Chapter 22
User Interfaces for File and Folder Selection

Abstract Running a script via an IDE requires some expertise, particularly when the script takes arguments. When you create scripts to share with others, your target audience may not be familiar with how to run a Python script. To make it easier, you can create a graphical user interface, a more intuitive means for the user to provide information for the script than command line arguments. *Graphical user interfaces* (GUIs) allow the user to interact with the script via windows labeled with explicit instructions. A Python script that uses GUIs to collect arguments can be run by double-clicking on it in Windows Explorer, avoiding an IDE altogether. Developing highly sophisticated custom GUIs is an advanced programming skill. However, you can generate certain GUIs with just a few lines of Python code. This chapter covers some easy to program GUIs for entering text or browsing to files.

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

- Get user input with graphical user interfaces.
- Get user input to open and save files.
- Customize interfaces using optional arguments.

22.1 A Simple Interface with `raw_input`

The built-in `raw_input` function launches a rudimentary GUI for text entry. When the script reaches a `raw_input` command, the user is prompted for input with a dialog box like the one shown in Figure 22.1. A *dialog box* is a type of GUI that uses a window to gather information from the user or inform the user of something, so called because it facilitates communications (a dialog) between the computer and the user. The dialog box in Figure 22.1 has two buttons, a label, and a text box. The label says 'Enter a workspace'. The white recessed box is called a *text box.*

This allows the user to type text which will be returned to the script that called the `raw_input` function.

The `raw_input` command in Example 22.1 is the code that generates the GUI in Figure 22.1. The `raw_input` function takes one argument, a string message to be displayed as the text box label (the argument in Example 22.1 is 'Enter a work-

L. Tateosian, *Python For ArcGIS*, DOI 10.1007/978-3-319-18398-5_22

Figure 22.1 `raw_input` dialog box.

space:' and this same message is displayed in the dialog box label). You can place any string message here, but the label is usually used to prompt the user to enter a value. The response typed by the user in the text box is returned to the script when the user selects the 'OK' button; This response is the return value of the `raw_input` function. In Example 22.1, the return value is being stored in `arcpy.env.workspace`. When the user types 'C:/gispy' in the text box and presses 'OK', the value of `arcpy.env.workspace` is set to 'C:/gispy'. The `raw_input` function return value is always a string, even if the user enters a number. You can cast numerical entries (This is similar to handling numeric `sys.argv` values). The `raw_input` function generates a simple general purpose GUI that can be used in a variety of situations with some limitations. One limitation is that the user is allowed to enter anything in the text box, so your script may need to anticipate common errors. For example, if the script casts a number garnered from the user, it can include code to handle `TypeError` exceptions. As a second example, there may be a typo in the workspace path. You can use error handling for this as well, but we'll discuss a GUI that constrains the input to avert this particular mistake altogether by using a specialized GUI. File names are such a common GIS scripting input need that it's worth learning about a few specialized GUIs for this purpose. The remainder of the GUIs discussed in chapter are designed for handling file and directory interactions.

Example 22.1

```
# askWorkspaceRaw.py
import arcpy
arcpy.env.workspace = raw_input('Enter a workspace:')
print arcpy.env.workspace
```

Printed output:

```
>>> C:/gispy
```

22.2 File Handling with `tkFileDialog`

This built-in `tkFileDialog` module is one of several `tk` (or tool kit) modules that provide functionality for interfaces. The `tkFileDialog` module functions generate *file dialog boxes* for browsing to files. This eliminates the need to type file

paths and enforces path accuracy. The appearance of the file dialog boxes is controlled by the operating system. For example, the screen shots shown in this chapter are generated by Windows 7. Windows 8 file dialogs may look different, but the functionality is the same. The file dialogs come in several different flavors for opening files, saving files, and choosing directories. There are several functions that get file or directory names from the user. Others functions return `file` objects open for reading or writing. Optional arguments can be used to control the appearance and behavior of the file dialogs. The upcoming sections discuss functions for getting file names and `file` objects, and the optional arguments for these functions.

22.2.1 Getting File and Directory Names

File paths are a common input for scripts. The `tkFileDialog` method `askopenfilename` creates a file browsing GUI, like the dialog box in Figure 22.2. When execution reaches the `askopenfilename` call in Example 22.2, this dialog is launched. When the user selects a file and clicks the 'Open' button, the full path name of the file is returned to the script. The assignment statement in Example 22.2 stores the return value in the `fc` variable. The script prints the file name, calls the `arcpy Describe` method, and uses the `Describe` object to print the data type of the input file. Optional arguments for `askopenfilename` allow the dialog to be customized. The two options used in Example 22.2 are setting the dialog box title and the type of files displayed by the browser. Here we only want the user to select a shapefile, so we specify this as the file type—more about these options in a moment.

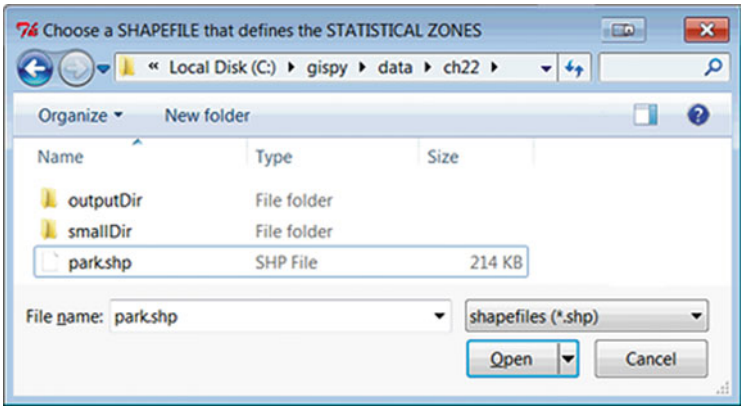

Figure 22.2 A dialog box launched with the Python code in Example 22.1.

Example 22.2

```
# getFileName.py
# Purpose: Get a shapefile name from the user and print the shape type.
import tkFileDialog, arcpy
fc = tkFileDialog.askopenfilename(filetypes=[('shapefiles','*.shp')],
    title='Choose a SHAPEFILE that defines the STATISTICAL ZONES')
print 'fc = {0}'.format(fc)
desc = arcpy.Describe(fc)
print 'Shape type = {0}'.format(desc.shapeType)
```

Printed output:

```
>>> fc = C:/gispy/data/ch22/park.shp
Shape type = Polygon
```

The `asksaveasfilename` method is similar to the `askopenfilename` method in that it returns a file name. However, since it's asking the user for a name for saving a file, it warns the user when a chosen name already exists. The user can cancel or overwrite the existing file. Calling this method doesn't actually create the file you name. Rather it returns a string file name that the script can use for some output it is creating. The return value from `asksaveasfilename` is used as an output file name in the following code which makes a copy of the input file:

```
fname = tkFileDialog.asksaveasfilename(initialfile='output.txt',
title='Save output as...')
arcpy.Copy_management('C:/gispy/data/ch22/precip.txt', fname)
```

Directory paths are another input that scripts often require. Example 22.3 uses the `tkFileDialog` module method, `askdirectory`, to set the geoprocessing workspace. The `initialdir` option sets the initial directory. The dialog box created by the code is shown in Figure 22.3.

The three methods described so far return file path names as strings. Additional methods discussed in this chapter return `file` objects. First though, we'll take a closer look at the optional arguments.

Example 22.3: askdirectory

```
# askDirectory.py
# Purpose: Get a directory from the user and set the workspace.
import tkFileDialog, arcpy
arcpy.env.workspace = tkFileDialog.askdirectory(initialdir='C:/',
            title='Select the FOLDER containing Landuse RASTERS')
print 'Workspace = {0}'.format(arcpy.env.workspace)
```

Printed output:

```
>>> Workspace = C:\buzzard
```

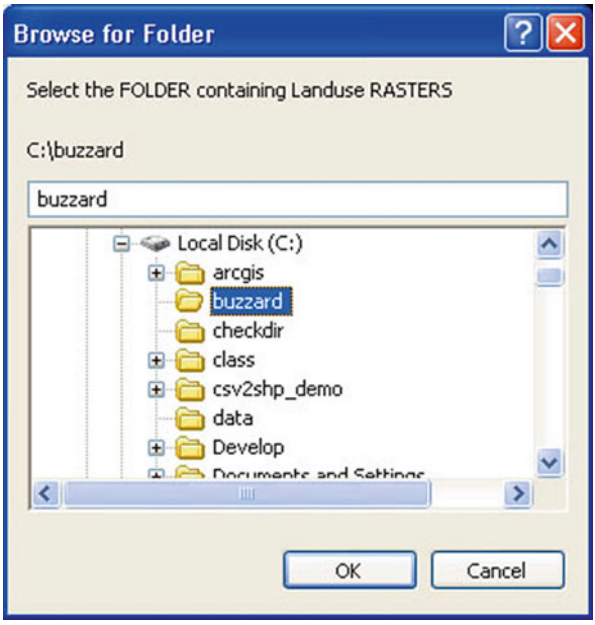

Figure 22.3 askdirectory dialog box.

Table 22.1 `tkFileDialog` method options.

`tkFileDialog` method	**Options**
askopenfilename	-defaultextension, -filetypes, -initialdir, -initialfile, -multiple, -parent, -title, -typevariable
askdirectory	-initialdir, -mustexist, -parent, or -title
asksaveasfilename	-defaultextension, -filetypes, -initialdir, -initialfile, -parent, -title, -typevariable
askopenfile	Same options as askopenfilename
asksaveasfile	Same options as asksaveasfilename

22.2.2 Options

`tkFileDialog` method options allow you to customize the appearance and behavior of the dialog boxes. These are specified as *keyword arguments*. All, none, or a subset of the options may be used. Throughout this book, we've been working with *positional arguments*. Positional arguments are ones which must be provided in the order specified in the function signature. Unlike positional arguments, the ordering of keyword arguments does not matter. Instead of order, assignment statements are used to specify which options are being set; The option name goes on the left of the equals sign and the value on the right. Arguments must still be separated by commas (e.g., the `askdirectory` call in Example 22.3 uses two options,

initialdir and title; The assignment statements for these two options are separated by a comma). The options for the methods discussed in this chapter are listed in Table 22.1.

22.2.2.1 File-Handling Options

Most of the options affect how the dialogs filter the files. The title option is the only option strictly for appearance; It specifies a string to display as the title of the dialog box (Compare the titles in Figures 22.2 and 22.3 to the title option settings in Examples 22.2 and 22.3). This section demonstrates the behavior of the file-handling options, filetypes, initialdir, initialfile, and multiple for the askopenfilename method and the mustexist option for the askdirectory method. The code snippets in this section assume that the tkFileDialog module has already been imported (The snippets are available in the sample script named 'fileDialogOptions.py').

File Types

The filetypes option filters the type of files the user can view and select in the browser. You can set this option to a single file type or multiple file types. If this option is not set, then all file types are available. To specify file types, set this option to a list of tuples. To restrict browsing to a single file type, use a list containing only one tuple. Each tuple should contain two items: the file type name and the file extension. The following code creates such a tuple with elements "shapefile" and "*.shp" and uses the tuple in an askopenfilename call:

```
>>> t = ('shapefiles','*.shp')
>>> fname1 = tkFileDialog.askopenfilename(filetypes=[t])
```

The filetypes option needs to be set to a Python list, even if only one tuple is specified. The square braces around the t embed the tuple in a list. The file type options appear in the lower right corner of the dialog. For example, the dialog box in Figure 22.2 was also generated with filetypes=[('shapefiles','*.shp')].

To specify more than one file type, use a list of tuples. The following code uses two tuples ('csv (Comma delimited)','*.csv') and ('Text Files','*.txt') to specify two file types:

```
>>> fname2 = tkFileDialog.askopenfilename(
                  filetypes=[('csv (Comma delimited)','*.csv'),
                             ('Text Files','*.txt')])
```

When multiple types are specified, you can click on the file type in the dialog and select alternative file types. The files shown in the browser are filtered by the selected type.

Initial Directory

The `initialdir` option specifies the initial browsing directory. Using `'./'` sets the initial directory to the Python current working directory. The following code makes 'C:/gispy' the default directory:

```
>>> fname3 = tkFileDialog.askopenfilename(initialdir = 'C:/gispy')
```

Initial File

The `initialfile` option specifies an initial filename. If this file does not exist in the specified directory, a warning is displayed and the user is forced to select another file, as in the following code, which points to a nonexistent file:

```
>>> fname4 = tkFileDialog.askopenfilename(initialfile = 'bogus.shp')
```

Multiple Files

The `multiple` option allows the user to select multiple files. This is a Boolean variable set to `False` by default, only allowing the user to select one file. When this option is set to `True`, the return value is a string of space separated file names. Example 22.4 lets the user select multiple files and loops to print the individual file names. The individual names are extracted by splitting the string on spaces. This returns a list of one or more file names (you'll need to take another approach if the file names contain spaces).

Example 22.4

```
# excerpt from fileDialogOptions.py
# Purpose: Vary file dialog options to get file and directory
#          names from user and print the results.

fnames = tkFileDialog.askopenfilename(
          title='Test multiple selections allowed', multiple=True)
files = fnames.split()
print 'Name list:'
for fname in files: # for each file selected by the user
    print '    {0}'.format(fname)
```

Existing Directories

The `mustexist` option specifies if the user must select an existing directory. This boolean variable is set to False by default. If the user types a non-existent directory name when this is set to True, a warning appears and the user is forced to cancel or select an existing directory. This option is not available for the `askopenfilename`, since this method is specifically designed to get existing file names (the `asksaveasfilename` method can be used instead to get a new file name). The following code restricts the user's directory selection to an existing directory:

```
>>> inputDir = tkFileDialog.askdirectory(mustexist = True)
```

Modify the code samples in the 'fileDialogOptions.py' to gain a deeper understanding of how these options work. One additional option, the `parent` option, is not related to file handling and requires some explanation. This is discussed next.

22.2.2.2 Close the Tk Window

The `parent` option enables you to force the Tk window to close. When you run the `tkFileDialog` examples in this chapter, you will notice that a small window entitled 'Tk' is launched in the background (see Figure 22.4). When you run the script from an IDE such as PythonWin or PyScripter, the 'Tk' window persists after you exit the script and won't close until you exit the IDE. This is the default window for the tk-based applications. If you were building a custom GUI, you would add buttons and text boxes to this window and these buttons could provide a means to close this window. But calling the `tkFileDialog` method directly does not use the 'Tk' window, so we need to add a few lines of code to force it to close.

Figure 22.4 The `tk` window is opened when a tk file dialog command is called, but does not close when the dialog box is exited.

Forcing the 'Tk' window to close involves setting the parent option and using another module called Tkinter (for interface tool kit). The Tkinter module has a Tk method that returns a Tk object. Setting the tkFileDialog parent option to that Tk object allows you to destroy it after the file dialog has been closed. The following code imports both tkFileDialog and Tkinter, creates a tk object named tkObj, sets the parent option of the askdirectory method, and then destroys the tk object, which closes the default window:

```
import tkFileDialog, Tkinter
tkObj = Tkinter.Tk()

inputDir = tkFileDialog.askdirectory(parent=tkObj)
tkObj.destroy()
```

In fact, by adding one more line of code, you can prevent the Tk window from being displayed at all. The Tk method named withdraw hides the window. You must still destroy the Tk window, else it will be running hidden in the background. Example 22.5 withdraws the Tk window and destroys it. There is nothing special about the name tkObj; Example 22.5 names the Tk object fatherWilliam (after the Lewis Carroll poem). The 'withdraw—set-parent—destroy' approach can be used for any of the tkFileDialog methods.

```
# askAndDestroy.py
# Purpose: Get a directory from the user
#              and suppress the default Tk window.

import tkFileDialog, Tkinter
# Get a tk object
fatherWilliam = Tkinter.Tk()

# Hide the tk window
fatherWilliam.withdraw()

inputDir = tkFileDialog.askopenfilename(parent=fatherWilliam)

# Destroy the tk window
fatherWilliam.destroy()
```

22.2.3 Opening Files for Reading and Writing

The askopenfilename, asksaveasfilename, and askdirectory methods return the names of the files. Other tkFileDialog methods, askopenfile and asksaveasfile, return a file object open for reading or writing.

`askopenfile` and `askopenfilename` both force the user to browse to an existing file; However, while `askopenfilename` returns a string filename, the `askopenfile` opens the file and returns a `file` object, which is opened in read mode. This `file` object can be treated just like a `file` object that is created using the built-in `open` command with mode set to `'r'`. In other words, you can use methods such as `read`, `readline`, and `readlines`. For example, the following code opens a file specified by the user, reads a line in the file, and closes the file.

```
fobject = tkFileDialog.askopenfile(
          filetypes=[('shapefiles','*.shp')],
          initialfile='data.txt', title='Open a data file...')
firstLine = fobject.readline()
fobject.close()
```

Analogously, the `asksaveasfile` returns a `file` object in write mode. Since these methods return `file` objects, they can be used within a WITH statement. The following code prompts the user for an output file name and writes a message in the file:

```
myTitle = 'Select an output file name'
with tkFileDialog.asksaveasfile(title=myTitle) as ofile:
    ofile.write('I like tkFileDialog')
```

22.3 Discussion

This chapter presented some simple GUI techniques, including the `raw_input` function and several `tkFileDialog` methods. The `raw_input` function returns a string version of the characters entered by the user. The `tkFileDialog` method provide several types of functionality. Several distinctions between the file dialog methods determine which method to select for use in a script.

One distinction is between those that return the names of files and those that return `file` objects. The `tkFileDialog` method names that end with 'filename' return the names of the files; These should be used when the script does not need to open the file for reading or writing. The `askdirectory` method also falls into this category (it returns a directory name). The `tkFileDialog` method names that end with 'file' return `file` objects open for reading or writing. When you call these methods, you need to use the `file` object `close` command or a WITH statement to avoid locking.

Another distinction is between those that are designed for input and output files. The methods names that start with 'askopen' are for getting an input file to read or use the file contents in some way. The method names that start with 'asksaveas' are for allowing the user to set output file names or create an empty file for writing.

Table 22.2 Selected `tkFileDialog` methods.

Method	Action	Return
`askopenfilename`	Ask for the *name* of an existing file	String file name or names
`askdirectory`	Ask for the *name* of a directory	String directory name
`asksaveasfilename`	Ask for a *name* to save a file	String file name
`askopenfile`	Ask for a file to *open for reading*	`file` object
`asksaveasfile`	Ask for a file to *open for writing*	`file` object

Table 22.2 summarizes the distinctions between the `tkFileDialog` methods discussed in this chapter.

The GUI implementations in this chapter do have some limitations. Only one function is called at a time. To collect both an input directory and an output file name, you need to call two methods sequentially and the GUIs will appear in succession. The `raw_input` method can handle any text input, but all values are returned as strings. When you collect a number with `raw_input` you need to cast it. Also, `raw_input` provides no validation. For example, when you ask for a number, the user can enter something other than a number, so you would have to check that in your script. When writing code for GUIs, there is a tradeoff between simplicity and customizability. There are a number of popular Python packages for building Python GUIs, such as Tk, wxWidgets, and pyQT, but these involve a steep learning curve. ArcGIS provides an alternative middle ground with a relatively easy to learn tool for building GUIs that are more complex than those presented in this chapter, though less flexible than stand-alone custom Python GUIs. Chapter 23 presents this ESRI alternative using ArcGIS toolboxes to build the interfaces.

22.4 Key Terms

Graphical user interface (GUI)
Dialog box
File dialog box
Text box
The built-in `raw_input` function
`filename` vs. `file` methods
`askopen` vs. `asksaveas` methods
positional arguments
keyword arguments

22.5 Exercises

1. **radiusRaw.py** Write a script which uses a `raw_input` GUI to get the radius of a circle and then calculates and prints the area of the circle. Use the `math` module to get the value of pi. Catch a named exception and set the radius to 1 if the user enters a non-numeric value. Print the results as in the following examples:

   Example1 input: User enters HELLO

   Example1 output:

   ```
   >>> Radius must be numeric; 'HELLO' is not numeric.
   Default radius of 1 used.
   Radius = 1  Area = 3.14159265359
   ```

   Example2 input: User enters 5

   Example2 output:

   ```
   >>> Radius = 5.0   Area = 78.5398163397
   ```

2. **boogieWoogie.py** Use sample script 'boogieWoogie.py' with the following instructions:

   (a) Run the script, take two screen shots of the file dialog that demonstrates each of the options and annotate the image, labeling the area in the file dialog affected by each option name: Label the first four options in the first screen shot (`filetypes`, `title`, `initialdir`, and `initialfile`) and use a second screen shot to label the area affected by the `multiple` option.
   (b) Add code to the script to hide and destroy the default 'Tk' window.
   (c) Add code to the script to print the file names returned by the file dialog, one per line.

3. **bufferDialog.py** Write a script that uses two `tkFileDialog` GUIs and one `raw_input` GUI to get an input shapefile name, an output shapefile name, and a buffer distance. Set the initial input file to 'park.shp', set the initial input directory to 'C:/gispy/data/ch22', set the initial output directory to 'C:/gispy/scratch', and set the initial output file to 'out.shp'. Finally, call the Buffer (Analysis) tool using the three values returned by the GUIs. The script should destroy the default 'Tk' window.

4. Match the task with the most appropriate GUI function.

Scripting task	**GUI function**
1. Count the number of lines in a file	A. `askdirectory`
2. Get a user's age	B. `askopenfilename`
3. Get the name of a KML file to import	C. `asksaveasfile`
4. Get the name of an output directory	D. `raw_input`
5. Get a name to use for a raster the script will create	E. `asksaveasfilename`
6. Write a log of `arcpy` messages in a file	F. `askopenfile`

5. **superman.py** The sample script 'superman.py' has some errors, but when they are repaired, it will use the `raw_input` function to prompt the user, giving the user three chances to guess who Clark Kent is.

   If you guess incorrectly three times it should print:

   ```
   >>> Haha! You don't know Superman!
   ```

   If you guess correctly (Superman), it should print:

   ```
   >>> You are right!
   ```

   Watch the video 'superman.swf' (found in the sample scripts directory) to see a demonstration of the correctly working script. Use the IDE to identify the syntax errors and the debugger to identify the logic errors (There are eight errors in total). Once you fix the syntax errors, turn on the debugging toolbar. Set a breakpoint on line 10, run to the breakpoint, and use the Watch window to watch `answer` and `count` as you step through the rest of the script.

6. **numChars.py** Correct the five errors in sample script 'numChars.py'. When corrected, the script should get an existing file from the user, read the contents, and print the number of characters contained in the file. Find a solution without using the built-in `open` function. Also, be sure to withdraw and destroy the default 'Tk' window.

   Example input: User selects 'precip.txt'

   Example output:
   ```
   >>> 'C:/gispy/data/ch22/precip.txt' has 43283 characters.
   ```

7. **dirGUI.py** Write a script that uses a `tkFileDialog` GUI to get an input directory from a user and prints a list of the raster files in the directory. Set the title for the GUI to 'Select a raster directory' and the initial directory to 'C:/gispy/data/ch22/smallDir'. The script should withdraw and destroy the default 'Tk' window.

   Example input: User selects smallDir

   Example output:
   ```
   >>> Directory = C:/gispy/data/ch22/smallDir
   [u'dog.JPG', u'pines.JPG', u'tree.gif']
   ```

8. **fieldFileGUI.py** Write a script which gets a single shapefile or dBASE file from the user with a GUI and prints the field names of the selected file. Set the title to 'Pick a file'. Set the initial directory to 'C:/gispy/data/ch22/'. Set the initial file to 'park.shp'. Accept files with '.shp' or '.dbf' extensions. The script should destroy the default 'Tk' window.

Example input: User selects ‘park.dbf’

Example output:

```
>>> C:/gispy/data/ch22/park.dbf has the following fields:
FID
Shape
COVER
RECNO
```

9. **batchAscii2Raster.py** Write a script which converts batches of ESRI ASCII format files to raster with the ASCII to Raster (Conversion) tool. Use a file dialog to let the user select one or more ASCII input files (with ‘.asc’ or ‘.txt’ file extensions). Name the output files ‘out0’, ‘out1’, and so forth. Set the GUI initial directory to ‘C:/gispy/data/ch22/’. Also use a file dialog to get an output directory from the user. Set the default directory to ‘C:/gispy/scratch’. The script should withdraw and destroy the default ‘Tk’ window.

   Example input1: User browses to ‘precip.txt’ and ‘ASCIIout.txt’

   Example printed output1:

```
>>> User selections: [u'C:/gispy/data/ch22/precip.txt',
u'C:/gispy/data/ch22/ASCIIout.txt']
Created output raster: C:/gispy/scratch/out0
Created output raster: C:/gispy/scratch/out1
```

   Example input2: User browses to ‘smallPrec.asc’

   Example print output2:

```
>>> User selections: [u'C:/gispy/data/ch22/smallPrec.asc']
Created output raster: C:/gispy/scratch/out0
```