

PROGRAMMING FOR HUMANISTS

STUART M. SHIEBER

CONTENTS

Part 1. Programming by imitation	2
1. Where we're headed	4
2. Installing Python	5
3. The synoptic gospels	6
Part 2. Programming from first principles	14
4. Python documentation	15
5. The Python interpreter	16
6. Expressions and nesting	18
7. Variables and the naming of values	19
8. Sequence data types	20
9. Functions	27
10. Words, types, and tokens	31
11. Files	32
12. Special characters	34
13. Splitting and joining strings	35
14. List comprehensions	37
15. Sets	39
16. Calculating with n -grams	40
17. Defining your own functions	41
18. Dictionaries	46
19. Loops and conditionals	48
20. A concordance	52
Appendix A. A concordance with description	55
Appendix B. Statistics	61
References	62
Index	63

Part 1. Programming by imitation

These notes are intended to provide an introduction to programming in the programming language `Python` for an audience of techno-savvy humanities scholars who are primarily interested in the use of computers for performing simple analyses of text. I originally prepared them for an audience of historians and philologists of premodern Europe, and the notes may reflect that audience, but should be appropriate for scholars from other disciplines as well.

There are two ways to learn a new language: by imitation and from first principles. This holds for both natural languages and programming languages. Under the `IMITATION` approach, learners see some examples and generate new examples by replacing parts of expressions they've seen. This approach has the benefit of allowing learners to use the language in interesting ways from early on, but they may do so without a full understanding of why the things they are saying work the way they do. Under the `FIRST PRINCIPLES` approach, learners study the elementary units of the language and how they are composed – the lexicon, grammar, and semantics of the language – and construct new examples from these first principles. This approach has the benefit that at every step the learner understands why the expressions work the way they do, but it may take a while to get to the point of being able to use the language to do much that is worthwhile.

For natural languages, the imitation approach is undoubtedly the preferred method. The lexicons and grammars of natural languages are large and complex and not well understood. Further, human beings have an ability to learn natural languages through immersion that allows even very young children to acquire a natural language with no explicit training in the first principles. Finally, the agents that understand natural languages are quite forgiving in their behavior. Fluent speakers can understand disfluent speech. So imperfections in the imitations don't have to hold up communication too much.

For programming languages, the case is somewhat different. Programming languages are artificial languages, and thus we cannot rely on innate language learning abilities. Furthermore, the agents that understand programming languages, computers, are quite unforgiving in their behavior. Even the most trivial variance from the well-formedness principles of the language may be met with utter failure to communicate the programmer's intent to the computer. On the other hand, the lexicons, grammars, and semantics of programming languages are much better understood than those of natural languages, because they have been explicitly designed and sometimes even specified with mathematical rigor. It is thus more practical to learn these first principles and apply them.

38 In these notes, I use both approaches, starting in this first part with the imitation
39 method to get started and build some intuition and sense of what can be done, and
40 then moving in the second part to the first principles that underly the language.

41 During this part, the idea is to merely get you used to the idea of commanding the
42 computer to carry out calculations. Don't worry about the details of the language.
43 Just let the code waft over you, like a pleasant sea breeze. Type the examples in
44 and marvel at the results even if you can't fully understand yet why they work.
45 Learn the following important lessons from the exercise:

- 46 (1) *There's nothing to fear here.* You won't damage your computer by typing the
47 wrong thing. You can experiment. If you wonder "what would happen
48 if", just try it.
- 49 (2) *First principles are important.* To really understand what's going on, the
50 zen-like approach of Part 1 is insufficient. If you're motivated, move on to
51 Part 2. Then go back to Part 1 afterwards and you'll see how much better
52 you understand what's going on.

1. WHERE WE'RE HEADED 53

The coverage of these notes is not sufficient to make you a proficient Python programmer. They do not even provide a basic understanding of the full language. But the notes should get you to the point of writing simple programs to do basic text analyses. To get a sense of what can be achieved, by the end of working through these notes you'll have written code to generate a concordance of the text in Figure 1 (page 31) as found in Appendix A.

You'll also have enough familiarity with Python programming that it should be a simpler transition to learning about and working with the **Natural Language Toolkit** (NLTK), a free and open source Python toolkit for language processing that comes with its own book *Natural Language Processing with Python*.

Like all skills, programming requires practice. You don't get it by reading about it but by doing it. I recommend that you do *all* of the exercises and problems in these notes in order, even the ones that feel trivial, as well as playing around with small problems and tasks of your own devising.

KEY CONCEPTS 1.1. **Conventions used in the notes.** First mentions of **KEY CONCEPTS** are shown in small caps and marked in the margins. You'll find them in the index at the end of the notes as well.

CLICKABLE LINKS The URLs provided in these notes, and some other items are **CLICKABLE**. Clickable links appear **like this**.

EXERCISES
PROBLEMS There are **EXERCISES** and **PROBLEMS** interspersed throughout. The problems are more difficult than the exercises.

 Advanced material that can be skipped on first reading is marked as here.

1.2. **Disclaimer.** I apologize ahead of time for the rather breathless nature of these notes. They go through things quickly, and may be incomplete in various ways. You may (in fact likely will) have to augment them with reading in the Python documentation. On the other hand, I'll be available in class to answer questions, so there's that.

If you find errors or disfluencies in the notes, please let me know so that I can correct them.

84 2. INSTALLING PYTHON

85 Go no further without getting access to a Python interpreter. You'll want to try
86 out the samples of Python code as they are presented and do your own experimen-
87 tation as well.

88 **Mac OS:** Python is available natively on Mac OS. From a window in the
89 Terminal application, type "python". The interpreter will be launched.

90 **Windows OS:** Python executables for Windows can be downloaded from
91 <https://www.python.org/downloads/windows/>. Good luck with that. In
92 case of failure, see the section below on web-based Python interpreters.

93 **Linux:** If you're running Linux, you're not going to need these notes.

94 **Web-based:** On any operating system with a browser, you can set up an
95 account at [Pythonanywhere](#) and run a Python interpreter from within your
96 browser. This will get you started for now.

```
>>> sys.version
'2.7.5 (default, Mar  9 2014, 22:15:05) \n[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)]'
>>> this_python_version
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'this_python_version' is not defined
```

97 The material in these notes is sufficiently straightforward that it probably makes
98 little difference which version of Python you are running. However, for concrete-
99 ness, all the examples below were run with Python 2.7.5.

100 **Exercise 1.** *Obtain access to a Python interpreter via one of the methods above.* □

101 **Exercise 2.** *Test that the Python interpreter is working by running it and typing in a*
102 *simple command for the interpreter to execute. You should see something like this:*

```
% python
Python 2.7.2 (default, Oct 11 2012, 20:14:37)
[GCC 4.2.1 Compatible Apple Clang 4.0 (tags/Apple/clang-418.0.60)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Finished loading pythonrc file
>>> 1+1
2
>>>
```

103

□

3. THE SYNOPTIC GOSPELS 104

We'll be looking primarily at text processing. Suppose we're interested in the synoptic gospels (and who isn't?). Each gospel is a text, which we can think of as a sequence of characters. Here, for instance, are the first four verses of the Gospel of Mark, generated using Python by *opening* a file named `Mark.txt` containing the Clementine Vulgate version of the Gospel of Mark, *reading* all of its lines into a list of lines, and then *extracting* the first four items in that list: 105 106 107 108 109 110

```
>>> open('Mark.txt').readlines()[:4]
```


```
['1:1 Initium Evangelii Jesu Christi, Filii Dei.\r\n', '1:2 Sicut scriptum est in Isaia propheta :
```

Let's give that list of lines a name. We'll call it `mark_lines`. 111

```
>>> mark_lines = open('Mark.txt').readlines()
```

```
>>> mark_lines[:4]
```

```
['1:1 Initium Evangelii Jesu Christi, Filii Dei.\r\n', '1:2 Sicut scriptum est in Isaia propheta :
```

 Notice that the expression `mark_lines[:4]` has exactly the same value as the previous expression `open('Mark.txt').readlines()[:4]`. This fact can be seen as an instance of Leibniz's law of the indiscernability of identicals. The command `mark_lines = open('Mark.txt').readlines()` has the effect of making `mark_lines` identical to `open('Mark.txt').readlines()`. Leibniz's law means that we can "substitute equals for equals". By substituting `mark_lines` for `open('Mark.txt').readlines()` in `open('Mark.txt').readlines()[:4]`, we get the equivalent `mark_lines[:4]`. 112 113 114 115 116 117 118 119 120 121

Viewing the list of lines that way isn't too readable. Here's a nicer presentation: 122

```
>>> for verse in mark_lines[:4]:
```

```
...     print verse,
```

```
...
```

```
1:1 Initium Evangelii Jesu Christi, Filii Dei.
```

```
1:2 Sicut scriptum est in Isaia propheta : [Ecce ego mitto angelum meum ante faciem tuam,/ qui pra
```

```
1:3 Vox clamantis in deserto :/ Parate viam Domini, rectas facite semitas ejus.]
```

```
1:4 Fuit Joannes in deserto baptizans, et praedicans baptismum poenitentiae in remissionem peccato
```

Exercise 3. *If the text of Matthew is in the file named `Matthew.txt`, how would you print out the first four verses of Matthew? The first six verses?* 123 124

Instead of a list of lines (verses), it might be useful to extract a list of words. We'll start by joining all of the lines together, separated by, say, a colon. 125 126

```
>>> mark_string = ':'.join(mark_lines)
```

We can then take a look at the first few characters of this string. (Restricting to the first few avoids the whole giant string running off the end of the page.) 127 128

```
>>> mark_string[:60]
'1:1 Initium Evangelii Jesu Christi, Filii Dei.\r\n1:2 Sicut s'
```

129 **Exercise 4.** *What do you think the [:60] at the end does? Try substituting different*
 130 *numbers, like [:5] or [:100] and see what happens.* □

Solution. *It extracts the first 60 characters of the string, which substring is then printed.*

□

131 **Exercise 5.** *Suppose instead that you wanted to join the lines together with a space instead*
 132 *of a colon. How would you do that?* □

Solution.

```
>>> mark_string = ' '.join(mark_lines)
>>> mark_string[:60]
'1:1 Initium Evangelii Jesu Christi, Filii Dei.\r\n 1:2 Sicut s'
```

□

133 Let's simplify and normalize the text a bit, by making it all lowercase.

```
>>> mark_lower = mark_string.lower()
>>> mark_lower[:60]
'1:1 initium evangelii jesu christi, filii dei.\r\n 1:2 sicut s'
```

134 **Exercise 6.** *How would you assign the name mark_upper to the uppercased text of Mark?*
 135 □

Solution.

```
>>> mark_upper = mark_string.upper()
>>> mark_upper[:60]
'1:1 INITIUM EVANGELII JESU CHRISTI, FILII DEI.\r\n 1:2 SICUT S'
```

□

136 The next step in extracting the words is to get rid of a bunch of characters that
 137 we aren't interested in – the chapter and verse markers for instance.

```
>>> mark_simple = mark_lower.translate(None, '0123456789:')
>>> mark_simple[:60]
' initium evangelii jesu christi, filii dei.\r\n sicut scriptu'
```

138 There are other characters we may want to remove, punctuation and newlines and
 139 such, so let's redo the process with a broader set of characters to exclude.

```
>>> mark_simple = mark_lower.translate(None, '\n\r,.;\|/')(?'0123456789:')
>>> mark_simple[:60]
' initium evangelii jesu christi filii dei sicut scriptum es'
```

Finally, let's get rid of any extraneous `WHITESPACE` – the nonprinting layout characters like spaces, tabs, and newlines – at the start and end of the string. 141


```
>>> mark_simple = mark_simple.strip()
>>> mark_simple[:60]
'initium evangelii jesu christi filii dei sicut scriptum est'
```

Now, we can split the string into the component words at the whitespace that separate the words. 142
143

```
>>> mark_words = mark_simple.split()
>>> mark_words[:7]
['initium', 'evangelii', 'jesu', 'christi', 'filii', 'dei', 'sicut']
```

Let's encapsulate this whole process of turning a file into the list of words by defining a *function* that carries out that process. 144
145

```
>>> def words_normed(filename):
...     return ' '.join(open(filename).readlines()) \
...             .lower() \
...             .translate(None, '\n\r,.;\|\/[]()?@123456789') \
...             .strip() \
...             .split()
... 
```

 The backslashes at the end of each of the lines are there to notify Python that the expression is not at that point finished, so that Python provides the opportunity to type some more input. Without the backslashes, Python would have gone ahead and evaluated the expression after the second line. 146
147
148
149
150

Now we can do that for several different documents. 151

```
>>> matthew = words_normed('Matthew.txt')
>>> mark = words_normed('Mark.txt')
>>> luke = words_normed('Luke.txt')
>>> john = words_normed('John.txt')
```

To make sure it worked, let's look at the first few words of each. 152

```
>>> matthew[:7]
['liber', 'generationis', 'jesu', 'christi', 'filii', 'david', 'filii']
>>> mark[:7]
['initium', 'evangelii', 'jesu', 'christi', 'filii', 'dei', 'sicut']
>>> luke[:7]
['quoniam', 'quidem', 'multi', 'conati', 'sunt', 'ordinare', 'narrationem']
>>> john[:7]
['in', 'principio', 'erat', 'verbum', 'et', 'verbum', 'erat']
```

Let's look at some contiguous word sequences from the gospels. Here's the third through fifth words in Mark. 153
154


```
>>> mark[2:5]
['jesu', 'christi', 'filii']
```


155 (Even though we want the third through fifth words, we use the numeric indices
156 2 and 5. You'll see why later in Section 8.1.)

157 How about generating a whole series of such three word sequences? Contiguous
158 sequences of n words in a document are called n -grams; in the case where n is 3,
159 they are called trigrams. Here are the first ten trigrams in Mark.

```
>>> mark10trigrams = [mark[i:i+3] for i in range(10)]
>>> for trigram in mark10trigrams:
...     print trigram
...
['initium', 'evangelii', 'jesu']
['evangelii', 'jesu', 'christi']
['jesu', 'christi', 'filii']
['christi', 'filii', 'dei']
['filii', 'dei', 'sicut']
['dei', 'sicut', 'scriptum']
['sicut', 'scriptum', 'est']
['scriptum', 'est', 'in']
['est', 'in', 'isaia']
['in', 'isaia', 'propheta']
```

160 We can define a process to generate a list of *all* of the trigrams in a list of words.

```
>>> def ngrams(lst, N=3):
...     return [lst[i:i+N] for i in range(len(lst)-N+1)]
... 
```

161  The argument specification $N=3$ means that the second argument
162 named N is **OPTIONAL**, and if it is not provided, a default value of 3 will be
163 used as its value. Thus `ngrams` by default computes trigrams, but can also
164 be used to compute n -grams for other values of n if desired.

OPTIONAL ARGUMENTS

165 **Exercise 7.** Why is the range limit $\text{len}(lst)-N+1$ rather than just $\text{len}(lst)$? What is
166 the point of the extra arithmetic? Hint: Try it with just $\text{len}(lst)$ and see what happens.

167 □

Solution. In a list of k items, there aren't k n -grams, one starting at each item, because the last few items in the list don't have enough items following them to make up a full n -gram. In fact, there are $n - 1$ of these leftover items. So the range of starting indices to use should subtract off these $n - 1$ items; it should end at $k - (n - 1)$, that is, $k - n + 1$, leading to the $-N+1$ in the code. □

168 Let's test it on Mark again, printing the first few trigrams found to verify that it
169 worked.

```

>>> mark_3grams = ngrams(mark)
>>> for trigram in mark_3grams[:5]:
...     print trigram
...
['initium', 'evangelii', 'jesu']
['evangelii', 'jesu', 'christi']
['jesu', 'christi', 'filii']
['christi', 'filii', 'dei']
['filii', 'dei', 'sicut']

```

For completeness, we can generate the trigrams in the other gospels as well. 170

```

>>> matthew_3grams = ngrams(matthew)
>>> luke_3grams = ngrams(luke)
>>> john_3grams = ngrams(john)

```

One way to measure the similarity of two documents is to examine what trigrams 171
(or other n -grams) they have in common. We start by defining the intersection of 172
two lists, that is, the items they have in common: 173

```

>>> def intersect(list1, list2):
...     return [item
...             for item in list1
...             if item in list2]
...

```

Now we can find all of the trigrams in common between Matthew and Mark: 174

```

>>> common_matthew_mark = intersect(matthew_3grams, mark_3grams)
>>> for common in common_matthew_mark[:5]:
...     print common
...
['jesu', 'christi', 'filii']
['quod', 'est', 'interpretatum']
['cum', 'illo', 'et']
['principes', 'sacerdotum', 'et']
['at', 'illi', 'dixerunt']

```

How many such common trigrams are there? 175

```

>>> len(common_matthew_mark)
1906

```

That's about 18 percent of the Mark trigrams. 176

Exercise 8. *Knowing the raw count of common n -grams may not be as useful as knowing 177
the proportion of common n -grams. How can you calculate the proportion of the Mark 178
trigrams that are also found in Matthew? □ 179*

Solution. *About 18 percent:*

```
>>> 100 * len(common_matthew_mark) / len(mark_3grams)
18
```

Python rounds to the nearest integer when performing integer arithmetic. To get more precision, we can convert one of the terms from an integer to a floating point number:

```
>>> 100.0 * len(common_matthew_mark) / len(mark_3grams)
18.48152816833123
```

□

180 Is that a lot? We can compare it against the proportion of trigrams found in some
181 other more or less unrelated Latin document. Let's use the *Vita Sancti Germani*.

```
>>> vsg_3grams = ngrams(words_normed('vsg.txt'))
>>> len(intersect(mark_3grams, vsg_3grams))
13
```

182 The 13 common trigrams accounts for only 0.13 percent. So (unsurprisingly) Mark
183 looks to be extremely similar to Matthew.

184 Let's make a table that shows how similar the gospels are to each other (at least
185 as measured by common trigrams).

```
>>> gospels = {'Matthew': matthew_3grams,
...           'Mark': mark_3grams,
...           'Luke': luke_3grams,
...           'John': john_3grams}
>>> N = 3
>>> for (g1, w1) in gospels.items():
...     for (g2, w2) in gospels.items():
...         print "{:10s} {:10s} {:10.3%}"\
...             .format(g1, g2,
...                     float(len(intersect(w1, w2)))
...                     / (len(w1) - N + 1))
...
Matthew Matthew 100.012%
Matthew Luke 12.985%
Matthew John 2.586%
Matthew Mark 11.517%
Luke Matthew 11.351%
Luke Luke 100.011%
Luke John 2.206%
Luke Mark 7.553%
John Matthew 3.286%
John Luke 3.485%
John John 100.014%
John Mark 3.116%
```

Mark	Matthew	17.127%
Mark	Luke	12.220%
Mark	John	3.065%
Mark	Mark	100.019%

Exercise 9. Which of the gospels is the outlier? That is, which is the most different from all the others? 186

□ 187

Solution. Again unsurprisingly, Matthew, Mark, and Luke are all relatively similar to each other, with common trigrams percentages in the double digit percentages. John, on the other hand, is quite different, sharing only a few percent of trigrams with the others. Still, John has about ten times as many shared trigrams as the gospels do with the Vita Sancti Germani. □

Exercise 10. What about common 5-grams? Generate the same table but for 5-grams. □ 188

Solution. All that needs to change is the value for N .

```
>>> N = 5
>>> def common_grams(list1, list2, N=3):
...     return intersect(ngrams(list1, N), ngrams(list2, N))
...
>>> for (g1, w1) in gospels.items():
...     for (g2, w2) in gospels.items():
...         print "{:10s} {:10s} {:.10.3%}"\
...             .format(g1, g2,
...                     float(len(common_grams(w1, w2, N)))
...                     / (len(w1) - N + 1))
...
Matthew Matthew 100.000%
Matthew Luke 2.538%
Matthew John 0.006%
Matthew Mark 1.988%
Luke Matthew 2.300%
Luke Luke 100.000%
Luke John 0.011%
Luke Mark 0.951%
John Matthew 0.007%
John Luke 0.014%
John John 100.000%
John Mark 0.028%
Mark Matthew 3.191%
Mark Luke 1.668%
```

<i>Mark</i>	<i>John</i>	<i>0.039%</i>
<i>Mark</i>	<i>Mark</i>	<i>100.000%</i>

□

Part 2. Programming from first principles

189

The first part of these notes should have given you an idea of how even a few lines of Python code can accomplish some serious textual analysis. But to really understand how to program, so that you can generate effective code directly and not merely program by analogy, you need to understand the first principles of the programming language. In this part, we present some of these first principles for Python in a graded manner with interspersed exercises.

190

191

192

193

194

195

196

4. PYTHON DOCUMENTATION


197 These notes are not self-contained – on purpose. Python is a large language,
198 with many built-in functions and add-on modules for doing all kinds of things.
199 All are well documented at the python.org web site. You’ll want to get in the habit
200 of heading there to look up aspects of the language that you need help with.

201 Here are some especially important bits:

- 202 • There is a tutorial on the language at [https://docs.python.org/2/
203 tutorial/index.html](https://docs.python.org/2/tutorial/index.html), which you may find complementary to these notes.
204 It does assume a bit of programming background.
- 205 • The language reference manual is at [https://docs.python.org/2/
206 reference/index.html](https://docs.python.org/2/reference/index.html).
- 207 • The Python standard library and modules are described at [https://docs.
208 python.org/2/library/index.html](https://docs.python.org/2/library/index.html). We use some of these below, for
209 instance, standard functions like `sorted` and the `pprint` module.

5. THE PYTHON INTERPRETER 210

INTERPRETER A Python **INTERPRETER** allows you to specify calculations as Python expressions or programs and calculates the result of those specifications. You type Python commands and expressions into the interpreter, and the interpreter executes the commands and calculates the values of the expressions printing a representation of the calculated values. 211-215

COMMAND  We distinguish commands and expressions. **COMMANDS** are executed for their side effects. **EXPRESSIONS** are executed for their values (though they may have side effects as well). The difference is revealed by the interpreter: after entering a command, no output is printed by the interpreter; after entering an expression, an output is printed, namely, the expression's value. 216-221

Here is a simple example of using a Python interpreter. The user's input is on the lines beginning '>>>' (or '.' for lines continuing a single input) and the interpreter's output immediately follows. 222-224

```
>>> 3 + 4 * 5
23
```

We've used the + symbol for addition and * for multiplication. You can find a larger listing of arithmetic operators at https://en.wikibooks.org/wiki/Python_Programming/Basic_Math. 225-227

Exercise 11. Enter the expression $3 + 4 * 5$ into the Python interpreter and verify that it works like it should. 228-229

Exercise 12. Use the Python interpreter to determine the values of the following arithmetic expressions: 230-231

- (1) $4/4 - 4/4$ 232
- (2) $\frac{4+4}{4+4}$ 233
- (3) $\frac{4*4}{4+4}$ 234

This exercise is inspired by the "four fours" puzzle, which involves constructing arithmetic expressions for each positive integer using four fours combined however you want. Feel free to generate more examples and use Python to verify them for you. 235-237

Solution.

```
>>> 4/4 - 4/4
0
>>> (4 + 4) / (4 + 4)
1
>>> (4 * 4) / (4 + 4)
2
```




6. EXPRESSIONS AND NESTING

238

One of the deep truths of linguistics, known since the time of Pāṇini in the fourth century BCE, is that the expressions of language have hierarchical structure. The recovery of that structure used to be a typical subject matter taught to students in “grammar school” through the exercise of sentence diagramming.

For instance, in the sentence “Some new cakes are nice” (the first proposition from Lewis Carroll’s *The Game of Logic*), the whole sentence is constituted of two primary parts, marked here:

$\overbrace{\text{Some new cakes are nice}}^{\hspace{1.5cm}}$

which parts in turn can, extending the structural hierarchy, be broken down further:

$\overbrace{\overbrace{\text{Some}}^{\hspace{0.5cm}} \overbrace{\text{new cakes are nice}}^{\hspace{1.5cm}}}^{\hspace{2.5cm}}$

And of course, the meaning of the utterance is determined in part by that structure. This fact accounts for the humor (of a sort) found in structurally ambiguous sentences:

$\overbrace{\text{I shot an elephant in my pajamas}}^{\hspace{2.5cm}}$
 $\overbrace{\text{I shot an elephant in my pajamas}}^{\hspace{2.5cm}}$

Python expressions, like the utterances of natural language, have structure as well. In the expression $3 + 4 * 5$, there is a subexpression $4 * 5$, but $3 + 4$ is not a subexpression. That is, the structure is

$\overbrace{3 + \overbrace{4 * 5}}^{\hspace{1.5cm}}$

and not

$\overbrace{\overbrace{3 + 4}^{\hspace{0.5cm}} * 5}^{\hspace{1.5cm}}$

Since $4 * 5$ is 20, the whole expression is 23, and not 35.

Just as the hierarchical structure of a natural-language utterance is crucial to deriving its meaning, so is the hierarchical structure of a Python expression crucial to deriving its.

264

7. VARIABLES AND THE NAMING OF VALUES

265 We can name the results of computations for later use. These names are called
266 **VARIABLES**. Variables are tokens made up of alphabetic characters, digits, and the **VARIABLES**
267 underscore (`_`), and not starting with a digit. By convention, variable names are
268 typically composed of lowercase letters, using the underscore to separate “words”
269 that make up the name.

270 **Exercise 13.** Which of these are not valid variable names in Python?

271 (1) `matthew`

272 (2) `sanctus_germanus`

273 (3) `1_samuel`

274 (4) `__name__`

275 (5) `n-grams`

276

□

Solution. All but examples 3 and 5 are valid variable names in Python. □

277 Here’s an example of the use of a variable (`large_square`) to name a value and
278 then using that value in later computations.


```
>>> large_square = 128 ** 2
>>> large_square / 2
8192
```

279 The first line constitutes an **ASSIGNMENT**; it assigns the name given on the left side **ASSIGNMENT**
280 of the `=` operator to the value specified by the expression on the right side. Thus
281 the variable `large_square` names the value 16384. Assignments are executed for
282 their *effect*, not their *value*. For that reason, the interpreter doesn’t print anything
283 after this line. (Don’t be confused. The `=` does not mean “is equal to”, as it does in
284 standard mathematical notation. It’s a kind of command, not a statement of fact.)

285 The second line then uses that variable by dividing its value by 2. The interpreter
286 prints the value specified by that last expression.

8. SEQUENCE DATA TYPES 287

It is conventional in defining programming languages to carefully distinguish the different types of data that programs can manipulate. We've seen one DATA TYPE already – numbers.

 In actuality, Python treats numbers as falling into a set of different data subtypes: integers, real numbers, complex numbers, each of which operates slightly differently.

Our primary application in these notes is analysis of text. We will therefore move quickly to look at the data type most useful for representing text, namely, strings. Strings are a kind of sequence data type; a string is essentially a sequence of characters. In fact, Python provides several different data types for sequences: strings of course, but also lists and tuples. These sequence data types share many properties, so we introduce them together.

8.1. Lists. The Python LIST data type is used to represent sequences of other data objects, sequences that can be adjusted in various ways, for instance, by adding or removing elements. The notation for lists is to place the individual listed objects, separated by commas and surrounded by brackets.

```
>>> [1, 2, 3]
[1, 2, 3]
>>> ex_list = [1, 4, 1, 5, 9, 2, 6]
>>> ex_list
[1, 4, 1, 5, 9, 2, 6]
```

Each item in a list has its own POSITION in the list. The individual items within a list can be extracted by INDEXING them based on their respective positions. We use the indexing notation `·[·]`. For instance, to retrieve the fifth item from `ex_list`, we use the notation `ex_list[4]`.

```
>>> ex_list[4]
9
```

Notice that the value of this expression is indeed the fifth item in the list, the number 9.

Why use the index 4 for the fifth item? Because we think of the positions as being numbered *starting from index zero*. Alternatively, you can think of the indices as numbering the points *between* the items, starting with zero, like in this picture.

1	4	1	5	9	2	6	
0	1	2	3	4	5	6	7

Under this conception, the indexing `ex_list[4]` extracts the item *following* position 4, that is, the fifth item.

316 **8.2. Sequence lengths.** We may want to know how many items there are in one
 317 of these kinds of sequences. We use the `len` function to calculate the length of a
 318 list. (We'll have much more to say about functions shortly, starting in Section 9.)

LEN FUNCTION

```
>>> len(ex_list)
7
```

319 Since the length of a list is a number, you can operate on it as you would any
 320 other number, applying arithmetic operations to it for instance.

```
>>> len(ex_list) * 2
14
```

321 **8.3. Strings.** We'll use the `STRING` data type for representing text. Strings in Python
 322 are specified by enclosing a sequence of characters within matching string `DELIM-`
 323 `ITERS`, such as single quotes.

STRING


DELIMITERS

```
>>> 'sanctus Germanus'
'sanctus Germanus'
```

324 Strings can be specified with other delimiters, such as double quotes, or triple
 325 double or single quotes.

```
>>> "This example uses double quotes"
'This example uses double quotes'
>>> """Triple quotes are
... often used for
... multi-line strings."""
'Triple quotes are\noften used for\nmulti-line strings.'
```

326 Note that Python always prints out the strings using the single quote delimiter.

327  This last string has some `NEWLINE` characters in it. They're specified
 328 with the `'\n'` characters. See Section 12 below.

NEWLINE

329 Strings can be concatenated using the `+` operator.

```
>>> "This" + ' that'
'This that'
```

330 (We can freely combine strings specified with the different delimiters.)

331 Like all data values, strings can be named by variables.

```
>>> ex_string = " be as it were as it"
>>> "Let it" + ex_string * 2 + " were"
'Let it be as it were as it be as it were as it were'
```

332 Interesting how Python uses the “multiplication” operator `*` for repeating
 333 strings, no? This “arithmetic” on strings works for lists as well.

```
>>> motto = [ "nihil", "agere", "delectat" ]
>>> motto
['nihil', 'agere', 'delectat']
```

```

>>> len(motto)
3
>>> motto + motto
['nihil', 'agere', 'delectat', 'nihil', 'agere', 'delectat']
>>> len(motto * 2) - len(motto) * 2
0
>>> ex_string
' be as it were as it'
>>> len(ex_string)
20

```

Exercise 14. *What will Python print in response to each of the following inputs?*

334

```

ex_list = [ "agere", "delectat", "nihil" ]
ex_list[2] + ex_list[0] + ex_list[1]
ex_list[2] + " " + ex_list[0] + " " + ex_list[1]
ex_list[1][1] + ex_list[2][2]
len(ex_list * 2) - len(ex_list) * 2

```

□ 335

Solution. *Python prints the following:*

```

>>> ex_list = [ "agere", "delectat", "nihil" ]
>>> ex_list[2] + ex_list[0] + ex_list[1]
'nihilageredelectat'
>>> ex_list[2] + " " + ex_list[0] + " " + ex_list[1]
'nihil agere delectat'
>>> ex_list[1][1] + ex_list[2][2]
'eh'
>>> len(ex_list * 2) - len(ex_list) * 2
0

```

□

8.4. Substrings. Strings, like lists, are sequences – in particular, sequences of characters. We can do many of the same operations on strings that we can on lists. For instance, we can extract a character from a string using the same indexing notation `[·]`. To retrieve the fifth character from `ex_string`, we use the notation `ex_string[4]`.

336

337

338

339

340

```

>>> ex_string = "sanctus Germanus"
>>> ex_string[4]
't'

```

341 As before we think of the indices as numbering the points *between* the characters,
 342 starting with zero, like in this picture.

```
343   s   a   n   c   t   u   s   _   G   e   r   m   a   n   u   s
      0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16
```

344 Under this conception, the indexing `ex_string[4]` extracts the character *following*
 345 string position 4, that is, the fifth character.

346 Substrings can be specified by a SLICING notation, similar to the indexing notation SLICING
 347 but providing both starting and ending positions within the full string, separated
 348 by a colon. For instance, to extract the substring between string positions 2 and 6
 349 (that is, the second through fifth characters):

```
>>> ex_string[2:6]
'nctu'
```

350 **Exercise 15.** *What strings are specified by the following Python expressions? Recall the*
 351 *value of `ex_string` defined above.*

- 352 (1) `ex_string[0:3]`
- 353 (2) `ex_string[3]`
- 354 (3) `ex_string[3:4]`
- 355 (4) `ex_string[3:3]`
- 356 (5) `ex_string[3:2]`
- 357 (6) `ex_string[:4]`
- 358 (7) `ex_string[4:]`
- 359 (8) `ex_string[4:-3]`
- 360 (9) `ex_string[3:100]`
- 361 (10) `ex_string[8:0:-1]`
- 362 (11) `ex_string[::-1]`

363 *We really haven't given enough detail about how the indexing notation works to determine*
 364 *all of these, so you'll have to experiment to figure them out.* □

Solution. *Here's what happens:*

```
>>> ex_string[0:3]
'san'
>>> ex_string[3]
'c'
>>> ex_string[3:4]
'c'
>>> ex_string[3:3]
''
>>> ex_string[3:2]
''
```

```

>>> ex_string[:4]
'sanc'
>>> ex_string[4:]
'tus Germanus'
>>> ex_string[4:-3]
'tus Germa'
>>> ex_string[3:100]
'ctus Germanus'
>>> ex_string[8:0:-1]
'G sutcna'
>>> ex_string[::-1]
'sunamreG sutcnas'

```

□

Exercise 16. *Based on your experiments with the previous exercise, how would you reverse a string in Python, that is, generate a string with the characters in the reverse order?* 365

□ 366

Exercise 17. *This method that allows extracting substrings from strings also allows extracting sublists from lists. Suppose the variable `vsg_list` names the value `['sanctus', 'Germanus', 'abba', 'et', 'martyr']`. How would you extract all but the first and last elements from the list?* 367

□ 368

□ 369

□ 370

Solution.

```

>>> vsg_list = ['sanctus', 'Germanus', 'abba', 'et', 'martyr']
>>> vsg_list[1:len(vsg_list)-1]
['Germanus', 'abba', 'et']

```

□

Exercise 18. *How would you extract the final two elements from `vsg_list`, without recourse to prior knowledge of the number of items in the list?* 371

□ 372

Solution.

Here are some methods:

```

>>> vsg_list[len(vsg_list)-2:len(vsg_list)]
['et', 'martyr']
>>> vsg_list[len(vsg_list)-2:]
['et', 'martyr']
>>> vsg_list[-2:]
['et', 'martyr']

```

□

TUPLES 8.5. **Tuples.** The final sequence data type we'll cover is the `TUPLE`. The name
 374 derives from the suffix seen in *quintuple*, *sextuple*, *septuple*, and the like.

375 A tuple in Python is specified like a list, with multiple elements separated by
 376 commas, but without the surrounding brackets. It is conventional (though not
 377 always required) to use grouping parentheses around the elements of the tuple.
 378 Here are a list and its corresponding tuple:

```
>>> ['sanctus', 'Germanus', 'abba', 'et', 'martyr']
['sanctus', 'Germanus', 'abba', 'et', 'martyr']
>>> ('sanctus', 'Germanus', 'abba', 'et', 'martyr')
('sanctus', 'Germanus', 'abba', 'et', 'martyr')
```

379 Lists can be converted to tuples using the `tuple` function, and tuples to lists using
 380 the `list` function.

TUPLE FUNCTION

LIST FUNCTION

```
>>> vsg_list
['sanctus', 'Germanus', 'abba', 'et', 'martyr']
>>> vsg_tuple = tuple(vsg_list)
>>> vsg_tuple
('sanctus', 'Germanus', 'abba', 'et', 'martyr')
>>> list(vsg_tuple)
['sanctus', 'Germanus', 'abba', 'et', 'martyr']
```

381 Like lists, tuples can be indexed, sliced, and (as we'll see later) iterated over.

```
>>> vsg_tuple[2]
'abba'
>>> vsg_tuple[-2:]
('et', 'martyr')
```

382 Since tuples and lists are so similar, why do both exist in the language? The
 383 distinction is a bit arcane. Lists are stored internally in such a way that they can be
 384 modified – items replaced, added, or removed. Tuples do not allow modification
 385 once created. Here's an example of the difference:

```
>>> vsg_list[2] = vsg_list[1]
>>> vsg_tuple[2] = vsg_tuple[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

386 The attempt to modify the tuple causes an error. Python won't allow it.

387 Data types that don't allow values to be modified are termed `HASHABLE`. (Num-
 388 bers and strings are also hashable data types.) Tuples (and other hashable data
 389 types) are thus useful in contexts in which it is important that a data object never
 390 change. For instance, in storing information by associating it with a special "key",
 391 it is important that the key not be changed; otherwise, the value associated with

HASHABLE

that key would become inaccessible. For that reason, keys are restricted to come from hashable data types such as tuples, as we will see when looking at dictionaries in Section 18.

Since it is the comma operator separating the elements that makes clear that a tuple is being specified, how do we specify a tuple of one element, or even zero elements? The zero-element `EMPTY TUPLE` is specified by parentheses enclosing nothing, `()`. A `SINGLETON TUPLE` uses a trailing comma within the parentheses, for instance, `(1,)`.

`EMPTY TUPLE`
`SINGLETON TUPLE`

400

9. FUNCTIONS

401 Data – numeric or string values, and all the other types of data that Python makes
 402 available – are manipulated through the application of `FUNCTIONS`, engines that take
 403 inputs, called `ARGUMENTS`, and transform them into an output, the `RESULT`. We’ve
 404 seen examples of such functions already: the arithmetic and string operators like
 405 `+` and `*`, indexing operators like `[:]`. These are special built-in functions that are
 406 invoked via special “idiomatic” notations. The arithmetic operators, for instance,
 407 are written infix, as, e.g., `1 + 2`, and the indexing operator is written with brackets.

FUNCTIONS

ARGUMENTS

RESULT

408 But in general, Python uses two notations that are more uniform for applying a
 409 function to its arguments.

410 (1) *Mathematical notation*: Mimicking a traditional `MATHEMATICAL NOTATION` the
 411 origin of which is attributed variously to Leibniz and Euler, a function,
 412 say `f`, applied to its arguments is notated by placing the comma-separated
 413 arguments after the function in parentheses, viz.,

MATHEMATICAL NOTATION

414


$$f(\langle arg1 \rangle, \langle arg2 \rangle, \dots) \quad .$$

415 (2) *Object notation*: A second notation, `OBJECT NOTATION`, derived from conven-
 416 tions used in so-called object-oriented programming languages, places the
 417 function *after* its first argument separated by a dot, with all other arguments
 418 following as in the mathematical notation, viz.,


OBJECT NOTATION

419

$$\langle arg1 \rangle.f(\langle arg2 \rangle, \dots) \quad .$$

420  The latter notation makes more sense once Python’s status as an
 421 object-oriented language is understood, but in the interest of introducing
 422 the least language for our purposes, we introduce it as just a fixed idiom.

423 Any given function uses either the first or second notation, in much the same
 424 way that any given Latin verb inflects as per one of a small set of conjugations. You
 425 might think of functions that use the mathematical notation as “first conjugation”
 426 functions and those using object notation “second conjugation”.

427  There are actually further “conjugations”, for infix operators like the
 428 `+` in `3 + 4` and prefix operators like the `-` in `- 5`. The operators specify
 429 functions, but they are not called using the mathematical notation, that
 430 is, `+(3, 4)` or `-(5)` (though the latter will work by happenstance since the
 431 parenthesized part will be treated as a grouping construct, not as part of
 432 the function application syntax).

433 As it turns out, Python makes available in the `operator` package equiv-
 434 alents to all such infix and prefix operators as regular functions called with
 435 the mathematical notation. For instance,

```

>>> import operator
>>> 3 + 4
7
>>> operator.add(3, 4)
7
>>> - 5
-5
>>> operator.neg(5)
-5

```

LEN FUNCTION An example of the mathematical notation is the built-in `len` function, which takes a single argument and returns its length. It can be applied to any kind of list, and in particular, to strings, for instance,

```

>>> len(ex_string)
16

```

Exercise 19. What are the values of the following Python expressions?

- (1) `ex_string[0:len(ex_string)]`
- (2) `ex_string[1:len(ex_string)]`
- (3) `ex_string[0:len(ex_string)-1]`

Can you find simpler ways of getting the same values?

Solution. The solution:

```

>>> ex_string[0:len(ex_string)]
'sanctus Germanus'
>>> ex_string[1:len(ex_string)]
'anctus Germanus'
>>> ex_string[0:len(ex_string)-1]
'sanctus Germanu'

```

SORTED FUNCTION Another useful function is the built-in `sorted` function, which takes a single argument representing a sequence (such as a list or string) and returns a corresponding object representing the elements of its argument in sorted order.

```

>>> sorted([3, 1, 4, 1, 5])
[1, 1, 3, 4, 5]

```

Exercise 20. Recall the value of `motto`, which is `['nihil', 'agere', 'delectat']`. What do the following Python expressions return?

- (1) `sorted(motto)`
- (2) `sorted(motto[0])`
- (3) `sorted(motto)[0]`

452

□

Solution. *The solution:*

```
>>> motto
['nihil', 'agere', 'delectat']
>>> sorted(motto)
['agere', 'delectat', 'nihil']
>>> sorted(motto[0])
['h', 'i', 'i', 'l', 'n']
>>> sorted(motto)[0]
'agere'
```

□

453 It is often useful to generate a list of sequential numbers. We'll see use examples
 454 later. The range function serves that purpose. Its two arguments specify the start
 455 and end of the range; the included numbers are obtained by starting with the first,
 456 and incrementing repeatedly until the second number is reached (or surpassed). If
 457 the first argument is left off, it is assumed to be 0. If a third argument is added, it
 458 is taken to be the increment used between numbers.

RANGE FUNCTION

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 10, 2)
[1, 3, 5, 7, 9]
>>> range(10, 0, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

459 **Exercise 21.** *Use the range function to generate the following lists:*

```
[10, 11, 12]
[10]
[2, 4, 6, 8]
[3, 2, 1, 0, -1, -2, -3]
[]
```

460

□

Solution. *Here's one way:*

```
>>> range(10, 13)
[10, 11, 12]
>>> range(10, 11)
[10]
```

```
>>> range(2, 10, 2)
[2, 4, 6, 8]
>>> range(3, -4, -1)
[3, 2, 1, 0, -1, -2, -3]
>>> range(1,1)
[]
```

□

COUNT FUNCTION

As a final example, we consider the `count` function (which uses the object notation), which counts the number of occurrences of its second argument as elements of its first list argument.

```
>>> motto.count('nihil')
1
>>> motto.count('ipsum')
0
>>> motto[0].count('i')
2
```

461

462

463

464

10. WORDS, TYPES, AND TOKENS

465 As we turn to processing of text, some standard terminology about words is
 466 useful, starting with the word “word” itself. The question of what is a word is
 467 itself somewhat fraught. For the time being, we’ll just consider the words in a text
 468 to be the maximal sequences of alphabetic characters separated by whitespace.
 469 (As it turns out, this is an exceptionally poor definition, but sufficient for the time
 470 being.)

471 We distinguish word types from word tokens. A text is made up of a series of
 472 word `TOKENS`. Each word token belongs to a word `TYPE`. Consider the text corpus in
 473 Figure 1, a sentence from Gertrude Stein’s 1929 poem “An Acquaintance With De-
 474 scription” (Stein, 1929). This corpus has 225 word tokens (ignoring punctuation),
 475 which are instances of just eight word types (if we conflate upper and lower case).
 476 The eight types, in decreasing order of frequency, are: “be”, “to”, “it”, “sure”, “let”,
 477 “mine”, “when”, “is”. Each of these word types has several occurrences as tokens
 478 in the poem.

`TOKENS`
`TYPE`

Let it be when it is mine to be sure let it be when it is mine when it is mine
 let it be to be sure when it is mine to be sure let it be let it be let it be to be
 sure let it be to be sure when it is mine to be sure let it to be sure when it
 is mine let it be to be sure let it be to be sure to be sure let it be to be sure
 let it be to be sure to be sure let it be to be sure let it be to be sure let it be
 to be sure let it be mine to be sure let it be to be sure to be mine to be sure
 to be mine to be sure to be mine let it be to be mine let it be to be sure to
 be mine to be sure let it be to be mine let it be to be sure let it be to be sure
 to be sure let it to be sure mine to be sure let it be mine to let it be to be
 sure to let it be mine when to be sure when to be sure to let it to be sure to
 be mine.

FIGURE 1. A sentence from Stein’s “An Acquaintance with Description” (1929).

11. FILES

479

11.1. **Strings from files.** Typing in the kinds of long strings we'll be analyzing, entire books in some cases, is painful. Better to store the text in a text file and load that file into Python. Let's imagine that we have a file called "stein.txt" that contains the line from Figure 1. We want to read that file into Python so that we can operate with it.

480

481

482

483

484

We'll use an idiom to get the contents of a text file into a variable. The idiom is this:

485

486

```
<variable> = open(<filename>).readlines()
```

We are using two different functions in this idiom, the `open` function, invoked using the mathematical function notation, and the `readlines` function, invoked using the object notation. The `open` function takes a single string as an argument, and returns as value an object that designates the file with that name. The `readlines` function's first argument is a file designator (as returned by `open`), and since it takes no further arguments, the parentheses for the remaining arguments are empty. The function returns a list, each component of which is a string containing a line of the file that was read in.

487

488

489

490

491

492

493

494


To read the Stein poem in, we can therefore use:

495

```
>>> stein_lines = open('stein.txt').readlines()
```

Now, let's examine what we've read in.

496

 Instead of just evaluating (and having the interpreter print the value of) `stein`, here we are "importing" a special "pretty-printing" facility, the `pprint` function, to print the value of `stein` in a more attractive manner.

497

498

499

PPRINT FUNCTION

```
>>> from pprint import pprint
```

```
>>> pprint(stein_lines)
```

```
['Let it be when it is mine to be sure let\n',
 'it be when it is mine when it is mine\n',
 'let it be to be sure when it is mine to\n',
 'be sure let it be let it be let it be to\n',
 'be sure let it be to be sure when it is\n',
 'mine to be sure let it to be sure when\n',
 'it is mine let it be to be sure let it\n',
 'be to be sure to be sure let it be to be\n',
 'sure let it be to be sure to be sure let\n',
 'it be to be sure let it be to be sure\n',
 'let it be to be sure let it be mine to\n',
 'be sure let it be to be sure to be mine\n',
 'to be sure to be mine to be sure to be\n',
 'mine let it be to be mine let it be to\n',
```



```
'be sure to be mine to be sure let it be\n',  
'to be mine let it be to be sure let it\n',  
'be to be sure to be sure let it to be\n',  
'sure mine to be sure let it be mine to\n',  
'let it be to be sure to let it be mine\n',  
'when to be sure when to be sure to let\n',  
'it to be sure to be mine.\n']
```

500 **Exercise 22.** *Read into Python the contents of a text file for some document you are*
501 *interested in. The Vita Sancti Germani comes to mind.* □

12. SPECIAL CHARACTERS

502

Let's examine the first line of the poem.

503

```
>>> stein_lines[0]
'Let it be when it is mine to be sure let\n'
```

It's a string of 41 characters.

504

Exercise 23. *How could you verify that length? Do it.*

□ 505

Solution. *To compute the length of the first line:*

```
>>> len(stein_lines[0])
41
```

□

Exercise 24. *Use Python to extract the last character from the first line of the poem.*

□ 506

Solution. *The most direct method to extract the last character is to compute the index of the last character in the first line and extract the character at that index:*

```
>>> stein_lines[0][len(stein_lines[0]) - 1]
'\n'
```

This method is inelegant in twice calculating `stein_lines[0]`. More “pythonic” is to take advantage of negative indices:

```
>>> stein_lines[0][-1]
'\n'
```

□

The last character of the first line is the newline character, which unlike all the “normal” characters, is notated with an ESCAPE SEQUENCE, a backslash followed by an n: `'\n'`. There are other escape sequences, used for characters that are otherwise hard to make clear in a printed representation, such as `'\t'` for the tab character or `'\''` for the single quote character (which is otherwise hard to put in a single-quoted string without prematurely terminating the string).

ESCAPE SEQUENCE

507

508

509

510

511

512

Exercise 25. *How would you notate the single-quoted string containing the possessive form of your first name?*

□ 514

Solution. *Here's how I would do it, with single-quoted and double-quoted strings:*

```
>>> 'Stuart\'s'
"Stuart's"
>>> "Stuart's"
"Stuart's"
```

□

515 13. SPLITTING AND JOINING STRINGS

516 We introduce some useful string manipulation functions. To concatenate to-
 517 gether a list of strings to form a single string, use the `join` function that takes a
 518 separator string and a list of strings to join and combines the strings in the list
 519 together separated by the separator string.

JOIN FUNCTION

```
>>> ' '.join(['sanctus', 'Germanus'])
'sanctus Germanus'
```

520 **Exercise 26.** Use `join` to generate the following strings from the list of number strings
 521 `['1', '2', '3']`.

522 (1) `'1-2-3'`

523 (2) `'1, 2, 3'`

524 (3) `'123'`

525 (4) `'3, 2, 1'`

526 For the last problem, recall Exercise 16. For further extra credit, start from the list of
 527 numbers themselves `[1, 2, 3]`. Check out the functions `map` and `str`. □

Solution. This solution starts with the number list, converting it first to the number string list, thereby solving both parts of the exercise.

```
>>> numbers = [1, 2, 3]
>>> numbers = map(str, numbers)
>>> numbers
['1', '2', '3']
>>> '-'.join(numbers)
'1-2-3'
>>> ', '.join(numbers)
'1, 2, 3'
>>> ''.join(numbers)
'123'
>>> ', '.join(numbers[::-1])
'3, 2, 1'
```

□

528 The converse of the `join` function is the `split` function. Again, `split` takes two
 529 arguments in object notation. The first is the string to be split up into substrings
 530 and the second is a string that specifies where to split. Each occurrence of the
 531 second string in the first string generates a split point. To split at the spaces in the
 532 string, then, the second argument would be the string `' '`:

SPLIT FUNCTION

```
>>> line = "He told me you had been to her and mentioned me to him"
>>> line.split(' ')[0:5]
['He', 'told', 'me', 'you', 'had']
```

The splitting can occur at any substring we want:

533

```
>>> line.split(' me ')
['He told', 'you had been to her and mentioned', 'to him']
```

Exercise 27. *Extra credit: What is this line from?*

□ 534

LOWER FUNCTION

Exercise 28. Use Python's `lower` function (*inter alia*) to generate the list of word tokens in `line` but with all words in lower case. Step one: Click on the link in this exercise to go to the Python documentation on the `lower` function. While you're there, look around at the range of *other string-processing functions* that may come in handy some day.

535

536

537

□ 538

Solution.

```
>>> pprint(line.lower().split(' '))
['he',
 'told',
 'me',
 'you',
 'had',
 'been',
 'to',
 'her',
 'and',
 'mentioned',
 'me',
 'to',
 'him']
```

□

Exercise 29. Use Python to split the first line of Stein's poem into its separate word tokens, storing the resulting list of tokens in the variable `stein_words1`.

539

□ 540

Solution.

```
>>> stein_words1 = stein_lines[0].strip().split(' ')
>>> pprint(stein_words1)
['Let', 'it', 'be', 'when', 'it', 'is', 'mine', 'to', 'be', 'sure', 'let']
```

□


541 14. LIST COMPREHENSIONS

542 We've seen the notation for specifying a list **EXTENSIONALLY**, that is, by enumer- **EXTENSIONAL**
 543 ating its elements explicitly. Here for instance are the first letters of the first few
 544 words (the first eight, say) in the first line of the Stein poem, enumerated explicitly:

```
>>> first_letters = ['L', 'i', 'b', 'w', 'i', 'i', 'm', 't']
>>> first_letters
['L', 'i', 'b', 'w', 'i', 'i', 'm', 't']
```

545 It's much more elegant and less error-prone to let Python do the work for you.
 546 We use **LIST COMPREHENSIONS** for the task. List comprehensions allow specifying **LIST COMPREHENSIONS**
 547 a single generic list element computation that captures all of the elements of the
 548 list. It allows defining lists **INTENSIONALLY** rather than extensionally. The list **INTENSIONAL**
 549 comprehension notation is

```
[ <generic element> for <variable> in <list> ]
```

551  For the mathematically inclined, it may be useful to think of this
 552 notation as analogous to the familiar mathematical notation for defining
 553 sets intensionally, for example,

```
{ x2 | 0 ≤ x < 10 }
```

555 which defines the set containing the first 10 squares. The braces become
 556 brackets in Python, and the vertical bar becomes the word **for**, which
 557 separates the generic element x^2 on its left from the specification of the
 558 possible values of x on its right.

559 For the current example, each element of the list can be calculated as `word[0]`
 560 where `word` is one of the first few words in the first line of the poem. (Recall that
 561 the words in the first few lines in the poem are named by the variable `stein_words`
 562 from Exercise 29.)

```
>>> first_letters = [word[0] for word in stein_words1[0:8]]
>>> first_letters
['L', 'i', 'b', 'w', 'i', 'i', 'm', 't']
```

563 Here, the variable `word` takes on each element of the list `stein_words[0:8]`, and
 564 for each one, an element of the list is computed as `word[0]`.

565 **Exercise 30.** *Generate a list each element of which is a list of all of the word tokens in a*
 566 *line of the Stein poem.* □

Solution.

```
>>> line_words = [line.strip().split(' ') for line in stein_lines]
>>> pprint(line_words[0:3])
[['Let', 'it', 'be', 'when', 'it', 'is', 'mine', 'to', 'be', 'sure', 'let'],
```

```
['it', 'be', 'when', 'it', 'is', 'mine', 'when', 'it', 'is', 'mine'],
['let', 'it', 'be', 'to', 'be', 'sure', 'when', 'it', 'is', 'mine', 'to']]
```

□

Exercise 31. Generate a list named `stein_words` of all the word tokens in the Stein poem. 567

STRIP FUNCTION Make sure that all the words are lower case. You may find the `strip` function to be useful. 568

You should be able to get the following behavior: 569

```
>>> stein_words[6:12]
['mine', 'to', 'be', 'sure', 'let', 'it']
```

□ 570

Solution.

```
>>> stein_words = ' '.join([line.lower().strip('\n .')]
...                          for line in stein_lines]).split(' ')
>>> stein_words[6:12]
['mine', 'to', 'be', 'sure', 'let', 'it']
```

□

Exercise 32. Generate a list of the first 10 squares (0, 1, 4, 9, etc.). Hint: You'll want to recall the `range` function. 571

□ 572

Solution.

```
>>> [x**2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

□

573

15. SETS

574 Time to introduce another data type, the `SET`. A set is a compound data type; SET
 575 like the list, each set contains elements. But the elements of a set are unique. A set
 576 does not contain multiple tokens of the same value. You can create a set from a list
 577 with the `set` function. SET FUNCTION

```
>>> set([1, 2, 3])
set([1, 2, 3])
>>> set([1, 2, 3, 2, 1])
set([1, 2, 3])
>>> set('it was the best of times it was the worst of times'.split(' '))
set(['of', 'it', 'times', 'worst', 'the', 'was', 'best'])
```

578 As you can see, the printed representation for a set shows a list of the elements but
 579 still marks it as a set.


580 Many of the same functions that apply to lists apply to sets as well: `len` for
 581 counting the number of elements, `+` for combining two sets (taking their `UNION`), UNION
 582 etc.

583 **Exercise 33.** Use Python to calculate how many word types (not tokens) there are in the
 584 Stein poem. Ignore case distinctions. Hint: The answer is 8. The hint is to emphasize that
 585 the point of the exercise is the code, not the answer. □

Solution.

```
>>> len(set(stein_words))
8
```

□

586  The elements of a set can be of many types – numbers, strings, and
 587 tuples, in particular – but unfortunately not lists or sets. Only hashable
 588 data types are allowed.

16. CALCULATING WITH n -GRAMS

589

n -GRAMS We'll spend some time looking at n -GRAMS, contiguous sequences of n words. 590

UNIGRAM When n is 1, 2, or 3, we call them UNIGRAMS, BIGRAMS, and TRIGRAMS, respectively. 591

BIGRAM Here are some examples of trigrams built from the vocabulary seen in Gertrude 592

TRIGRAM Stein's poem: 593

(1) let it be 594

(2) it is mine 595

(3) it is sure 596

(4) to be sure 597

Problem 34. Generate a list of all of the trigram tokens in the Stein poem. You'll want to use the word list you generated in Exercise 31. 599

Solution.

```
>>> N = 3
>>> stein_ngrams = [stein_words[i:i+N]
...                 for i in range(len(stein_words)-N+1)]
>>> pprint(stein_ngrams[:4])
[['let', 'it', 'be'],
 ['it', 'be', 'when'],
 ['be', 'when', 'it'],
 ['when', 'it', 'is']]
```

Problem 35. How many times do each of the four sample trigrams above occur in the poem? If you resort to counting them yourself, go back to the beginning of these notes and start over. 602

Solution.

```
>>> [stein_ngrams.count(probe.split(' ')) for probe in
...   ['let it be', 'it is mine', 'it is sure', 'to be sure' ] ]
[24, 6, 0, 31]
```

Exercise 36. How many unique trigrams are there in the Stein poem? (You may want to look at the earlier discussion about hashable data types.) 604

Solution.

```
>>> len(set([' '.join(ngram) for ngram in stein_ngrams]))
37
```


605 17. DEFINING YOUR OWN FUNCTIONS

606 Functions like `len`, `sorted`, `count`, and the like can be fabulously useful. If
 607 there's a function that does just what you need, a single line of code can accomplish
 608 your purposes.

609 Sadly, there often is not a function tailor-made for your purposes. But you can
 610 write your own. Indeed, writing functions is the heart of computer programming
 611 (in spite of the fact that it took until page 41 to get to the topic).

612 In Python, you can define your own function of zero or more arguments using
 613 the **def** command. The notation is as follows: DEF COMMAND

```
def <function name>(<arguments>):
    <function body>
```

614 Within the body of the function, the **return** command generates the value to return RETURN COMMAND
 615 as the result of the function.

616 For example, here we define a function to calculate the first letter of a string.

```
>>> def first_letter(ex_string):
...     return ex_string[0]
... 
```

617 We can use this function by **CALLING** it just as we would a built-in function using FUNCTION CALL
 618 mathematical notation:

```
>>> first_letter('nihil')
'n'
>>> first_letter(stein_lines[0])
'L'
```

619 **Exercise 37.** Define and test a function that returns the last letter of the first word in a
 620 string. □

Solution.

```
>>> def last_of_first(ex_string):
...     return ex_string.split(' ')[0][-1]
... 
```

```
>>> last_of_first("He told me you had been to her")
'e'
```

□

621 **Exercise 38.** Define and test a function that returns the reversal of a string or list. □

Solution.

```
def reverse(ex_list):
...     return ex_list[::-1]
```

```

...
>>> reverse(motto)
['delectat', 'agere', 'nihil']
>>> reverse(motto[0])
'lihin'

```

□

Exercise 39. Define and test a function that returns the alphabetically first word in a string. 622

□ 623

Solution.

```

>>> def lex_first_word(ex_string):
...     return sorted(ex_string.split(' '))[0]
...
>>> lex_first_word("a b c d")
'a'
>>> lex_first_word("c b a d")
'a'
>>> lex_first_word("single")
'single'
>>> lex_first_word("")
''
>>> lex_first_word("he told me you had been to her")
'been'

```

□

Exercise 40. Define and test a function that returns the middle element of a list, that is, the element that has the same number of elements before and after it. (If the list has an even number of elements, the chosen element should have one more element before than after.) 624

625

□ 626

Solution.

```

>>> def mid_word(ex_list):
...     return ex_list[len(ex_list)/2]
...
>>> mid_word([1, 2, 3, 4, 5])
3
>>> mid_word([1, 2, 3, 4])
3
>>> mid_word([1])
1
>>> mid_word([ ])

```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in mid_word
IndexError: list index out of range
```

□

- 627 **Exercise 41.** Define and test a function that returns a list of all the trigram tokens in a
 628 list of tokens. For instance, it should have the following behavior:

```
>>> pprint(ngrams(motto * 2))
[('nihil', 'agere', 'delectat'),
 ('agere', 'delectat', 'nihil'),
 ('delectat', 'nihil', 'agere'),
 ('nihil', 'agere', 'delectat')]
```

- 629 Test it on the Stein poem. □

Solution.

```
>>> def ngrams(items, N=3):
...     return [tuple(items[i:i+N]) for i in range(len(items)-N+1)]
...
>>> pprint(ngrams(stein_words)[:10])
[('let', 'it', 'be'),
 ('it', 'be', 'when'),
 ('be', 'when', 'it'),
 ('when', 'it', 'is'),
 ('it', 'is', 'mine'),
 ('is', 'mine', 'to'),
 ('mine', 'to', 'be'),
 ('to', 'be', 'sure'),
 ('be', 'sure', 'let'),
 ('sure', 'let', 'it')]
>>> len(ngrams(stein_words))
223
```

This version uses Python's ability to have optional arguments with defaults (the argument *N* for the *n*-gram length). The code can thus be used to generate lists of *n*-grams of other lengths.

```
>>> pprint(ngrams(stein_words, 5)[:10])
[('let', 'it', 'be', 'when', 'it'),
 ('it', 'be', 'when', 'it', 'is'),
 ('be', 'when', 'it', 'is', 'mine'),
 ('when', 'it', 'is', 'mine', 'to'),
 ('it', 'is', 'mine', 'to', 'be'),
```

```

('is', 'mine', 'to', 'be', 'sure'),
('mine', 'to', 'be', 'sure', 'let'),
('to', 'be', 'sure', 'let', 'it'),
('be', 'sure', 'let', 'it', 'be'),
('sure', 'let', 'it', 'be', 'when')]

```

□

Exercise 42. Define and test a function that returns a set of all trigram types in a list of tokens. For instance, it should have the following behavior: 630

```

>>> pprint(ngram_set(motto * 2))
set([('agere', 'delectat', 'nihil'),
     ('delectat', 'nihil', 'agere'),
     ('nihil', 'agere', 'delectat')])

```

631

Test it on the first line of the Stein poem. □ 632

Solution.

```

>>> def ngram_set(items, N=3):
...     return set(ngrams(items, N))
...
>>> pprint(ngram_set(stein_words))
set([('be', 'let', 'it'),
     ('be', 'mine', 'let'),
     ('be', 'mine', 'to'),
     ('be', 'mine', 'when'),
     ('be', 'sure', 'let'),
     ('be', 'sure', 'mine'),
     ('be', 'sure', 'to'),
     ('be', 'sure', 'when'),
     ('be', 'to', 'be'),
     ('be', 'when', 'it'),
     ('is', 'mine', 'let'),
     ('is', 'mine', 'to'),
     ('is', 'mine', 'when'),
     ('it', 'be', 'let'),
     ('it', 'be', 'mine'),
     ('it', 'be', 'to'),
     ('it', 'be', 'when'),
     ('it', 'is', 'mine'),
     ('it', 'to', 'be'),
     ('let', 'it', 'be'),
     ('let', 'it', 'to'),

```

```
('mine', 'let', 'it'),  
( 'mine', 'to', 'be'),  
( 'mine', 'to', 'let'),  
( 'mine', 'when', 'it'),  
( 'mine', 'when', 'to'),  
( 'sure', 'let', 'it'),  
( 'sure', 'mine', 'to'),  
( 'sure', 'to', 'be'),  
( 'sure', 'to', 'let'),  
( 'sure', 'when', 'it'),  
( 'sure', 'when', 'to'),  
( 'to', 'be', 'mine'),  
( 'to', 'be', 'sure'),  
( 'to', 'let', 'it'),  
( 'when', 'it', 'is'),  
( 'when', 'to', 'be')])
```

□

18. DICTIONARIES

633

DICTIONARY A **DICTIONARY** is a data structure for associating one kind of data with another. 634
 We might want to associate words with their locations in a document, or n -grams 635
 with their number of occurrences, or any of a variety of other associations. 636

In Python, a dictionary can be specified extensionally using a notation with 637
 braces. Here, we build a dictionary that associates a few words with their length. 638

```
>>> lengths = { 'the': 3, 'a': 1, 'is': 2, 'an': 3 }
>>> lengths
{'a': 1, 'the': 3, 'is': 2, 'an': 3}
```

KEYS Notice that when the dictionary is printed, the association between **KEYS** (the words) 639
VALUES and their **VALUES** (the lengths) is preserved, but the order of presentation is not. 640
 Dictionaries are important for the association, not the ordering. (That's what lists 641
 are for.) 642

The value for a given key can be recovered using the indexing notation we've 643
 already used, but now we're indexing not by numeric positions but by keys to 644
 retrieve the corresponding values. 645

```
>>> lengths['the']
3
>>> lengths['an']
3
>>> lengths['some']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'some'
```

You may have noticed a problem with the lengths list: One of the values is 646
 wrong. That's what you get when building things extensionally. Better to build 647
 the dictionary intensionally. First, we build a list of pairs of words and their lengths 648
 using a list comprehension. 649

```
>>> len_list = [ (word, len(word)) for word in ['the', 'a', 'is', 'an'] ]
>>> len_list
[('the', 3), ('a', 1), ('is', 2), ('an', 2)]
```

DICTIONARY FUNCTION Then we convert this list of pairs into a dictionary using the dict function. 650

```
>>> len_dict = dict(len_list)
>>> len_dict['the']
3
>>> len_dict['an']
2
>>> len_dict['some']
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
KeyError: 'some'
```

651 **Exercise 43.** Build a dictionary named `first_letters` of words and their first letters.

652 The word types should be taken from the Stein poem. The result should look like this:

```
>>> first_letters
{'be': 'b', 'sure': 's', 'is': 'i', 'when': 'w', 'it': 'i', 'mine': 'm', 'to': 't', 'let': 'l'}
```

653

□

Solution.

```
>>> first_letters = dict( [(word, word[0]) for word in stein_words] )
>>> first_letters
{'be': 'b', 'sure': 's', 'is': 'i', 'when': 'w', 'it': 'i', 'mine': 'm', 'to': 't', 'let': 'l'}
```

□

654 There are a few additional functions for manipulating dictionaries that may

655 prove useful. The `keys` function returns a list of all of the keys defined in a KEYS FUNCTION

656 dictionary

```
>>> first_letters.keys()
['be', 'sure', 'is', 'when', 'it', 'mine', 'to', 'let']
```

657 and the `values` function returns a list of all of the values in a dictionary. VALUES FUNCTION

```
>>> first_letters.values()
['b', 's', 'i', 'w', 'i', 'm', 't', 'l']
```

658 Finally, the `items` function returns a list of key-value pairs from the dictionary. ITEMS FUNCTION

```
>>> first_letters.items()
[('be', 'b'), ('sure', 's'), ('is', 'i'), ('when', 'w'), ('it', 'i'), ('mine', 'm'), ('to', 't'),
```

19. LOOPS AND CONDITIONALS

659

It's now page 48, and I've postponed as long as possible a discussion of the kind of control structures that many people think of as the hallmark of computer programming, such constructs as loops and conditionals. The style of programming I've been implicitly using – a kind of functional programming over compound data structures – eschews these kinds of structures. But for the next steps, we'll need to use them a bit.

660

661

662

663

664

665

FOR LOOP The **FOR LOOP** allows executing a block of code several times, once *for* each value that a certain variable takes on. The notation is as follows:

666

667

```
for <variable> in <list or set or other iterable data>:
    <body>
```

For example,

668

```
>>> for letter in 'sanctus':
...     print letter
...
s
a
n
c
t
u
s
```

INDENTATION Note the **INDENTATION**. It is crucial. Python uses indentation to convey the structure of the program. What constitutes the body of a for loop, for instance, is exactly the sequence of textual lines that follow the first line and that are *indented more deeply*. Similarly for other constructs in the language. Indentation is important; pay attention to it.

669

670

671

672

673

PRINT COMMAND The **print** command (it's not a function) used above, when executed, has the side effect of presenting the printed representation of the comma-separated items following it (they're not really arguments) to the screen. I've used it inside the loop so that we can see what's happening inside the loop.

674

675

676

677


CONDITIONAL The **CONDITIONAL** allows different code to be executed depending on whether a particular condition holds or not. We test the condition, and if it holds execute one branch of the conditional, otherwise executing the other branch.

678

679

680

```
if <condition>:
    <true branch>
else:
    <false branch>
```


681  The `else:` and *(else branch)* can be dropped if nothing needs to be
682 done in case the condition is false.

683 Here's an (admittedly artificial) example:

```
>>> occurs = {}
>>> for letter in 'sanctus':
...     if letter in 'Germanus':
...         occurs[letter] = True
...     else:
...         occurs[letter] = False
...
>>> occurs
{'a': True, 'c': False, 'n': True, 's': True, 'u': True, 't': False}
```


684 **Exercise 44.** *What does this snippet of code do?* □

Solution. *It tells us which letters in the string 'sanctus' are also in the string 'Germanus'. Apparently, the two have several letters in common, in fact all of the letters in 'sanctus' except for 'c' and 't'.* □

685 What kinds of expressions can be in the test part of a conditional? Any expres-
686 sion whose value is a truth value, or `BOOLEAN`. The Boolean data type contains just `BOOLEAN`
687 two values: `True` and `False`. (In the above snippet, the values in the dictionary
688 were also Booleans.) There are several functions that return Boolean values. Here
689 are just a few:

- 690 • `x in y`: Returns `True` just in case the value `x` is one of the values in the list, `IN FUNCTION`
691 set, or other iterable data object `y`. Otherwise, it returns `False`.
- 692 • `x == y`: Returns `True` just in case `x` and `y` are the same value. `== FUNCTION`
- 693 • `x < y`: Returns `True` just in case the value `x` is less than the value `y` un- `< FUNCTION`
694 der whatever ordering is appropriate for their data type (numerically for
695 numbers, lexicographically for strings).
- 696 • `x and y`: Returns `True` just in case both `x` and `y` have the value `True`. `AND FUNCTION`

697 There are many other built-in functions that return Booleans, and of course you
698 can define your own.

699  The Boolean data type is named after George Boole, whose work on
700 what is now called Boolean algebra provided a mathematical basis for a
701 logic of truth and falsity.

702 **Exercise 45.** *Define and test a function `is_palindrome` that returns a Boolean: `True` if*
703 *its argument is a palindromic string, and `False` otherwise.* □

Solution. A solution that takes advantage of the `reverse` function provided in the solution to Exercise 38:

```
>>> def is_palindrome(ex_list):
...     return ex_list == reverse(ex_list)
...
>>> is_palindrome('radar')
True
>>> is_palindrome('radio')
False
```

□

Exercise 46. Define and test a function `print_palindromes` that prints all of the words in its list argument that are palindromes, one palindrome per line. 704

□ 705

Solution.

```
>>> def print_palindromes(ex_list):
...     for item in ex_list:
...         if is_palindrome(item):
...             print item
...
>>> print_palindromes(['madam', 'i\'m', 'adam', 'radar'])
madam
radar
```

□

Exercise 47. Define and test a function `common_letters` that takes two string arguments and returns a string containing all of the letters that its two arguments have in common. Demonstrate it on the two strings `'disproportionableness'` and `'absolutism'`. 706
707
708
Hint: The answer is 'isotabl'. 709

□

Solution.

```
>>> def common_letters(s1, s2):
...     common = ''
...     for letter in s1:
...         if letter in s2 and not letter in common:
...             common = common + letter
...     return common
...
>>> common_letters('disproportionableness', 'absolutism')
'isotabl'
```



710 A useful idiom is to loop over all of the key-value pairs in a dictionary by taking
711 advantage of the fact that the `items` function returns an iterable list:

```
>>> for (key, value) in first_letters.items():  
...     print key, "has first letter", value  
...  
be has first letter b  
sure has first letter s  
is has first letter i  
when has first letter w  
it has first letter i  
mine has first letter m  
to has first letter t  
let has first letter l
```


731 **Problem 49.** Generate a dictionary that for a given list of words (the words in the Stein
 732 poem, say) associates each position or index with the word at that index. The dictionary
 733 should associate the number 0 with 'let' (because the Stein poem has the word 'let' at index
 734 0), the number 1 with 'it', and so forth. Then invert the dictionary. The inverted dictionary
 735 will map words to a list of positions where that word occurs – a concordance! □

Solution.

```
>>> wordsbyposition = \
...     dict( [(i, stein_words[i])
...            for i in range(0, len(stein_words)-1)] )

>>> wordsbyposition[5]
'is'
>>> invert_dict(wordsbyposition)
{'be': [2, 8, 12, 23, 25, 32, 36, 39, 42, 44, 48, 50, 57, 62, 70, 72, 76, 78, 81, 85, 87, 91, 93,
```

□ The backslash at the end of the first line tells Python that the line has been split across multiple lines. No backslash is needed on the second line as it is clear from the unclosed open bracket and indentation that the list comprehension has not been completed.

□

736 Finally, we can keep track not only of the index of each word, but also its context,
 737 the few words surrounding it.

738 **Problem 50.** Choose an appropriate dictionary structure that, when inverted, associates
 739 with each word a list of pairs. Each pair has an index and a surrounding n -gram at that
 740 position. Create such a dictionary and invert it. Write some code to print out the contents
 741 of that dictionary in a nice format. The output of such a concordance generator operating
 742 on the Stein poem can be found in Appendix A. □

Solution.

```
>>> N = 5
>>> midwords = dict( [(i, tuple(stein_words[i - N/2: i + N/2 + 1])),
...                  stein_words[i])
...                  for i in range(N/2, len(stein_words) - N - 1)] )

>>> concordance = invert_dict(midwords)

>>> def print_concordance(concordance):
...     for (k, v) in concordance.iteritems():
...         print "{}:".format(k)
...         for (pos, context) in v:
```

```
...         print "{:8d} -- {}".format(pos, " ".join(context))  
...
```

See Appendix A for the result of executing
`print_concordance(concordance)`

□

743

APPENDIX A. A CONCORDANCE WITH DESCRIPTION

```
>>> print_concordance(concordance)
be:
    91 -- let it be to be
    139 -- mine to be sure to
    210 -- when to be sure when
    136 -- sure to be mine to
     57 -- mine to be sure let
     72 -- be to be sure let
    201 -- be to be sure to
     2 -- let it be when it
     62 -- it to be sure when
    114 -- be to be sure let
    121 -- mine to be sure let
    108 -- be to be sure let
    125 -- let it be to be
     93 -- be to be sure to
     25 -- be to be sure when
    152 -- let it be to be
     81 -- sure to be sure let
    100 -- let it be to be
    160 -- mine to be sure let
    194 -- let it be mine to
    190 -- mine to be sure let
     70 -- let it be to be
    148 -- be to be mine let
     50 -- be to be sure when
     76 -- let it be to be
    170 -- let it be to be
     78 -- be to be sure to
     87 -- be to be sure let
    214 -- when to be sure to
     32 -- mine to be sure let
    176 -- let it be to be
    199 -- let it be to be
    172 -- be to be sure let
    118 -- let it be mine to
     39 -- let it be let it
    146 -- let it be to be
    133 -- mine to be sure to
     96 -- sure to be sure let
    130 -- sure to be mine to
```

42 -- let it be to be
 8 -- mine to be sure let
 12 -- let it be when it
 154 -- be to be sure to
 157 -- sure to be mine to
 127 -- be to be sure to
 178 -- be to be sure to
 48 -- let it be to be
 102 -- be to be sure let
 142 -- sure to be mine let
 106 -- let it be to be
 44 -- be to be sure let
 186 -- it to be sure mine
 166 -- be to be mine let
 36 -- let it be let it
 181 -- sure to be sure let
 164 -- let it be to be
 206 -- let it be mine when
 112 -- let it be to be
 85 -- let it be to be
 23 -- let it be to be

sure:

211 -- to be sure when to
 33 -- to be sure let it
 115 -- to be sure let it
 73 -- to be sure let it
 9 -- to be sure let it
 128 -- to be sure to be
 103 -- to be sure let it
 97 -- to be sure let it
 82 -- to be sure let it
 134 -- to be sure to be
 179 -- to be sure to be
 122 -- to be sure let it
 182 -- to be sure let it
 58 -- to be sure let it
 215 -- to be sure to let
 88 -- to be sure let it
 187 -- to be sure mine to
 94 -- to be sure to be
 63 -- to be sure when it
 140 -- to be sure to be


```
79 -- to be sure to be
161 -- to be sure let it
26 -- to be sure when it
109 -- to be sure let it
173 -- to be sure let it
51 -- to be sure when it
155 -- to be sure to be
191 -- to be sure let it
45 -- to be sure let it
202 -- to be sure to let
is:
66 -- when it is mine let
19 -- when it is mine let
54 -- when it is mine to
5 -- when it is mine to
29 -- when it is mine to
15 -- when it is mine when
when:
208 -- be mine when to be
212 -- be sure when to be
13 -- it be when it is
17 -- is mine when it is
3 -- it be when it is
64 -- be sure when it is
27 -- be sure when it is
52 -- be sure when it is
it:
14 -- be when it is mine
47 -- sure let it be to
99 -- sure let it be to
69 -- mine let it be to
60 -- sure let it to be
184 -- sure let it to be
90 -- sure let it be to
175 -- sure let it be to
193 -- sure let it be mine
38 -- be let it be let
11 -- sure let it be when
205 -- to let it be mine
18 -- mine when it is mine
75 -- sure let it be to
105 -- sure let it be to
```

111 -- sure let it be to
163 -- sure let it be to
169 -- mine let it be to
35 -- sure let it be let
84 -- sure let it be to
4 -- be when it is mine
117 -- sure let it be mine
198 -- to let it be to
28 -- sure when it is mine
41 -- be let it be to
124 -- sure let it be to
65 -- sure when it is mine
151 -- mine let it be to
22 -- mine let it be to
218 -- to let it to be
53 -- sure when it is mine
145 -- mine let it be to

mine:

67 -- it is mine let it
137 -- to be mine to be
188 -- be sure mine to be
119 -- it be mine to be
30 -- it is mine to be
131 -- to be mine to be
6 -- it is mine to be
167 -- to be mine let it
195 -- it be mine to let
158 -- to be mine to be
207 -- it be mine when to
55 -- it is mine to be
143 -- to be mine let it
16 -- it is mine when it
20 -- it is mine let it
149 -- to be mine let it

to:

141 -- be sure to be mine
159 -- be mine to be sure
135 -- be sure to be mine
31 -- is mine to be sure
171 -- it be to be sure
61 -- let it to be sure
129 -- be sure to be mine

```
189 -- sure mine to be sure
 95 -- be sure to be sure
138 -- be mine to be sure
 56 -- is mine to be sure
 24 -- it be to be sure
209 -- mine when to be sure
203 -- be sure to let it
 49 -- it be to be sure
156 -- be sure to be mine
 86 -- it be to be sure
113 -- it be to be sure
 71 -- it be to be sure
165 -- it be to be mine
  7 -- is mine to be sure
147 -- it be to be mine
177 -- it be to be sure
 92 -- it be to be sure
185 -- let it to be sure
200 -- it be to be sure
180 -- be sure to be sure
132 -- be mine to be sure
216 -- be sure to let it
126 -- it be to be sure
101 -- it be to be sure
213 -- sure when to be sure
 43 -- it be to be sure
 80 -- be sure to be sure
120 -- be mine to be sure
 77 -- it be to be sure
196 -- be mine to let it
153 -- it be to be sure
107 -- it be to be sure
```

let:

```
 10 -- be sure let it be
168 -- be mine let it be
 40 -- it be let it be
150 -- be mine let it be
 34 -- be sure let it be
110 -- be sure let it be
197 -- mine to let it be
 37 -- it be let it be
144 -- be mine let it be
```

83 -- be sure let it be
116 -- be sure let it be
74 -- be sure let it be
46 -- be sure let it be
174 -- be sure let it be
68 -- is mine let it be
123 -- be sure let it be
204 -- sure to let it be
104 -- be sure let it be
21 -- is mine let it be
183 -- be sure let it to
59 -- be sure let it to
89 -- be sure let it be
162 -- be sure let it be
217 -- sure to let it to
98 -- be sure let it be
192 -- be sure let it be

744

APPENDIX B. STATISTICS

745

Running time of included Python examples: 249.44 seconds.

REFERENCES

746

Gertrude Stein. *An Acquaintance with Description*. Seizin Press, 1929. URL <http://books.google.com/books?id=YpFuQgAACAAJ>. 747

748

INDEX

- < function, 49
- = operator, *see* assignment
- == function, 49

- addition, 16
- and function, 49
- arguments, 27
- arithmetic operators, 16
- assignment, 19

- bigram, 40
- Boolean, 49

- Carroll, Lewis, 18
- clickable links, 4
- command, 16
 - def, 41
 - for, 48
 - if, 48
 - print, 48
 - return, 41
- conditional, 48
- count function, 30

- data type, 20
- def command, 41
- delimiters, 21
- dict function, 46
- dictionary, 46

- empty tuple, 26
- escape sequence, 34
- Euler, Leonhard, 27
- exercises, 4
- expression, 16
- extensional, 37

- False value, 49
- first principles, 2
- for loop, 48
- four fours, 16
- function
 - <, 49
 - ==, 49
 - and, 49
 - count, 30
 - dict, 46
 - in, 49
 - items, 47
 - join, 35
 - keys, 47
 - len, 21, 28
 - list, 25
 - lower, 36
 - pprint, 32
 - range, 29
 - set, 39
 - sorted, 28
 - split, 35
 - strip, 38
 - tuple, 25
 - values, 47
- function call, 41
- functions, 27

- hashable, 25

- if command, 48
- imitation, 2
- in function, 49
- indentation, 48
- indexing, 20
- intensional, 37
- interpreter, 16
- items function, 47

- join function, 35

- key concepts, 4
- keys, 46
- keys function, 47

- Leibniz, Gottfried, 27
- len function, 21, 28
- list, 20
- list comprehensions, 37
- list function, 25
- lower function, 36

- Marx, Groucho, 18
- mathematical notation, 27
- multiplication, 16

n-grams, 40

newline, 21

object notation, 27

optional arguments, 9

position, 20

pprint function, 32

print command, 48

problems, 4

range function, 29

result, 27

return command, 41

set, 39

set function, 39

singleton tuple, 26

slicing, 23

sorted function, 28

split function, 35

Stein, Gertrude, 31

string, 21

strip function, 38

tokens, 31

trigram, 40

True value, 49

tuple, 25

 empty, 26

 singleton, 26

tuple function, 25

type, 31

unigram, 40

union, 39

values, 46

values function, 47

variables, 19

whitespace, 8

