Chapter 4, Fancy Tools

John M. Morrison

January 6, 2020

Contents

0 Introduction

This chapter will add zip to your Python programs. You will be able manipulate collections with comprehensions, and do custom sorting, and find needles in haystacks with regualr expressions.

We will apply this to a variety of interesting computational problems. Don't be shy about attempting the exercises. Get your hands dirty.

1 Comprehensions

Python has a compact and clear way to transform and filter collections that is a better way than the old `map` and `filter` options. Suppose you want to find the longest word in the Scrabble dictionary, `scrabble.txt`. You could do this.

```
with open("scrabble.txt", "r") as fp:
    blob = fp.readlines()
longest = 0
for word in blob:
    if len(word) > longest:
        longest = len(word)
print("The longest word in the scrabble dictionary is {0} letters long".format(longest))
```

Note that the words in this file come one to a line, and that there is a newline character at the end of each line, so the longest word actually contains 15 characters.

Compare that to this; notice how easy it is to account for the end-of-line character.

```
fp = open("scrabble.txt", "r")
n = max([len(k) - 1 for k in fp])
print("The longest word in the scrabble dictionary is {0} letters long".format(n))
```

So, here is what happened. If you have a list `x` expression `e(x)`, you can butter the expression across the list as follows.

```
[e(k) for k in x]
```

This code is shorter and less error-prone than the explicit loop.

Now, to become a power Scrabble player, you need to know all of the legal two-letter words. Here is one way to query the file for them.

```
def wordsOfLength(n):
    out = []
    with open("scrabble.txt", "r") as fp:
        for word in fp:
            if len(word) == n + 1:  #newline adjustment
                out.append(word)
    return out
print(wordsOfLength(2))
```

Here is another.

```
def fastWordsOfLength(n):
    fp = open("scrabble.txt", "r")
    return [word for word in fp if len(word) == n + 1]
print(fastWordsOfLength(2))
```

If you don't want the newline characters, you can do this

```
def fastWordsOfLength(n):
    fp = open("scrabble.txt", "r")
    return [word[:-1] for word in fp if len(word) == n + 1]
print(fastWordsOfLength(2))
```

So, now we have juiced this up. The general form of a list comprehension is the following

```
[e(k) for k in x if p(k)]
```

where `e(k)` is an expression in `k`, and `p(k)` is a predicate.

Now let us turn this loose. Suppose you want to search a lot of text files for a particular string and tell which one have that string in them. Let us begin with a design.

We will specify a string, a file extension, and a directory as commmand line arguments. We will give the directory a default argument of "." so the current working directory is searched by default.

We can also use comprehensions on dictionaries and sets. The item we are "inning" must be an iterable. Here we create a set from a list and a set from a set.

```
>>> x = [1, -1, 4, 5,-2, 2, 6]
>>> y = {k*k for k in x}
>>> y
{1, 4, 36, 16, 25}
>>> z = {k for k in y if k < 20}
>>> z
{16, 1, 4}
```

Let's take this for a spin with a dictionary. You can extract all kinds of screwy stuff from a dictionary with a simple comprehension.

```
>>> d = {"cat":"meow", "dog": "woof", "pig":"squeal", "chicken":"cluck", "duck":"quack", "co
>>> e = {k for k in d if d[k].endswith("k")}
>>> e
{'chicken', 'duck'}
```

```
>>> f = {len(k) for k in d if len(d[k]) == 5}
>>> f
{4, 7}
>>> g = {d[k].upper() for k in d if len(k) < 4}
>>> g
{'SQUEAL', 'WOOF', 'MEOW'}
>>>
```

2 Custom Sorting with Keys

List objects have a `sort` method which performs an in-place sort and a `sorted` method that returns a sorted copy of the list. Let us make a list and sort it.

```
>>> roster = ["Vazquez", "gabbrielli", "roberts",
    "monahan", "hernandez", "bullard", "regalis", "Alter"]
>>> sorted(roster)
['Alter', 'Vazquez', 'bullard', 'gabbrielli', 'hernandez',
    'monahan', 'regalis', 'roberts']
```

Yuck. The default is asciicographical. What if we would like to ignore case? This method has two keyword arguments, `key` and `reverse`. The keyword `reverse` takes a boolean; marking it true causes the sort to occur in reverse order. The object `key` is a function that is applied to each entry before comparisons are made.

```
>>> sorted(roster, key=str.lower)
['Alter', 'bullard', 'gabbrielli', 'hernandez',
    'monahan', 'regalis', 'roberts', 'Vazquez']
```

This caused all items to be lower-cased before comparison so they ended up being sorted case-insensitive. The object `str.lower` is an example of a *method reference*. You might ask, "How do you to a primary and secondary sort? Try this.

```
>>> alpha = sorted(roster, key=str.lower)
>>> alpha
['Alter', 'bullard', 'gabbrielli', 'hernandez', 'monahan', 'regalis', 'roberts', 'Vazquez']
>>> beta = sorted(alpha, key=len)
>>> beta
['Alter', 'bullard', 'monahan', 'regalis', 'roberts', 'Vazquez', 'hernandez', 'gabbrielli']
>>> [k for k in beta if len(k)== 7]
['bullard', 'monahan', 'regalis', 'roberts', 'Vazquez']
```

We first sorted on the secondary key, then on the primary.

Why does this work? Sorts in Python are guaranteed to be *stable*; this means that when multiple records have the same key, their order is preserved.

You can use a `for` loop on a dictionary; it iterates through the dictionary's keys. Here is something to make a webmaster happy.

```
>>> for k in d:
    ...print("<tr><td>{0}</td><td>{1}</td></tr>".format(k, d[k]))
...
<tr><td>gabbrielli</td><td>chemistry</td></tr>
<tr><td>brisk</td><td>chemistry</td></tr>
<tr><td>spells</td><td>chemistry</td></tr>
<tr><td>brenner</td><td>humanities</td></tr>
<tr><td>vazquez</td><td>math</td></tr>
```

How sad. They are not alphabetical. But this is an easy fix.

```
>>> roster = sorted(list(d.keys()))
>>> roster
['brenner', 'brisk', 'gabbrielli', 'spells', 'vazquez']
>>> for k in roster:
    ...print("<tr><td>{0}</td><td>{1}</td></tr>".format(k, d[k]))
<tr><td>brenner</td><td>humanities</td></tr>
<tr><td>brisk</td><td>chemistry</td></tr>
<tr><td>gabbrielli</td><td>chemistry</td></tr>
<tr><td>spells</td><td>chemistry</td></tr>
<tr><td>vazquez</td><td>math</td></tr>
```

Vita est bona!

A Room With a View Consider these three methods.

```
>>> d.values()
dict\_values(['chemistry', 'chemistry', 'chemistry',
        'humanities', 'math'])
>>> d.keys()
dict\_keys(['gabbrielli', 'brisk', 'spells',
        'brenner', 'vazquez'])
>>> d.items()
dict\_items([('gabbrielli', 'chemistry'), ('brisk', 'chemistry'),
    ('spells', 'chemistry'), ('brenner', 'humanities'),
    ('vazquez', 'math')])
```

They return objects called `views`. A view is an interable that walks through the items in a collection, avoiding the necessity of constructing a whole new

object in memory that would contain the items you wish to traverse. A view is much more lightweight than a full-fledged container, especially when there are thousands of objects to be viewed. Views can be cast as lists or sets.

3 Characterclassese

In this section and the next, we are going to learn a mini-language for specifying patterns in text that you wish to search for or filter into or out of collection. The first step in this process is to understand *character classes*, which represent a wildcard for a single character.

Here are examples of classes of characters we might like to represent.

- an alphabetical character
- a punctuation mark
- a decimal digit
- 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

A character class is a character wildcard that can stand for one or more characters. We will learn a language called Characterclassese that is the language used to create character wildcards.

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. Oddly enough, these are known as *list character classes* 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 `[` metacharacters. 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.

Now let's see how this works in Python. We will import the `re`, or regular expression, module into our session. 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("z"))
False
>>> bool(seeker.search("e"))
True
>>> bool(seeker.search("E"))
False
```

The seeker using the character class [aeiou] is seeking a lower-case regular vowel.

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. For the purposes of this section, this will work very well.

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

Sometimes raw is better Recall that python has raw strings, which supresses the action of magic characters, including `\`. To make a raw string, you just put an `r` in front of the string. This will save your regex strings from having tons of irksome double backslashes.

3.1 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]
```

3.2 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.

3.3 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.

3.4 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.

3.5 "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.

3.6 Repetition Operators

There are repetitions operators for regular expressions. These are all postfix operators. They all have higher precedence than juxtaposition.

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 than juxtaposition. Hence the need for parentheses when having a regex with more than one character class being acted on by a multiplicity operator.

3.7 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

```
unix> python reg.py
True , expected:  True
False , expected:  False
True , expected:  True
unix>
```

Programming Exercises

1. Write a function called `xWordCheat` which accepts as an argument a string of the form `--te--au-` and which returns all words in the Scrabble dictionary in a list that match the string with a single character for each dash. In this case, CATERWAUL should be in your list
2. Write a function called `containsInOrder` which takes a comma-separated list of at least one string as arguments and which retuns `True` if the second and subsequent strings are find in order inside of the first string Examples

   ```
   containsInOrder("asciicographical", "ci", "cog", "ica") -> True
   containsInOrder("zither", "th", "er", "zi") -> False
   ```

3. A valid IPv4 number is a string of the form xxx.xxx.xxx.xxx, where each xxx is a decimal number 0-255. Write a function that accepts a string as an argument and whic returns `True` when the string passed in is a valid IP4.
4. A valid hex code for a color is a six-digit hex number, which might be preceded by a prefix of `0x` or a `#`. Write a function that determines if a string is a valid hex color code.

4 Case Study: Writing a Concordance

A concordance is a list of all distinct words used in a document, along with the number of times each words is used. Concordances are very useful for the

forensic study of writing, as well as for deciding how to construct an index for a book.

This will be a heavy-duty program that will be able to produce a concordance for such a document as Tolstoy's *War and Peace*, or a King James Bible, which, in plain text, is about 4 megabytes.

So, we begin by breaking the problem down. We will open a file one line at time so as not to be a memory hog. Each line must be split into words. To avoid duplication, we should lower-case all of the words found. Annoyances remain: what about punctuation marks that adhere to words like the question mark you see here? So, we need to do some sanitization of the text prior to processing it.

Next we create a dictionary to hold what we find. The keys are words and the values are the number of times each word is found. Once the words are sanitized, we check each new word to see if it is in the dictionary; if it is not, add it and give it a value of 1. If it is, increment its value. The question is, what do we do with this dictionary once we create it? A King James Bible will have a vocabulary with thousands of words. It would be bad manners to simply spew them to `stdout`. We will put them to a file, which we will give an extension of `.conc`, which will be tacked onto the end of the input file's name.

So, let us begin by getting the apparatus necessary to process a single line of text into a list of "clean" strings. Our legitimate alphabetical characters include lower-case letters, upper-case letters, and the apostrophe. Don't forget that. We wil screen out numbers, too. A character class for such a thing is this.

```
[^a-zA-Z']
```

Note that we'd like to weed out all instances of any-sized consecutive blob of them so our regex looks like this

```
[^a-zA-Z']+
```

Next we avail ourselves of the method `re.sub` to replace any contiguous group of punctuation marks wiht a space. So far, our function looks like this.

```
import re
def lineToCleanList(s):
    s = s.lower()
    puncts = "[^a-zA-Z']+"
    s = re.sub(puncts, " ", s)
    return s
#temporary test code
s = "This is a test for our method. Let's see what it does."
print("{0}\n{1}".format(s, lineToCleanList(s)))
```

It returns the line of text, lower-cased and with all punctuation removed. Run

this and see for yourself. The final step is to return a list of strings with the individual words in them. We now achieve this.

```
import re
def lineToCleanList(s):
    s = s.lower()
    puncts = "[^a-zA-Z']+"
    s = re.sub(puncts, " ", s)
    return s.split()
```

Now run this and see that it does the job. It is now time to see where we are in the process. If we open a file, we can grab a line and ship ot to `lineToCleanList` to get it ready. We have everything in position to create the dictionary.

Before opening a file, let's put a toy exmple in the main routine of our program and run on that. Here is the new state of our program.

```
import re
def lineToCleanList(s):
    s = s.lower()
    puncts = "[^a-zA-Z']+"
    s = re.sub(puncts, " ", s)
    return s.split()
def build(s):
    out = {}
    return out
toy = """Here is an example for our concordance builder
to go to work on.  It has some useless numbers such as
59 and a bunch of #>*!@ punctuation to annoy us. We will see
how this all turns out; heaven forfend it works correctly. """
```

Let us now feed in our toy example and see what happens. For each element in the list, if the element is a key in the dictionary, we increment its value. If not, we enroll it and give it a value of 1.

```
import re
def lineToCleanList(s):
    s = s.lower()
    puncts = "[^a-zA-Z']+"
    s = re.sub(puncts, " ", s)
    return s.split()
def build(s):
    out = {}
    for word in lineToCleanList(s):
        if word in out:
            out[word] += 1
```

```
        else:
            out[word] = 1
    return out
toy = """Here is an example for our concordance builder
to go to work on.  It has some useless numbers such as
59 and a bunch of #$%!@ punctuation to annoy us. We will see
how this all turns out; heaven forfend it works correctly. """
dictionary = build(toy)
for key in dictionary:
    print("{0}\t{1}".format(key, dictionary[key]))
```

Now run

```
unix> python concordance.py
concordance         1
of          1
work          1
useless         1
for          1
we          1
is          1
turns         1
on          1
to          3
example         1
it          2
bunch         1
this          1
punctuation         1
some          1
forfend         1
as          1
and          1
numbers         1
heaven         1
such          1
builder         1
here          1
works          1
our         1
out         1
correctly         1
annoy         1
has         1
how         1
all         1
```

```
will        1
an        1
us        1
go        1
a       1
see         1
```

Now it's time to add the file IO to this program. Get rid of the old main routine and insert this.

```
if len(argv) < 2:
    print("You need to specify the name of a text file for me to process")
else:
    fileName = argv[1]
    fpin = open(fileName, "r")
    fpout = open(fileName+".conc", "w")
```

Next, we are going to modify `build` so it accepts a file object that is open for reading. We will need to wrap its loop in another loop. Make this file, `testFile.txt`.

```
Hello, I am a test file with some !*!@@
junk in it and and some repeated words are repeated
so we can test both branches of execution
in the routine that sticks items into the dictionary.
I hope this is a go.
```

Let us print out the dictionary constructed for us.

```
unix >python concordance.py testFile.txt
{'go': 1, 'that': 1, 'a': 2, 'are': 1, 'am': 1,
'file': 1, 'so': 1, 'routine': 1, 'dictionary': 1,
    'test': 2, 'can': 1, 'branches': 1, 'sticks': 1,
    'we': 1, 'it': 1, 'this': 1, 'of': 1, 'words': 1,
    'with': 1, 'execution': 1, 'both': 1, 'is': 1, 'in': 2,
    'the': 2, 'hello': 1, 'hope': 1, 'junk': 1, 'repeated': 2,
    'some': 2, 'and': 2, 'into': 1, 'items': 1, 'i': 2}
```

It worked. But it seems as if the entries come out willy-nilly in some kind of chaotic order. We can fix that with a little key sorting chicanery, along with an invocation of the `items` method for a dictionary. We will also send our output to the file we opened for writing. Here we see the entire program in its full glory.

```
import re
from sys import argv
```

```
import os
def lineToCleanList(s):
    s = s.lower()
    puncts = "[^a-zA-Z']+"
    s = re.sub(puncts, " ", s)
    return s.split()
def build(inFile):
    out = {}
    for s in inFile:
        for word in lineToCleanList(s):
            if word in out:
                out[word] += 1
            else:
                out[word] = 1
    inFile.close()
    return out.items()
if len(argv) < 2:
    print("You need to specify the name of a text file for me to process")
else:
    fileName = argv[1]
    fpin = open(fileName, "r")
    fpout = open(fileName+".conc", "w")
    pairs = list(build(fpin))
    pairs.sort(key=lambda p:p[0])
    for p in pairs:
        fpout.write("{0}\t{1}\n".format(p[0], p[1]))
    fpout.close()
```

In 29 lines of code, we have created an application with some serious firepower that can get real work done.

Programming Exercises

1. Add a function you pass the output file object to that will put the entries in sorted by frequency first then by alphabetical order second.
2. Create a similar program that creates a "concordance" of numbers in a file.
3. Write a program that accepts a file containing an HTML table and which extracts the data to a CSV file. Here is an example input file.

```
<table>
<tr><th>x</th><th>y</th></tr>
<tr><td>1.5</td><td>3.4</td></tr>
<tr><td>4.5</td><td>17.4</td></tr>
<tr><td>3.9</td><td>2.4</td></tr>
```

```
</table>
```

Here is the output file.

```
x,y
1.5,3.4
4.5,17.4
3.9,2.4
```