Chapter 10

Creating Custom Geoprocessing Tools

In this chapter, we will cover the following exercises:

- Creating a custom geoprocessing tool
- Creating a Python toolbox

In addition to accessing the system geoprocessing tools provided by ArcGIS Pro, you can also create your own custom tools. These tools work in the same way that system tools do and can be used in ModelBuilder, Python window, or a standalone Python scripts. Many organizations build their own library of tools that perform geoprocessing operations specific to their data. In ArcGIS Pro there are two ways that you can create custom geoprocesisng tools: custom script tools in a custom toolbox and custom script tools in a Python toolbox. Both methods essentially accomplish the same thing, but the workflow for creating the custom tools differs significantly. We'll cover both methods in this chapter.

Exercise 1: Creating a custom geoprocessing tool

Getting ready

In this exercise, you will learn to create custom geoprocessing script tools by attaching a Python script to a custom toolbox. There are a number of advantages to creating a custom script tool. When you take this approach, the script becomes a part of the geoprocessing framework, which means that it can be run from a model, command line, or another script. In addition to this, the script has access to ArcGIS Pro environment settings and help documentation. Other advantages include a nice, easy-to-use user interface and error prevention capabilities. Error prevention capabilities provided include a dialog box that informs the user of certain errors.

These custom developed script tools must be added to a custom toolbox that you create, because the system toolboxes provided with ArcGIS Pro are read-only toolboxes and thus can't accept new tools.

In this exercise, you are going to be provided with a pre-written Python script that reads wildfire data from a comma-delimited text file, and writes this information to a point feature class called `FireIncidents`. References to these datasets have been hardcoded, so you are going to alter the script to accept dynamic variable input. You'll then attach the script to a custom tool to give your end users a visual interface for using the script.

How to do it...

The custom Python geoprocessing scripts that you write can be added to custom toolboxes. You are not allowed to add your scripts to any of the system toolboxes, such as **Analysis** or **Data Management**. However, by creating a new custom toolbox, you can add these scripts.

1. Open PyCharm
2. Select **File | New | Python File**.
3. Name the file `InsertWildfiresToolbox` and click **OK**. The file should be written to your default project location of `c:\Student\ProgrammingPro\Scripts` folder.
4. Import the `arcpy` and `os` modules and set up a basic `try/except` structure.

```
import arcpy, os
try:

except Exception as e:
   print("Error: " + e.args[0])
```

5. Create variables to hold the output feature class, feature class template, and text file. Use the `arcpy.GetParameterAsText()` function to retrieve these values from the script tool dialog.

```
import arcpy, os

try:
   outputFC = arcpy.GetParameterAsText(0)
   fClassTemplate = arcpy.GetParameterAsText(1)
```

```
    f = open(arcpy.GetParameterAsText(2), 'r')

except Exception as e:
    print("Error: " + e.args[0])
```

6. Create a new feature class using the input provided by the user in the first parameter.

```
import arcpy, os

try:
    outputFC = arcpy.GetParameterAsText(0)
    fClassTemplate = arcpy.GetParameterAsText(1)
    f = open(arcpy.GetParameterAsText(2), 'r')
    arcpy.CreateFeatureclass_management(
        os.path.split(outputFC)[0],
        os.path.split(outputFC)[1],
        "point",
        fClassTemplate)

except Exception as e:
    print("Error: " + e.args[0])
```

7. Read each of the lines in the text file into a Python list.

```
import arcpy, os

try:
    outputFC = arcpy.GetParameterAsText(0)
    fClassTemplate = arcpy.GetParameterAsText(1)
    f = open(arcpy.GetParameterAsText(2), 'r')
    arcpy.CreateFeatureclass_management(
        os.path.split(outputFC)[0],
        os.path.split(outputFC)[1],
        "point",
        fClassTemplate)
    lstFires = f.readlines()

except Exception as e:
    print("Error: " + e.args[0])
```

8. Defines a Python list containing the fields to be used when inserting geometry and attributes.

```
import arcpy, os

try:
   outputFC = arcpy.GetParameterAsText(0)
   fClassTemplate = arcpy.GetParameterAsText(1)
   f = open(arcpy.GetParameterAsText(2), 'r')
   arcpy.CreateFeatureclass_management(
      os.path.split(outputFC)[0],
      os.path.split(outputFC)[1],
      "point",
      fClassTemplate)
   lstFires = f.readlines()
   fields = ["SHAPE@XY", "CONFIDENCEVALUE"]

except Exception as e:
   print("Error: " + e.args[0])
```

9. Create an `InsertCursor` from the feature class, loop through each of the fires found in the text file, extract the coordinates and confidence value attributes, and insert the information into the output feature class.

```
try:
   outputFC = arcpy.GetParameterAsText(0)
   fClassTemplate = arcpy.GetParameterAsText(1)
   f = open(arcpy.GetParameterAsText(2), 'r')
   arcpy.CreateFeatureclass_management(
      os.path.split(outputFC)[0],
      os.path.split(outputFC)[1],
      "point",
      fClassTemplate)
   lstFires = f.readlines()
   fields = ["SHAPE@XY", "CONFIDENCEVALUE"]
   with arcpy.da.InsertCursor(outputFC, fields) as cur:
      for fire in lstFires:
         if 'Latitude' in fire:
            continue
         vals = fire.split(",")
         latitude = float(vals[0])
         longitude = float(vals[1])
         confid = int(vals[2])
         row_values = [(longitude, latitude), confid]
```

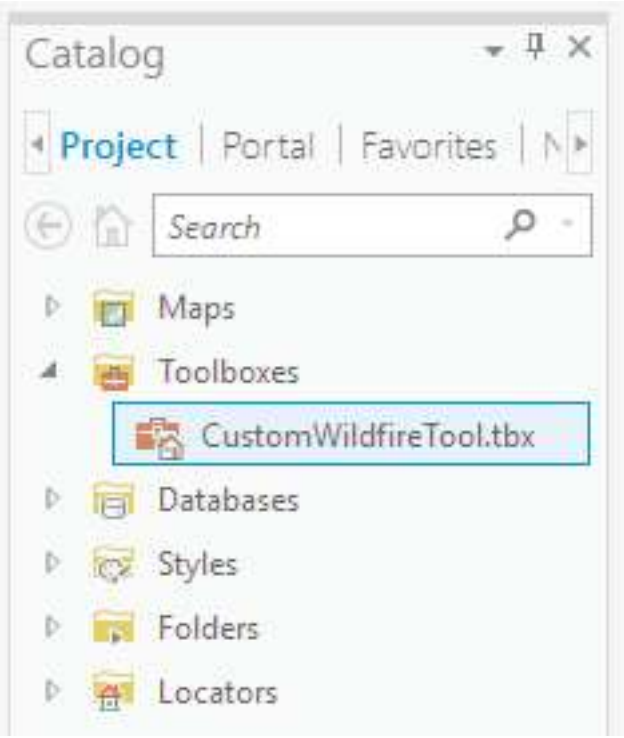
```
                cur.insertRow(row_values)

        except Exception as e:
            print("Error: " + e.args[0])
```

10. You can check your code against a solution file found at `c:\Student\ProgrammingPro\Solutions\Scripts\InsertWildfiresToolbox.py`.

    Now we'll attach the script to a tool using the ArcGIS Pro interface.

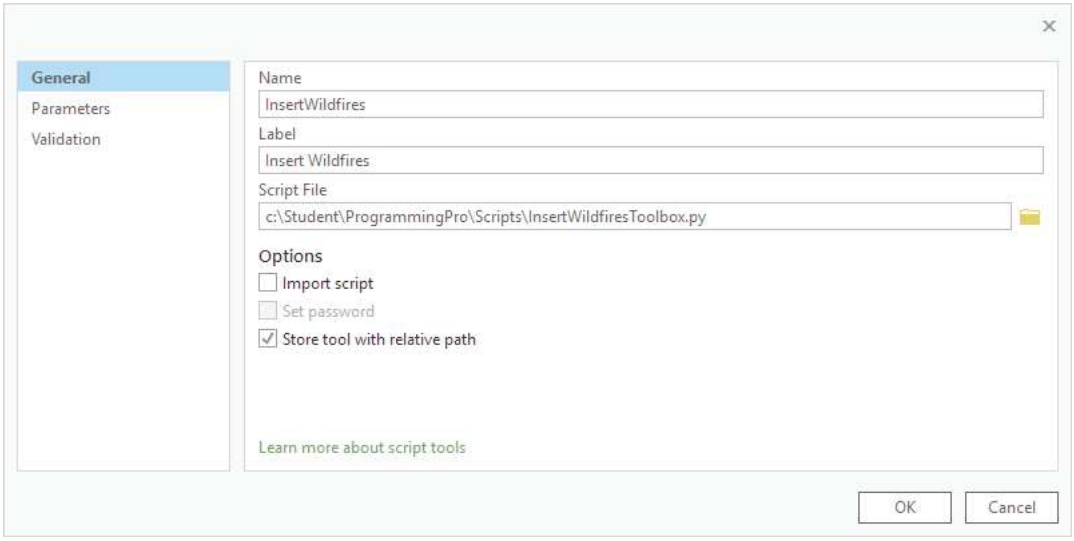
11. Open **ArcGIS Pro** and create a new project using the `Map.aptx` template. Name the project `CustomWildfireTool` and save it in the `c:\Student\ProgrammingPro\My Projects` folder.

12. In the **Catalog pane** open the **Toolboxes** folder and find the `CustomWildfireTool.tbx` toolbox.



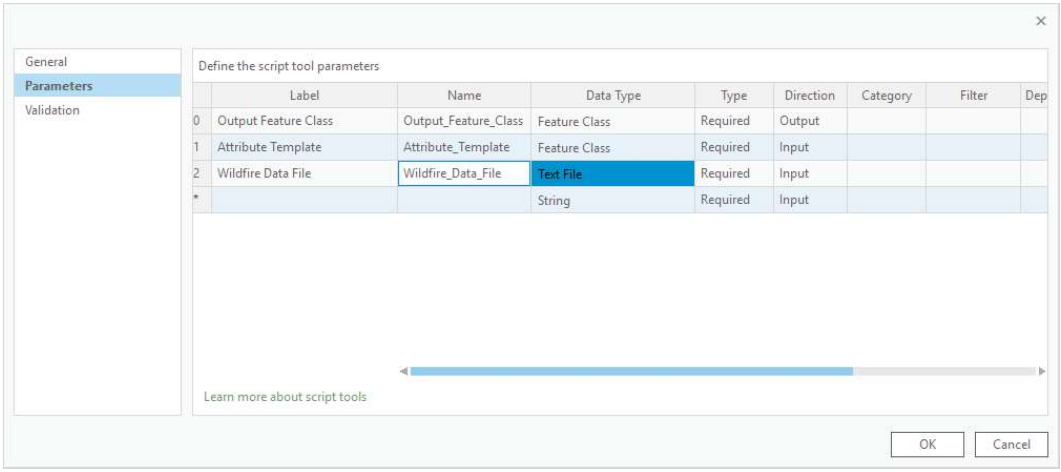
13. Right click `CustomWildfireTool.tbx` and select **New | Script**.

14. The **General** dialog will be displayed initially. Here you will enter the following parameters.

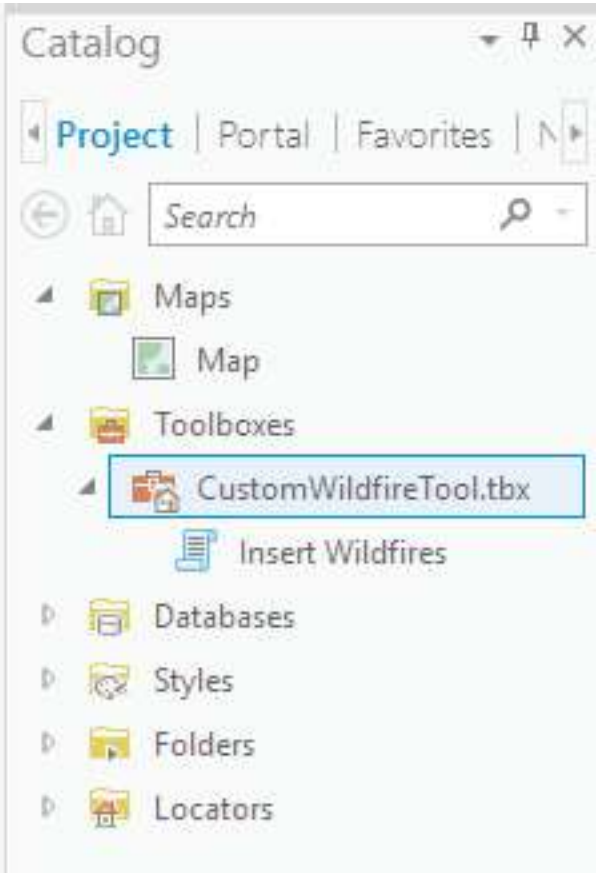


- Name: `InsertWildfires`
- Label: `Insert Wildfires`
- Script File: `c:\Student\ProgrammingPro\Scripts\InsertWildfiresToolbox.py`

15. Click the **Parameters** tab and enter the parameters seen in the screenshot below.



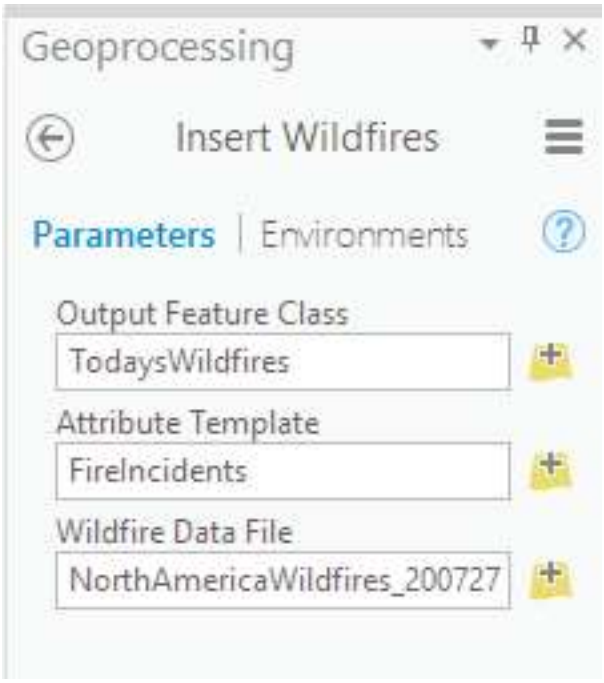
16. This tool won't include any custom validation so you can click **OK** to create the custom script tool seen in the screenshot below.



17. Before running the tool you'll need to create a connection to the geodatabase containing the `Wildfire` geodatabase. In the **Catalog** pane, Open the **Databases** folder and select **Add Database**.

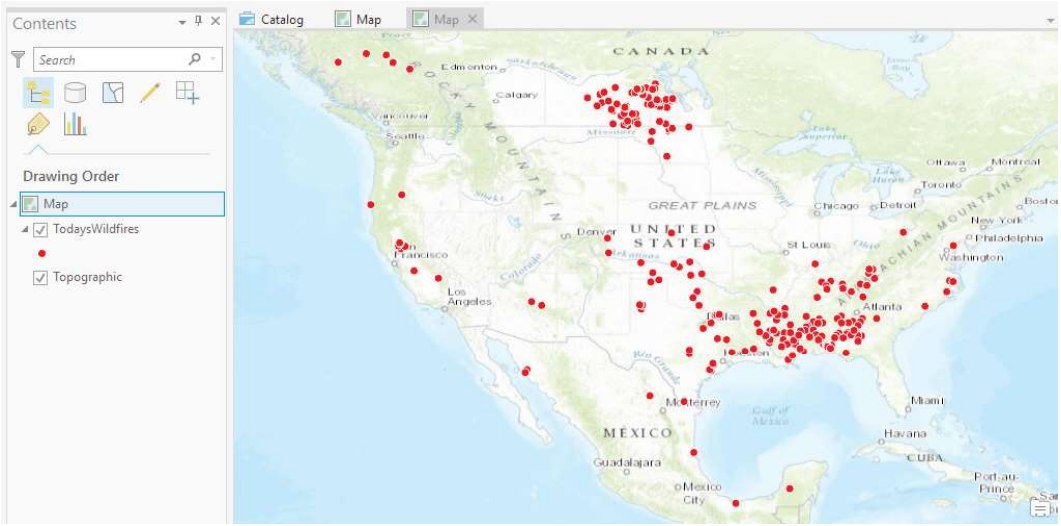
18. Navigate to the `c:\Student\ProgrammingPro\Databases` folder and select `WildlandFires.gdb`.

19. Double click the **InsertWildfires** tool found in the **CustomWildfireTool** toolbox and fill in the parameters as defined below:



- Output Feature Class – `C:\Student\ProgrammingPro\Databases\WildlandFires.gdb\TodaysWildfires`
- Attribute Template – `C:\Student\ProgrammingPro\Databases\WildlandFires.gdb\FireIncidents`
- Wildfire Data File – `C:\Student\ProgrammingPro\Databases\NorthAmericaWildfires_2007275.txt`

20. Click **Run** to execute the tool. If everything goes correctly you should see the **TodaysWildfires** layer added to the **Contents** pane.



21. Open the attribute table to verify that the `CONFIDENCEVALUE` field has been populated.

In conclusion…

Custom script tools in a custom toolbox are used to provide a visual interface to your Python scripts. End users of these tools don't need to understand anything about Python to use the tools. They look just like every other tool in the ArcGIS Pro toolbox. These tools can be used to accomplish a wide variety of geoprocessing tasks.

Exercise 2: Creating a Python toolbox

Getting ready

There are two ways to create toolboxes in **ArcGIS Pro**: script tools in custom toolboxes that we covered in the last exercise, and script tools in Python toolboxes. Python toolboxes were introduced at version 10.1 of ArcGIS Desktop and they encapsulate everything in one place: parameters, validation code, and source code. This is not the case with custom toolboxes, which are created using a wizard and a separate script that processes the business logic.

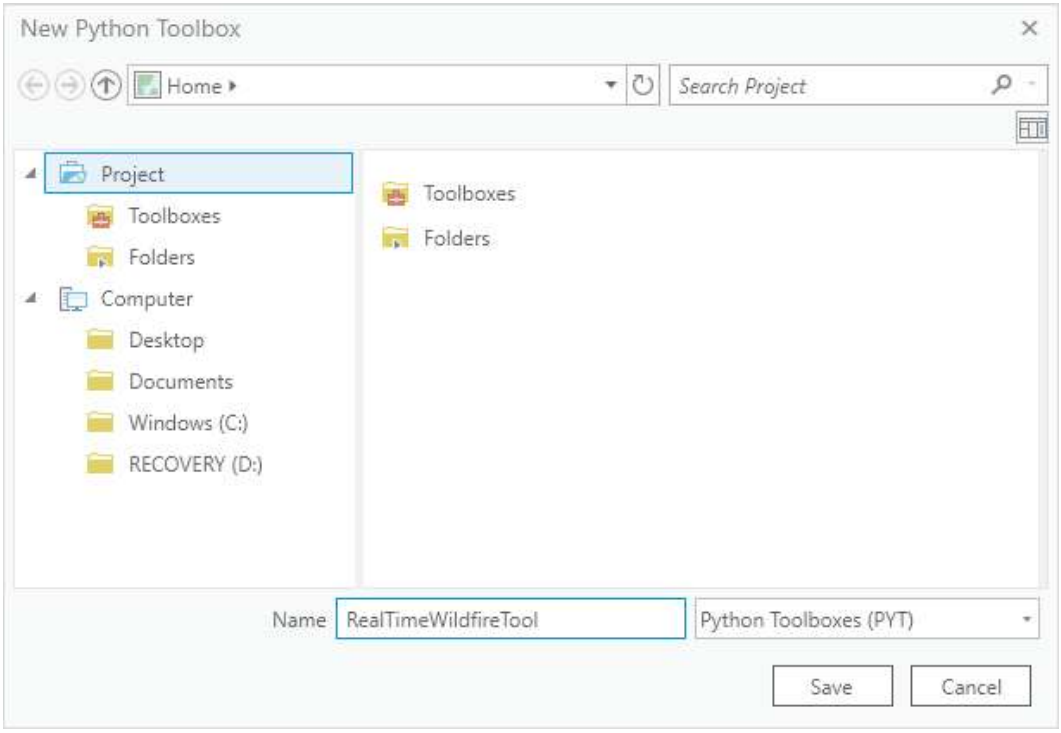
A **Python Toolbox** functions like any other toolbox but it is created entirely in Python and has a file extension of `.pyt`. It is created programmatically as a class named `Toolbox`. In this exercise you will learn how to create a **Python Toolbox** and add a custom tool.

In this exercise you'll create a custom Python toolbox that connects to a live USGS map service that contains real time wildfire information. After creating the basic structure of the `Toolbox` and `Tool` you'll complete the functionality of the tool by adding code that connects to an **ArcGIS Server** map service, downloads real time data, and inserts it into a feature class.

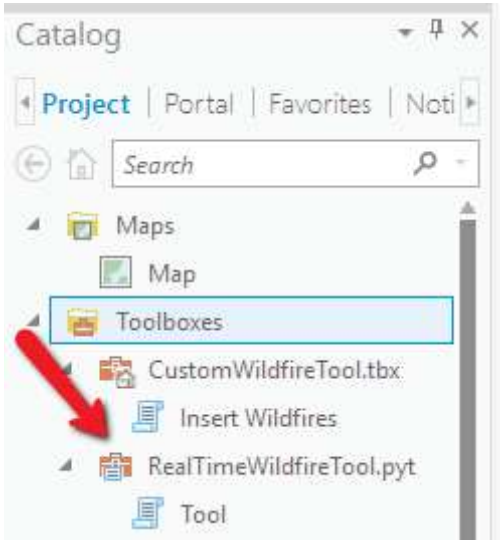
How to do it...

Complete the steps below to create a **Python Toolbox** and create a custom tool that connects to an ArcGIS Server map service, downloads real time data, and inserts it into a feature class.

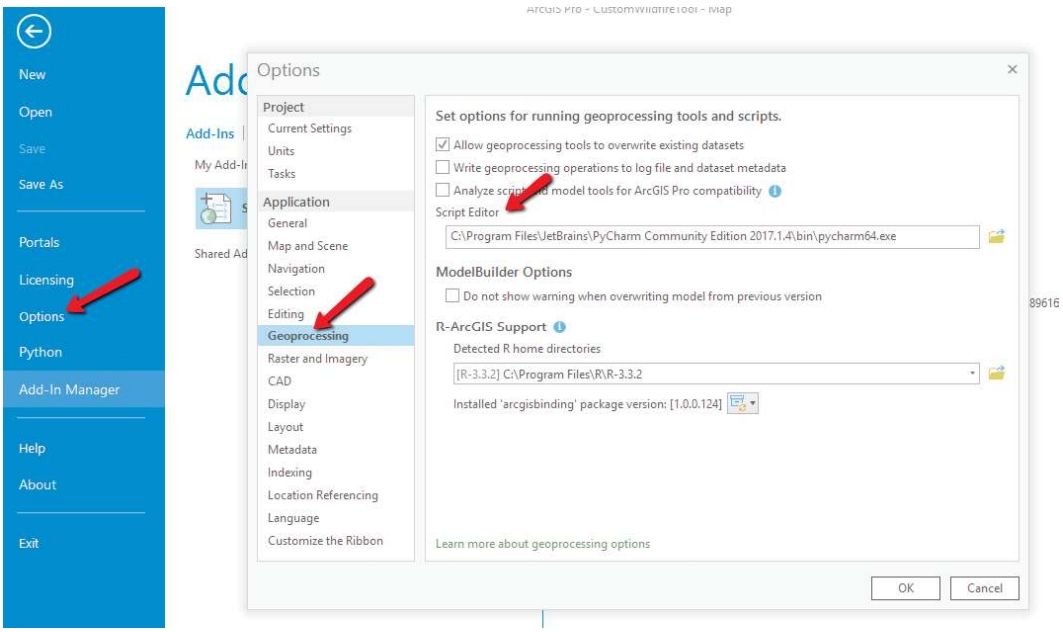
1. Open the **CustomWildfireTool** project in **ArcGIS Pro** and find the **Toolboxes** folder in the Catalog pane.
2. Right click the **Toolboxes** folder and select **New Python Toolbox**.
3. Name the toolbox **RealTimeWildfireTool** and place it into the **Toolboxes** folder for the **Project** as seen in the screenshot below.



4. Click **Save** to create the Python toolbox seen in the screenshot below.



5. By default, when you right click on the toolbox and select **Edit**, **ArcGIS Pro** will automatically open your code in **Notebook.** To change this to the development environment of your choice. You can go to **Project** | **Options** | **Geoprocessing** and enter the path to your development environment as seen in the screenshot below. In this example I've provided the path to the Community Edition of PyCharm. You'll want to update this to the path specific to your environment and development environment.



6. Right click **RealTimeWildfireTool.pyt** and select **Edit**. This will open your development environment. Your development environment will vary depending upon the editor that you have defined or simply default to **Notepad** if you haven't defined a development environment.

7. Remember that you will not be changing the name of the class, which is `Toolbox`. However, you will rename the `Tool` class to reflect the name of the tool you want to create. Each tool will have various methods including `__init__()` which is the constructor for the tool along with `getParameterInfo()`, `isLicensed()`, `updateParameters()`, `updateMessages()`, and `execute()`. You can use the `__init__()` method to set initialization properties like the tool's label and description.

Look for the `Tool` class and change the name to `USGSDownload`. Also, set the `label`, and `description` as seen in the code below:

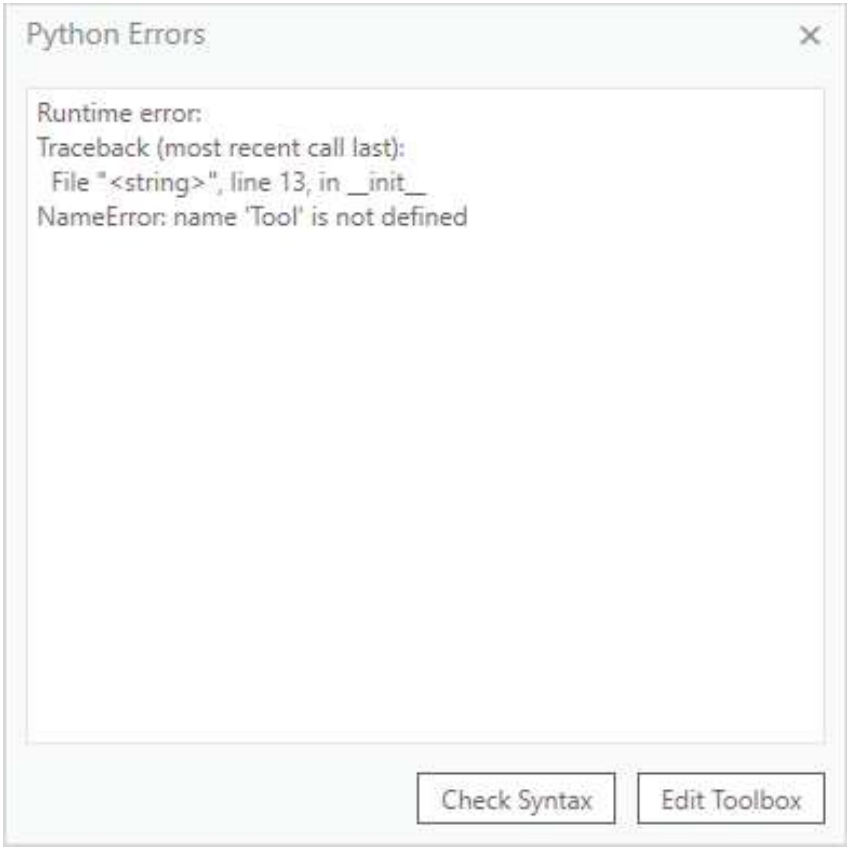
```
class USGSDownload(object):
   def __init__(self):
      """Define the tool (tool name is the name of
      the class)."""
      self.label = "USGS Download"
   self.description = "Download from USGS ArcGIS
   Server instance"
```

8. You can use the `Tool` class as a template for other tools you'd like to add to the toolbox by copying and pasting the class and it's methods. We're not going to do that in this particular exercise, but I wanted you to be aware of this. You will need to add each tool to the tools property of the `Toolbox`. Add the `USGS Download` tool as seen in the code below.

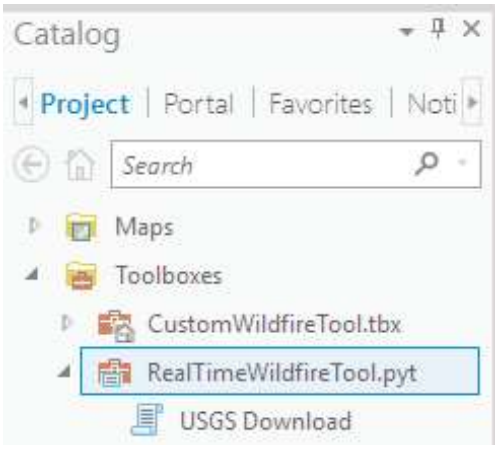
```
class Toolbox(object):
   def __init__(self):
      """Define the toolbox (the name of the toolbox
      is the name of the
      .pyt file)."""
      self.label = "Toolbox"
      self.alias = ""

      # List of tool classes associated with this toolbox
      self.tools = [USGSDownload]
```

9. When you close the code editor your **Toolbox** should automatically be refreshed. You can also manually refresh a toolbox by right clicking the toolbox and selecting **Refresh**.

10. You shouldn't have any errors at this time, but you can check by right clicking the toolbox and select **Check Syntax**.



11. Assuming that you don't have any syntax error you should see the following Toolbox/Tool structure.



12. Almost all tools have parameters, and you set their values on the tool dialog box or within a script. When the tool is executed, the parameter values are sent to your tool's source code. Your tool reads these values and proceeds with its work. You use the `getParameterInfo()` method to define the parameters for your tool. Individual `Parameter` objects are created as part of this process. Add the following parameters in the `getParameterInfo()` method and then we'll discuss.

```
def getParameterInfo(self):
    """Define parameter definitions"""

    # First parameter
    param0 = arcpy.Parameter(
        displayName="ArcGIS Server Wildfire URL",
        name="url",
        datatype="GPString",
        parameterType="Required",
        direction="Input")
    param0.value = "https://wildfire.cr.usgs.gov/arcgis/
    rest/services/geomac_dyn/MapServer/0/query"

    # Second parameter
    param1 = arcpy.Parameter(
        displayName="Output Feature Class",
        name="out_fc",
        datatype="DEFeatureClass",
        parameterType="Required",
        direction="Input")

    params = [param0, param1]
    return params
```

13. Each `Parameter` object is created using `arcpy.Parameter` and is passed a number of arguments that define the object.
For the first `Parameter` object (`param0`) we are going to capture a URL to an **ArcGIS Server** map service containing current wildfire data. We give it a display name (**ArcGIS Server Wildfire URL**), which will be displayed on the dialog box for the tool, a name for the parameter, a data-type, parameter type (Required), and direction.

In the case of the first parameter (`param0`) we also assign an initial value, which is the URL to an existing map service containing wildfire data.
For the second parameter we're defining an output feature class where the wildfire data that is read from the map service will be written. An empty feature class for storing the data has already been created for you. Finally, we added both parameters to a Python list called `params` and return the list to the calling function

14. The main work of a tool is done inside the `execute()` method. This is where the geoprocessing of your tool takes place. The `execute()` method, seen below, can accept a number of arguments including the tool (self), parameters, and messages.

```
def execute(self, parameters, messages):
"""The source code of the tool. """
return
```

15. To access the parameter values that are passed into the tool you can use the `valueAsText()` method. Add the following code to access the parameter values that will be passed into your tool. Remember from a previous step that the first parameter will contain a URL to a map service containing wildfire data and the second parameter is the output feature class where the data will be written.

```
def execute(self, parameters, messages):
   inFeatures = parameters[0].valueAsText
   outFeatureClass = parameters[1].valueAsText
```

16. At this point you have created a Python toolbox, added a tool, defined the parameters for the tool, and created variables that will hold the parameter values that the end user has defined. Ultimately this tool will use the URL that is passed into the tool to connect to an **ArcGIS Server** map service, download the current wildfire data, and write the wildfire data to a feature class. We'll do that next.

17. Next, add the code that connects to the wildfire map service to perform a query. In this step you will also define the `QueryString` parameters that will be passed into the query of the map service. First we'll import the `requests` and `json` modules by adding the code below. The `requests` module is part of the standard conda installation associated with **ArcGIS Pro**. The import statements should go at the very top of your script.

```
import requests, json
```

18. Then, create the payload variable that will hold the `QueryString` parameters. Notice that in this case we have defined a where clause so that only fires where the

acres are greater than 5 will be returned. The `inFeatures` variable holds the URL.

```
def execute(self, parameters, messages):
   inFeatures = parameters[0].valueAsText
   outFeatureClass = parameters[1].valueAsText

   agisurl = inFeatures

   payload = { 'where': 'acres > 5','f':
      'pjson', 'outFields':
      'latitude,longitude,
      incidentname,acres'}
```

19. Submit the request to the **ArcGIS Server** instance and the response should be stored in a variable called `r`.

```
def execute(self, parameters, messages):
   inFeatures = parameters[0].valueAsText
   outFeatureClass = parameters[1].valueAsText

   agisurl = inFeatures

   payload = { 'where': 'acres > 5','f': 'pjson',
   'outFields': 'latitude,longitude,incidentname,acres'}

   r = requests.get(inFeatures, params=payload)
```

20. Let's test the code to make sure we're on the right track. Add the code below to your script.

```
def execute(self, parameters, messages):
   inFeatures = parameters[0].valueAsText
   outFeatureClass = parameters[1].valueAsText

   agisurl = inFeatures

   payload = { 'where': 'acres > 5','f': 'pjson',
   'outFields': 'latitude,longitude,incidentname,acres'}

   r = requests.get(inFeatures, params=payload)
   arcpy.AddMessage(r.text)
```

21. Save the file and refresh your toolbox in the Catalog pane. Execute the tool and accept the default URL. If everything is working as expected you should see a JSON object output to the progress dialog. To view the output you'll need to mouse over the message on the progress dialog. Your output will probably vary somewhat because we're pulling live data.

22. Return to the `execute()` method and convert the JSON object to a Python dictionary. Also, comment out the `arcpy.AddMessage()` function call you implemented in the last step.

```
def execute(self, parameters, messages):
    inFeatures = parameters[0].valueAsText
    outFeatureClass = parameters[1].valueAsText

    agisurl = inFeatures

    payload = { 'where': 'acres > 5','f': 'pjson',
    'outFields': 'latitude,longitude,incidentname,acres'}

    r = requests.get(inFeatures, params=payload)


    #arcpy.AddMessage(r.text)
    decoded = json.loads(r.text)
```

23. Create an `InsertCursor` by passing in the output feature class defined in the tool dialog along with the fields that will be populated. We then start a `for` loop that loops through each of the features (wildfires) that have been returned from the request to the **ArcGIS Server** map service. The decoded variable is a Python dictionary. Inside the `for` loop we retrieve the `incidentname`, `latitude`, `longitude`, `acres` from the `attributes` dictionary. Finally, we call the `insertRow()` method to insert a new row into the feature class along with the fire name and acres as attributes. Progress information is written to the **Progress Dialog** and the counter is updated.

```
def execute(self, parameters, messages):
   inFeatures = parameters[0].valueAsText
   outFeatureClass = parameters[1].valueAsText

   agisurl = inFeatures

   payload = { 'where': 'acres > 5','f': 'pjson',
   'outFields': 'latitude,longitude,incidentname,acres'}

   r = requests.get(inFeatures, params=payload)

   decoded = json.loads(r.text)

   with arcpy.da.InsertCursor(outFeatureClass,
   ("SHAPE@XY", "NAME", "ACRES")) as cur:
      cntr = 1
      for rslt in decoded['features']:
         fireName = rslt['attributes']['incidentname']
         latitude = rslt['attributes']['latitude']
         longitude = rslt['attributes']['longitude']
         acres = rslt['attributes']['acres']

         cur.insertRow([(longitude,latitude),
         fireName, acres])
   arcpy.AddMessage("Record number: " + str(cntr) + "
   written to feature class")
```

24. You can check your code against a solution file found at `c:\Student\ProgrammingPro\Solutions\Scripts\RealTimeWildfireTool.py`.

25. Save the file and refresh your Python toolbox

26. Double click the **USGS Download** tool.

27. Leave the default URL and select the **RealTimeFires** feature class in the **WildlandFires** geodatabase. The **RealTimeFires** feature class is empty and has fields for `NAME` and `ACRES`.

28. Click **OK** to execute the tool. The number of features written to the feature class will vary depending upon the current wildfire activity. Remember that this tool is pulling real time data from a USGS map service. Most of the time there is at least a little activity but it is possible (but not likely) that there wouldn't be any wildfires in the U.S. depending upon the time of year you run the tool.