Python programs

This lab covers

- [Creating a very basic program]
- [Making a program directly executable on Linux/UNIX]
- [Writing programs on macOS]
- [Selecting execution options in Windows]
- [Combining programs and modules]
- [Distributing Python applications]

Up until now, you've been using the Python interpreter mainly in interactive mode. For production use, you'll want to create Python programs or scripts. Several of the sections in this lab focus on command-line programs. If you come from a Linux/UNIX background, you may be familiar with scripts that can be started from a command line and given arguments and options that can be used to pass in information and possibly redirect their input and output. If you're from a Windows or Mac background, these things may be new to you, and you may be more inclined to question their value.

It's true that command-line scripts are sometimes less convenient to use in a GUI environment, but the Mac has the option of a UNIX command-line shell, and Windows also offers enhanced command-line options. It will be well worth your time to read the bulk of this lab at some point. You may find occasions when these techniques are useful, or you may run across code you need to understand that uses some of them. In particular, command-line techniques are very useful when you need to process large numbers of files.

Creating a very basic program

Any group of Python statements placed sequentially in a file can be used as a program, or *script*. But it's more standard and useful to introduce additional structure. In its most basic form, this task is a simple matter of creating a controlling function in a file and calling that function.

Listing 11.1. File script1.py

```
def main():
    print("this is our first test script file")
main()
```

In this script, [main] is the controlling---and only---function. First, it's defined, and then it's called. Although it doesn't make much difference in a small program, this structure can give you more options and control when you create larger applications, so it's a good idea to make using it a habit from the beginning.

Starting a script from a command line

If you're using Linux/UNIX, make sure that Python is on your path and you're in the same directory as your script. Then type the following on your command line to start the script:

```
python script1.py
```

If you're using a Macintosh running OS X, the procedure is the same as for other UNIX systems. You need to open a terminal program, which is in the Utilities folder of the Applications folder. You have several other options for running scripts on OS X, which I discuss shortly.

If you're using Windows, open Command Prompt (this can be found in different menu locations depending on the version of Windows; in Windows 10, it's in the Windows System menu) or PowerShell. Either of these opens in your home folder, and if necessary, you can use the [cd] command to change to a subdirectory. Running script1.py if it was saved on your desktop would look like this:

```
C:\Users\naomi> cd Desktop

C:\Users\naomi\Desktop> python script1.py
this is our first test script file

C:\Users\naomi\Desktop>
```

I look at other options for calling scripts later in this lab, but stick with this option for now.

Command-line arguments

A simple mechanism is available for passing in command-line arguments.

Listing 11.2. File script2.py

```
import sys
def main():
    print("this is our second test script file")
    print(sys.argv)
main()
```

If you call this with the line

```
python script2.py arg1 arg2 3
```

you get

```
this is our second test script file
['script2.py', 'arg1', 'arg2', '3']
```

You can see that the command-line arguments have been stored in [sys.argv] as a list of strings.

Redirecting the input and output of a script

You can redirect the input and/or the output for a script by using command-line options. To show this technique, I use this short script.

Listing 11.3. File replace.py

```
import sys
def main():
   contents = sys.stdin.read()
   sys.stdout.write(contents.replace(sys.argv[1], sys.argv[2]))
main()
```

This script reads its standard input and writes to its standard output whatever it reads, with all occurrences of its first argument replaced with its second argument. Called as follows, the script places in [outfile] a copy of [infile] with all occurrences of [zero] replaced by [0]:

```
python replace.py zero 0 < infile > outfile
```

Note that this script works on UNIX, but on Windows, redirection of input and/or output works only if you start a script from a command-prompt window.

In general, the line

```
python script.py arg1 arg2 arg3 arg4 < infile > outfile
```

has the effect of having any [input] or [sys.stdin] operations directed out of [infile] and any [print] or [sys.stdout] operations directed into [outfile]. The effect is as though you set [sys.stdin] to [infile] with ['r'] (read) mode and [sys.stdout] to [outfile] with ['w'] (write):

```
python replace.py a A < infile >> outfile
```

This line causes the output to be appended to [outfile] rather than to overwrite it, as happened in the previous example.

You can also *pipe* in the output of one command as the input of another command:

```
python replace.py 0 zero < infile | python replace.py 1 one > outfile
```

This code results in [outfile] containing the contents of [infile], with all occurrences of [0] changed to [zero] and all occurrences of [1] changed to [one].

The argparse module

You can configure a script to accept command-line options as well as arguments. The [argparse] module provides support for parsing different types of arguments and can even generate usage messages.

To use the [argparse] module, you create an instance of [ArgumentParser], populate it with arguments, and then read both the optional and positional arguments. This listing illustrates the module's use.

Listing 11.4. File opts.py

```
from argparse import ArgumentParser

def main():
    parser = ArgumentParser()
    parser.add_argument("indent", type=int, help="indent for report")
    parser.add_argument("input_file", help="read data from this file")    1
    parser.add_argument("-f", "--file", dest="filename",                  2
                  help="write report to FILE", metavar="FILE")
    parser.add_argument("-x", "--xray",
                  help="specify xray strength factor")
    parser.add_argument("-q", "--quiet",
                  action="store_false", dest="verbose", default=True,     3
                  help="don't print status messages to stdout")

    args = parser.parse_args()

    print("arguments:", args)
main()
```

This code creates an instance of [ArgumentParser] and then adds two positional arguments, [indent] and [input_file], which are the arguments entered after all of the optional arguments have been parsed. *Positional arguments* are those without a prefix character (usually (["-"]) and are required, and in this case, the [indent] argument must also be parsable as an [int] ***1***.

The next line adds an optional filename argument with either ['-f'] or ['--file'] ***2***. The final option added, the ["quiet"] option, also adds the ability to turn off the verbose option, which is [True] by default ([action="store_false"]). The fact that these options begin with the prefix character ["-"] tells the parser that they're optional.

The final argument, "[-q]", also has a default value ([True], in this case) that will be set if the option isn't specified. The [action="store_false"] parameter specifies that if the argument *is* specified, a value of [False] will be stored in the destination. ***3***

The [argparse] module returns a Namespace object containing the arguments as attributes. You can get the values of the arguments by using dot notation. If there's no argument for an option, its value is [None]. Thus, if you call the previous script with the line

```
python opts.py -x100 -q -f outfile 2 arg2
```

the following output results:

```
arguments: Namespace(filename='outfile', indent=2, input_file='arg2',
     verbose=False, xray='100')
```

If an invalid argument is found, or if a required argument isn't given, [parse_args] raises an error:

```
python opts.py -x100 -r
```

This line results in the following response:

```
usage: opts.py [-h] [-f FILE] [-x XRAY] [-q] indent input_file
opts.py: error: the following arguments are required: indent, input_file
```

Using the fileinput module

The [fileinput] module is sometimes useful for scripts. It provides support for processing lines of input from one or more files. It automatically reads the command-line arguments (out of [sys.argv]) and takes them as its list of input files. Then it allows you to sequentially iterate through these lines. The simple example script in this listing (which strips out any lines starting with [##]) illustrates the module's basic use.

Listing 11.5. File script4.py

```
import fileinput
def main():
    for line in fileinput.input():
        if not line.startswith('##'):
            print(line, end="")
main()
```

Now assume that you have the data files shown in the next two listings.

Listing 11.6. File sole1.tst

```
## sole1.tst: test data for the sole function
0 0 0
0 100 0
##
0 100 100
```

Listing 11.7. File sole2.tst

```
## sole2.tst: more test data for the sole function
12 15 0
```

```
##
100 100 0
```

Also assume that you make this call:

```
python script4.py sole1.tst sole2.tst
```

You obtain the following result with the comment lines stripped out and the data from the two files combined:

```
0 0 0
0 100 0
0 100 100
12 15 0
100 100 0
```

If no command-line arguments are present, the standard input is all that is read. If one of the arguments is a hyphen (-), the standard input is read at that point.

The module provides several other functions. These functions allow you at any point to determine the total number of lines that have been read ([lineno]), the number of lines that have been read out of the current file ([filelineno]), the name of the current file ([filename]), whether this is the first line of a file ([isfirstline]), and/or whether standard input is currently being read ([isstdin]). You can at any point skip to the next file ([nextfile]) or close the whole stream ([close]). The short script in the following listing (which combines the lines in its input files and adds file-start delimiters) illustrates how you can use these functions.

Listing 11.8. File script5.py

```
import fileinput
def main():
    for line in fileinput.input():
        if fileinput.isfirstline():
            print("<start of file {0}>".format(fileinput.filename()))
        print(line, end="")
main()
```

Using the call

```
python script5.py file1 file2
```

results in the following (where the dotted lines indicate the lines in the original files):

```
<start of file file1>
.......................
.......................
<start of file file2>
.......................
.......................
```

Finally, if you call [fileinput.input] with an argument of a single filename or a list of filenames, they're used as its input files rather than the arguments in [sys.argv]. [fileinput.input] also has an [inplace] option that leaves its output in the same file as its input while optionally leaving the original around as a backup file. See the documentation for a description of this last option.

Quick Check: Scripts and arguments

Match the following ways of interacting with the command line and the correct use case for each:

Multiple argurments and options sys.agrv No arguments or just one argument Use file_input module Processing multiple files Redirect standard input and output Using the script as a filter Use argparse module

---

Making a script directly executable on UNIX

If you're on UNIX, you can easily make a script directly executable. Add the following line to its top, and change its mode appropriately (that is, [chmod +x replace.py]):

```
#! /usr/bin/env python
```

Note that if Python 3.x isn't your default version of Python, you may need to change the [python] in the snippet to [python3, python3.6], or something similar to specify that you want to use Python 3.x instead of an earlier default version.

Then if you place your script somewhere on your path (for example, in your bin directory), you can execute it regardless of the directory you're in by typing its name and the desired arguments:

```
replace.py zero 0 < infile > outfile
```

On UNIX, you'll have input and output redirection and, if you're using a modern shell, command history and completion.

If you're writing administrative scripts on UNIX, several library modules are available that you may find useful. These modules include [grp] for accessing the group database, [pwd] for accessing the password database, [resource] for accessing resource usage information, [syslog] for working with the syslog facility, and [stat] for working with information about a file or directory obtained from an [os.stat] call. You can find information on these modules in the *Python Library Reference*.

Scripts on macOS

In many ways, Python scripts on macOS behave the same way as they do on Linux/UNIX. You can run Python scripts from a terminal window exactly the same way as on any UNIX box. But on the Mac, you can also run Python programs from the Finder, either by dragging the script file to the Python Launcher app or by configuring Python Launcher as the default application for opening your script (or, optionally, all files with a .py extension.)

You have several options for using Python on a Mac. The specifics of all the options are beyond the scope of this course, but you can get a full explanation by going to the www.python.org website and checking out the Mac section of the "Using Python" section of the documentation for your version of Python. You should also see [section 11.6] of the documentation, "[Distributing Python applications]," for more information on how to distribute Python applications and libraries for the Mac platform.

If you're interested in writing administrative scripts for macOS, you should look at packages that bridge the gap between Apple's Open Scripting Architectures (OSA) and Python. Two such packages are [appscript] and [PyOSA].

Script execution options in Windows

If you're on Windows, you have several options for starting a script that vary in their capability and ease of use. Unfortunately, exactly what those options might be and how they are configured can vary considerably across the various versions of Windows currently in use. This book focuses on running Windows from a command prompt or PowerShell. For information on the other options for running Python on your system, you should consult the online Python documentation for your version of Python and look for "Using Python on Windows."

Starting a script from a command window or PowerShell

To run a script from a command window or PowerShell window, open a command prompt or PowerShell window. When you're at the command prompt and have navigated to the folder where your scripts are located, you can use Python to run your scripts in much the same way as on UNIX/Linux/MacOS systems:

```
> python replace.py zero 0 < infile > outfile
```

Python doesn't run?

If Python doesn't run when you enter [python] at the Windows command prompt, it probably means that the location of the Python executable isn't on your command path. You either need to add the Python executable to your system's PATH environment variable manually or rerun the installer to have it do the job. To get more help on setting up Python on Windows, refer to the Python Setup and Usage section of the online Python documentation. There, you'll find a section on using Python on Windows, with instructions for installing Python.

This is the most flexible of the ways to run a script on Windows because it allows you to use input and output redirection.

Other Windows options

Other options are available to explore. If you're familiar with writing batch files, you can wrap your commands in them. A port of the GNU BASH shell comes with the Cygwin tool set, which you can read about at www.cygwin.com and which provides UNIX-like shell capability for Windows.

On Windows, you can edit the environment variables (see the previous section) to add .py as a magic extension, making your scripts automatically executable:

```
PATHEXT=.COM;.EXE;.BAT;.CMD;.VBS;.JS;.PY
```

Try this: Making a script executable

Experiment with executing scripts on your platform. Also try to redirect input and output into and out of your scripts.

Programs and modules

For small scripts that contain only a few lines of code, a single function works well. But if the script grows beyond this size, separating your controlling function from the rest of the code is a good option to take. The rest of this section illustrates this technique and some of its benefits. I start with an example using a simple controlling function. The script in the next listing returns the English-language name for a given number between 0 and 99.

Listing 11.9. File script6.py

```
#! /usr/bin/env python3
import sys
# conversion mappings
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
def num2words(num_string):
    if num_string == '0':
        return'zero'
    if len(num_string) > 2:
        return "Sorry can only handle 1 or 2 digit numbers"
    num_string = '0' + num_string
    tens, ones = num_string[-2], num_string[-1]
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def main():
    print(num2words(sys.argv[1]))

main()
```

If you call it with

```
python script6.py 59
```

you get this result:

```
fifty nine
```

The controlling function here calls the function [num2words] with the appropriate argument and prints the result ***2***. It's standard to have the call at the bottom, but sometimes you'll see the controlling function's definition at the top of the file. I prefer this function at the bottom, just above the call, so that I don't have to scroll back up to find it after going to the bottom to find out its name. This practice also cleanly separates the scripting plumbing from the rest of the file, which is useful when combining scripts and modules.

People combine scripts with modules when they want to make functions they've created in a script available to other modules or scripts. Also, a module may be instrumented so it can run as a script either to provide a quick interface to it for users or to provide hooks for automated module testing.

Combining a script and a module is a simple matter of putting the following conditional test around the controlling function:

```
if __name__ == '__main__':
    main()
else:
    # module-specific initialization code if any
```

If it's called as a script, it will be run with the name [__main__], and the controlling function, [main], will be called. If the test has been imported into an interactive session or another module, its name will be its filename.

When creating a script, I often set it as a module as well right from the start. This practice allows me to import it into a session and interactively test and debug my functions as I create them. Only the controlling function needs to be debugged externally. If the script grows, or if I find myself writing functions I might be able to use elsewhere, I can separate those functions into their own module or have other modules import this module.

The script in [listing 11.10] is an extension of the previous script but modified to be used as a module. The functionality has also been expanded to allow the entry of a number from 0 to 999999999999999 rather than just from 0 to 99. The controlling function ([main]) does the checking of the validity of its argument and also strips out any commas in it, allowing more user-readable input like 1,234,567.

Listing 11.10. File n2w.py

```python
#! /usr/bin/env python3
"""n2w: number to words conversion module: contains function
   num2words. Can also be run as a script
usage as a script: n2w num
           (Convert a number to its English word description)
           num: whole integer from 0 and 999,999,999,999,999 (commas are
           optional)
example: n2w 10,003,103
           for 10,003,103 say: ten million three thousand one hundred three
"""
import sys, string, argparse
_1to9dict = {'0': '', '1': 'one', '2': 'two', '3': 'three', '4': 'four',
             '5': 'five', '6': 'six', '7': 'seven', '8': 'eight',
             '9': 'nine'}
_10to19dict = {'0': 'ten', '1': 'eleven', '2': 'twelve',
               '3': 'thirteen', '4': 'fourteen', '5': 'fifteen',
               '6': 'sixteen', '7': 'seventeen', '8': 'eighteen',
               '9': 'nineteen'}
_20to90dict = {'2': 'twenty', '3': 'thirty', '4': 'forty', '5': 'fifty',
               '6': 'sixty', '7': 'seventy', '8': 'eighty', '9': 'ninety'}
_magnitude_list = [(0, ''), (3, ' thousand '), (6, ' million '),
                  (9, ' billion '), (12, ' trillion '),(15, '')]
def num2words(num_string):
    """num2words(num_string): convert number to English words"""
    if num_string == '0':
        return 'zero'
    num_string = num_string.replace(",", "")
    num_length = len(num_string)
    max_digits = _magnitude_list[-1][0]
    if num_length > max_digits:
        return "Sorry, can't handle numbers with more than  " \
               "{0} digits".format(max_digits)
    num_string = '00' + num_string                                      #5
    word_string = ''                                                    #7
    for mag, name in _magnitude_list:
        if mag >= num_length:
            return word_string
        else:
            hundreds, tens, ones = num_string[-mag-3], \
                 num_string[-mag-2], num_string[-mag-1]
            if not (hundreds == tens == ones == '0'):
                word_string = _handle1to999(hundreds, tens, ones) + \
                                            name + word_string
def _handle1to999(hundreds, tens, ones):
    if hundreds == '0':
        return _handle1to99(tens, ones)
    else:
        return _1to9dict[hundreds] + ' hundred ' + _handle1to99(tens, ones)
def _handle1to99(tens, ones):
    if tens == '0':
        return _1to9dict[ones]
    elif tens == '1':
        return _10to19dict[ones]
    else:
        return _20to90dict[tens] + ' ' + _1to9dict[ones]
def test():
    values = sys.stdin.read().split()
    for val in values:
        print("{0} = {1}".format(val, num2words(val)))
def main():
    parser = argparse.ArgumentParser(usage=__doc__)
    parser.add_argument("num", nargs='*')
    parser.add_argument("-t", "--test", dest="test",
                        action='store_true', default=False,
                        help="Test mode: reads from stdin")
    args = parser.parse_args()
    if args.test:
        test()
    else:
        try:
            result = num2words(args.num[0])
        except KeyError:
            parser.error('argument contains non-digits')
        else:
            print("For {0}, say: {1}".format(args.num[0], result))
if __name__ == '__main__':
    main()
else:
    print("n2w  loaded as a module")
```

If it's called as a script, the name will be [__main__]. If it's imported as a module, it will be named [n2w] ***12***.

This [main] function illustrates the purpose of a controlling function for a command-line script, which in effect is to create a simple UI for the user. It may handle the following tasks:

- [Ensure that there's the right number of command-line arguments and that they're of the right types. Inform the user, giving usage information if not. Here, the function ensures that there is a single argument, but it doesn't explicitly test to ensure that the argument contains only digits.]
- [Possibly handle a special mode. Here, a ['--test'] argument puts you in a test mode.]
- [Map the command-line arguments to those required by the functions, and call them in the appropriate manner. Here, commas are stripped out, and the single function [num2words] is called.]
- [Possibly catch and print a more user-friendly message for exceptions that may be expected. Here, [KeyErrors] are caught, which occurs if the argument contains nondigits.^[[1]]^] A better way to do this would be to explicitly check for nondigits in the argument using the regular expression module that will be introduced later. This would ensure that we don't hide [KeyErrors] that occur due to other reasons.
- [Map the output if necessary to a more user-friendly form, which is done here in the [print] statement. If this were a script to run on Windows, you'd probably want to let the user open it with the double-click method---that is, to use the [input] to query for the parameter, rather than have it as a command-line option and keep the screen up to display the output by ending the script with the line]

  ```
          input("Press the Enter key to exit")
  ```

```
But you may still want to leave the test mode in as a command-line
option.
```

The test mode in the following listing provides a regression test capability for the module and its [num2words] function. In this case, you use it by placing a set of numbers in a file.

Listing 11.11. File n2w.tst

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 98 99 100
101 102 900 901 999
999,999,999,999,999
1,000,000,000,000,000
```

Then type

```
python n2w.py --test < n2w.tst > n2w.txt
```

The output file can be easily checked for correctness. This example was run several times during its creation and can be rerun any time [num2words] or any of the functions it calls are modified. And yes, I'm aware that full exhaustive testing certainly didn't occur. I admit that well over 999 trillion valid inputs for this program haven't been checked!

Often, the provision of a test mode for a module is the only function of a script. I know of at least one company in which part of the development policy is to always create one for every Python module developed. Python's built-in data object types and methods usually make this process easy, and those who practice this technique seem to be unanimously convinced that it's well worth the effort.

Another option is to create a separate file with only the portion of the [main] function that handles the argument and import [n2w] into this file. Then only the test mode would be left in the [main] function of [n2w.py].

Quick Check: Programs and modules

What issue is the use of [if __name__ == "__main__":] meant to prevent, and how does it do that? Can you think of any other way to prevent this issue?

Distributing Python applications

You can distribute your Python scripts and applications in several ways. You can share the source files, of course, probably bundled in a zip or tar file. Assuming that the applications were written portably, you could also ship only the bytecode as .pyc files. Both of those options, however, usually leave a lot to be desired.

Wheels packages

The current standard way of packaging and distributing Python modules and applications is to use packages called wheels. Wheels are designed to make installing Python code more reliable and to help manage dependencies. The details of how to create wheels are beyond the scope of this lab, but full details about the requirements and the process for creating wheels are in the Python Packaging User Guide at https://packaging.python.org.

zipapp and pex

If you have an application that's in multiple modules, you can also distribute it as an executable zip file. This format relies on two facts about Python.

First, if a zip file contains a file named [__main__.py], Python can use that file as the entry point to the archive and execute the [__main__.py] file directly. In addition, the zip file's contents are added to [sys.path], so they're available to be imported and executed by [__main__.py].

Second, zip files allow arbitrary contents to be added to the beginning of the archive. If you add a shebang line pointing to a Python interpreter, such as [#!/usr/bin/env python3], and give the file the needed permissions, the file can become a self-contained executable.

In fact, it's not that difficult to manually create an executable zipapp. Create a zip file containing a [__main__.py], add the shebang line to the beginning, and set the permissions.

Starting with Python 3.5, the zipapp module is included in the standard library; it can create zipapps either from the command line or via the library's API.

A more powerful tool, pex, isn't in the standard library but is available from the package index via pip. pex does the same basic job but offers many more features and options, and it's available for Python 2.7, if needed. Either way, zip file apps are convenient ways to package and distribute multifile Python apps ready to run.

py2exe and py2app

Although it's not the purpose of this course to dwell on platform-specific tools, it's worth mentioning that [py2exe] creates standalone Windows programs and that [py2app] does the same on the macOS platform. By *standalone*, I mean that they're single executables that can run on machines that don't have Python installed. In many ways, standalone executables aren't ideal, because they tend to be larger and less flexible than native Python applications. But in some situations, they're the best---and sometimes the only---solution.

Creating executable programs with freeze

It's also possible to create an executable Python program that runs on machines that don't have Python installed by using the [freeze] tool. You'll find the instructions for this in the Readme file inside the freeze directory in the Tools subdirectory of the Python source directory. If you're planning to use [freeze], you'll probably need to download the Python source distribution.

In the process of "freezing" a Python program, you create C files, which are then compiled and linked using a C compiler, which you need to have installed on your system. The frozen application will run only on the platform for which the C compiler you use provides its executables.

Several other tools try in one way or another to convert and package a Python interpreter/environment with an application in a standalone application. In general, however, this path is still difficult and complex, and you probably want to avoid it unless you have a strong need and the time and resources to make the process work.

Summary

- [Python scripts and modules in their most basic form are just sequences of Python statements placed in a file.]
- [Modules can be instrumented to run as scripts, and scripts can be set up so that they can be imported as modules.]
- [Scripts can be made executable on the UNIX, macOS, or Windows command lines. They can be set up to support command-line redirection of their input and output, and with the [argparse] module, it's easy to parse out complex combinations of command-line arguments.]
- [On macOS, you can use the Python Launcher to run Python programs, either individually or as the default application for opening Python files.]
- [On Windows, you can call scripts in several ways: by opening them with a double-click, using the Run window, or using a command-prompt window.]
- [Python scripts can be distributed as scripts, as bytecode, or in special packages called wheels.]
- [[py2exe], [py2app], and the [freeze] tool provide an executable Python program that runs on machines that don't contain a Python interpreter.]
- [Now that you have an idea of the ways to create scripts and applications, the next step is looking at how Python can interact with and manipulate filesystems.]