain the unique ID and the geometry of each polyline. The `length` property of the geometry represents the length of the polyline. The straight-line distance between the first and last vertex of the polyline is calculated using the `firstPoint` and `lastPoint` properties of the geometry, which return a `Point` object. The x,y coordinates of these vertices are used to calculate the distance on the basis of the Pythagorean theorem. The two distances are divided to obtain the sinuosity index, and for display purposes, the values are rounded to three decimal places.

Consider the following stream network, as shown in the figure.



The result of the script is a printout of the sinuosity index of each segment.

```
Stream ID 1 has a sinuosity index of 1.246
Stream ID 2 has a sinuosity index of 1.118
Stream ID 3 has a sinuosity index of 1.059
Stream ID 4 has a sinuosity index of 1.329
Stream ID 5 has a sinuosity index of 1.067
Stream ID 6 has a sinuosity index of 1.092
Stream ID 7 has a sinuosity index of 1.026
Stream ID 8 has a sinuosity index of 1.139
Stream ID 9 has a sinuosity index of 1.128
Stream ID 10 has a sinuosity index of 1.655
Stream ID 11 has a sinuosity index of 1.178
Stream ID 12 has a sinuosity index of 1.285
Stream ID 13 has a sinuosity index of 2.186
Stream ID 14 has a sinuosity index of 1.3
Stream ID 15 has a sinuosity index of 1.209
Stream ID 16 has a sinuosity index of 1.279
Stream ID 17 has a sinuosity index of 1.065
```