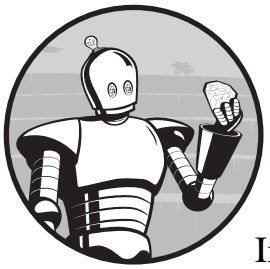
1

GENERAL PURPOSE UTILITIES



In any programming language, scripting is the solution to frequently performed tasks. If you find yourself asking *Couldn't a robot or well-trained monkey do this job?*, then scripting with Ruby just might be the next best solution. Writing scripts for frequently performed tasks makes your job and computing experience as efficient as it can be. Who wouldn't want to get the job done in less time with less effort? As you work through these examples, I encourage you to write down ideas for your own scripts. Once you've finished this book, you will probably have a list of scripts you want to write, or at the very least, some useful revisions of mine. Are you ready? Let's get started!

#1 Check for Changed Files

changedFiles.rb

The purpose of this script is to validate a file's integrity. While it sounds like a humble end use, its applications are broad: If you can't trust the contents of files on your computer, you can't trust your computer. Would you know if a malicious worm or virus modified a file on your system? If you think your

antivirus has you covered, think again—most only go as far as checking for known viruses and their signatures. File integrity validation is used every day for real-world tasks such as digital forensics and tracking the behavior of malicious logic. One method of tracking file integrity is shown below.

The Code

```
require 'find'
require 'digest/md5'

unless ARGV[0] and File.directory?(ARGV[0])
    puts "\n\n\nYou need to specify a root directory:  changedFiles.rb
<directory>\n\n\n"
    exit
end

❶ root = ARGV[0]
  oldfile_hash = Hash.new
  newfile_hash = Hash.new
  file_report = "#{root}/analysis_report.txt"
  file_output = "#{root}/file_list.txt"
  oldfile_output = "#{root}/file_list.old"

❷ if File.exists?(file_output)
    File.rename(file_output, oldfile_output)
    File.open(oldfile_output, 'rb') do |infile|
      while (temp = infile.gets)
        line = /(.+)\s{5,5}(\w{32,32})/.match(temp)
        puts "#{line[1]}  --->  #{line[2]}"
        oldfile_hash[line[1]] = line[2]
      end
    end
  end

❸ Find.find(root) do |file|
      next if /^\./.match(file)
      next unless File.file?(file)
      begin
          newfile_hash[file] = Digest::MD5.hexdigest(File.read(file))
      rescue
          puts "Error reading #{file} --- MD5 hash not computed."
      end
  end

  report = File.new(file_report, 'wb')
  changed_files = File.new(file_output, 'wb')

  newfile_hash.each do |file, md5|
    changed_files.puts "#{file}     #{md5}"
  end

❹ newfile_hash.keys.select { |file| newfile_hash[file] == oldfile_hash[file]
  }.each do |file|
```

```
    newfile_hash.delete(file)
    oldfile_hash.delete(file)
end

❺ newfile_hash.each do |file, md5|
    report.puts "#{oldfile_hash[file] ? "Changed" : "Added"} file: #{file}
#{md5}"
    oldfile_hash.delete(file)
end

❻ oldfile_hash.each do |file, md5|
    report.puts "Deleted/Moved file: #{file}     #{md5}"
end

report.close
changed_files.close
```

Running the Code

Execute this script by typing:

```
ruby changedFiles.rb /path/to/check/
```

You can add more than one directory to crawl, but subdirectories will automatically be verified. The script will automatically determine if a directory exists and then add it to the crawler's queue.

The Results

The script will initially produce two separate files (*changed.files* and *file_report.txt*). Both will contain a list of the names and MD5 hashes for all of the files scanned by the script:

```
Added file: fileSplit.rb d79c592af618266188a9a49f91fe0453
Added file: fileJoin.rb 5aedfe682e300dcc164ebbdebdcd8875
Added file: win32RegCheck.rb c0d26b249709cd91a0c8c14b65304aa7
Added file: changedFiles.rb c2760bfe406a6d88e04f8969b4287b4c
Added file: encrypt.rb 08caf04913b4a6d1f8a671ea28b86ed2
Added file: decrypt.rb 90f68b4f65bb9e9a279cd78b182949d4
Added file: file_report.txt d41d8cd98f00b204e9800998ecf8427e
Added file: changed.files d41d8cd98f00b204e9800998ecf8427e
Added file: test.txt a0cbe4bbf691bbb2a943f8d898c1b242
Added file: test.txt.rsplit1 35d5b2e522160ce3b3b98d2d4ad2a86e
Added file: test.txt.rsplit2 a65dde64f16a4441ff1619e734207528
Added file: test.txt.rsplit3 264b40b40103a4a3d82a40f82201a186
Added file: test.txt.rsplit4 943600762a52864780b9b9f0614a470a
Added file: test.txt.rsplit5 131c8aa7155483e7d7a999bf6e2e21c0
Added file: test.txt.rsplit6 1ce31f6fbeb01cbed6c579be2608e56c
```

After the script is run a second time, three files will appear in the root directory. Two of the files, *changed.files* and *old_changed.files*, are where the MD5 hashes are stored; the third, *file_report.txt*, is a text file showing the

results. The script will compare the MD5 hashes for all of the files listed in *changed.files* with those in *old_changed.files* and return any differences found. Here is an example:

```
Changed file: old_changed.files 45de547aef9366eeaeb1b565dff1e1a3
Deleted/Moved file: test.txt.rsplit4 943600762a52864780b9b9f0614a470a
Deleted/Moved file: test.txt.rsplit5 131c8aa7155483e7d7a999bf6e2e21c0
Deleted/Moved file: test.txt.rsplit6 1ce31f6fbeb01cbed6c579be2608e56c
Deleted/Moved file: test.txt.rsplit1 35d5b2e522160ce3b3b98d2d4ad2a86e
Deleted/Moved file: test.txt.rsplit2 a65dde64f16a4441ff1619e734207528
Deleted/Moved file: test.txt.rsplit3 264b40b40103a4a3d82a40f82201a186
```

How It Works

This script is great for verifying the contents of your hard drive and ensuring they haven't been manipulated. The script starts by confirming that the user-supplied arguments were included and that a valid directory was given. Next is the initialization of variables used in the script. The `root` variable contains the root directory to scan, two hashes are created that will be used for comparing the files and their MD5 hashes, and, finally, the names of the files to be used are specified ❶. The script output is saved in two or three files, depending on whether the script has been run before. The main file, *file_report.txt*, is used for reading the output, and the other two files are used to store the list of MD5 hashes.

Next, the script checks to see if it's been run before by looking for *file_list.txt* ❷. If the file is not found, the script moves on. If it finds *file_list.txt*, the script immediately renames the file. The renamed file is then opened and the contents are read. For every line in the file, the script reads a filename and MD5 hash and stores these in the `oldfile_hash` for later comparison. Once the `oldfile_hash` has been populated, the script is ready to begin computing new MD5 hashes and comparing results.

As the script works its way through the directory tree, it will iterate through each object ❸. The `Find.find` method is a powerful recursive way to retrieve files in a directory and subdirectories. The code block will be run on every file found. The first statement is looking for the "." and ".."—which are skipped for obvious reasons. If the object is a directory, the script will give it the skip treatment and press on. If the item is a file, the hash is generated and stored for later use. The hashing process is surrounded by a `begin/rescue` block to save us if something goes terribly wrong.

The bulk of the information gathering is now complete. All that is left is to determine the status of each file. If a file has the same name and MD5 hash, it is unchanged and the script will remove the filename from the output hash. There are three categories that a file can fit into aside from *Unchanged*. The first is *Deleted or Moved*, which is determined by a file's presence in the past scan but not the current one ❹. Next is the *Changed* category. If the filename exists and the MD5 hash is not the same as in the previous scans, the file has been changed ❺. At this point, for the sake of readability in the code, I used the *ternary operator*, which is an abbreviation of the `if/then/else` statement. So this says *if* the file exists in `oldfile_hash`, *then*

label it *Changed*, *else* label it *Added*; since the filename doesn't exist previously, it has been added since the last scan ❻. All of the data is saved, and a report is generated so the user is aware of each file's status. If anything is out of the ordinary, further analysis is required.

There are several software packages that perform similar computations for security purposes, but the method above is a nice alternative, and the price is right, too. For enhanced security, you can store the output files on a separate medium, but I generally leave them in the top-level directory for simplicity's sake.

Hacking the Script

This script can be modified to use any number of hashing algorithms. I chose MD5 because it is the most popular for checking a file's integrity (even though its hashes are vulnerable to a collision attack). This script works on both Microsoft Windows and Unix-like systems. Cross platform scripts are always a plus!

Other potential changes to the script include encrypting the hashed files for added protection or interfacing the results into a database. The script has many potential uses, and I'll leave it to you to investigate further. If you are curious about encryption, check out the next script.

#2 Encrypt a File

encrypt.rb

How often have you heard about people selling their computers on an auction site, only to later discover that their sensitive information had been exposed on the Internet? And what about corporate espionage, or all those missing government laptops? If you talk to security experts, one of the first recommendations they make is to encrypt sensitive information. You could always buy a program that does this for you, but that's no fun. Let's write our own encryption script! There are many encryption algorithms from which to choose, all with varying levels of strength. In this example, I will be using Blowfish, a very fast, symmetric block cipher.

The Code

```
❶ require 'crypt/blowfish'

  unless ARGV[0]
      puts "Usage: ruby encrypt.rb <filename.ext>"
      puts "Example: ruby encrypt.rb secret.stuff"
      exit
  end

  #take in the file name to encrypt as an argument
  filename = ARGV[0].chomp
  puts filename
❷ c = "Encrypted_#{filename}"

❸ if File.exists?(c)
```

```
    puts "File already exists."
    exit
end

❹ print 'Enter your encryption key (1-56 bytes): '
kee = gets.chomp

❺ begin
❻     blowfish = Crypt::Blowfish.new(kee)
❼     blowfish.encrypt_file(filename.to_str, c)
      puts 'Encryption SUCCESS!'
❽ rescue Exception => e
      puts "An error occurred during encryption: \n #{e}"
end
```

Running the Code

You must have the Ruby gem *crypt* installed on your system—use the command `gem install crypt` at the console to install the crypt library. This encryption script is accessed through a command prompt. To run, type:

```
ruby encryption.rb /path/of/file/to/encrypt
```

You will be prompted for a password:

```
Enter your encryption key (1-56 characters):
```

WARNING *Remember your password, or you won't be able to decrypt your file!*

Now press ENTER and, if the encryption was successful, you will see this message:

```
Encryption SUCCESS!
```

Look in the folder where this script resides; you will see the new, encrypted file, named *Encrypted_<filename>*.

The Results

For the example above, I used a plaintext file with the following contents:

```
Wicked Cool Ruby Scripts
```

After the script has finished encrypting the file, it will output a success message. You can then attempt to view the file. Good luck deciphering it if you forgot your password:

```
qo”.1[>°‹|šã_8tÃhÞí}“ƒ-‰1ð»=ðrÞ¡.¸
```

As you can see, the results don't resemble the original plaintext at all.

How It Works

In the first line, I include the library used for encryption: crypt/blowfish ❶. Note that you could change this to use another algorithm, such as Rijndael or GOST. Line ❷ starts the creation of our encrypted file. Creating files in Ruby is very simple. As you can see, I used a shortcut to name the file by including the variable (`filename`) *in line* with my string, `Encrypted_#{filename}`. I enjoy having the option of including variables in line with a text string, so you will see I use them throughout this book.

Next, we check to see if the encrypted filename already exists. We don't want the script overwriting files arbitrarily—data gets lost very easily that way. If there is no conflict, the script continues on ❸. Now that the script knows the encrypted file hasn't already been created, an *encryption key*, or password, needs to be provided by the user. The script asks for a key that is between 1 and 56 characters ❹. Once all the ncessary information has been collected, the script starts a `begin/rescue` error-handling block ❺. The last and most important piece of the script is the actual encryption of the data. A new encryption object is created with the encryption key passed as an argument ❻. Then the file is passed to the `encrypt_file` method, and *poof*—the file is encrypted ❼. If any errors were encountered during the encryption phase, the rescue block is there to catch them and exit the script gracefully, reporting the specific error ❽.

Hacking the Script

You can modify this script in many different ways. For example, you can make it a modular part of another program, change the encryption algorithm, layer the encryption, automatically delete the plaintext file after encryption, or encrypt entire directories.

Next, we will look at how to reverse the process and get our information back.

#3 Decrypt a File

decrypt.rb

This code is structured much like the encryption algorithm, so I will focus on the differences between the two. I am using the same algorithm for decryption as used during encryption. As mentioned earlier, you can use any number of encryption algorithms—just be sure to use the corresponding decryption algorithm. Don't forget your password, or else you will have to write your own brute force script if you ever want to see your data again!

The Code

```
require 'crypt/blowfish'

unless ARGV[0]
    puts "Usage: ruby decrypt.rb <Encrypted_filename.ext>"
    puts "Example: ruby decrypt.rb Encrypted_secret.stuff"
      exit
```

```
end

❶ filename = ARGV[0].chomp
puts "Decrypting #{filename}."
p = "Decrypted_#{filename}"

❷ if File.exists?(p)
    puts "File already exists."
    exit
end

❸ print 'Enter your encryption key: '
kee = gets.chomp

begin
❹   blowfish = Crypt::Blowfish.new(kee)
    blowfish.decrypt_file(filename.to_str, p)
    puts 'Decryption SUCCESS!'
❺ rescue Exception => e
    puts "An error occurred during decryption: \n #{e}"
end
```

Running the Code

The code is simple to execute; just type the name of decryption script followed by the file you wish to decrypt:

```
ruby decrypt.rb encrypted_filename.ext
```

The Ruby script will prompt you for the encryption key. Remember that you must have the key used to encrypt the file in order to decrypt it. If you don't, then there is no way to recover the file other than brute force, which can take much longer than you probably want to spend.

The Results

```
File Content Before: qo”.1[>°‹|šã_8tÃhÞí}“ƒ-‰1ð»=ðrÞ¡.¸
File Content After: Wicked Cool Ruby Scripts
```

As expected, the decryption script took the cipher text and cleanly translated it back into plaintext. If you have time, try using the wrong key and examine the output. It will look as cryptic as the cipher text.

How It Works

The script starts by grabbing the filename from the command-line argument and initializing the variables that will be used ❶. Whenever a file is created, you should always check to see if there is already a file with the same name ❷. After the algorithms have been initialized, the script will ask for a key ❸.

Up to this point in the script, everything looks as it did for the encryption script. Even if you type the wrong encryption key, the script will decrypt the file based on that incorrect key, with results as cryptic as they were before. If all goes well, you'll be able use the file that was previously encrypted.

The actual decryption happens using the `decrypt` method from the crypt library ❹, which is just the reverse of the encryption.

If there are no errors or exceptions, the output will display `Decryption SUCCESS!` and the program will exit. If there is an issue, our `begin/rescue` block will catch the error and enter our `rescue` case. The `rescue` case displays an error message and notifies the user that the file has not yet been decrypted ❺.

Any modifications you make to the encryption script must also be made to the decryption script. If you do a task in the encryption script and forget to undo it in the decryption script, your data will be history.

#4 File Splitting

fileSplit.rb

A cool use of Ruby scripting is to split a large file into several smaller, symmetric files. I wrote this script for a friend who was having trouble sending files into and out of his corporate network since the network administrators wouldn't allow files over a certain size to be transferred—presumably for bandwidth reasons. This script worked like a charm.

The Code

```
❶ if ARGV.size != 2
      puts "Usage: ruby fileSplit.rb <filename.ext> <size_of_pieces_in_bytes>"
      puts "Example: ruby fileSplit.rb myfile.txt 10"
      exit
  end

  filename = ARGV[0]
  size_of_split = ARGV[1]

❷ if File.exists?(filename)
      file = File.open(filename, "r")
      size = size_of_split.to_i

      puts "The file is #{File.size(filename)} bytes."

❸     temp = File.size(filename).divmod(size)
      pieces = temp[0]
      extra = temp[1]

      puts "\nSplitting the file into #{pieces} (#{size} byte) pieces and 1
  (#{extra} byte) piece"

❹     pieces.times do |n|
          f = File.open("#{filename}.rsplit#{n}", "w")
          f.puts file.read(size)
```

```
    end

❺   e = File.open("#{filename}.rsplit#{pieces}", "w")
    e.puts file.read(extra)
else
    puts "\n\nFile does NOT exist, please check filename."
end
```

Running the Code

It's easiest to run this script in a fresh directory with the file you want to split. Like the previous scripts, start by typing:

```
ruby fileSplit.rb path/to/file size_of_pieces_in_bytes
```

If you want to split up a 10KB file into 1,000-byte (or 1KB) pieces, the script will make 10 separate files labeled *<filename>.rsplit<#1-10>*. To do this, type:

```
ruby fileSplit.rb test.txt 1000
```

The Results

The initial file used in this example is called *test.txt*, and the results are shown below:

```
test.txt.rsplit0
test.txt.rsplit1
test.txt.rsplit2
test.txt.rsplit3
test.txt.rsplit4
test.txt.rsplit5
test.txt.rsplit6
test.txt.rsplit7
test.txt.rsplit8
test.txt.rsplit9
```

How It Works

If you are faced with a pesky corporate network policy that has limited the size of files allowed to be transferred, or if you are looking for a more reliable way to transfer large files, this utility will save the day. I was faced with the corporate scenario, and I knew the file size limit, so I was able to hard code the file sizes. However, you can use whatever size you need or make it an option in the script.

The script starts by reading the first two items out of the ARGV array: the name of the file to split and the size of each section. If the two variables, `filename` and `size`, aren't specified, the script will display `correct usage` ❶.

Next, the script ensures that you are trying to split a real file ❷. It's tough to divide by zero and even more difficult to split a file that doesn't exist. If the file cannot be found, the script exits and displays an error message

letting the user know something is wrong with the filename. Hopefully, the file is found, and the script begins to set up for the splits.

As you know, files can be any size, and rarely are they perfectly divisible by whatever number of bytes you chose. In order to deal with dynamic file sizes, the script uses `divmod`—`divmod` will divide two numbers, passing back an array containing the quotient and modulus. In this script, `pieces` is the quotient and the `extra` is the modulus ❸.

To maintain the file's integrity, the split pieces are created by reading in one byte at a time and writing binary to the output. This section is where the magic happens ❹. The whole pieces are written first, and then the extra piece ❺.

Hacking the Script

If you want to extend the code, a perfect addition would be to add a compression routine before the file is split. I'll talk more about compression later. Another spin on this script, giving it more flexibility, is to add an option for splitting the file into a specific number of pieces, regardless of the size. You could also modify this script to create file pieces sized to the media format of your choice, whether it's 700MB CDs or 2.88MB floppies.

#5 File Joining

fileJoin.rb

This script was also written for my friend, knowing he would be pretty upset if he didn't have a way to reconstruct his files. This is a companion script for the file-splitting one, and both scripts can be put together in a wrapper if you prefer. (A *wrapper* is code that brings both scripts together in one utility.) I separated them here for instructional purposes. This file-joining script will only work for files that were previously split (see "#4 File Splitting" on page 9); however, you can adjust it to suit your needs.

The Code

```
❶ if ARGV.size != 1
      puts "Usage: ruby fileJoin.rb <filename.ext>"
      puts "Example: ruby fileJoin.rb myfile.txt"
      exit
  end

  file = ARGV[0]
  piece = 0
  orig_file = "New.#{file}"

❷ if File.exists?("#{file}.rsplit#{piece}")
❸     ofile = File.open(orig_file, "w")
❹     while File.exists?("#{file}.rsplit#{piece}")
          puts "Reading File: #{file}.rsplit#{piece}"
❺         ofile << File.open("#{file}.rsplit#{piece}","r").read.chomp
          piece+=1
      end
      ofile.close
```

```
    puts "\nSUCCESS!  File reconstructed."
else
    puts "\n\nCould not find #{file}.rsplit#{piece}."
end
```

Running the Code

The script does not support a change in directory, so make sure it is located in the same directory as the files you want to join. To run the script, type:

```
ruby fileJoin.rb filename.ext
```

The Results

Using the files output by the file-splitting script, the input should be the name of the file to be reassembled as shown below:

```
Reading File: test.txt.rsplit0
Reading File: test.txt.rsplit1
Reading File: test.txt.rsplit2
Reading File: test.txt.rsplit3
Reading File: test.txt.rsplit4
Reading File: test.txt.rsplit5
Reading File: test.txt.rsplit6
Reading File: test.txt.rsplit7
Reading File: test.txt.rsplit8
Reading File: test.txt.rsplit9
SUCCESS!  File Reconstructed.
```

After the script has run, the assembled file will be called *New.test.txt*.

The new file that was joined will be found in the same directory as the script. Each *.rsplit* piece will still exist, in case there were any errors reconstructing the file. Once you locate the file and open it, the contents should be exactly as they were before you split the file. You can compare the old and new MD5 hashes to see for yourself (see "#1 Check for Changed Files" on page 1).

How It Works

The script starts by getting the original filename of the file that was split. If a name was not provided as a command-line argument, the script will complain, and you'll have to try again ❶. If a filename is provided, then the script checks to see if there are any pieces that correspond to that filename ❷. If not, it will again complain, saying the file couldn't be found.

After the first piece is located, the script creates the output file ❸. Next, a while loop is used to ensure that only the next consecutive piece is appended to the main body ❹. As long as there is a "next piece," the script will continue appending to the output file. Since the data of each split piece has a newline at the end, we use the `chomp` method to ensure only raw data is being streamed ❺.

The output file is closed after all the pieces have been appended to it. A nice success message is displayed and the script exits. Now you can check the new file to verify that it is perfectly restored.

Hacking the Script

If you trust the script, you can tweak it to clean up after itself, erasing all of the *.rsplit* pieces. You could also compute the MD5 hash of the file before and after the split to verify its authenticity.

#6 Windows Process Viewer

listWin Processes.rb

The process viewer in Windows Task Manager can be extremely frustrating, due to a lack of information. If you have ever used the `ps` command on a Unix-like system, you know how much more information is available besides the process name, CPU/memory usage, and process owner. Some applications make nice, detailed entries in the process viewer, and those tasks are easy to identify, but other applications have some ambiguous name that doesn't do you any favors. Having an alternative way to view the processes is handy because you can customize the script to show exactly what is important to you. This script demonstrates how to retrieve every available process property.

The Code

```
❶ require 'win32ole'

❷     ps = WIN32OLE.connect("winmgmts:\\\\.")
❸     ps.InstancesOf("win32_process").each do |p|
❹     puts "Process: #{p.name}"
      puts "\tID: #{p.processid}"
      puts "\tPATH:#{p.executablepath}"
      puts "\tTHREADS: #{p.threadcount}"
      puts "\tPRIORITY: #{p.priority}"
      puts "\tCMD_ARGS: #{p.commandline}"
    end
```

Running the Code

The script is written so that it runs autonomously and displays information about each process. Add and remove properties in the script as needed.

The Results

```
Process: winlogon.exe
    ID: 1296
    PATH:C:\WINDOWS\system32\winlogon.exe
    THREADS: 22
    PRIORITY: 13
    CMD_ARGS: winlogon.exe
```

```
Process: services.exe
    ID: 1348
    PATH:C:\WINDOWS\system32\services.exe
    THREADS: 15
    PRIORITY: 9
    CMD_ARGS: C:\WINDOWS\system32\services.exe
Process: explorer.exe
    ID: 1240
    PATH:C:\WINDOWS\Explorer.EXE
    THREADS: 14
    PRIORITY: 8
    CMD_ARGS: C:\WINDOWS\Explorer.EXE
Process: svchost.exe
    ID: 3836
    PATH:C:\WINDOWS\System32\svchost.exe
    THREADS: 8
    PRIORITY: 8
    CMD_ARGS: C:\WINDOWS\System32\svchost.exe -k HTTPFilter
Process: firefox.exe
    ID: 2140
    PATH:C:\Program Files\Mozilla Firefox\firefox.exe
    THREADS: 7
    PRIORITY: 8
    CMD_ARGS: "C:\Program Files\Mozilla Firefox\firefox.exe"
Process: cmd.exe
    ID: 1528
    PATH:C:\WINDOWS\system32\cmd.exe
    THREADS: 1
    PRIORITY: 8
    CMD_ARGS: "C:\WINDOWS\system32\cmd.exe"
Process: ruby.exe
    ID: 244
    PATH:c:\ruby\bin\ruby.exe
    THREADS: 4
    PRIORITY: 8
    CMD_ARGS: ruby ListWinProcesses.rb
```

How It Works

For most interactions with the Windows Operating System, I use the win32ole library ❶. This library is very useful, and I'll demonstrate more automation with it in later chapters. The first part of the script is the initialization of *winmgmts*, which lets the script interact with the Windows internal methods ❷. *Winmgmts* is the *Windows Management Interface (WMI)*. WMI has a lot of useful tools that you can explore further if you're interested in scripting for Windows. I called my instance of WMI `ps` because it reminds me of the `ps` method in Unix-style systems.

Next, the script iterates all instances of `win32_process`. This is where all of the processes are found and information can be extracted ❸. The properties I used for the script are `process name`, `id`, `path`, `threads running`,

priority, and command line arguments. I find knowing command-line arguments useful in case I want to invoke the program from some other script or from the command line ❹.

Hacking the Script

If you want to view everything about each process, you can include the properties listed below from the WMI properties class. There are a lot of possible configurations to suit your needs.

WMI Process Class Properties		
Caption	OSCreationClassName	QuotaPeakPagedPoolUsage
CommandLine	OSName	ReadOperationCount
CreationClassName	OtherOperationCount	ReadTransferCount
CreationDate	OtherTransferCount	SessionId
CSCreationClassName	PageFaults	Status
CSName	PageFileUsage	TerminationDate
Description	ParentProcessId	ThreadCount
ExecutablePath	PeakPageFileUsage	UserModeTime
ExecutionState	PeakVirtualSize	VirtualSize
Handle	PeakWorkingSetSize	WindowsVersion
HandleCount	Priority	WorkingSetSize
InstallDate	PrivatePageCount	WriteOperationCount
KernelModeTime	ProcessId	WriteTransferCount
MaximumWorkingSetSize	QuotaNonPagedPoolUsage	WorkingSetSize
MinimumWorkingSetSize	QuotaPagedPoolUsage	
Name	QuotaPeakNonPagedPoolUsage	

While the list above contains all the properties for the WMI process class, there are several other operating system classes—each with its own properties. To use the same script with a different operating system class, replace the win32_process at ❸ with another WMI class. For example, registry would be win32_registry.

#7 File Compressor

compress.rb

Being able to effectively compress a file is a serious asset when you start talking about data storage. The more efficient the compression, the more information can be stored in the same amount of space. There are two popular Ruby compression libraries in use today. The first is ruby-zlib, and the second is rubyzip. Both have their advantages and disadvantages, and I'll leave it to you to choose a compression algorithm that fits your purpose. I will be using rubyzip in the following script.

The Code

```
require 'zip/zip'

❶ unless ARGV[0]
    puts "Usage: ruby compress.rb <filename.ext>"
    puts "Example: ruby compress.rb myfile.exe"
    exit
end

file = ARGV[0].chomp

❷ if File.exists?(file)
    print "Enter zip filename:"
    zip = "#{gets.chomp}.zip"
❸   Zip::ZipFile.open(zip, true) do |zipfile|
❹       begin
            puts "#{file} is being added to the archive."
❺       zipfile.add(file,file)
❻       rescue Exception => e
            puts "Error adding to zipfile: \n #{e}"
        end
    end
else
    puts "\nFile could not be found."
end
```

Running the Code

This script allows users to compress different file types, either to save space or for easy archiving. Call the script with the following command:

```
ruby compress.rb /path/to/file
```

The Results

The script will create a compressed archive of the file specified on the command line using the name the user provides at the prompt. For this example, I compressed *chapter1.odt* into *nostarch.zip*. Before compression, *chapter1.odt* was 29.1KB, and after the compression, it was 26.3KB. The file will be stored in the same directory as the script is executed.

How It Works

When the script is run, the first error handling check is made to ensure the user has provided a file to compress ❶. If a filename has been provided, the file is checked for availability. As always, there is no sense in continuing if the object we want to manipulate is not available. If the file doesn't exist, the script alerts the user and promptly exits ❷. If the file does exist and has been validated, the user is asked to name the Zip file. After the user types the filename

and presses ENTER, the script continues. That press of the ENTER key is added to the character stream and, in turn, sent to the script as user input. You'll note that `chomp` is used to remove the `\n` (*newline character*) that is added when the user strikes ENTER.

The code used to compress the file is straightforward. As seen above on line ❸, the section will open an existing Zip file if it is available and the second parameter is set to `true`. A new Zip file will be created if the file doesn't already exist. These options are similar to the `open` method in the File library.

Sometimes errors happen. The most vulnerable spot in this script is during the compression while adding files to the Zip file. The `begin/rescue` block at ❹ is used to handle unforeseen errors and provide information to the user about any issues. If an error does occur, the `rescue` block will be executed and the script will exit, displaying the error message ❻.

Each file that is being added to the Zip file is saved using the `add` method ❺. You can create directories in the Zip file from this section or write entirely new files on the fly. Basically, the Zip filesystem can be treated like any normal directory on your computer. The syntax is a little different, but the results are the same.

The rubyzip library is wonderful because you can open the Zip file and manipulate the contents without having to decompress the entire archive. Also, instead of grouping files and then compressing them, as tar and gz do, rubyzip will do all of this with just one command.

#8 File Decompression

decompress.rb

This script shows you the basics of decompressing a file. The rubyzip library does all of the work for you. On a standard Unix-like system, you would have to manually unzip the file, carry out your task, and then re-compress the file. With rubyzip, you can work with files in an archive using one seamless library. This script completely decompresses an archive into the user-specified directory.

The Code

```
require 'zip/zip'
require 'fileutils'

unless ARGV[0]
    puts "Usage: ruby decompress.rb <zipfilename.zip>"
    puts "Example: ruby decompress.rb myfile.zip"
    exit
end

archive = ARGV[0].chomp

❶ if File.exists?(archive)
      print "Enter path to save files to (\'.\' for same directory): "
❷     extract_dir = gets.chomp
```

```
❸   begin
❹       Zip::ZipFile::open(archive) do |zipfile|
            zipfile.each do |f|
❺               path = File.join(extract_dir, f.name)
                FileUtils.mkdir_p(File.dirname(path))
                zipfile.extract(f, path)
            end
        end
❻   rescue Exception => e
        puts e
    end
else
    puts "An error occurred during decompression: \n #{e}"
end
```

Running the Code

The decompression script is invoked like the compression script, with the file to decompress as the command-line argument.

```
ruby decompress.rb /path/to/compressed/file
```

The Results

All files that were originally put into the Zip file will be decompressed in the same structure they had before compression. For this example, I decompressed *nostarch.zip* into *chapter1.odt*. The compressed Zip file *chapter1.odt* was 26.3KB, and after decompression, the file went back to the original 29.1KB.

How It Works

Similar to the compression script, this script expects the zipped file to be provided as a command-line argument. If the archive file cannot be located, the script will present the user with an error message ❶. The major difference between the scripts is that, instead of asking for the name of the Zip file to be created, the decompression script requests the target path where the unzipped files should go ❷.

The next step is the start of a `begin`/`rescue` block ❸. As with the compression script, the decompression is a vulnerable section of code. The first part of decompression is to open the zipped file ❹. After that, each file is decompressed. The decompression routine recreates the directory structure as it was before compression ❺. So, if there were two subfolders before compression, there will also be two folders after this script has completed. As long as no errors are encountered, the script will output each file into the directory specified by the user. The last part of the script is the `rescue` block, which will catch and report any errors that occur during decompression ❻.

#9 Mortgage Calculator

mortgageCalc.rb

I recently began house shopping. Being a first-time home buyer, the task seemed daunting, especially when I considered the financing options. So, I decided to write a script to help me get a handle on mortgage rates—at least this way, I could estimate my monthly payments. Even though Ruby didn't solve all of the issues related to buying a home, this script helped me get a handle on my financing options.

The Code

```
print "Enter Loan amount: "
loan = gets.chomp.to_i
print "Enter length of time in months: "
time = gets.chomp.to_i
print "Enter interest rate: "
rate = gets.chomp.to_f/100

❶ i = (1+rate/12)**(12/12)-1
❷ annuity = (1-(1/(1+i))**time)/i

❸ payment = loan/annuity

❹ puts "\n$%.2f per month" % [payment]
```

Running the Code

This script is interactive and therefore runs without any parameters. It walks the user through each piece of information needed to come up with the correct monthly payment. No command-line arguments are needed.

The Results

```
Enter Loan amount: 250000
Enter length of time in months: 360
Enter interest rate: 6.5

$1580.17 per month
```

How It Works

Mortgage calculations always seemed a bit cryptic to me, and I thought I needed a wall of degrees to understand the formulas. Thankfully, calculating a mortgage payment isn't like solving differential equations! It's quite a bit easier, once you understand the basic formulas. The calculations of a mortgage payment are broken down into two main formulas (that can be combined

into one formula, if you are feeling especially daring). The first calculates the *interest rate* per month using the following equation ❶ (don't forget that ** is the Ruby way of expressing exponentiation):

```
i = (1+rate/12)**(12/12)-1
```

The next piece of information that we need is the annuity factor ❷. Basically, the *annuity factor* is the current value of $1 for each period of time. The *time* is received in months. So, the calculations are:

```
annuity = (1-(1/(1+i))**time)/i
```

Now that the annuity factor has been computed, monthly payments are really what we are after. A simple division of the loan by the annuity factor will reveal the final answer ❸. All that's left is some formatting to make the information easier to read. As with other programming languages, Ruby gives programmers the ability to specify how output should be formatted. In this case, for monetary values, I am interested in two decimal places for the cents in addition to the *integer*, or whole dollar, value ❹.

Hacking the Script

One way to hack this script would be to give a variance of interest rates or loan amounts, so the output could display several possible monthly payments instead of just one—maybe +/−0.05 percent. Usually, when you are looking for a mortgage, you compare a lot of financial information. The more information you can present in one interface, the better the decision you can make.