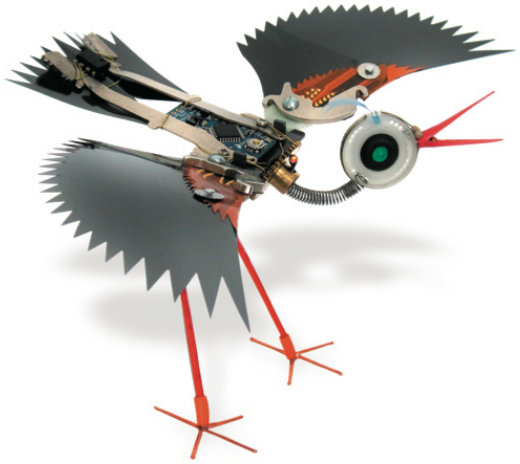

practical computing for biologists

Steven H. D. Haddock
The Monterey Bay Aquarium Research Institute, and University of California, Santa Cruz

Casey W. Dunn
Department of Ecology and Evolutionary Biology, Brown University

ISBN 978-0-87893-391-4

Sinauer Associates, Inc. • Publishers
Sunderland, Massachusetts U.S.A.

http://practicalcomputing.org/

Available at Sinauer and Amazon.

Appendix 2

REGULAR EXPRESSION SEARCH TERMS

Regular expressions—ways to perform adaptive searches and replacements—are described in Chapters 2 and 3. Here we provide a quick reference to some of the more common regular expression terms. This table and the text of the book itself do not encompass the entire range of regular expressions. There are many other useful constructs, for example, embedding miniature scripts into your replacement terms, and searching for A or B in a string using the syntax `(sword|jelly)fish`. If you would like to delve deeper, there are many online references, and there is even an in-depth reference guide built into the Help menu of TextWrangler.

There is some variation in the terms supported from program to program and from language to language. The most widespread terms, which can be used almost anywhere that regular expressions are supported, are the POSIX Extended Regular Expressions. These include `.`, `*`, `+`, `{}`, `()`, `[]`, `[^]`, `^`, `$`, `?`, and `|`. While quite a bit can be accomplished with the POSIX terms, in many implementations the language has been supplemented with some nonstandard terms. Most of these nonstandard terms are based on Perl regular expressions. These include many of the character class wildcards listed in the tables below, such as `\d`, `\w`, and `\n`. These extra wildcards make it easier to write clear regular expressions. Lack of support for Perl-like regular expressions is one of the most common causes of confusion when moving to a new programming context.

If you are using regular expressions in a new context but find that they don't behave as expected, or that they generate errors, check to see which regular expressions are supported by the tool you are using. POSIX does define its own set of wildcards, but the syntax is different from the Perl-style `\w` format that we use in this book. These wildcards include `[:digit:]` in place of `\d` and `[:alpha:]` instead of `\w` that we use in this book (though not including the digits). These POSIX character classes can be used in some contexts where Perl classes aren't available, including SQL queries and the command-line tool `grep`. If you don't want to switch between wildcard types, a more universal solution is to replace character class wildcards with an explicit character range, such as `[0-9]` or `[A-Z]`.


Wildcards	
`\w`	Letters, numbers and _
`.`	Any character except `\n \r`
`\d`	Numerical digits
`\t`	Tab
`\r`	Return character. Also used as the generic end-of-line character in TextWrangler
`\n`	Line-feed character. Also used as the generic end-of-line character in Notepad++
`\s`	Space, tab, or end of line
`[A-Z]`	A single character of the ranges indicated in square brackets
`[^A-Z]`	A single character including all characters *not* in the brackets. Note that this will include `\n` unless otherwise specified, and may cause you to match across lines
`\`	Used to escape punctuation characters so they are searched for as themselves, not interpreted as wildcards or special symbols
`\\`	The \ symbol itself, escaped
Boundaries	
`^`	Match the start of the line, i.e., the position before the first character
`$`	Match the last position before the end-of-line character

Quantifiers, used in combination with characters and wildcards	
`+`	Look for the longest possible match of one or more occurrences of the character, wildcard, or bracketed character range immediately preceding. The match will extend as far as it can while still allowing the entire expression to match.
`*`	As above, matches as many of the previous character to occur, but allows for the character not to occur at all if the match still succeeds
`?`	Modifies greediness of + or * to match the shortest possible match instead of longest
`{}`	Specify a range of numbers to repeat the match of the previous character. For example: `\d{2,4}` matches between 2 and 4 digits in a row `[AC]{4,}` matches 4 or more of the letter A or C in a row
Capturing and replacing	
`()`	Capture the search results between the parentheses for use in the replacement term
`\1` `$1`	Substitute the contents of the matched into the replacement term, in numerical order. Syntax depends on the text editor or language that you are using.

Appendix 3

SHELL COMMANDS

Terminal operations are described in Chapters 4–6, 16, and 20. Many of the built-in `bash` shell commands are summarized here for quick reference. To get more information about a command and its options, type `man`, followed by the name of the command. If you are not sure which command applies, you can also search the contents of the help files using `man -k` followed by a keyword term.

Command	Description	Usage
`ls`	List the files in a directory	`ls -la`
	Parameters that follow can be folder names (use `*` as a wildcard)	`ls -1 *.txt`
		`ls -FG scripts`
		`ls ~/Documents`
	`-a` Show hidden files	`ls /etc`
	`-l` Show dates and permissions	
	`-1` List the file names on separate lines. Useful as a starting point for regexp into a list of commands	
	`-G` Enable color-coding of file types	
	`-F` Show a slash after directory names	
`cd`	Change directory	
	Without a slash, names are relative to the current directory	`cd scripts`
	With a preceding slash (/) names start at the root level	`cd /User`
	Tilde (~/) starts at the user's home directory	`cd ~/scripts`
	Two dots (`..`) goes "up" to the enclosing directory	`cd My\ Documents`
	One dot refers to the current directory	`cd 'My Documents'`
	Minus sign goes to the previously occupied directory	`cd ../..`
	Use tab key (see below) to auto-complete partially typed paths	`cd ..`
		`cd -`
	Use backslash before spaces or strange characters in the directory name, or put the whole name in quotes	

Command	Description	Usage
`pwd`	Print the working directory (the path to the folder you are in)	
↑	↑ key to step back through previously typed commands The cursor can be repositioned with the ← and → keys, and commands can then be edited Press *return* from anywhere in the line to re-execute. On OS X you can also reposition by *option*-clicking at a cursor location	
tab	Auto-complete file, folder, or script names at the command line	`cd ~/Doc` *tab*

`less`	Show contents of a file, page by page
	These commands also apply to viewing the results of `man`
	While `less` is running:
	`q` Quit viewing
	space Next page
	`b` Back a page
	`15 g` Go to line 15
	`G` Go to the end
	↑ or ↓ Move up or down a line
	`/abc` Search file for text `abc`
	`n` After an initial search, find next occurrence of the search item
	`?` Find previous occurrence of the search item
	`h` Show help for `less`
		`less data.txt`

`mkdir`	Make a new directory (a new folder)	`mkdir scripts`
`rmdir`	Remove a directory (folder must be empty)	`rmdir ~/scripts`
`rm`	Remove file or files Use the `-f` flag to delete without confirmation (careful!) Use the `-r` flag to recursively delete the files in a directory and then the directory itself	`rm test.txt` `rm -f *_temp.dat`

`man`	Show the manual pages for a Unix command
	Use `-k` to search for a term within all the manuals
	The result is displayed using the `less` command above, so the same shortcuts allow you to navigate through
		`man mkdir`
		`man -k date`
		`man chmod`

Command	Description	Usage

`cp`	Copy file, leaving original intact
	Does not work on folders themselves
	Single period as destination copies file to current directory, using same name
		`cp test1.txt test1.dat`
		`cp temp ../temp`
		`cp ../test.py .`

`mv`	Move file or folder, renaming or relocating it Unlike cp, this does work on directories	`mv test1.txt test1.dat` `mv temp ../temp2`
`|`	Pipe output of one command to the input of another command	`history | grep lucy`
`>`	Send output of a command to a file, overwriting existing files Do not use a destination file that matches a wildcard on the left side	`ls -1 *.py > files.txt`
`>>`	Send output of a command to a file, appending to existing files	`echo "#Last line" >> data.txt`
`<`	Send contents of a file into command that supports its contents as input	`mysql -u root midwater < data.sql`
`./`	Represents the current directory in a path—the same location as `pwd` Trailing slash is optional Can execute a file in the current directory even when the file directory is not included in the `PATH`	`cp ../*.txt ./` `./myscript.py`
`cat`	Concatenate (join together) files without any breaks. Streams the contents of the file list across the screen	`cat README` `cat *.fta > fasta.txt`
`head`	Show the first lines of a file or command Use the `-n` flag to specify the number of lines	`head -n 3 *.fasta` `ls *.txt | head`
`tail`	Show the last lines of a file or output stream Use the `-n` flag to specify the number of lines to show With a plus sign, skip that number of lines and show to the end. Use `-n +2` to show from the second line of the file to the end, skipping one header line	`tail -n 20 *.fta` `tail -n +3 data.txt`
`wc`	Count lines, words, and characters in an output stream or file	`wc data.txt` `ls *.txt | wc`
`which`	Show the location of executable files in the system path	`which man`

Command	Description	Usage

`grep`	Search for phrase in a list of files or pipe and show matching lines:
	`grep -E "searchterm" filelist`
	Often used in conjunction with piped output: `command | grep searchterm`
	Use quotes around search terms, especially spaces or punctuation like >, &, #, and others
	To search for tab characters, type ctrl V followed by tab inside the quotes
	Optional flags:
	`-c` Show only a count of the results in the file
	`-v` Invert the search and show only lines that do not match
	`-i` Match without regard to case
	`-E` Use full regular expressions
	Terms should be enclosed in quotes. Use `[]` to indicate a character range rather than the wildcards of Chapters 2 and 3
	General wildcard equivalents:
	`\s [[:space:]]`
	`\w [[:alpha:]]`
	`\d [[:digit:]]`
	`-l` List only the filenames containing matches
	`-n` Show the line numbers of the match
	`-h` Hide the filenames in the output	

`agrep`	Search for approximate matches, allowing insertions, deletions, or mismatched characters. (Must be installed separately.) See Chapter 21 Optional flags include: `-d ","` Use comma as delimiter between records `-2` Return results with up to 2 mismatches. Maximum is 8 mismatches `-B -y` Return the best match without specifying a number of mismatches `-l` Only list file names containing matches `-i` Match without regard to case	`agrep -d "\>" -B -y ATG seqs.fta` `agrep -3 siphonafore taxa.txt`

`chmod`	Change access permissions on a file (usually to make a script executable or Web accessible)
	First option is one of `u`, `g`, `o` for user, group, other
	Second option after the plus or minus is `r`, `w`, or `x`, for read, write, or execute. Can also use binary encoding as explained in Appendix 6
		`chmod u+x file.pl`
		`chmod 644 myfile.txt`
		`chmod 755 myscript.py`

Command	Description	Usage
`set`	Show environmental variables, including functions that have been defined	
`$HOME`	The environmental variable containing the path user's home directory	`echo $HOME` `cd $HOME`
`$PATH`	The user's `PATH` variable, where the directories to search for commands are stored	`export PATH=$PATH:/usr/local/bin`
`nano`	Invoke the text editor. Control key sequences include: ctrl X Exit `nano` (will be prompted to save) ctrl O Save file without exiting ctrl Y Scroll up a page ctrl V Scroll down a page ctrl C Cancel operation ctrl G Show help and list of commands	`nano filename.txt`
ctrl C	Interrupt the current process	

`sort`	Sort lines of a file
	`-k N` Sort using column number *N* instead of starting at the first character. Columns are delimited by a series of white space characters
	`-t ","` In conjunction with –k, use commas as the delimiter to define columns
	`-n` Sort by numerical value instead of alphabetical
	`-r` Sort in reverse order
	`-u` Return only one unique representative from a series of identical sorted lines
		`sort -k 3 data.txt`
		`sort -k 2 -t "," F1.csv`
		`sort -nr numbers.txt`
		`sort A.txt > A_sort.txt`

`uniq`	Return a single line for each consecutive instance of that line in a file or output stream. To remove all duplicates from anywhere in the file, it must be sorted before being piped to the `uniq` command Use `-c` flag to return a count along with the repeated element	`uniq -c records.txt` `sort names | uniq -c`

Command	Description	Usage

`cut`	Extract one or more columns of data from a file
	`-f 1,3` Return columns 1 and 3, delimited by tabs
	`-d ","` Use commas as the field delimiter instead of tabs. Used in combination with `-f`
	`-c 3-8` Return characters 3 through 8 from the file or stream of data
		`cut -c 5-15 data.txt`
		`cut -f 1,6 data.csv`
		`cut -f2 -d ":" > Hr.txt`

`curl`	Retrieve the contents of a URL from over the network. URL should be placed in quotes. Without additional parameters, will stream contents to the screen For some Linux versions, `wget` offers similar functionality See `man curl` for ways to send user login information at the same time `-o` Set the name of the output file to save individual files for the data. See `#1` below `-m 30` Set a time out of 30 seconds `[01-25]` In the URL, substitute two digit numbers from 01 to 25 into the address in succession `{22,33}` `{A,C,E}` Substitute items in brackets into URL `#1` The substituted value, for use in generating the filename	`curl "www.myloc.edu" > myloc.html` `curl "http://www.nasa.gov/weather[01-12]{1999,2000}" -m 30 -o weather#1_#2.dat`
`sudo`	Run the command that follows as a superuser with privileges to write to system files	`sudo python setup.py install` `sudo nano /etc/hosts`
`alias`	Define a shortcut for use at the command line. To make persistent, add to startup settings file `.bash_profile` or equivalent	`alias cx='chmod u+x'`

`function`	Create a shell function—like a small script
	`$1` is the first user argument supplied after the command is typed
	`$@` is all the parameters—useful for loops as below
	Variable names are defined with the format `NAME=` with no spaces. They are retrieved with `$NAME`
	Save it in `.bash_profile` to make it permanent
		`myfunction() {`
		`  # insert commands here`
		`  echo $1`
		`}`

`;`	In a command or script, equivalent to pressing return and starting a new line	`date; ls`

Command	Description	Usage

`for`	Perform a `for` loop in the shell. Can be useful in the context of a function	`for ITEM in *.txt; do`
		`  echo $ITEM`
		`done`

`if`	An `if` statement in a shell function:
	`if [ test condition ]`
	`then`
	`  # insert commands`
	`else`
	`  # alternate command`
	`fi`
	Comparison operators are `eq` for equals, `lt` for less than and `gt` for greater than
		`if [ $# -lt 1 ]`
		`then`
		`  echo "Less than"`
		`else`
		`  echo "greater than 1"`
		`fi`

`` ` ` ``	Backtick symbols surrounding a command cause the command to be executed and then substitute the output into that place in the shell command or script	`cd `which python`/..` `nano `which script.py``
`host`	Return IP number associated with a hostname, or the hostname associated with an IP address, if available	`host www.sinauer.com` `host 127.0.0.1`
`ssh`	Start a secure remote shell connection	`ssh lucy@pcfb.org`
`scp`	Securely copy files to or from a remote location	`scp localfile user@host/path/remotefile` `scp user@host/home/file.txt localfile.txt`
`sftp`	Start a file transfer connection to a remote site. The prompt changes to an ftp prompt, at which the following commands can be used: `open` From the prompt, open a new `sftp` connection `get` Bring a remote file to the local server `put` Place a local file on the remote system `cd` Change directory on the remote server `lcd` Change directory on the local machine `quit` Exit the `sftp` connection	`sftp user@remotemachine`

`gzip`
`gunzip`
`zip`
`unzip`
	Compress and uncompress files	`gzip files.tar`
		`gunzip files.tar.gz`
		`unzip archive.zip`

`tar`	Create or expand an archive containing files or folders `-cf` Create `-xvf` Expand `-xvfz` Expand and uncompress `gzip`	`tar -cf archive.tar ~/scripts` `tar -xvfz arch.tar.gz`

Command	Description	Usage
`&`	When placed at the end of a command, runs it in the background	
`ps`	Show currently running processes. Flags controlling the output vary greatly by system. Usually a good starting point is `-ax`. See `man ps` for more	`ps -ax | grep lucy`
`top`	Show current processes sorted by various parameters, most useful of which is processor usage `-u`	`top -u`
`kill -9`	Terminate a process emphatically, using its process ID. Retrieve PID from the `ps` or `top` command	`kill -9 5567`
`killall`	Terminate processes by name	`killall Firefox`
`nohup`	Run command in background and don't terminate it when logging out or closing the shell window Use in this odd format shown, to prevent program output to cause the command to quit	`nohup command 2> /dev/null < /dev/null &`
ctrl Z	Suspend the operation to move it into the background or perform other operations	
`jobs`	Show backgrounded or suspended jobs, won't show normal active processes	
`bg`	Move a suspended process into the background. Optional number after it in the format `%1` will specify the job number	

`apt-get`
`yum`
`rpm`
`port`
	Package installers for various Unix distributions. Search for and install remote software packages. Typically used with `sudo`	`sudo apt-get install agrep`
		`yum search imagemagick`

Appendix 4

PYTHON QUICK REFERENCE

Conventions for this appendix

In the examples below, italicized terms are not real variable or function names, but are stand-ins for an actual name. If a function name is shown as `.function()` then the dot means it is used as a method, coming after the variable name, as in `MyString.upper()`.

Format, syntax, and punctuation in Python

- Indented lines define blocks of statements that are executed in loops, decisions, and functions.
- Comments are marked by # and extend from that symbol to the end of the line. Multi-line comments can be bracketed on both sides by three quote marks.
- To continue a statement on the next line, use the \ character at the end of a line.
- Parentheses `()` pass parameters to functions.[1]
- Square brackets [ ] define lists and retrieve subsets of values from strings, lists, dictionaries, and other types.
- Curly brackets `{}` define dictionary entries.

Python scripts begin with the shebang line, and can include an optional line to enable support of Unicode characters:

```
#! /usr/bin/env python
# coding: utf-8
```

[1] They also are used to define tuples, non-changeable list-like variables that we don't address in this book.

The command-line interpreter

Start by typing `python` at the command line. Cycle up through history of previous Python commands using ↑. Use `quit()` or ctrl D to exit (ctrl Z in Windows).

You should be able to paste entire programs into the interpreter, but sometimes the indented block of a loop or conditional statement might not be carried over properly. Pasting commands at the Python prompt also does not work well for things involving user input or reading and writing files. In addition, the buffer of your terminal program may not keep up with large pasted blocks, resulting in errors on the text pasted.

Command summary

Variable types and statistics

Changing variable types and getting information	
Convert numbers and other types to strings This conversion is required for the `.write()` function used with a file or the `sys.stderr.write()` function	`str()`
Convert integers or strings to floating point	`float()`

Can specify the base in alternate base systems. To specify the number in hex, use `int(MyString,16)`	`int(3.14)`
	`int("3")`
	`int("4F",16)`

Give the length of a string, list, or dictionary	`len("ABCD")`
	`len([1,2,4,8])`
	`len(Diction)`

Strings

Defining and formatting strings	

Strings are defined by pairs of single (') or double (") quotation marks, not curly quotes (“”)	`Location = "Hawai'i"`
	`Region = "3'-polyA"`
	`Genus = 'Gymnopraia'`

Multi-line strings are defined by three quote marks in a row	`MultiString = """`
	`Triple-quoted strings`
	`can span several lines.`
	`They also act like comments`
	`"""`

Convert from number to string	`str(100.5)`
Find the ASCII code for a string character with `ord()`	`ord('A')`

Manipulating strings	
Change case with `.upper()` and `.lower()`	`MyString.upper()` `MyString.lower()`
Join two strings with +	`MyString + YourString` `'Value' + str(MyValue) + '\n'`
Repeat a string with *	`print '='*30` `==============================`
Literal substitution (not using wildcards or regular expressions) with `.replace()`	`MyString.replace('jellyfish','medusa')`
Count occurrences of `'A'` in `MyString` with `.count()`	`MyString.count('A')`
Remove all white space from rightmost end of string with `.rstrip()` Remove only linefeeds, not tabs	`MyString.rstrip()` `MyString.rstrip('\n')`
Strip all white space from both sides of string with `.strip()`	`MyString.strip()`

See *Working with lists* in this appendix for converting strings or characters to lists and *Searching with regular expressions*, also in this appendix, for advanced search and replace techniques.

Gathering user input

Get user input during execution of program	`raw_input("Enter a value:")`
Get space-separated parameters given when program is run at the command line. You can pass parameters with wildcards, like `dive*.csv`	`import sys` `sys.argv`
The script or program name, using the zeroth parameter	`sys.argv[0]`
All subsequent command-line arguments	`sys.argv[1:]`
Determine how many command-line parameters were provided, via the `len()` function	`if len(sys.argv) > 1:`

Building strings

Printing strings	
Print variables separated by a space	`print MyString, MyNumber`
Print variables not separated by space	`print MyString + str(MyNumber)`

Generating strings with the formatting operator, `%`:

```
MyString = '%s %.2f %d' % ("Value",4.1666,256)
           ↳ Substitution points  ↳ Values to insert
```

This creates the string: `'Value 4.17 256'`

Given the string `s = '%x' % (4.13)` where `%x` is a placeholder listed below:		
Placeholder	**Type**	**Result**
`%s`	String variable	`'four'`
`%d`	Integer digits	`'4'`
`%5d`	Integer padded to at least five spaces	`'    4'`
`%f`	Floating point	`'4.130000'`
`%.2f`	Float with precision of two decimal points	`'4.13'`
`%5.1f`	Float with one decimal, padded to at least five total spaces (includes decimal point)	`'  4.1'`

Comparisons and logical operators

Comparison operators[a]	
Comparison	**Is `True` if...**
`x == y`	x is equal to y
`x != y`	x is not equal y
`x > y`	x is greater than y
`x < y`	x is less than y
`x >= y`	x is greater than or equal to y
`x <= y`	x is less than or equal to y

[a]These operators return `True` (`1`) or `False` (`0`) based on the result of the comparison.

Logical operators[a]	
Logical operator	**Is `True` if...**
`A and B`	Both `A` and `B` are `True`
`A or B`	Either `A` or `B` is `True`
`not B`	`B` is `False` (inverts the value of `B`)
`(not A) or B`	`A` is `False` or `B` is `True`
`not (A or B)`	`A` and `B` are both `False`

[a]In this table, `A` and `B` represent a `True`/`False` comparison like those listed in the previous table.

Note that in Python, when an expression involving logical operators is found to be true, the value returned is that of the first true item being tested, not `True` itself.

```
>>> 1 and 2
2
>>> 3 or 4
3
```

Math operators

Normal order of precedence applies. Operations involving only integers produce only integers, even at the expense of accuracy.

Addition	`+`
Subtraction	`-`
Multiplication	`*`
Division	`/`
Modulo (remainder after division)	`%` `7 % 2 → 1`
Power	`**` `2**8=256`
Truncated division (result without remainder)	`//` `7//2.0 = 3.0`
Increment a variable by a value	`+=` `X += 2`

Decisions

The `if`, `elif`, and `else` commands control the flow of a program according to logical tests. Statements built on these commands end with a colon. Below is a description of each, with example code on the right.

```
if logical1:
  # do indented lines
  # if logical1 is True

elif logical2:
  # if logical1 is False
  # and logical2 is True

else:
  # do if all tests
  # above are False
```

```
A=5
if A < 0:
     print "Negative number"

elif A > 0:
     print "Zero or positive number"

else:
     print "Zero"
```

Loops

`For` and `while` loop definitions end with a colon. Use `for` loops to step through ranges and lists. Below are a series of loop examples, with code shown on the right.

`for` loop using `range()`

```
for Num in range(10):
   print Num * 10
```

`for` loop with a list

```
for Item in MyList:
   print Item
```

`for` loop with a string

```
for Letter in "FEDCBA":
    print Letter
```

`while` loop

```
X=0
while X < 11:
    print X
    X = X + 2
```


Searching with regular expressions

Regexp to find matching subsets in a string

Use regexp within your program to extract and substitute portions of a string. The basic format is:

```
Results = re.search(query,string)
```

The query is a text string containing the regular expressions pattern that you would enter into a Find dialog box.

Import the module	`import re`
Define a search query, using raw string	`MyRe = r"(\w)(\w+)"`
String to search	`MyString = "Agalma elegans"`
Search and save matches	`MyResult = re.search(MyRe, MyString)`
All the matches together	`MyResult.group(0)`
The first captured match	`MyResult.group(1)`
All matches as separate items	`MyResult.groups()`

Regexp to substitute into a string

The basic format is:

```
re.sub(query, replacement, string)
```

When used in a program, this is the same as a Replace All command for that string.

Import the module	`import re`
Define a search query, using a raw string	`MyRe = r"(\w)(\w+) (.*)"`
Define the replacement term, using `\1`, `\2`, etc., to represent entities captured with parentheses	`MySub = r"\1. \3"`
String to search	`MyString = "Agalma elegans"`
Search and save matches	`NewString = re.sub(MyRe, MySub, MyString)`
The result saved in `NewString`	`"A. elegans"`

Working with lists

Lists are ordered collections of objects. Items in a list can be of any type, including other lists and heterogeneous mixes of variable types. The first element has an index of 0; so, for example, a list with five members does not have an item at index 5.

Creating lists	
Create a list from string or other variable type If the variable is a string, the list elements will be each character of the string	`list(MyString)`
Define with square brackets	`MyList = [1,2,3]` `OtherList = [[2,4,6],[3,5,7]]`
Define an empty list; required before the list can be appended to	`MyList=[]`

Define numerical lists with the `range()` function
The left element is included in the retrieval, the right index is not
Given one parameter, `range(N)` creates `N` elements, from `0` to `N-1`. A third parameter optionally sets the step size between elements, positive or negative
	Function **Result**
	`range(5)` `[0, 1, 2, 3, 4]`
	`range(1,8,2)` `[1, 3, 5, 7]`
	`range(5,0,-1)` `[5, 4, 3, 2, 1]`

Parse strings into lists with `.split()` Default delimiter is any amount of white space, or specify delimiter character in the `()`	`MyList = MyString.split()`
Add elements with `.append()`	`MyList.append(10)`

Insert elements with a single index repeated on both sides of the colon	`MyList=range(5)`
	`MyList[3:3]=[9,8,7]`
	`>>> MyList`
	`[0, 1, 2, 9, 8, 7, 3, 4]`

Delete elements from list with `del` Assign `=[]` to delete indexed elements	`del MyList[2:5]` `MyList[2:5]=[]`

Accessing list elements	
Extract elements with `[]` Index range: Start element is retrieved, finish element is not Indices can count from either the beginning, or, using negative numbers, the end of the list	`MyList[Start:Finish]` `MyList[begin:end+1:step]`
Skip first element of a list	`MyList[1:]`
All but last element	`MyList[:-1]`
Return list elements in reverse order, leaving the original list unchanged Sort list in place, modify original	`MyList[::-1]` `MyList.reverse()`

Extract even or odd elements	`MyList=range(8)`
	`MyList [1::2]`
	`[1, 3, 5, 7]`
	`MyList[0::2]`
	`[0, 2, 4, 6]`

Unpacking two or more values at once	`a,b=MyList[0,1]`

List information and conversions	

Convert lists of strings to strings with `.join()`
The `.join()` method works a bit backwards, acting on the character used to join, with the list as a parameter
	`''.join(MyList)`
	`MyList = ['A', 'B', 'C', 'D']`
	`print '-'.join(MyList)`
	`A-B-C-D`

Test if an item is in a list with the `in` operator	`print 'A' in MyList` `True`

Create a list of unique elements of a list with `set()`	`MyList=list('aabbbcdaa')`
	`print list(set(Mylist))`
	`['a','b','c','d']`

Sort lists
Return a sorted list, leaving original list unaltered
Sort in place, modifying original list
	`NewList=MyList.sorted()`
	`MyList.sort()`
	`Keys=Diction.keys()`
	`Keys.sort()`

Retrieve elements and their indices together, using `enumerate()`	`Ind, Elem = enumerate(MyList)`

List comprehension

Performs an operation on each item in a list, and returns a list of the results. List comprehensions are very useful for manipulating lists in Python.

```
Squares = [Val**2 for Val in MyList]
Strings = [str(Val) for Val in MyList]
```

Dictionaries

Dictionaries are somewhat like lists, except that instead of values being accessed by sequential numerical keys (indexes), they are accessed by non-sequential keys defined as you wish. Keys and values can be of many types, including numbers, strings, or lists, and they can occur together in one dictionary. Only one instance of a key is allowed in a dictionary, but values can occur repeatedly; that is, it is keys that are required to be unique, not values. Dictionaries have no intrinsic order to their contents, and values are returned only by key, not by position or order of entry.

Defining dictionaries	

Define entries within curly brackets with the format `{key: value}`
Key–value pairs are separated by commas
Between the brackets, the definition can span several lines and indentation is not important
	`Diction = {1:'a', 2:'b'}`
	`Diction={`
	`'Lilyopsis' :3, 'Resomia'  :2,`
	`'Rhizophysa':1, 'Gymnopraia':3 }`

A list of keys and a list of values having the same number of elements can be zipped together to form a dictionary	`SiphKeys = ['Lilyopsis','Rhizophysa',`
	`  'Resomia','Gymnopraia']`
	`SiphVals = [3,1,2,3]`
	`Diction = dict(zip(SiphKeys,SiphVals))`

Add entries using indexed values with square brackets Requires a pre-existing dictionary, which can have no entries	`Diction={}` `Diction['Marrus'] = 2`
Delete dictionary entries with `del` The method used to clear list elements by assigning to `[]` does not work with dictionaries. The key will still exist	`del Diction['Marrus']`

Extracting values from a dictionary	
Index with square brackets `[]` and the key	`print Diction['Resomia']` `2`
If the key is not present, results in an error	`print Diction ['Erenna']` `...KeyError: 'Erenna'`

Retrieve with `.get()`
Optionally, provide a value to return if the key is not present
	`print Diction.get('Resomia')`
	`2`
	`print Diction.get('Erenna',-99)`
	`-99`

Information about a dictionary	

Get a list of keys or values with `.keys()` and `.values()`, but not in any predictable order
The order, however, will be internally consistent between the two lists
	`Diction.keys()`
	`['Resomia','Lilyopsis',`
	`'Gymnopraia','Rhizophysa' ]`
	`Diction.values()`
	`[2, 3, 3, 1]`

Number of entries in a dictionary	`len(Diction)`

Creating functions

Define the function in the program before it is used, or in an external file which is imported. Functions can be generated with or without additional parameters, and parameters can be assigned default values.

```
def function_name(Parameter = Defaultvalue):
    # insert statements that calculate values
    return Result   # send back the result
```

Call the function from within the program, passing values in parentheses:

```
MyValue = function_name(200)
```

Working with files

Reading from a file	
Open the connection to the file	`InFile = open(FileName, 'rU')`
Read lines in succession	`for Line in InFile:` `  # perform operation on Lines`
Alternatively, read all lines into a list at once. (This can't be used after the command above since `InFile` is already at the end of the file)	`AllLines = InFile.readlines()`
Close the file connection	`InFile.close()`

An example of a short file-reading program in action:

```
FileName="/Users/lucy/pcfb/examples/FPexcerpt.fta"
InFile = open(FileName, 'rU')
for Line in InFile:
    MyLine = Line.strip()
    if MyLine[0]==">":
        print MyLine[1:]
InFile.close()
```

Getting information about files	
Use the os module	`import os`
Check if string is path to a file; fails if it is not found or if it is a folder rather than a file	`os.path.isfile('/Users/lucy/pcfb/')`

Check if a folder or file exists
 	
Fails with ~/ as part of path
	`os.path.exists('/Users/lucy/pcfb/')`
	`True`
	`os.path.exists('~/pcfb/')`
	`False`

Get a list of files matching the parameter, using * as a wildcard	`import glob` `FileList = glob.glob('pcfb/*.txt')`

Writing to a file	
Open file stream, overwriting existing file if it exists	`OutFile = open(FileName, 'w')`
Open file stream, appending to the end of a file if it already exists	`OutFile = open(FileName, 'a')`
Write a string to the specified `OutFile` Line endings are not automatically appended, and numbers must be converted to strings beforehand, using the `str()` function or the format operator `%`	`OutFile.write('Text\n')`
Close the `OutFile` when done writing	`OutFile.close()`

Using modules and functions

First import the module, then call the function, usually followed by parentheses.

Ways to import functions from a module	
Import all the functions and use them thereafter by appending the function name to the module	`import themodule` `themodule.thefunction()`
Import a module, but use a different name for it within the program	`import longmodulename as shortname` `shortname.thefunction()`
Import all the functions from a module, and use them with only the function name	`from themodule import *` `thefunction()`
Import a particular function, and use it with just its name	`from themodule import thefunction` `thefunction()`
To see a list of commands in the module, after importing in the Python interactive environment	`dir(modulename)` `help(modulename)`

To create your own modules, use `def` to define functions as indicated above, place them in their own file, and save with a filename ending in `.py` somewhere in your `PATH`. Import them into your script using the filename without the `.py` extension.

Some built-in modules	
`random`	Random sampling and random number generation
`urllib`	Downloading and interacting with Web resources
`time`	Information related to the current time and elapsed time
`math`	Some basic trigonometric functions and constants
`os`	Items related to the file system
`sys`	System-level commands, such as command-line arguments
`re`	The regular expressions library for search and replace
`datetime`	Date conversion and calculation functions
`xml`	Reading and writing XML files
`csv`	Read in a comma-delimited file using the function `csv.reader()`

Other installable modules	
`MySQLdb`	Interact with a `mysql` database
PySerial	Connect through the serial port to external devices. Use with `import serial`
`matplotlib`	MATLAB-like plotting functionality
`numpy`, `Scipy`	Large package of numerical and statistical capabilities
Biopython	Functions for dealing with molecular sequence files and searches. Use with `import Bio` or `from Bio import Seq`

Miscellaneous Python operations

Presenting warnings and feedback

```
sys.stderr.write()
```

Sends output to screen (but does not send output to a file when a redirect such as >> is used).

Catching errors

Statements indented under a `try:` function will be executed until an error occurs. If there is an error, then the block of code indented under a subsequent `except:` statement will be executed.

Shell operations within Python

```
os.popen("rmdir sandbox")
```

The shell command specified in parentheses is executed. If you want to read the results the command would usually print to the screen, append `.read()`:

```
Contents = os.popen("ls -l").read()
```

For example, `os.popen(pwd)` will try to operate whether or not there is printed feedback.

Reference and getting help

- From the `python` command line, use `dir(item)` to see functions within a variable or imported module. Use `type(item)` to get a simple statement of the variable type.
- Depending on the variable, `help(item)` may give you the information pages related to a function or a variable, showing you information pertinent to its type.
- Consult Web sites such as `diveintopython.org` when stuck.

Appendix 7

SQL COMMANDS

SQL, short for Structured Query Language, is the language used to interact with relational databases, as discussed in Chapter 15. Although our specific examples are drawn from MySQL, learning the basics of SQL can help you work with nearly any database system. MySQL has excellent online references, tutorials, and examples. Many are at the site: `dev.mysql.com/doc/refman/5.1/en/`.

Installing MySQL is described in Chapter 15. The commands listed in the tables below would be entered at the `mysql>` prompt, launched using the command:

See Appendix 1 for installation and launching instructions.

```
mysql -u root
```

If you have assigned a password to the `root` account, the command above should end with `-p`. You can also log in as a user other than `root` if you have configured other users.

Databases are organized into tables containing fields (corresponding to columns), which in turn contain values of related information organized into rows.

Working at the MySQL prompt

Purpose	Example

Entering commands
Commands can span several lines. They are only executed when the line is terminated with a semicolon. Indentation and capitalization are just for readability and are not interpreted
	`SELECT genus FROM specimens`
	`  WHERE vehicle LIKE 'Tib%'`
	`  AND depth > 100`
	`;`

Interrupt a command or cancel a partially typed command. Do not type ctrl C, which will end your entire `mysql` session	`\c` return
Quit MySQL	`EXIT;` `\q` return

Get general help, or help on a command or topic	`HELP`
	`HELP SELECT`
	`HELP LOAD DATA`

Selected MySQL data types

Data type	Description
`INTEGER`	An integer. Also abbreviated as `INT`
`FLOAT`	A floating point number, including scientific notation
`DATE`	A date in `'YYYY-MM-DD'` format
`DATETIME`	A date and time in `'YYYY-MM-DD HH:MM:SS'` format
`TEXT`	A string containing up to 65,535 characters
`TINYTEXT`	A string containing up to 255 characters
`BLOB`	A binary object, including images or other non-text data

Creating databases and tables

Make a new blank database	`CREATE DATABASE databasename;`
Select a database as the target of subsequent commands	`USE databasename;`
Make a new table containing field type definitions	`CREATE TABLE tablename (fieldname1 TYPE, fieldname2 TYPE2);`
Make a new table with an autoincrementing primary key, then other column definitions	`CREATE TABLE tablename (primarykeyname INTEGER NOT NULL AUTO_INCREMENT PRIMARY KEY, nextfield TYPE, anotherfield TYPE);`

Adding data into table fields

Import formatted text data whose columns correspond exactly to predefined table fields	`LOAD DATA LOCAL INFILE 'path/to/infile';`
Add a row of values to a table in the order that matches the predefined fields	`INSERT INTO tablename VALUES (1,"Beroe",5.2,"1865-12-18");`
Redefine values based on another criterion	`UPDATE tablename SET values = x WHERE othervalues = y;`

Database and table information

List the names of the databases or tables	`SHOW DATABASES;` `SHOW TABLES;`
Show name, type, and other information about the fields of a table	`DESCRIBE tablename;`
Show the number of entries in the table	`SELECT COUNT(*) FROM tablename;`

Extracting data from tables with `SELECT`

List all the rows in all columns of a table. The rows retrieved can be refined with `WHERE` statements at the end of the line	`SELECT * FROM tablename;`
Show the values of the listed columns from the table	`SELECT vehicle,date FROM specimens;`
Show the unique values of a named column	`SELECT DISTINCT vehicle FROM specimens;`
Show a count of the values in a named table	`SELECT COUNT(*) FROM specimens;`
Show a count of the values in a named field, clustered by the unique values of that field. Like `SELECT DISTINCT`, but with counts	`SELECT vehicle,COUNT(*) FROM specimens GROUP BY vehicle;`

Qualifying which rows to retrieve using `WHERE`

`WHERE` refines the records (rows) retrieved from a `SELECT` command. Criteria include comparisons like greater than and less than, or comparisons of equality, which can apply to numbers or strings. Use `!=` for not equal	`SELECT vehicle FROM specimens WHERE depth > 500 AND dive < 600 ;`
Find approximate matches, using `%` as a wildcard of any characters	`WHERE vehicle LIKE "Tib%"`
Find matches using regular expressions. Wildcards are not all supported, but beginning and end of line, `. [ ] + ` are supported	`WHERE field REGEXP query` `WHERE vehicle REGEXP "^T"` `WHERE species REGEXP "galma$"`
Combine criteria with logical operators Use parentheses to group logical entities	`SELECT vehicle from specimens WHERE (vehicle LIKE "Ven%") OR (vehicle LIKE "JSL%");`

Mathematical and statistical operators

Basic math operators	`+, –, *, , /`
Basic comparisons	`<, >, =, !=`
Average of the values	`AVG()`
Count of the values	`COUNT()`
Maximum value	`MAX()`
Minimum value	`MIN()`
Standard deviation	`STD()`
Sum of the values	`SUM()`

Deleting entries and tables	
Clear all entries from a table	`DELETE FROM tablename;`
Clear entries matching `WHERE` criteria	`DELETE FROM tablename WHERE vehicle LIKE "Tib%";`
Delete an entire table. Use with caution. Can't undo it	`DROP tablename;`
Saving to a file	
Save the results from a query into a tab-delimited file	`SELECT * FROM midwater INTO OUTFILE '/export.txt' FIELDS TERMINATED BY '\t' LINES TERMINATED BY '\n' ;`
Export the entire database to an archive. This command is run at the shell prompt, not the `mysql` prompt. The resulting file has all the commands necessary to recreate the original database tables	`mysqldump -u root databasename > datafile.sql`
Read back in a database created via dump Read in a file of SQL commands This command is also run at the `bash` prompt, and the target database must already exist	`mysql -u root targetdb < mw.sql`
User management[a]	
Set the password for the current user (from the `mysql` prompt). Remember the equal sign	`SET PASSWORD = PASSWORD('mypass');` `SET PASSWORD FOR 'python_user'@'localhost' = PASSWORD('newpass') OLD_PASSWORD('oldpass');`
Add a new user with defined addresses that they can connect from and a preset password	`CREATE USER 'newuser'@'localhost' IDENTIFIED BY 'newpassword';`
Give a user privileges. The capabilities, database and tables, and user and host are specified. Host IP ranges use `%` as the wildcard character	`GRANT SELECT, INSERT, UPDATE, CREATE, DELETE ON midwater.* TO 'newuser'@'localhost';`
Log in with password (from the shell prompt)	`mysql -u newuser -p`

[a]These commands can also be accomplished from within the Dashboard or SQuirrelSQL GUIs.