PROGRAMMING FOR HUMANISTS

STUART M. SHIEBER

Contents

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.

IMITATION

FIRST PRINCIPLES

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.

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

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

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

1. Where we're headed

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 25) 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.

2. Installing Python

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

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

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

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

Web-based: On any operating system with a browser, you can set up an account at Pythonanywhere and run a Python interpreter from within your 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
```

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

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

Exercise 2. *Test that the Python interpreter is working by running it and typing in a 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
>>>
```

□

3. The synoptic gospels

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:

```
>>> 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`.

```
>>> 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]`.

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

```
>>> 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?* □

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.

```
>>> 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.)

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

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

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

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'
```

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

The next step in extracting the words is to get rid of a bunch of characters that 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 scriptum'
```

There are other characters we may want to remove, punctuation and newlines and 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 est'
```

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. WHITESPACE

```
>>> 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.

```
>>> 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.

```
>>> def words_normed(filename):
...     return ' '.join(open(filename).readlines())                  \
...                   .lower()                                       \
```

```
...                    .translate(None, '\n\r,.:;\\/[]()?0123456789') \
...                    .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.

Now we can do that for several different documents.

```
>>> 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.

```
>>> 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.

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

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

How about generating a whole series of such three word sequences? Contiguous sequences of n words in a document are called n-grams; in the case where n is 3, 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']
```

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)]
...
```

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

OPTIONAL ARGUMENTS

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

□

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

```
>>> 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 (or other n-grams) they have in common. We start by defining the intersection of two lists, that is, the items they have in common:

```
>>> 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:

```
>>> 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?

```
>>> len(common_matthew_mark)
1906
```

That's about 18 percent of the Mark trigrams.

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

Is that a lot? We can compare it against the proportion of trigrams found in some 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
```

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

Let's make a table that shows how similar the gospels are to each other (at least 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?* □

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

Part 2. Programming from first principles

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.

4. Python documentation

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

Here are some especially important bits:

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

5. The Python interpreter

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.

COMMAND

EXPRESSION

> 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.

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.

```
>>> 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.

Exercise 11. *Enter the expression* `3 + 4 * 5` *into the Python interpreter and verify that it works like it should.* □

Exercise 12. *Use the Python interpreter to determine the values of the following arithmetic expressions:*

(1) $4/4 - 4/4$
(2) $\frac{4+4}{4+4}$
(3) $\frac{4 \cdot 4}{4+4}$

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. □

6. Expressions and nesting

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:

Some new cakes are nice

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

Some new cakes are nice

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:

I shot an elephant in my pajamas

I shot an elephant in my pajamas

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

`3 + 4 * 5`

and not

`3 + 4 * 5`

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.

7. Variables and the naming of values

VARIABLES

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

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

(1) *`matthew`*
(2) *`sanctus_germanus`*
(3) *`1_samuel`*
(4) *`__name__`*
(5) *`n-grams`*

□

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

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

ASSIGNMENT

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

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

8. Sequence data types

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. *DATA TYPE*

> 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. *LIST*

```
>>> [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]`. *POSITION* *INDEXING*

```
>>> 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.

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

LEN FUNCTION

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

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

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

STRING

DELIMITERS

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

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

Strings can be specified with other delimiters, such as double quotes, or triple 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.'
```

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

NEWLINE

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

Strings can be concatenated using the + operator.

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

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

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'
```

Interesting how Python uses the "multiplication" operator `*` for repeating 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?*

```
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
```

□

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]`.

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

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

```
  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
```

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

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

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

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

(1) `ex_string[0:3]`
(2) `ex_string[3]`
(3) `ex_string[3:4]`
(4) `ex_string[3:3]`
(5) `ex_string[3:2]`
(6) `ex_string[:4]`
(7) `ex_string[4:]`
(8) `ex_string[4:-3]`
(9) `ex_string[3:100]`
(10) `ex_string[8:0:-1]`
(11) `ex_string[::-1]`

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

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?* □

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?* □

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?* □

TUPLE

8.5. **Tuples.** The final sequence data type we'll cover is the TUPLE. The name derives from the suffix seen in quin*tuple*, sex*tuple*, sep*tuple*, and the like.

A tuple in Python is specified like a list, with multiple elements separated by commas, but without the surrounding brackets. It is conventional (though not always required) to use grouping parentheses around the elements of the tuple. 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')
```

TUPLE FUNCTION
LIST FUNCTION

Lists can be converted to tuples using the `tuple` function, and tuples to lists using the `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']
```

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

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

Since tuples and lists are so similar, why do both exist in the language? The distinction is a bit arcane. Lists are stored internally in such a way that they can be modified – items replaced, added, or removed. Tuples do not allow modification 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
```

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

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

HASHABLE

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

9. Functions

FUNCTIONS
ARGUMENTS
RESULT

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

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

MATHEMATICAL NOTATION

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

`f(`⟨*arg1*⟩`,` ⟨*arg2*⟩`, ...)` .

OBJECT NOTATION

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

⟨*arg1*⟩`.f(`⟨*arg2*⟩`, ...)` .

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

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

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

As it turns out, Python makes available in the `operator` package equivalents to all such infix and prefix operators as regular functions called with 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? □

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]`*

□

RANGE FUNCTION

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

```
>>> 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]
```

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]
[]
```

□

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
```

10. Words, types, and tokens

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

We distinguish word types from word tokens. A text is made up of a series of word TOKENS. Each word token belongs to a word TYPE. Consider the text corpus in Figure 1, a sentence from Gertrude Stein's 1929 poem "An Acquaintance With Description" (Stein, 1929). This corpus has 225 word tokens (ignoring punctuation), which are instances of just eight word types (if we conflate upper and lower case). The eight types, in decreasing order of frequency, are: "be", "to", "it", "sure", "let", "mine", "when", "is". Each of these word types has several occurrences as tokens 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

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.

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

```
⟨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.

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

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

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

PPRINT FUNCTION

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.

```
>>> 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']
```

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

12. Special characters

Let's examine the first line of the poem.

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

It's a string of `41` characters.

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

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

ESCAPE SEQUENCE

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.

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

13. Splitting and joining strings

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

JOIN FUNCTION

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

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

(1) `'1-2-3'`
(2) `'1, 2, 3'`
(3) `'123'`
(4) `'3, 2, 1'`

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

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

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

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

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.* □

LOWER FUNCTION

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`*.* □

14. List comprehensions

EXTENSIONAL

We've seen the notation for specifying a list EXTENSIONALLY, that is, by enumerating its elements explicitly. Here for instance are the first letters of the first few 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']
```

LIST COMPREHENSIONS

INTENSIONAL

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

$$[\ \langle \textit{generic element} \rangle\ \texttt{for}\ \langle \textit{variable} \rangle\ \texttt{in}\ \langle \textit{list} \rangle\ ] \quad .$$

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

$$\{ x^2 \mid 0 \leq x < 10 \}$$

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

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

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

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

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

STRIP FUNCTION

Exercise 31. *Generate a list named* `stein_words` *of all the word tokens in the Stein poem. Make sure that all the words are lower case. You may find the* `strip` *function to be useful. You should be able to get the following behavior:*

```
>>> 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.* □

15. Sets

SET

Time to introduce another data type, the SET. A set is a compound data type; like the list, each set contains elements. But the elements of a set are unique. A set does not contain multiple tokens of the same value. You can create a set from a list 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'])
```

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

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

UNION

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

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

16. Calculating with n-grams

We'll spend some time looking at n-grams, contiguous sequences of n words. When n is 1, 2, or 3, we call them unigrams, bigrams, and trigrams, respectively. Here are some examples of trigrams built from the vocabulary seen in Gertrude Stein's poem:

n-grams
unigram
bigram
trigram

(1) let it be
(2) it is mine
(3) it is sure
(4) to be sure

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.* □

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.* □

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.)* □

17. Defining your own functions

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

Sadly, there often is not a function tailor-made for your purposes. But you can write your own. Indeed, writing functions is the heart of computer programming (in spite of the fact that it took until page 34 to get to the topic).

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

DEF COMMAND

def ⟨*function name*⟩(⟨*arguments*⟩):
    ⟨*function body*⟩

RETURN COMMAND

Within the body of the function, the **`return`** command generates the value to return as the result of the function.

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

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

FUNCTION CALL

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

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

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

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

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

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.)* □

Exercise 41. *Define and test a function that returns a list of all the trigram tokens in a 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')]
```

Test it on the Stein poem. □

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:*

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

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

18. Dictionaries

DICTIONARY

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

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

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

KEYS
VALUES

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

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

```
>>> 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 wrong. That's what you get when building things extensionally. Better to build the dictionary intensionally. First, we build a list of pairs of words and their lengths using a list comprehension.

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

DICT FUNCTION

Then we convert this list of pairs into a dictionary using the `dict` function.

```
>>> 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'
```

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

□

There are a few additional functions for manipulating dictionaries that may prove useful. The `keys` function returns a list of all of the keys defined in a dictionary

KEYS FUNCTION

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

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']
```

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

It's now page 38, 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.

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:

```
for ⟨variable⟩ in ⟨list or set or other iterable data⟩:
    ⟨body⟩
```

For example,

```
>>> 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.

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.

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.

```
if ⟨condition⟩:
    ⟨true branch⟩
else:
    ⟨false branch⟩
```

> The **else:** and ⟨*else branch*⟩ can be dropped if nothing needs to be done in case the condition is false.

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}
```

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

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

Boolean

- x **in** y: Returns `True` just in case the value x is one of the values in the list, set, or other iterable data object y. Otherwise, it returns `False`. (in function)
- $x == y$: Returns `True` just in case x and y are the same value. (== function)
- $x < y$: Returns `True` just in case the value x is less than the value y under whatever ordering is appropriate for their data type (numerically for numbers, lexicographically for strings). (< function)
- x **and** y: Returns `True` just in case both x and y have the value `True`. (and function)

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

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

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

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.* □

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'`. *Hint: The answer is* `'isotabl'`. □

A useful idiom is to loop over all of the key-value pairs in a dictionary by taking 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
```

20. A concordance

In this section, you'll put together code to generate a simple keyword-in-context (KWIC) concordance, which lists for each word in a text all of the contexts in which it occurs.

Recall the dictionary you built in Exercise 43. This dictionary associates each word with its first letter. Of course, in a traditional dictionary (in the nontechnical sense of the word 'dictionary'), the association is the other way around: Each letter is associated with a list of the words that it is the first letter of. We could generate such a dictionary from the one we already built if we had a way of "inverting" dictionaries. Such a dictionary inverter will turn out to be useful for other tasks as well.

Problem 48. *Write a function that takes a dictionary as its argument and returns a new dictionary that is the "inversion" of its argument. The keys in the new dictionary are the values in the original, and the values for a key x is the list of all keys in the original whose value in the original was x.* □

If you've done this problem properly, you should get the following behavior:

```
>>> pprint(invert_dict(first_letters))
{'b': ['be'],
 'i': ['is', 'it'],
 'l': ['let'],
 'm': ['mine'],
 's': ['sure'],
 't': ['to'],
 'w': ['when']}
```

We've turned our `first_letters` dictionary into a dictionary in the conventional sense, a mapping from letters to words they start with.

Now a slightly more sophisticated case.

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

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

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

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
```

Appendix B. Statistics

Running time of included Python examples: `93.19` seconds.

References

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

Index

School of Engineering and Applied Sciences, Harvard University