Chapter 4, Power Programming Tools

John M. Morrison

October 30, 2019

Contents

1 Introduction

In this chapter we shall introduce some new modules and features that will make Python a far more useful applications programming tool. You can go to the Python site and find any of these module's documentation in the Global Module Index. This is the place to look for information about any module you would like to learn about. The major topics in the section are largely independent of one another. It pays to begin peeking at this chapter early in your Python programming experience, as some of the tools here do not require much knowledge of Python.

We begin by introducing raw strings, which are a real convenience when working with fileIO and regular expressions. We then show a couple of useful features in the `sys` module.

The first major topic is *regular expressions*, a powerful pattern-matching language that eliminates a great deal of tedious conditional and looping logic. Regular expressions (or regexes for short) take some effort to learn, but they are a powerful work–saving tool. Languages such as PHP, Java, Ruby and JavaScript all use regexes, so you may as well learn them early.

We then will study the `os` and `os.path` modules, which allow you to interact with your computer's operating system and do system calls from Python, and which grant access to and control over your file system.

In this chapter we shall see how Python can be used to create professional–looking programs that perform useful tasks. We will begin to look at programs that consist of a small number of functions that work together to perform a specific task. Pay attention to the case studies created in the exercises; these are meant to help you build some useful utilities.

1.1 A Helpful Tool: Raw Strings

Python supports a version of strings called *raw strings*. To make a raw string literal, just prepend with an `r`. When Python encounters a raw string, all backslashes are read literally. No special meaning is given them by the language. This interactive session shows how it works.

```
>>> path = 'C:\nasty\mean\oogly'
>>> print (path)
```

```
C:
asty\mean\oogly
>>> path = r'C:\nasty\mean\oogly'
>>> print (path)
C:\nasty\mean\ugly
>>>
```

Notice that in the raw string, the `\n` did not expand to a newline; it was a literal backslash-n. This is a great convenience when dealing with file paths in Windoze and for writing regular expressions.

Warning! You may not end a raw string with a `\`. This causes the close-quote to be escaped to a literal character and causes a string-delimiter leak. Think for a moment: there is an easy work-around for this!

2 Introducing the sys module

Recall that a UNIX command has three parts, name, options, and arguments. So far, we have created Python programs and simply run them. If we have a program `foo.py`, we type

```
$ python foo
```

at the UNIX command line to run the program. We have used the `raw_input` feature to obtain information from the user.

The disadvantage to this appears when you want to automate a process that involves running a Python program. You need to tell the script running your program all the information it needs; there is no one to type the input for it. The first new feature we will look at allows Python programs to accept command line arguments. Place this program in a file called `cl.py`.

```
#!/usr/bin/python3
import sys
for k in sys.argv:
    print (k)
```

We will use the shebang line and make this file executable. Observe what happens in the following UNIX session.

```
$ chmod u+x cl.py
$ ./cl.py a b c d e f
./cl.py
```

```
a
b
c
d
e
f
$
```

What we see here is that the list `sys.argv` of strings is put to `stdout` by the `for` loop in the program. This list of strings is a list of command–line arguments. You can gain access to these by using the `[]` operator. For example `sys.argv[1]` evaluates to `a`. The value `sys.argv[0]` evaluates to `./cl.py`, the name of the command that invoked it.

A Command Line for Windoze Users You also have access to this tool in Windows. You need to edit your environment variables, and add the Python executable to your search path. Having done this, you can invoke Python from a `cmd` window just as we have done in this section. You will find this interface to be very useful.

Be reminded if you want to use the command–line arguments as numbers, you must cast them to the appropriate type. Here is an example of this at work.

Enter the following in a file named `f2c.py`, save, quit and make the file executable.

```
#!/usr/bin/python3
import sys
Fahrenheit = float(sys.argv[1])
print (farenheit, "degrees farenheit is equal to", )
print (5.0/9*(farenheit - 32), "degrees Celsius.")
```

Then run it like so.

```
$ ./f2c.py 212
212.0 degrees farehheit is equal to 100.0 degrees celsius.
```

2.1 This Way to the Egress!

Suppose some silly end-user decides to enter a foolish command-line argument, or that some other dissaster occurs and you just want to bail out. The function `sys.exit("error message")` gives you an expeditioius route to escape. Here is an elaboration on our temperature scale converter example. Place it in a file named `safef2c.py`

```
#!/usr/bin/python3
import sys
farenheit = sys.argv[1]
if farenheit.count(".") > 1:
    sys.exit("Malformed number: Two or more decimal points")
for k in farenheit:
    if not (k in "0123456789."):
        sys.exit("Malformed number: non-numerical character")
farenheit = float(farenheit)
print("%s degrees farenheit is equal to %s degrees centigrade"% (farenheit, 5.0/9*(farenheit
```

Now we run our shiny new program. Look at the nasty, inelegant snarl of code we wrote to safeguard our innocent comand–line argument from evil end-users with foul intent. Next see how it works nicely to prevent problems and graceless uninformative surly error messages.

```
$ chmod u+x safef2c.py
$ ./safef2c.py 212
212.0 degrees farehheit is equal to 100.0 degrees celsius.
$ ./safef2c.py 4.55
4.55 degrees farenheit is equal to -15.25 degrees Celsius.
$ ./safef2c.py 4.55.23
Malformed number: Two or more decimal points
$ ./safef2c.py cowpie
Malformed number: non-numerical character
$
```

We an improve this slightly by adding two functions. By so doing, our code becomes easier to read. The main routine is no longer puncuated by error and conversion code. The main routine is playing "conductor," orchstrating the actions of the functions that do much of the actual work. The code for `isLegalFloat` is a nice, reusable item for a variety of situations.

This is good abstraction at work. The main routine simply delegates the work of checking validity of input and of conversion of the temperature to the functions.

```
#!/usr/bin/python3
import sys
def convert(x):
    """converts x in farenheit to centigrade"""
    return 5.0/9*(x - 32)
def isLegalFloat(x):
    """ends program if x is not a legal float.
Otherwise it returns True"""
```

```
    if x.count(".") > 1:
        sys.exit("Malformed number: Two or more decimal points")
    for k in farenheit:
        if not (k in "0123456789."):
            sys.exit("Malformed number: non-numerical character")
        return True
farenheit = sys.argv[1]
if isLegalFloat(farenheit):
        farenheit = float(farenheit)
print (farenheit, "degrees farehheit is equal to",)
print (convert(farenheit), "degrees celsius.")
```

Here we see that the program has all of its original functionality.

```
$ ./betterf2c.py 45.3
45.3 degrees farenheit is equal to 7.38888888889 degrees Celsius.
$ ./betterf2c.py 212
212.0 degrees farehheit is equal to 100.0 degrees celsius.
$ ./betterf2c.py 45.6.7
Malformed number: Two or more decimal points
$ ./betterf2c.py 56a
Malformed number: non-numerical character
$
```

2.2 How can I change Python's Recursion Limit?

If you have been fooling around with recursion, you probably have notice that Python limits the call stack to a height of 1000. The recursion limit prevents runaway function calls that can give your box some pretty serious agita. Here is a sample interactive session with `getrecursionlimit` and `setrecursionlimit` at work.

```
>>> import sys
>>> sys.getrecursionlimit()
1000
>>> sys.setrecursionlimit(5002)
>>> def factorial(n):
...     if n in [0,1]: return 1
...     return n*factorial(n - 1)
...
>>> print (factorial(5000))
4228577926605543522201064 .....   0000000
```

Use this feature judiciously. This kind of stack overflow can be indicative of a serious bug in your program. However, here, we are fully aware that the factorial function needs sufficient depth of the call stack to do its job.

3 Python File IO

File operations in Python are handled by the built-in function `open`. This function opens a pipe between your program and a file. You can specify the file using a relative or absolute path. It is an error to attempt to do operations on a file or into a directory for which you do not have permission. For example, you can open a file for reading if you have read permissions. You cannot open a file for writing if you lack have write permission.

3.1 File Opening Modes

The open function has two arguments: a filename (a string) and a mode (string). We will concentrate on three modes for opening files. We show them in the table here.

Python File Open Modes

Mode	Explanation
a	This mode will open a file for writing and append that which you write to the end of the file. If the file does not exist, it is created for you. It is an error for you to attempt to write in a directory in which you do not have write permissions. It is an error to attempt to append to a file for which you lack write permissions.
w	This mode will open a file for writing and overwrite the file you specify if it exists and it will create it otherwise. Be warned: *opening an existing file for writing means it will be clobbered.* It is an error for you to attempt to write to a file in a directory in which you do not have write permissions.
r	This mode will open a file for reading; it is an error to attempt to read a nonexistent file or to read from a file for which you do not possess read permissions.

Usage In what follows the file name and mode are both strings. The syntax for opening a file is as follows.

```
filePipe = open(filename, mode)
```

3.2 Writing to a File

You will often see the name `outFilePipe` for a file to write or append to and `inFilePipe` for a file we are reading into our programs. Think of the pipe as a one–way connection between your program and the file. The `write` method sends the characters you write down the pipe and into the file. The `read` method will hoover up the characters from a file into your program. We shall begin by working with the write mode.

```
outFilePipe = open("out.txt", "w")
outFilePipe.write("Hello")
outFilePipe.write("World")
outFilePipe.close()
```

When you open the resulting file `out.txt`, you will see that it contains this text.

```
HelloWorld
```

Notice that there is no newline placed by the write method. If you want newlines, you must put them yourself! The `write` method just pushes the new bytes into the file. We now revise our program as follows

```
outFilePipe = open("out.txt", "w")
outFilePipe.write("Hello\n")
outFilePipe.write("World\n")
outFilePipe.close()
```

and you will now have the newlines you wanted. Do not fail to close the file or it may never write and your work will be lost!

Danger! If you open a file for writing which already exists, it is clobbered. Later, when we learn about the `os` module, we will see how to prevent this. Recall that such clobbering can occur in recipient files when invoking the `mv` and `cp` commands. This feature is present because the destructive overwrite is often a desired side effect of the command.

3.3 Reading from a File

Suppose we have a file named `in.txt` with this Haiku stored in it.

```
I know am an innie
I therefore collect lint daily
It is my lot in life.
```

Now we show how to open it for reading using the `"r"` mode.

```
inFilePipe = open("in.txt", "r")
stuff = inFilePipe.read();
inFilePipe.close()
print (stuff)
print ("len(stuff) = " , len(stuff))
```

The `read()` method reads the file in as one giant string. This string will contain the newlines necessary to reconstruct the original structure of the file.

```
$ python read.py
I know am an innie
I therefore collect lint daily
It is my lot in life.

len(stuff) =  72
```

If you count the characters with spaces, you will come up short characters. Do not forget that there are invisible `'\n'`s that act as end-of-line characters.

The object returned by open is a file object; you can check in an interactive session. Calling the `type()` function on a file pipe results in the reply

```
type<'file'>
```

It is best to think of a file object as a pipe to a file; this pipe is outgoing if we are writing or appending and incoming if we are reading. You can have as many file objects present in your program as you wish, each slurping from and spewing data into different places. These file objects can open and close as needed throughout the lifetime of your program.

3.4 A Bigger Example: `copy.py`

We shall now produce an example of a program that emulates the action of the UNIX command `cp` for files. Recall that the `cp` command needs a *donor file*, which is the file being copied, and a *recipient file*, the destination of the data being copied from the donor. We begin by specifying the *behavior* of our program. We want something like this

```
$ ./copy.py source sink
```

The file `source` should exist. We do not yet have the ability to check this, but that will change before the end of the chapter. The file *sink* will be overwritten by the contents of `source` if it exists. Otherwise it will be created and the contents of `source` will be placed in it.

Create a file named `copy.py` and put this outline of comments in it. In this file, we see the broad details of what we need to do to accomplish our stated task. If you run this program (clearly) it will do nothing.

```
#get the name of the donor file
#get the name of the recipient file
#read in the donor file
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Let's start at the beginning. We shall begin by just getting the file names from `stdin`. Let us add the shebang line and make the file executable too.

```
#!/usr/bin/python3
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Run the program we have so far and see the donor and recipient file getting requested. So far, nothing has really happened, other than the garnering of the file names.

```
$ chmod u+x copy.py
$ ./copy.py
Enter a donor file: foo.txt
Enter a destination file:  goo.txt
$
```

Next, we open the file for reading, get everything and get out.

```
#!/usr/bin/python3
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
```

```
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
inFilePipe.close()
#print its contents to the recipient
#make sure all is closed when we are done or else.
```

Run the program again to check our progress.

```
$ python copy.py
Enter a donor file: in.txt
Enter a destination file:  noplace.txt
```

Since we are doubting Thomases here, we will put in some temporary code to print `buf` to `stdout` and to verify that all is in good order. Now let's write it all into the recipient file.

```
#!/usr/bin/python3
#get the name of the donor file
#get the name of the recipient file
donor = raw_input("Enter a donor file: " )
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
inFilePipe.close()
#print its contents to the recipient
print (buf)     ##temporary code to test file open.
#make sure all is closed when we are done or else.
```

All is working so far.

```
$ python copy.py
Enter a donor file: in.txt
Enter a destination file:  nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.

$
```

Notice the extra line at the end; it is the `\n` at the end of the last line of the file that is doing this. Now we will get rid of the temporary code and go for the real thing. Notice how we remembered to open the recipient file for writing.

```
#!/usr/bin/python3
```

```
#get the name of the donor file
donor = raw_input("Enter a donor file: " )
#get the name of the recipient file
recipient = raw_input("Enter a destination file:  ")
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
#open to write to the recipient file  (can't forget this)
outFilePipe = open(recipient, "w")
#print its contents to the recipient
outFilePipe.write(buf)
#make sure all is closed when we are done or else.
inFilePipe.close()
outFilePipe.close()
```

Here is a shell session that shows all.

```
$ ./copy.py
Enter a donor file: in.txt
Enter a destination file:  nowhere.txt
$ more in.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$ more nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$
```

We will do one more thing to add polish: we will use the command–line argument feature to give this program a professional look.

```
#!/usr/bin/python3
import sys
#get the name of the donor file
donor = sys.argv[1]
#get the name of the recipient file
recipient = sys.argv[2]
#read in the donor file
inFilePipe = open(donor, "r")
buf = inFilePipe.read()
#open to write to the recipient file  (can't forget this)
outFilePipe = open(recipient, "w")
#print its contents to the recipient
outFilePipe.write(buf)
```

```
#make sure all is closed when we are done or else.
inFilePipe.close()
outFilePipe.close()
```

Our program has all of the slickness of a nice UNIX command.

```
$ ./copy.py in.txt nowhere.txt
$ cat in.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$ cat nowhere.txt
I know am an innie
I therefore collect lint daily
It is my lot in life.
$
```

4 Some Useful Techniques for File Input

We begin by showing some useful methods for `file` objects. Each method is shown with the modes for which it applies.

The file reading mechanism has a *file pointer*, which keeps track of how much of the file has been read. As the file read, the file pointer advances.

Python File Open Methods		
Method	Mode	Explanation
read()	r	This reads in the entire file in a giant string. The newlines in the file will be embedded in this string. The file pointer will advance to the end of the file.
read(*size*)	r	This reads *size* bytes in as a string. The file pointer will then be advanced to the end of the bytes returned.
readlines()	r	This reads in the entire file into a giant list of strings. Each line in the file constitutes an item.
readline()	r	This reads one line from the file then points at the next one. The file pointer winds up at the end of the file.
seek(*size*)	r	This moves the file pointer to the *size*th byte of the file. If *size* is 0, it resets the file pointer to the beginning of the file.
write(*string*)	w	This writes in the string passed it into the file. If you want a newline, you must explicitly put it in.
close	w, a, r	This closes the file pipeway and cleans up. Do it as soon as you finish with the file, as it conserves system resources.

You can traverse a file all at once, a line at a time, a character at a time, or in pieces of any size you specify.

4.1 Methods of Traversing Files with `for` loops

Here is a basic technique for traversing a file opened for reading.

```
f = open("foo.txt", "r")
for k in f:
    ##do something with each line in the file
```

This `for` loop concludes its business when it reaches the end of the file. It works on a file a line at a time. The `for` loop thinks of the file as a collection of strings, each line in the file being a string. It iterates through those strings in a natural order, the order in which they are present in the file.

Python 2/3 note This next loop is ideal for large files in Python 2. The xreadlines() method reads the lines in seriatum and it terminates when the file ends. In Python 3, this is actually what happens.

```
for k in open("foo.txt").xreadlines():
    ##do stuff to each line
```

5 Introducing the `re` module

Python supports a powerful feature called *regular expressions*, which allow you to quickly and easily match textual patterns. Witness the ungainliness of the code for `isLegalFloat` in the last section. Can we accomplish this without a lot of error–prone complex looping and forking? In short, the answer is, "yes." However, you will need to expend a substantial effort to learn two new languages: *Characterclassese* and *Regexese*.

This effort is one you will want to make

Characterclassese enables you to create wildcards for representing a single character. Character classes are the "bricks" of regular expressions. Regexese is the "building;" it controls what happens when character classes are assembled together to recognize patterns . This section will be a quick–and–dirty introduction to regular expressions that will allow you to accomplish a wide variety of routine tasks.

5.1 Characterclassese

This is the language spoken inside of character classes. These are examples of classes of characters we might like to represent. A character class is a character wildcard that can stand for one or more characters. We might like to create wildcards to represent these types of classes.

- an alphabetical character
- a punctuation mark
- a numeral
- lower–case letters
- upper–case letters
- an octal digit
- a hex digit
- any old list of characters you'd like to create
- any whitespace character

The simplest character class is just a character by itself. For example, the character `a` is the character class standing for the character `a`. This is called a *literal* character class.

The the next simplest character class shows an explicit list of characters. For example

```
[aeiou]
```

represents any of the lower–case letters a, e, i, o or u. Notice how this character class lives in a “house” made of `[]`. The characters `]` and `[` are *metacharacters* or “magic characters.” They play the role of being the exterior walls of a character class’s house. We will make a complete list of metacharacters in Characterclassese later in this section.

A lone character class is a regex, so we can test it in Python. This will return `True` any line that contains any of a, e, i, o or u. To use Python’s regex capabilities, you must first import the library re.

```
>>> import re
>>> seeker = re.compile("[aeiou]")
>>> bool(seeker.search("syzygy"))
False
>>> bool(seeker.search("sheck"))
True
>>> bool(seeker.search("EYEBALL"))
False
```

The seeker using the character class [aeiou] is seeking a lower-case regular vowel. You can see that ”syzygy” lacks these and that ”Sheck” has one. However, ”EYEBALL” fails, because all of its letters are upper-case.

The code

```
seeker = re.compile("[aeiou]")
```

creates the seeker, which is a Python regular expression object. To get the seeker to work, you use its search method. This returns a match object, which is a fairly complicated creature. However, you can cast a match to a bool and see if the seeker found the quarry it sought. The `seek` method returns a match object casting to `False` if it does not see the characters a, e, i, o, or u in the string it is searching.

You can put any list of characters inside of `[....]` and use it to to create a match object. For instance,

```
[aWybe05]
```

will match any of the characters a, W, y, b, e, 0 or 5. Be warned, however that Characterclassese has some magic (meta-) characters. We shall address this issue here so you know what to do.

`[`	begin a character class formed with a list of characters
`]`	end a character class formed with a list of characters
`-`	produces a range (see below)
`\`	defangs any magic character (ex: `\[` for a `[` character)
`^`	special “not” character This is only magic at the start of a list character class

5.2 Ranges

The metacharacter - creates a range of characters. The ordering of characters is determined by ASCII values. A character class can contain zero or more ranges. Let's begin with a simple example.

```
[a-z]
```

represents the lower case alphabetical characters. Range is keyed on ASCII value. If you are unsure about the ASCII value of a character, use Python's ord function.

```
>>> ord("a")
97
>>> ord("z")
122
>>> ord("A")
65
```

You can represent any alphabetical character with

```
[A-Za-z]
```

and any hex digit with

```
[0-9A-Fa-f]
```

Any octal digit may be represented with

```
[0-7]
```

You can include the - character in a character class by using

```
[\- ({\it and any other characters you want})]
```

Let's be exclusionary and sNOTty! We can write "negative" character classes with the exclusion operator.

The not character `^` must appear FIRST, right after `[`. This example represents any character that is not a lower case alpha.

```
[^a-z]
```

5.3 Escape now!

We can toggle magic with the escape character

```
\
```

This turns magic off for magic characters and turns it on for certain non-magic characters. For example, `n` is not magic, but `\n` is the newline metacharacter.

5.4 Special Character Classes

Some character classes exist that take the form \someCharacter. Here we present a short table of the most useful ones.

`\s`	This stands for any single whitespace character.
`\S`	This stands for any single non-whitespace character.
`\d`	This stands for any single decimal numeral.
`\D`	This stands for any character that is NOT a single decimal numeral.
`\w`	This stands `[a-zA-Z_0-9]`; this represents the characters allowed for Python identifiers in the positions beyond the first.

5.5 Regexese

Be warned: The rules for Regexese are different from Characterclassese! This is because Regexese is a whole new language. When in character classes, speak Characterclassese, when outside, speak Regexese. Context is everything! Let us begin with the metacharacters of Regexese.

Basic Metacharacters (One Keystroke)	
Metacharacter	Action
`[`	begin delimiting a character class
`]`	end delimiting a character class
`^`	beginning-of-line charcter
`$`	end-of-line charcter
`|`	or
`\`	escape character The escape character can make other characters into metacharacters, or it can remove magic from a metacharacter.
`(`	left delimiter
`)`	right delimiter
`.`	any single character except for a newline
`* + ?`	repetition metacharacters (later)

To turn off any magic character, precede it with the escape character `\`. This rule is exactly the same as the corresponding rule in Characterclassese.

5.6 "And then immediately"

We shall now see our first regex for matching a sequence of characters. Juxtaposition in a regex means "and then immediately". The the regex

```
[a-iA-I][1-9]
```

matches a string that contains of a letter a-i and then a digit 0-9. The digit must immediately follow the letter. For example, baaa5 matches and Q3 does not.

Battleship! In the game of Battleship, we specify coordinates with letters a-i and digits 0-9. The regex

```
^[a-iA-I]\d$
```

matches any string that is a legit Battleship coordinate. For example it will match a5, A9 or B3, but not Q4. This regex demands the following: begining-of-line andthenimmediately a-i or A-I andthenimmediately a decimal digit andthenimmediately an end-of-line.

String Literals The character 'a' is the same as the character class `[a]`. The regex

```
CUSIP[0-9]
```

contains the string literal `"CUSIP"`; this portion of the string requires an exact match of a the substring `"CUSIP"`. Therefore the regex here matches CUSIP5, but not CUSIp5, CUSIP55 or CUSIP. Read it as CUSIP andthenimmediately a digit.

5.7 Repetition Operators

There are repetitions operators for regular expressions. These are all postfix operators.

Repetition Operators	
Operator	Action
`?`	expression appears 0 or 1 times
`+`	expression appears at one or more times consecutively
`*`	expression appears at zero or more times consecutively
`{n}`	expression appears exactly n times
`{m,n}`	expression appears at least m but not more than n times

To match a social security number, you needs three digits, followed by a dash, two more digits and then a dash and then four digits. This is an easy job when you use the repetition operators.

```
^[0-9]{3}-[0-9]{2}-{0-9}{4}$
```

Observe that - is *not* a magic character in Regexese. Now we will bring the delimiters `(...)` into the picture. The regex

```
^([a-c][0-9])+$
```

matches any string containing a character a-c followed by a digit any number of times. Here are some matches

```
>>> import re
>>> seeker = re.compile("^([a-c][0-9])+$")
>>> bool(seeker.search("a3b2c5")
True
>>> bool((seeker.search(""))
True
>>> bool(seeker.search("b4"));
False
```

Notice that the multiplicity operators have a higher order of precedence that juxtaposition. Hence the need for parentheses when having a regex with more than one character class being acted on by a multiplicity operator.

5.8 Using or

The operator — means ”or”. When using it, ALWAYS enclose the things you are “orring” in parens! This is a strict style expectation; adhere to it. It protects you from all manner of stupidity. The or operator is piggy and if you do not use parens, you do not control its ardor.

Let’s plunge in with an example. Notice how we escape the magic character . to defang its magic (any character).

```
>>> import re
>>> seeker = re.compile("^(Morrison|Sheck) is a nut\.$")
>>> bool(seeker.search("Morrison is a nut"));  ##no period.
False
>>> bool(seeker.search("Morrison is a nut."));
True
>>> bool(seeker.search("Sheck is a nut."));
True
```

Two-Keystroke Metacharacters There are some characters that can be preceded by a to give a special interpretation. Here is table of some of them.

Two-Keystroke Metacharacters	
Metacharacter	Matches
`\d`	any decimal integer
`\D`	Any character not a decimal integer
`\s`	any whitespace character
`\S`	any non-whitespace character

For example the regex

```
^\s*-?[0-9]+\s*$
```

matches any string that has an integer in it that may or may not be surrounded by whitespace.

Turbo! Repetition operators can be applied to regexes, not just character classes. This is accomplished by using parentheses.

This regex will do a case-insensitive (note i after /) check and return true if the string passed it alternates a letter a-c followed by a digit zero or more times. Note: it must start with a letter and end with a digit. Enter this into a file named `reg.py`.

```
import re
seeker = re.compile("^([a-c]\d)*$", re.IGNORECASE)
print (bool(seeker.search("c4")), ", expected:  True")
print (bool(seeker.search("poop")),  ", expected:  False")
print (bool(seeker.search("A5b4")),  ", expected:  True")
```

Notice the second argument to `re.compile`; it causes the case of the string being scanned to be ignored. Running this program we see

```
$ python reg.py
True , expected:  True
False , expected:  False
True , expected:  True
$
```