APPENDIX C

Distributing Modules and Programs

Now that you know how to build some great libraries and programs, it's natural to share them with others. It's pretty easy to just send someone your Python code, but there are a couple of problems with that approach:

- **The other person has to have Python installed.** It's not *that* hard to install Python, but it might not be realistic to expect Grandma to be able to install Python just so she can see the game you wrote. Remember, to run the games, you'll also need pygame, so that adds a little complexity to the installation as well.
- **Your code could get messed up.** A Python program is just a text file. What if the user accidentally opens your masterwork in Notepad and then randomly hits the keyboard? If the code gets typed over or something gets inserted, the code will no longer work correctly. Using code in a form that's easy to change (and break) is kind of risky.
- **You also have to consider additional files.** Most games also incorporate graphics and sound effects, which come as separate files. All these files need to be stored in the right place on the user's machine in order to work correctly. It would be great if you had a system to recreate your own set of resources on the user's machine, including the appropriate directory structure.

- **Modules have their own issues.** If you've built a module like gameEngine, it will be used only by Python programmers wanting to add gameEngine functionality. You can count on these users having Python, but you have other issues: How do you install a custom module so you can access it from any program, like you do with pygame?

Introducing the distutils and py2exe modules

Python has a really nice built-in feature for handling these kinds of problems. There's a special package called distutils that contains some really handy features for packaging up your programs so they can be run from another machine. There's an extension to distutils called py2exe that goes one step farther. It packs up a Python program with all its dependencies and extra files, and packages those files in a Windows executable file (an EXE file that can be run simply by clicking on its icon). The user doesn't have to have Python installed on his machine at all. All she needs to do is unzip the package and run the executable file you created. A custom version of Python is embedded into the program, and all the images and sound effects are automatically incorporated. (Pretty handy, huh?)

There's a couple of different ways you can package things up, depending on how the code is supposed to behave. If you're writing a module like gameEngine, you're packaging it up for programmers to use. You can assume they have Python installed, and they just want to add your module to their system so they can use it to write their own programs. If you're writing a program for end users (who typically don't have Python installed and don't care to do so), you'll package things up a bit differently. I explain both techniques here.

Step Into the Real World

The techniques I show you in this chapter work well, but they are not very efficient. Every executable you create with this system incorporates its own miniature Python distribution. When you take a simple script and turn it into a Windows executable, the py2exe system generates a large complex series of directories and files containing Python itself. All these files need to be retained in your executable programs. I've included everything on the Web site so you can see what gets created. Follow my instructions carefully, and you'll see that I create only a few standard Python scripts by hand. All the other stuff is being created by the installation process.

Creating a Module

The gameEngine module described in this book is a very handy tool. So far, you probably re-used it by copying it into whatever directory you are currently using. There are a couple of problems with that approach.

- **It gets tedious to keep copying the program.** It would be great if it were automatically available to all your Python programs, like pygame is.
- **It can be risky to have several copies of the same library.** If you have five different games using gameEngine and you decide to make an improvement to the module, you have to either copy that change to all five versions of gameEngine.py or you have confusing inconsistencies, because one version of gameEngine does things the other versions don't do.

Installable Module Overview

Fortunately, it's pretty easy to set up a module so it can be installed into a programmer's Python system. If you want to make an installable module, follow these steps:

1. **Prepare the module.**

   It's best to put a copy of the module into its own directory. Otherwise you'll likely end up confused about what goes into the module and what doesn't. Put any other files that module needs in the same directory. (For example, gameEngine uses freesansbold.ttf, so I put a copy of that font file in the directory.)

2. **Write a `setup.py` script.**

   Create a simple Python program to manage your module. This program (traditionally called setup.py) doesn't really do much on the surface, but it accesses an incredibly powerful Python routine. The setup.py program passes information about your module to another function called distutils.core.setup(). Here's my setup.py program for the gameEngine module:

```
# setup.py
from distutils.core import setup
setup(name = "gameEngine",
      version = "1.0",
      py_modules=["gameEngine"],
      author="Andy Harris",
      author_email="aharris@cs.iupui.edu",
      url="http://www.cs.iupui.edu/~aharris/pgl"
      data_files=[('',["freesansbold.ttf"])]
    )
```

Information Kiosk

Don't panic if this code doesn't make sense to you yet. I'll explain the `setup.py` script in the next section.

3. **Save the `setup.py` script in the same directory as the program file.**

   Generally it should be called `setup.py`.

4. **Run the `setup.py` script.**

   Go to the command line and run `setup.py` with its distribution option, like this:

   ```
   python setup.py sdist
   ```

5. **Examine the results.**

   `setup.py` will create two new directories. The `build` directory is used during the process of creating the module. Once your module is finished, you can safely delete the `build` directory if you want. The other directory is called `dist`. Inside that directory, you'll find one ZIP file containing everything you need. (If you're using Linux or Mac OS, you will have a tarball rather than a ZIP file.) Note that the program may complain about the lack of a MANIFEST or README file. If you want to avoid this warning, you can simply incorporate text files with these names. MANIFEST is normally a simple list of the files included in a package, and README is a text file describing how to install and use the program. Neither is absolutely necessary.

6. **Make a Windows installer if you anticipate users will be running Windows.**

   If you know your audience uses Windows, you can run a variation that creates an installer for you.

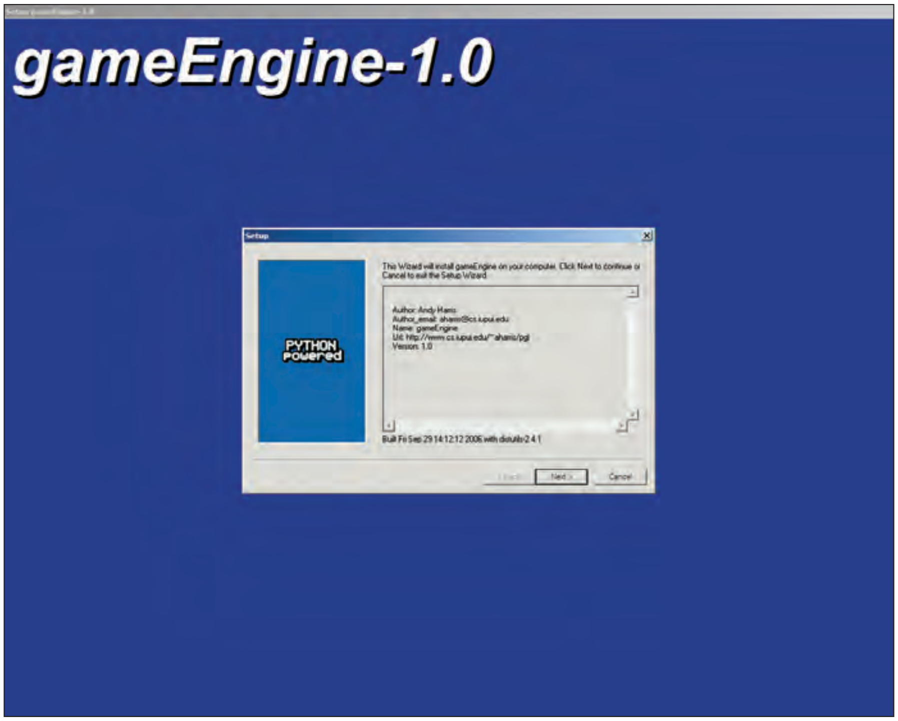
   ```
   python setup.py bdist_wininst
   ```

Watch Your Step

You should use a Windows machine to create a Windows installer, because the process uses several DLL files peculiar to Windows.

When you use this option, look in the `dist` directory, and you'll see an executable (EXE) file. If the user runs this program, your module will automatically install, and the user won't ever need to manually run the `setup.py` file.

Figure C-1 shows the installation package running in Windows.



This Wizard will install gameEngine on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

```
Author: Andy Harris
Author_email: aharris@cs.iupui.edu
Name: gameEngine
Url: http://www.cs.iupui.edu/~aharris/pgl
Version: 1.0
```

Built Fri Sep 29 14:12:12 2006 with distutils-2.4.1

Figure C-1: Using the bdist option gives you a professional installation option for your module.

7. **Run the `setup.py` installation.**

   Mac and Linux users will not be able to use the Windows installer. Fortunately, the same `setup.py` script you use to create the distribution can be used to install the source distribution that works on every machine. The ZIP file created by `setup.py` also includes a copy of the same `setup.py` program. To use it, have your users unpack the compressed file and run this variant of `setup.py`:

   ```
   python setup.py install
   ```

Step Into the Real World

The `sdist` qualifier is a *command-line* parameter. This tells `setup.py` that it should create a source (or plain code in text files) distribution package. As you see in step 6, there are also binary distribution commands. When users want to install the module, they'll use a different command that installs the module. This is described in step 7.

Step Into the Real World

The Windows installer is a terrific option, but this particular tool is only used for adding a custom module to a system that already includes Python. If you want to do something similar with your finished games, you need to create an executable version of the game, and add an installer to it. I show both these techniques in this appendix.

This runs the same `setup.py` program but uses `install` mode, which automatically installs the files in the appropriate place on the hard drive.

Writing the setup.py script

The `setup.py` script begins with an `import` statement. After ensuring you have access to `distutils.setup`, simply call the `setup()` function with a set of custom parameters. Of course, the parameters are really the key to a successful `setup.py` program. The documentation for `setup.py` lists many parameters, but those in Table C-1 are the most important.

Table C-1 Setup.py Parameters

Name	Type	Description
`name`	string	The name of the module or program as it will be seen by Python.
`version`	string representation of number	Used to keep track of different versions of your program. It's a very good idea to incorporate version numbers so users can tell when they should upgrade.
`py_modules`	list of strings	List containing modules in your distribution (used primarily in modules).
`author`	string	Author's name.
`author_email`	string	E-mail address.
`url`	string	Project's main URL.
`data_files`	list of (`dir`: directory string `fileList`: list of filenames) tuples	Describes where any other files needed by the distribution can be found and should be located in the final version. Use `"."` to indicate currentdirectory.
`console`	string filename	Name of Python program that should be run in the console (used only in `py2exe`).

The `setup.py` program for installing a module is usually pretty simple. As an example, here again is the `setup.py` program for the `gameEngine` module followed by the steps to create it:

```
# setup.py for gameEngine
from distutils.core import setup
setup(name = "gameEngine",
      version = "1.0",
      py_modules=["gameEngine"],
      author="Andy Harris",
      author_email="aharris@cs.iupui.edu",
      url="http://www.cs.iupui.edu/~aharris/pgl",
      data_files=[('',["freesansbold.ttf"])]
    )
```

1. **Import the `setup()` function from `distutils`.**

   The `setup` function is part of the `distutils` module that comes with Python. Import the utility so you can use it.

   ```
   from distutils.core import setup
   ```

2. **Run the `setup()` function to start the whole process.**

   Inform the `setup()` function of the name of your module.

   ```
   setup(name = "gameEngine",
   ```

3. **Determine a version.**

   Although the version isn't strictly necessary, it can be very handy, especially if your module gets popular and people start using it.

   ```
         version = "1.0",
   ```

4. **List the modules to include.**

   Since this is a very simple package with only one module, I simply include that module's name.

   ```
         py_modules=["gameEngine"],
   ```

5. **Include contact information.**

   Add these values so that users can reach you (presumably with gifts, because your module is so useful).

   ```
         author="Andy Harris",
         author_email="aharris@cs.iupui.edu",
         url="http://www.cs.iupui.edu/~aharris/pgl",
   ```

6. **List any other files that should be included.**

   I found that `gameEngine` works better if it avoids `SysFont` and directly loads the default font instead. By adding this directive, I ensure that the default pygame font will always be installed with `gameEngine`.

   ```
         data_files=[('',["freesansbold.ttf"])]
   ```

Step Into the Real World

The data_files parameter looks really complicated, but it isn't quite as messy as it looks. Data_files contains a list of values. Each element of this list is a tuple, representing a subdirectory and the files in that directory. The tuple has two values: the directory name ('' for the current directory) and *another* list containing names of files in the directory that should be included. In my example, I want to include one file called freesansbold.ttf, which is found in the same directory as the gameEngine.py program itself.

You can use the data_files directive to add any other resources that your module needs. These resources should generally be in the same directory as the current file.

Transfer

I explain the data_files directive more completely in the next section on creating executables.

Creating a Windows Executable

The py2exe program extends the distutils module by allowing you to build a complete Windows executable from your game. With this system in place, the user won't need to have Python or pygame installed. When they install the game, it simply runs.

Optimizing a game for py2exe

I have discovered a couple of things that cause problems when building executables with py2exe. Your code will be much easier to turn into an executable program if you follow these guidelines:

- **Put the game in its own directory.** Put the game file and any other files it is dependent upon in its own directory. If you need graphics, external fonts, or sound effects, put them all in the primary directory or as simple a subdirectory structure as you can get away with. Remove anything that's not a critical part of the game from the main directory structure.
- **Use external fonts.** The biggest headaches seem to come from problems with the Font class. I consistently have trouble with SysFonts working correctly in compiled games. The easiest way to do this is to simply use TTF files (using the standard Font class). If you want to use the standard pygame font, copy it from the pygame distribution (look in Lib/site-packages/pygame for freesansbold.ttf). If you're using gameEngine, it will include a copy of this font.

As an example, I've written a simple game called Bugs (`bugs.py` to be more specific) that illustrates a famous problem in computer science and statistics called the "random walk." I made it more interesting by creating a hundred bugs dashing around the screen. Figure C-2 illustrates the Bugs program.

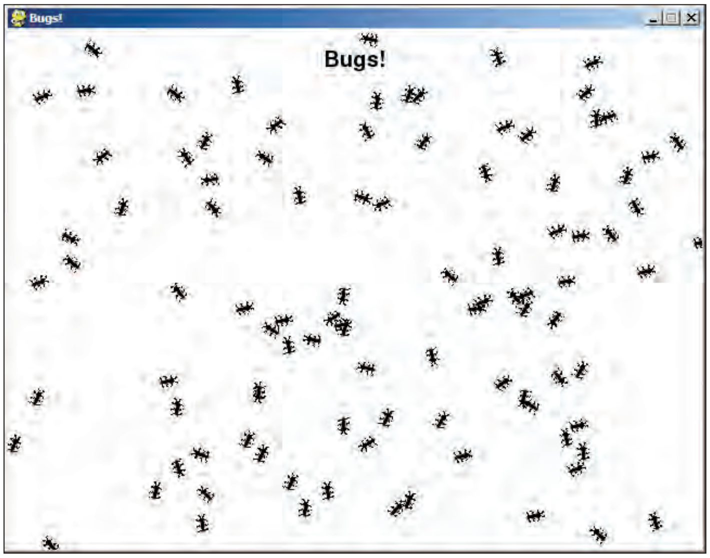


Figure C-2: This is one program that's *supposed* to have a lot of bugs in it.

The code for the Bugs simulation is pretty simple, because `gameEngine` does most of the work:

```
""" bugs.py
    random bug simulation
    using gameEngine
"""

import random, pygame, gameEngine

class Bug(gameEngine.SuperSprite):
    def __init__(self, scene):
        gameEngine.SuperSprite.__init__(self, scene)
        self.image = self.setImage("bug.gif")
        self.setPosition((320, 240))
        self.setSpeed(1)

    def checkEvents(self):
        """ move semi-randomly """
        self.speedUp(random.randint(-1, 1))
```

```
            self.turnBy(random.randint(-15, 15))

def main():
    game = gameEngine.Scene()
    game.setCaption("Bugs!")
    game.background.fill((0xff, 0xff, 0xff))

    lblTitle = gameEngine.Label()
    lblTitle.center = (320, 30)
    lblTitle.font = pygame.font.Font                                    ↩
("freesansbold.ttf", 20)
    lblTitle.text = "Bugs!"

    bugs = []
    for i in range(100):
        bugs.append(Bug(game))

    game.sprites = [bugs, lblTitle]
    game.start()

if __name__ == "__main__":
    main()
```

Even though the program is relatively simple, there's several things that must be considered when you try to make a standalone version of the game.

- **It needs a copy of Python and pygame.** All the gory details of Python and pygame need to be incorporated into the executable, so your program can run on a computer that doesn't have Python installed.
- **Pygame requires SDL and other dependencies.** Remember that pygame is actually a wrapper to the SDL library, which your user may not have. The executable needs to incorporate this as well.
- **Your game has its own resources that need to be included.** Your program will need access to the sound effects, images, and fonts used in the original program. These files should be preserved in the appropriate data structure so the distributed version of your program can still find them.
- **Don't forget `gameEngine`!** Since this program uses the custom `gameEngine` module, the executable version must have access to special module as well.

Fortunately, `py2exe` and the `setup.py` program automate most of this for you. You just have to tell them what to do with a properly written `setup.py` program.

Writing setup.py for your game

The basic process for creating an executable is similar to building an installable module. The process is still based on the `distutils.setup()` method, but this time, it gets a little help from an add-on called `py2exe`. Install this package into your Python distribution, and then write a `setup.py` program for your game.

Here's the `setup.py` for the Bugs game:

```
""" setup.py for bugs game """

from distutils.core import setup
import py2exe

setup(console = ["bugs.py"],
      author="Andy Harris",
      author_email="aharris@cs.iupui.edu",
      url="http://www.cs.iupui.edu/~aharris/pgl",
      data_files=[('.', ["bug.gif",
                          "freesansbold.ttf"]
                  )]
    )
```

The `setup.py` code for an executable is slightly different from the module version:

1. **The `setup.py` code requires that you to install the `py2exe` package on your development system.**

   ```
   import py2exe
   ```

   By including this import, you make the `py2exe` package available when you run the `setup.py` program.

2. **You must indicate a console or window name.**

   When you create an executable, if you give it a *console name,* the program will appear with the DOS command line visible. If you don't want the console to appear, specify `window` rather than `console`. In either case, the value is a list containing a string with the name of the main Python file you want to run. The EXE file will have this name. Note the console and window parameters replace the module parameter, so you don't need it when you're making a standalone application.

   ```
   setup(console = ["bugs.py"],
   ```

3. **Include contact information.**

   Use the standard parameters to include contact information for your program:

   ```
         author="Andy Harris",
         author_email="aharris@cs.iupui.edu",
         url="http://www.cs.iupui.edu/~aharris/pgl",
   ```

4. **Incorporate any other files you need.**

   Since all my other resources exist in the same directory as my main program, I have a list containing one tuple of the current directory '.') and a list of files from that directory that must be included:

   ```
   ["bug.gif", "freesansbold.ttf"]
   ```

   If you have other directories with files that need to be included, create a new tuple for each directory and a list of files in that directory.

   ```
         data_files=[('.', ["bug.gif",
                            "freesansbold.ttf"]
                    )]
   ```

Information Kiosk

The directory structure you indicate will be recreated on the user's computer, so all the file references in the original program will still point to the right places.

Running the setup for py2exe

The actual executable program is created by running `setup.py`. This time, use the `py2exe` argument made available when you installed `py2exe` on your system.

1. **Go to the command line.**

   It's easiest to run `setup.py` from the command line because you'll need to send parameters to the `setup.py` program, and you'll want to see all the results in the console. Make sure you're in the directory that contains your game program and the `setup.py` program you just wrote.

2. **Run `setup.py`.**

   Use the following command:

   ```
   python setup.py py2exe
   ```

3. **View the flood of information in the console.**

Step Into the Real World

The end of the process includes some scary-looking warnings about "non-included dependencies." These are files normally included with Windows that `py2exe` does not include for licensing reasons. The good news is that most users will already have the needed files installed, because they are installed with Windows itself.

If all goes well, you'll see a lot of information scream by in the console. Once it's done, you can scroll back through it and see the relatively complete reporting on what went on through the process. Essentially, the program encapsulates all the main Python files, examines all the dependencies your program may need, copies over any files you requested, and builds the executable file.

4. **Move to the `dist` directory.**

   The `py2exe` process will create two new subdirectories of your game directory, `build` and `dist`. You can safely ignore the `build` directory, as it is used only during the process of creating the executable. The `dist` directory contains everything your users will really need.

5. **Test the executable program.**

   You'll find a bunch of support files and one executable in the `dist` directory. You should give the executable file the same name as your program (in my case, `bugs.exe`). Run it and see if it works.

Distributing your executable in a ZIP file

You have two main ways to package your executable. You can

- **Put your entire `dist` directory into a ZIP file.** That way users can install it on their own systems wherever they want.
- **Create an installation program.** This option is a lot easier on the user, but it requires a little more work from you.

If you want to simply zip up the files, use your favorite compression program (I like IZarc, a freeware utility for Windows) to stuff all the files in your `dist` directory into a file called `bugs.zip`.

Step Into the Real World

If everything worked up to here, but your executable fails to run, look carefully at the error message in the console. Usually the problem is a file that failed to get transferred. Fonts are especially prone to this, especially if your program uses a `SysFont`. You may have to tweak your `setup.py` file a bit and run it a few times to get it right. You might also have to alter your primary program a little bit. For example, I replaced all the calls to `pygame.font.SysFont("None")` to `pygame.font.Font("freesansbold.ttf")`, and things worked fine.

Step Into the Real World

Users will be baffled by a ZIP file containing all these terrifying filenames. Be sure to include a `readme.txt` or `readme.html` file explaining that all the files must be installed, but the game is played by simply selecting the executable file.

Making an Installation Program

An *Installation Program* is a special program that aids in the installation of some other software. Almost all commercial software uses this type of tool, which allows you to install a program. It copies all the necessary files, (sometimes) makes shortcuts, and shields the user from the drudgery of setting up a program. An installer also often includes an *uninstaller* that allows the user to safely and easily remove your program from the system when it is no longer needed.

You can make your program just a little easier for your users by adding an installer to the process. Software installation can be a daunting process, but basic Windows installers aren't too hard to write if you're using the terrific NSIS (Null Soft Installation System) installation system from NullSoft. This open-source tool is used to create incredibly sophisticated installation programs, but you can also use it to make a simple installer for your game. Here's the overview:

1. **Prepare your executable.**

   Presumably you've done this through `py2exe`. Everything you need is stored in your `dist` directory.

2. **Write an NSI script.**

   An NSI script is a program in a specialized programming language specifically created for building installers. (It may sound daunting to learn a new programming language just for making installations, but don't worry — I have a couple of tricks that can help you. Read on.)

3. **Compile the NSI script.**

   The NSIS package comes with a compiler that reads the script and uses it to package all of the files you want into an executable file. If you want, you can also have it make changes to the Start menu, desktop shortcuts, and uninstallers.

4. **Test and distribute.**

   You'll need to test your installation on computers that don't have Python or pygame installed to make sure everything works as expected.

Installing NSIS

NSIS is an open-source installation environment for Windows from NullSoft. This package is essentially a programming language with its own compiler, designed to help you build custom installation packages. The system can be a bit daunting, but it is really pretty easy to make it work. Here's how to get started:

1. **Install NSIS.**

   Download and install the complete NSIS package.

2. **Write an NSI script.**

   NSI scripts can be complex, but the most basic type of installation (that simply encapsulates all the files into one executable and installs them in a specific spot) is not terribly complex. I describe exactly such a script in the section (cleverly) called "Writing a basic NSI script."

3. **Run the main NSIS menu program.**

   NSIS is really a number of programs, which can all be accessed from the main program as shown in Figure C-3.

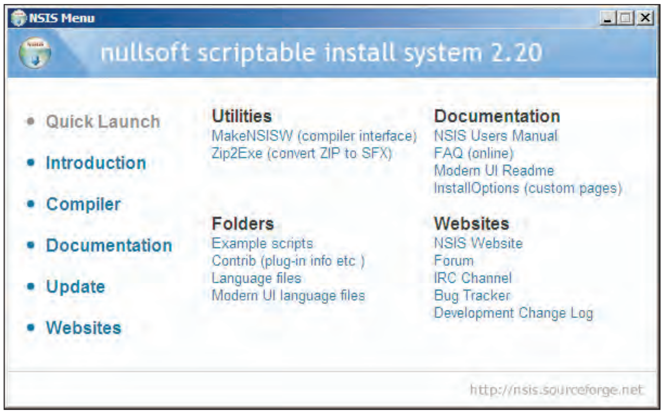


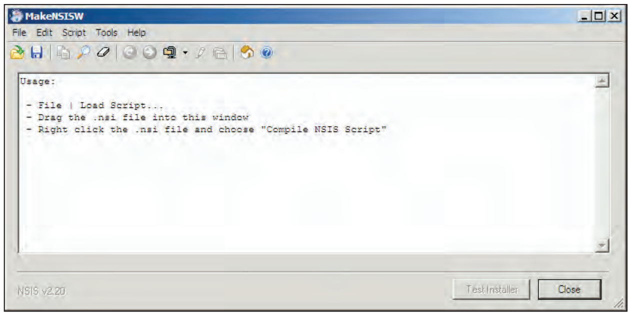
Figure C-3: The NSIS main menu.

4. **Run the compiler by clicking the MakeNSISW button in the NSIS menu.**

   A window that looks like Figure C-4 will appear.

5. **Compile the NSI script.**

   Choose Load Script from the File menu of the NSISW compiler window. If all goes well, you'll see a stream of commentary regarding the success or failure of the process. If all goes well, you can ignore this text. If something went wrong, a look at the report might help resolve problems.



```
Usage:

- File | Load Script...
- Drag the .nsi file into this window
- Right click the .nsi file and choose "Compile NSIS Script"
```

Figure C-4: The MakeNSISW compiler interface.

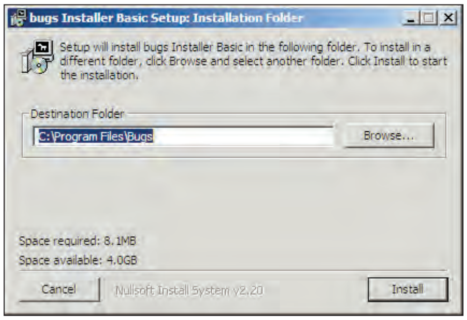
6. **Test the installer.**

   When the script has finished correctly, the program will offer a Test Installer button. Click this button to run the resulting installation program.

7. **Test again on a new computer.**

   You can't be sure the installation works until you take it to a new machine that doesn't have any of the needed files already on it.

When you run the installer, it will look like Figures C-5 and C-6.


bugs Installer Basic Setup: Installation Folder

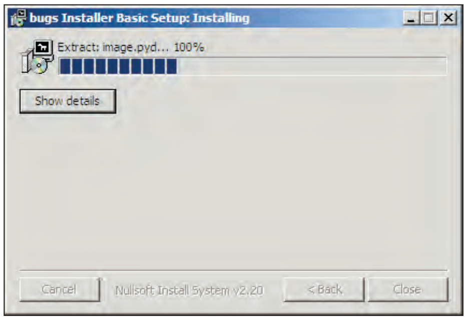
Setup will install bugs Installer Basic in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\Program Files\Bugs

Space required: 8.1MB
Space available: 4.0GB

Browse...
Cancel
Nullsoft Install System v2.20
Install

Figure C-5: The installer prompts for an installation directory.


bugs Installer Basic Setup: Installing

Extract: image.pyd... 100%

Show details
Cancel
Nullsoft Install System v2.20
< Back
Close

Figure C-6: All the appropriate files are installed in the right place.

Writing a basic NSI script

Here’s the simplest NSI script that will work for the Bugs program. I wrote this script by modifying the basic example script that comes with NSIS, digging around in the documentation, and some educated guesswork. Take a look at the whole script first:

```
; bugsBasic.nsi
; modified from example1.nsi from nullsoft

;--------------------------------

; The name of the installer
Name "bugs Installer Basic"

; The file to write
OutFile "bugInstallBasic.exe"

; The default installation directory
InstallDir $PROGRAMFILES\Bugs

;--------------------------------

; Pages

Page directory
Page instfiles

;--------------------------------

; The stuff to install
Section "" ;No components page, name is not important


  ;create desktop shortcut
  CreateShortCut "$DESKTOP\Bugs.lnk" "$INSTDIR\bugs.exe"

  ; Set output path to the installation directory.
  SetOutPath $INSTDIR

  ; Put files there

  File "bugs.exe"
  File "base.pyd"
  File "bug.gif"
  File "bz2.pyd"
  File "cdrom.pyd"
  File "constants.pyd"
  File "display.pyd"
  File "draw.pyd"
  File "event.pyd"
  File "fastevent.pyd"
  File "font.pyd"
  File "freesansbold.ttf"
  File "image.pyd"
  File "imageext.pyd"
  File "joystick.pyd"
  File "key.pyd"
  File "library.zip"
  File "mixer.pyd"
  File "mixer_music.pyd"
  File "mouse.pyd"
  File "movie.pyd"
  File "msvcr71.dll"
  File "multiarray.pyd"
  File "overlay.pyd"
  File "python24.dll"
  File "rect.pyd"
  File "rwobject.pyd"
  File "SDL.dll"
  File "SDL_image.dll"
  File "SDL_mixer.dll"
  File "SDL_ttf.dll"
  File "smpeg.dll"
  File "sndarray.pyd"
  File "surface.pyd"
  File "surfarray.pyd"
  File "surflock.pyd"
  File "time.pyd"
  File "transform.pyd"
  File "umath.pyd"
  File "unicodedata.pyd"
  File "w9xpopen.exe"
  File "zlib.pyd"
  File "_dotblas.pyd"
  File "_GLU__init__.pyd"
  File "_GL__init__.pyd"
  File "_numpy.pyd"

SectionEnd ; end the section
```

Here's how this very basic program works:

1. **Give the installer a name.**

   The Name variable indicates the title of the installation dialog box.

   ```
   ; The name of the installer
   Name "bugs Installer Basic"
   ```

2. **Indicate the filename of the installer program.**

   The `OutFile` variable determines the filename of the installation program itself.

   ```
   ; The file to write
   OutFile "bugInstallBasic.exe"
   ```

3. **Specify a default installation directory.**

   The installation program will allow the user to choose a specific directory, but it's customary to suggest a location. The `$PROGRAMFILES` constant indicates the user's standard Program Files directory.

   ```
   ; The default installation directory
   InstallDir $PROGRAMFILES\Bugs
   ```

4. **Specify the pages to display.**

   Each dialog box that pops up in this simple script is called a *page* in NSIS vocabulary; the next section indicates which of these pages should be displayed. In this case, I want to display the `directory` page (which prompts the user for an installation directory) and an `instfiles` page (which tracks progress as the various files are installed).

   ```
   Page directory
   Page instfiles
   ```

5. **Create the main section.**

   In a complex installation, you may want to allow the user to install specific parts of the program (for example, language extensions or graphics filters). Each component requires its own section in the NSI code. Since this is such a simple program, it will have only one section.

   ```
   ; The stuff to install
   Section "" ;No components page, name is not important
   ```

6. **Create a desktop shortcut.**

   If you want, you can have your installer add shortcuts to the Start menu and/or the desktop. I personally don't like Start menu shortcuts, but I do like having a shortcut on the desktop (that I can then move wherever I want). The `createShortcut` command makes this pretty easy to do:

   ```
     ;create desktop shortcut
     CreateShortCut "$DESKTOP\Bugs.lnk"                    ↩
   "$INSTDIR\bugs.exe"
   ```

Information Kiosk

You can also change the shortcut's icon to something else, if you don't like the generic icon provided by `py2exe`. Look up the details in the online help that comes with NSI.

7. **Set the output path.**

   Use the directory obtained from the `directory` page for the subsequent file-extraction commands.

   ```
   ; Set output path to the installation directory.
   SetOutPath $INSTDIR
   ```

   Information Kiosk

   Once the `directory` page has run, the variable `$INSTDIR` contains the directory the user has chosen for installing the program.

8. **Extract the files.**

   Indicate each of the files you want to extract to the directory. This is how the program will know which files to include in the installation. It's easiest if the NSI script is in the same directory as the files you want to transfer, so you can use a simple reference to indicate the filename.

   ```
   File "bugs.exe"
   File "base.pyd"
   ```

   Information Kiosk

   I only included a couple of files here, to show the basic idea. The complete NSI script has a list of all the files in the `dist` directory created by `py2exe`. Be sure to incorporate all the files in the `dist` directory that were generated when you ran `py2exe`. Most of the files are the various parts of Python that must be included with your original program.

9. **Close the section.**

   The last required element is to close up the main (and only) section.

   ```
   SectionEnd ; end the section
   ```

Creating a more elaborate script with HM NIS Edit

You can use NSIS to create much more elaborate installations, if you spend the time to learn the language. If you want anything more elaborate than the basic script indicated here, you should download the excellent HM NIS Edit program by Hector Mauricio Rodriguez Segura (`http://hmne.sourceforge.net`). This program is a text editor specializing in the NSI scripting language. It includes a wizard that makes very professional installations easily. Follow these steps:

1. **Download and install HM NIS Edit.**

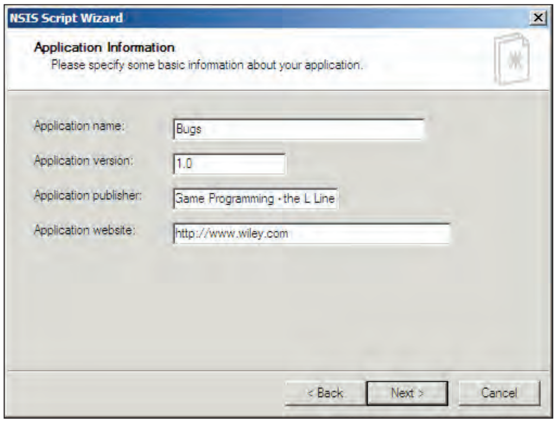
   Install the program normally and start it up.

2. **Create a new script using the wizard.**

   From the File menu, pick New Script from Wizard.

3. **Fill in general application information.**

   Figure C-7 shows the Application Information screen.



NSIS Script Wizard

Application Information

Please specify some basic information about your application.

Application name: Bugs

Application version: 1.0

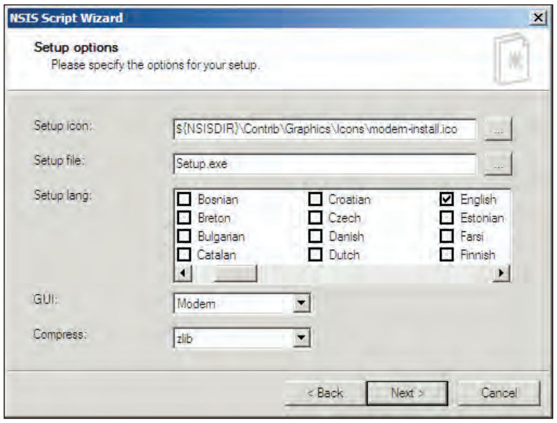
Application publisher: Game Programming - the L Line

Application website: http://www.wiley.com

Figure C-7: Enter basic information about your application.

4. **Choose setup options**.

   Determine the name and icon for your `setup.py` file, and the language(s) you want to include in your setup, as in Figure C-8. (If you aren't sure of any of these options, you can use the defaults pretty safely.)



NSIS Script Wizard

Setup options

Please specify the options for your setup.

Setup icon: ${NSISDIR}\Contrib\Graphics\Icons\modern-install.ico

Setup file: Setup.exe

Setup lang: ☐ Bosnian ☐ Breton ☐ Bulgarian ☐ Catalan ☐ Croatian ☐ Czech ☐ Danish ☐ Dutch ☑ English ☐ Estonian ☐ Farsi ☐ Finnish

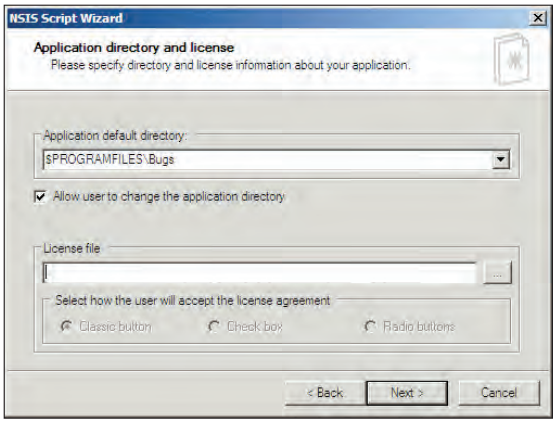
GUI: Modern

Compress: zlib

Figure C-8: Determine setup options.

5. Indicate the default directory and the license file.

Figure C-9 shows this dialog box. The default directory is the suggested installation directory of your file. The license file can be any text file. If you remove all text from the License file textbox, no license will be shown.



NSIS Script Wizard

Application directory and license

Please specify directory and license information about your application.

Application default directory: $PROGRAMFILES\Bugs

☑ Allow user to change the application directory

License file

Select how the user will accept the license agreement

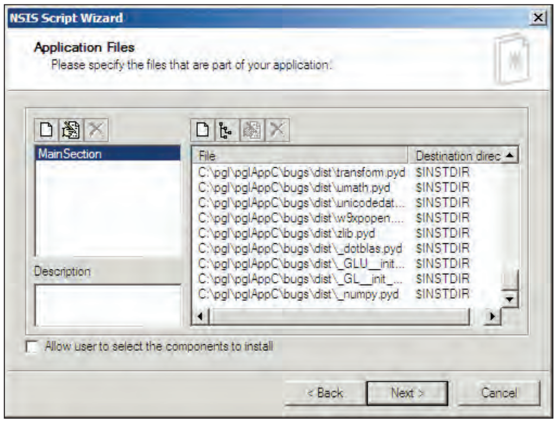
Classic button · Check box · Radio buttons

< Back · Next > · Cancel

Figure C-9: Set up a default directory and indicate a license file if appropriate.

6. Pick the application files to install.

Pick all the files you want to incorporate in your final project. You can pick files individually, or all files in a directory. Figure C-10 shows the file screen.



NSIS Script Wizard

Application Files

Please specify the files that are part of your application.

MainSection

Description

File	Destination direc
C:\pgl\pglAppC\bugs\dist\transfom.pyd	$INSTDIR
C:\pgl\pglAppC\bugs\dist\umath.pyd	$INSTDIR
C:\pgl\pglAppC\bugs\dist\unicodedat...	$INSTDIR
C:\pgl\pglAppC\bugs\dist\w9xpopen....	$INSTDIR
C:\pgl\pglAppC\bugs\dist\zlib.pyd	$INSTDIR
C:\pgl\pglAppC\bugs\dist_dotblas.pyd	$INSTDIR
C:\pgl\pglAppC\bugs\dist_GLU__init...	$INSTDIR
C:\pgl\pglAppC\bugs\dist_GL__init_...	$INSTDIR
C:\pgl\pglAppC\bugs\dist_numpy.pyd	$INSTDIR

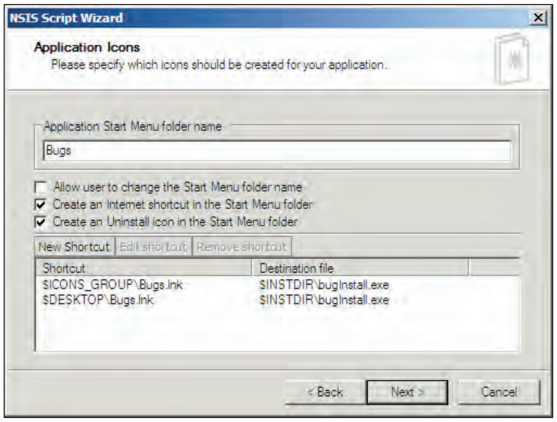
☐ Allow user to select the components to install

< Back · Next > · Cancel

Figure C-10: Determine files to include.

7. Set application links.

Pick the types of shortcuts to include with the application, as shown in Figure C-11.


NSIS Script Wizard

Application Icons

Please specify which icons should be created for your application.

Application Start Menu folder name: Bugs

☐ Allow user to change the Start Menu folder name
☑ Create an Internet shortcut in the Start Menu folder
☑ Create an Uninstall icon in the Start Menu folder

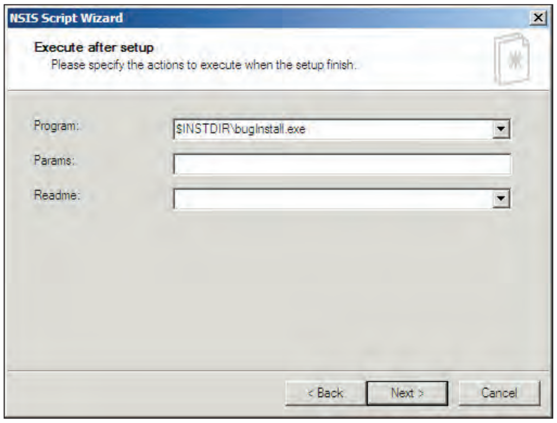
Shortcut	Destination file
$ICONS_GROUP\Bugs.lnk	$INSTDIR\bugInstall.exe
$DESKTOP\Bugs.lnk	$INSTDIR\bugInstall.exe

Figure C-11: Determine which shortcut should be added.

Although the dialog talks about choosing an Icon, you're really determining the shortcut at this point. The icon is pre-determined.

8. **Execute the application after setup.**

   You might want the program to run after the setup, or a `readme` file to appear. You can choose either option in the form shown in Figure C-12.


NSIS Script Wizard

Execute after setup

Please specify the actions to execute when the setup finish.

Program: $INSTDIR\bugInstall.exe

Params:

Readme:

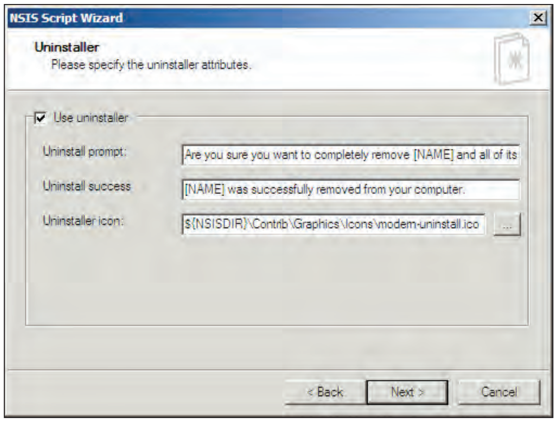
Figure C-12: Choose any final actions you want to occur after setup.

9. **Create an uninstaller.**

   Uninstallers are nice because they undo any modifications made to the system. Add an uninstaller automatically with this option, as shown in Figure C-13.

10. **Save, modify, and test.**

   When the wizard finishes, it produces an NSI script in the NIS editor. You can modify this script, compile and test it from within the editor, as shown in Figure C-14.



NSIS Script Wizard

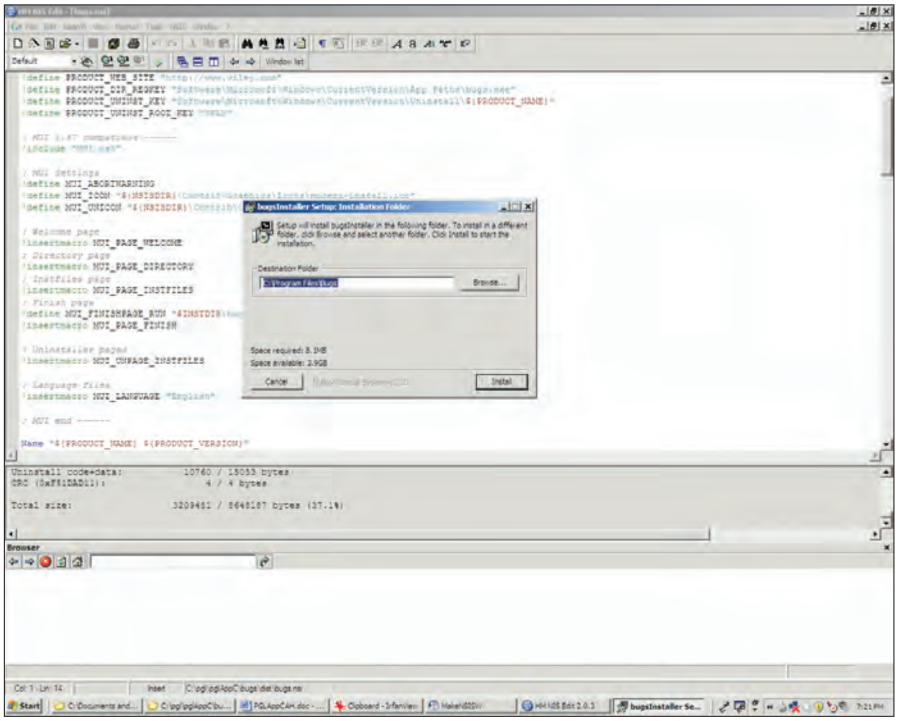
Uninstaller
Please specify the uninstaller attributes.

[✓] Use uninstaller

Uninstall prompt:	Are you sure you want to completely remove [NAME] and all of its
Uninstall success	[NAME] was successfully removed from your computer.
Uninstaller icon:	${NSISDIR}\Contrib\Graphics\Icons\modern-uninstall.ico

Figure C-13: Create an uninstaller if you wish.

The triangular Run button runs the script, and you can see the installation running in the foreground of Figure C-14.



```
!define PRODUCT_WEB_SITE "http://www.wiley.com"
!define PRODUCT_DIR_REGKEY "Software\Microsoft\Windows\CurrentVersion\App Paths\bugs.exe"
!define PRODUCT_UNINST_KEY "Software\Microsoft\Windows\CurrentVersion\Uninstall\${PRODUCT_NAME}"
!define PRODUCT_UNINST_ROOT_KEY "HKLM"

; MUI 1.67 compatible ------
!include "MUI.nsh"

; MUI Settings
!define MUI_ABORTWARNING
!define MUI_ICON "${NSISDIR}\Contrib\Graphics\Icons\modern-install.ico"
!define MUI_UNICON "${NSISDIR}\Contrib\

; Welcome page
!insertmacro MUI_PAGE_WELCOME
; Directory page
!insertmacro MUI_PAGE_DIRECTORY
; Instfiles page
!insertmacro MUI_PAGE_INSTFILES
; Finish page
!define MUI_FINISHPAGE_RUN "$INSTDIR\
!insertmacro MUI_PAGE_FINISH

; Uninstaller pages
!insertmacro MUI_UNPAGE_INSTFILES

; Language files
!insertmacro MUI_LANGUAGE "English"

; MUI end ------

Name "${PRODUCT_NAME} ${PRODUCT_VERSION}"
```

bugsInstaller Setup: Installation Folder

Setup will install bugsInstaller in the following folder. To install in a different folder, click Browse and select another folder. Click Install to start the installation.

Destination Folder: C:\Program Files\bugs

Space required: 8.1MB
Space available: 2.9GB

```
Uninstall code+data:          10760 / 15053 bytes
CRC (0xF81DAD11):                 4 / 4 bytes

Total size:                 3209481 / 8648187 bytes (37.1%)
```

Figure C-14: Test your installation to make sure it works correctly.