

# Exercises in Programming Style

---

# Contents

---

[Preface to the Second Edition](#)

[Preface to the First Edition](#)

[Prologue](#)

[Author](#)

## [PART I Historical](#)

---

[CHAPTER 1 • Good Old Times](#)

[CHAPTER 2 • Go Forth](#)

[CHAPTER 3 • Arrays](#)

## [PART II Basic Styles](#)

---

[CHAPTER 4 • Monolithic](#)

[CHAPTER 5 • Cookbook](#)

[CHAPTER 6 • Pipeline](#)

[CHAPTER 7 • Code Golf](#)

## [PART III Function Composition](#)

---

[CHAPTER 8 · Infinite Mirror](#)

[CHAPTER 9 · Kick Forward](#)

[CHAPTER 10 · The One](#)

## **[PART IV Objects and Object Interaction](#)**

---

[CHAPTER 11 · Things](#)

[CHAPTER 12 · Letterbox](#)

[CHAPTER 13 · Closed Maps](#)

[CHAPTER 14 · Abstract Things](#)

[CHAPTER 15 · Hollywood](#)

[CHAPTER 16 · Bulletin Board](#)

## **[PART V Reflection and Metaprogramming](#)**

---

[CHAPTER 17 · Introspective](#)

[CHAPTER 18 · Reflective](#)

[CHAPTER 19 · Aspects](#)

[CHAPTER 20 · Plugins](#)

## **[PART VI Adversity](#)**

---

[CHAPTER 21 · Constructivist](#)

[CHAPTER 22 · Tantrum](#)

CHAPTER 23 · Passive Aggressive

CHAPTER 24 · Declared Intentions

CHAPTER 25 · Quarantine

## **PART VII Data-Centric**

---

CHAPTER 26 · Persistent Tables

CHAPTER 27 · Spreadsheet

CHAPTER 28 · Lazy Rivers

## **PART VIII Concurrency**

---

CHAPTER 29 · Actors

CHAPTER 30 · Dataspaces

CHAPTER 31 · Map Reduce

CHAPTER 32 · Double Map Reduce

## **PART IX Interactivity**

---

CHAPTER 33 · Trinity

CHAPTER 34 · Restful

## **PART X Neural Networks**

---

CHAPTER 35 · Dense, Shallow, under Control

CHAPTER 36 · Dense, Shallow, out of Control

CHAPTER 37 · Bow Tie

CHAPTER 38 · Neuro-Monolithic

CHAPTER 39 · Sliding Window

CHAPTER 40 · Recurrent

**Index**

---

# Preface to the Second Edition

---

In the six years since the first edition of this book was published, two things happened that made me want to update the book. The first one was the wide adoption of Python 3. The original edition had all its code in Python 2, which has now reached its official end of life. This second edition updates all the code to Python 3.

But the second, and most important, thing that happened since 2014 was the dizzying developments in machine learning, more specifically, in neural networks (NNs). By 2018, I felt it was my duty, and personal challenge, to capture the basic programming concepts in neural networks in exactly the same way I had done for all other concepts: by doing term frequency in neural networks. This led me into a fascinating tour of that field, focused, as I was, on exploring it using a problem for which neural networks are not typically used – term frequency is a well-specified problem for which we know the exact logic. This second edition includes a whole new part, Part X, featuring several basic programming concepts in NNs.

In the process of doing this tour of neural networks for the term frequency problem, four things became clear. **First**, I had to break the problem down into its smaller components, and show how to solve those smaller sub-problems using NNs. This is because the solution of the complete problem is essentially a pipeline of functions that requires knowledge of too many NN concepts at once. **Second**, although *learning* is the magic sauce for which neural networks are popular, I found myself being even more fascinated by the concept of networks as computing

machines. As much as I admire the power of statistics for making predictions based on existing data, the computer engineer in me absolutely wants to program these networks by hand by setting the weights manually. I couldn't help it! Part X is full of neural networks that are programmed manually, without learning. **Third**, the most popular framework for programming NNs, TensorFlow, uses *array programming* concepts at its core. This is not surprising, given that we are dealing, essentially, with linear algebra operations. I realized the first edition had missed this historically important programming style – array programming – so I added a new chapter about it in Part I, Historical. (Hello, APL, apologies for having missed you the first time!) Finally, the **fourth** thing I realized was that I could easily write an entire new book just covering neural network programming concepts. I had to stop myself at six chapters in Part X, but these chapters don't even begin to cover the immense and fertile field of programming ideas in neural networks.

NNs require a fundamentally different way of thinking about computing that is at the same time very low level and very powerful. I am now convinced that every programmer needs to learn this connectionist computing model, not just for the hype that its applications currently enjoy, but in spite of it.

Pierre Baldi was instrumental in my developing an interest in neural networks, and in my ability to navigate that field as an outsider. I thank him for the many conversations we had about all that is covered in Part X. In these past six years, my daughter Julia grew up, and, at points, helped me stay focused on finishing this second edition of the book. I thank her for that. Thank you, also, to my Department Chair, André van der Hoek, and to my Dean, Marios Papaefthymiou, for letting me go on sabbatical in 2018. That allowed me to dive into the world of machine learning. Finally, I want to thank the hundreds of students who have taken my course, and who have enthusiastically provided all sorts of feedback.

**Cristina Videira Lopes**

*Irvine, February 29, 2020*



---

# Preface to the First Edition

---

## THE CODE

---

This book is a companion text for code that is publicly available at <http://github.com/crista/exercises-in-programming-style>

## WHO WILL BENEFIT FROM THIS BOOK

---

The collection of code that is the foundation of this book is for everyone who enjoys the art of programming. I've written this book in order to complement and explain the raw code, as some of the idioms may not be obvious. Software developers with many years of experience may enjoy revisiting familiar programming styles in the broad context of this book and learning about styles that may not be part of their normal repertoire.

This book can be used as a textbook for advanced programming courses in computer science and software engineering. Additional teaching materials, such as lecture slides, are also available. The book is not designed for introductory programming courses; it is important for students to be able to crawl (i.e. learn to program under the illusion that there's only one way of doing things) before they can run (i.e. realize that there's a lot more variety). I expect that many of the readers will be students in their junior/senior years or in their early stages of graduate study. The exercise list at the end of each chapter is a good mechanism for testing the reader's

understanding of each style. The suggested further readings are more appropriate for graduate students.

This book may also be of interest to writers, especially those who know a little programming or have a strong interest in programming technology. Despite important differences, there are many similarities between writing programs and writing in general.

## MOTIVATION FOR THESE EXERCISES

---

In the 1940s, the French writer Raymond Queneau wrote a jewel of a book called *Exercises in Style*, featuring 99 renditions of the exact same story, each written in a different style. The book is a masterpiece of writing technique, as it illustrates the many different ways a story can be told. The story being fairly trivial and always the same, highlights form, rather than content; it illustrates how the decisions we make in telling a story affect the perception of that story.

Queneau's story is trivially simple and can be told in two sentences: The narrator is on the "S" bus and notices a man with a long neck who is wearing a hat, and who gets into an altercation with the man sitting next to him. Two hours later, the narrator sees this same man near the Saint Lazare train station, with a friend, and the friend is giving this man some advice regarding an extra button on his overcoat. That's it! He then goes through 99 renditions of this story using, for example, litotes, metaphors, animism, etc.

Over the years, as an instructor of many programming-intensive courses, I noticed that often students have a hard time understanding the different ways of writing programs and of designing systems, in general. They have been trained in one, at most two, programming languages, so they understand only the styles that are encouraged by those languages, and have a hard time wrapping their heads around other styles. It's not their fault. Looking at the history of programming languages and the lack of pedagogical material on style in most computer science programs, one hardly gets exposed to the issue until after an enormous amount of experience is accumulated. Even then, style is seen as an intangible property of programs that

remains elusive to explain to others – and over which many technical arguments ensue. So, in order to give programming styles the proper due, and inspired by Queneau, I decided to embark on the project of writing the exact same computational task in as many styles as I have come across over the years.

So what is *style*? In Queneau’s circle of intellectuals, a group known as *Oulipo* (for French *Ouvroir de la littérature potentielle*, roughly translated as “workshop of potential literature”), style was nothing but the consequence of *creating under constraints*, often based on mathematical concepts such as permutations or lipograms. These constraints are used as a means to create something intellectually interesting besides the story itself. The ideas caught on, and over the years, several literary works have been created using Oulipo’s constraints.

In this book, too, programming style is what results from writing programs under a set of constraints. Constraints can come from external sources or they can be self-imposed; they can capture true challenges of the environment or they can be artificial; they can come from past experiences and measurable data or they can come from personal preferences. Independent of their origin, constraints are the seeds of style. By honoring different constraints, we can write a variety of programs that are virtually identical in terms of *what* they do, but that are radically different in terms of *how* they do it.

In the universe of all things a good programmer must know, I see collections of programming styles as being as important as any collection of data structures and algorithms, but with a focus on human effects rather than on computing effects. Programs convey information not just to the computers but, more importantly, to the people who read them. As with any form of expression, the consequences of *what* is being said are shaped and influenced by *how* they are being said. An advanced programmer needs not be able to just write correct programs that perform well; he/she needs to be able to choose appropriate styles for expressing those programs for a variety of purposes.

Traditionally, however, it has been much easier to teach algorithms and data structures than it is to teach the nuances of programming expression. Books on data structures and

algorithms all follow more or less the same formula: pseudo-code, explanation, and complexity analysis. The literature on programming tends to fall into two camps: books that explain programming *languages* and books that present collections of design or architectural *patterns*. However, there is a continuum in the spectrum of how to write programs that go from the concepts that the programming languages encourage/enforce to the combination of program elements that end up making up the program; languages and patterns feed on each other, and separating them as two different things creates a false dichotomy. Having come across Queneau's body of work, it seemed to me that his focus on *constraints* as the basis for explaining expression styles was a perfectly good model for unifying a lot of important creative work in the programming world.

I should note that I'm not the first one to look at constraints as a good unifying principle for explaining style in software systems. The work on *architectural styles* has taken that approach for a long time. I confess that the notion that style arises from constraints (some things are disallowed, some things must exist, some things are limited, etc.) was a bit hard to understand at first. After all, who wants to write programs under constraints? It wasn't until I came across Queneau's work that the idea made perfect sense.

Like Queneau's story, the computational task in this book is trivial: given a text file, we want to produce the list of words in the file and their frequencies, and print them out in decreasing order of frequency. This computational task is known as **term frequency**. This book contains 33 different styles for writing the term frequency task, one in each chapter. Unlike Queneau's book, I decided to verbalize the constraints in each style and explain the example programs. Given the target audience, I think it's important to provide those insights explicitly rather than leaving them to the reader's interpretation. Each chapter starts by presenting the constraints of the style, then it shows an example program; a detailed explanation of the code follows; most chapters have additional sections regarding the use of the style in systems design and another section on the historical context in which the programming style emerged. History is

important; a discipline should not forget the origins of its core ideas. I hope the readers will be curious enough to follow through some of the suggested further readings.

Why 33 styles? I chose 33 as a bounded personal challenge. Queneau's book has 99 styles. Had I set my goal to writing a book with 99 chapters, I probably never would have finished it! The public repository of code that is the basis for this book, however, is likely to continue to grow. The styles are grouped into nine categories: historical, basic, function composition, objects and object interactions, reflection and metaprogramming, adversity, data-centric, concurrency, and interactivity. The categories emerged as a way to organize the book, grouping together styles that are more related to each other than to the others. Other categorizations would be possible.

Similar to Queneau's book, these exercises in programming style are exactly that: *exercises*. They are the sketches, or arpeggios, of software; they aren't the music. A piece of real software usually employs a variety of styles for the different parts of the system. Furthermore, all these styles can be mixed and matched, creating hybrids that are interesting in themselves.

Finally, one last important remark. Although Queneau's book was the inspiration for this project, software is not exactly the same as the language arts; there are utility functions attached to software design decisions, i.e. some expressions are better than others for specific objectives.<sup>1</sup> In this book I try to stand clear of judgments of good and bad, except in certain clear cases. It is not up to me to make those judgments, since they depend heavily on the context of each project.

## ACKNOWLEDGMENTS

---

I would like to thank the following people for valuable feedback on earlier drafts of this book: Richard Gabriel, Andrew Black, Guy Steele, James Noble, Paul Steckler, Paul McJones, Laurie Tratt, Tijs van der Storm, and the students of INF 212 / CS 235 (Winter 14) at UC Irvine, especially Matias Giorgio and

David Dinh.

Thanks also to members of the IFIP Working Group 2.16, where I first presented the idea of this book, and whose reactions were critical for shaping the material.

A special thanks to the contributors to the exercises-in-style code repository so far: Peter Norvig, Kyle Kingsbury, Sara Triplett, Jørgen Edelbo, Darius Bacon, Eugenia Grabielova, Kun Hu, Bruce Adams, Krishnan Raman, Matias Giorgio, David Foster, Chad Whitacre, Jeremy MacCabe, and Mircea Lungu.

---

<sup>1</sup> Maybe that's also the case for the language arts, but I'm afraid I don't know enough!

---

# Prologue

---

## TERM FREQUENCY

---

**L**IKE QUENEAU'S STORY, the computational task in this book is trivial: given a text file, we want to display the N (e.g. 25) most frequent words and corresponding frequencies ordered by decreasing value of frequency. We should make sure to normalize for capitalization and to ignore stop words like "the," "for," etc. To keep things simple, we don't care about the ordering of words that have equal frequencies. This computational task is known as **term frequency**.

Here is an example of an input file and corresponding output after computing the term frequency:

Input:

```
White tigers live mostly in India
Wild lions live mostly in Africa
```

Output:

```
live - 2
mostly - 2
africa - 1
india - 1
lions - 1
tigers - 1
white - 1
wild - 1
```

If we were to run this flavor of term frequency on Jane Austen's *Pride and Prejudice* available from the Gutenberg Collection, we would get the following output:

```
mr - 786
elizabeth - 635
very - 488
darcy - 418
such - 395
mrs - 343
much - 329
more - 327
bennet - 323
bingley - 306
jane - 295
miss - 283
one - 275
know - 239
before - 229
herself - 227
though - 226
well - 224
never - 220
sister - 218
soon - 216
think - 211
now - 209
time - 203
good - 201
```

This book's example programs cover this term frequency task. Additionally, all chapters have a list of exercises. One of those exercises is to write another simple computational task using the corresponding style. Some suggestions are given below.



These computational tasks are simple enough for any advanced student to tackle easily. Algorithmic difficulties out of the way, the focus should be on following the constraints that underlie each style.

## WORD INDEX

---

Given a text file, output all words alphabetically, along with the page numbers on which they occur. Ignore all words that occur more than 100 times. Assume that a page is a sequence of 45 lines. For example, given *Pride and Prejudice*, the first few entries of the index would be:

abatement - 89  
abhorrence - 101, 145, 152, 241, 274, 281  
abhorrent - 253  
abide - 158, 292  
...

## WORDS IN CONTEXT

---

Given a text file, display certain words alphabetically and in context, along with the page numbers of the pages in which they occur. Assume that a page is a sequence of 45 lines. Assume that context consists of the preceding and succeeding two words. Ignore punctuation. For example, given *Pride and Prejudice*, the words “concealment” and “hurt” would result in the following output:

perhaps this **concealment** this disguise - 150  
purpose of **concealment** for no - 207  
pride was **hurt** he suffered - 87  
must be **hurt** by such - 95  
and are **hurt** if i - 103  
pride been **hurt** by my - 145  
must be **hurt** by such - 157  
infamy was **hurt** and distressed - 248

Suggestion of words for the words in context task: concealment, discontented, hurt, agitation, mortifying, reproach, unexpected, indignation, mistake, and confusion.

## PYTHONISMS

---

The example code used in this book is all written in Python, but expertise in Python is not necessary in order to understand the styles. In fact, one of the exercises in all of the chapters is to write the example program in another language. As such, the reader needs only to be able to *read* Python without needing to *write* in Python.

Python is relatively easy to read. There are, however, a few corners of the language that may confuse readers coming from other languages. I explain some of them here.

- **Lists.** In Python, a list is a primitive data type supported by dedicated syntax that is normally associated with arrays in C-like languages. Here is an example of a list: `mylist = [0, 1, 2, 3, 4, 5]`. Python doesn't have an array as a primitive data type,<sup>2</sup> and most situations that would use an array in C-like languages use a list in Python.
- **Tuples.** A tuple is an immutable list. Tuples are also primitive data types supported by dedicated syntax that is normally associated with lists in Lisp-like languages. Here is an example of a tuple: `mytuple = (0, 1, 2, 3, 4)`. Tuples and lists are handled in similar ways, except for the fact that tuples are immutable, so the operations that change lists don't work on tuples.
- **List indexing.** List and tuple elements are accessed by index like this: `mylist[some_index]`. The lower bound of a list is index 0, like in C-like languages, and the list length is given by `len(mylist)`. Indexing a list can be much more expressive than this simple example suggests. Here are some more examples:
  - `mylist[0]` – first element of the list

- `mylist[-1]` - last element of the list
  - `mylist[-2]` - next-to-last element of the list
  - `mylist[1:]` - a list starting at index 1 until the end of `mylist`
  - `mylist[1:3]` - a list starting at index 1 and stopping before index 3 of `mylist`
  - `mylist[::2]` - a list containing every other element of `mylist`
  - `mylist[start:stop:step]` - a list containing every `step` element between `start` and `stop` indexes of `mylist`
- **Bounds.** Indexing an element beyond the length of a list results in an `IndexError`. For example, trying to access the 4th element of a list of 3 elements (e.g. `[10, 20, 30][3]`) results in an `IndexError`, as expected. However, many Python operations on lists (and collections in general) are constructivist with respect to indexing. For example, obtaining a list consisting of the range from 3 to 100 in a list with only 3 elements (e.g. `[10, 20, 30][3:100]`) results in an empty list (`[]`) rather than an `IndexError`. Similarly, any range that partially covers a list results in whatever part of the list is covered, with no `IndexError` (e.g. `[10, 20, 30][2:10]` results in `[30]`). This constructivist behavior may be puzzling at first for people used to more intolerant languages.
  - **Dictionaries.** In Python, a dictionary, or map, is also a primitive data type supported by dedicated syntax. Here is an example of a dictionary: `mydict = {'a' : 1, 'b' : 2}`. This particular dictionary maps two string keys to two integer values; in general, keys and values can be of any type. In Java, these kinds of dictionaries can be found in the form of the `HashMap` class (among others), and in C++ they can be found in the form of the class template `map` (among others).

- **self.** In most object-oriented languages, the reference that an object has to itself is implicitly available through special syntax. For example, `this` in Java and C++, `$this` in PHP, or `@` in Ruby. Unlike these languages, Python has no special syntax for it. Moreover, instance methods are simply class methods that take an object as the first parameter; this first parameter is called `self` by convention, but not by special mandate of the language. Here is an example of a class definition with two instance methods:

---

```
1 class Example:
2     def set_name(self, n):
3         self._name = n
4     def say_my_name(self):
5         print self._name
```

---

Both methods have a parameter named `self` in the first position, which is then accessed in their bodies. There is nothing special about the word `self`, and the methods could use any other name, for example `me` or `my` or even `this`, but any word other than `self` will be frowned upon by Python programmers. Calling instance methods, however, may be surprising, because the first parameter is omitted:

```
e = Example()
e.set_my_name(``Heisenberg'`)
e.say_my_name()
```

This mismatch on the number of parameters is due to the fact that the dot-notation in Python (`.'`) is simply syntactic sugar for this other, more primitive form of calling the methods:

```
e = Example()
Example.set_my_name(e,
``Heisenberg'`')
```

Example.say\_my\_name(e)

- **Constructors.** In Python, a constructor is a regular method with the name `__init__` (two underscores on each side of the word). Methods with this exact name are called automatically by the Python runtime right after object creation. Here is one example of a class with a constructor, and its use:

---

```
1 class Example:
2     # This is the constructor of this class
3     def __init__(self, n):
4         self._name = n
5     def say_my_name(self):
6         print self._name
7
8 e = Example('Heisenberg')
9 e.say_my_name()
```

---

<sup>2</sup> There is an array data object, but it's not a primitive type of the language and it doesn't have any special syntax. It's not used as much as lists.

---

# Author

---

**Cristina (Crista) Videira Lopes** is a Professor of Software Engineering at the Donald Bren School of Information and Computer Sciences, University of California, Irvine. Her research focuses on software engineering for large-scale data and systems. Early in her career, she was a founding member of the team at Xerox PARC that developed Aspect-Oriented Programming and AspectJ. Along with her research program, she is also a prolific software developer. Her open source contributions include acoustic software modems, and the virtual world server OpenSimulator. She is a co-founder of a company specializing in online virtual reality for early-stage sustainable urban redevelopment projects. She developed and maintains a search engine for OpenSimulator-based virtual worlds.

Dr. Lopes has a PhD from Northeastern University, and MS and BS degrees from Instituto Superior Técnico in Portugal. She is the recipient of several National Science Foundation grants, including a prestigious CAREER Award. She is an ACM Distinguished Scientist and an IEEE Fellow.

I

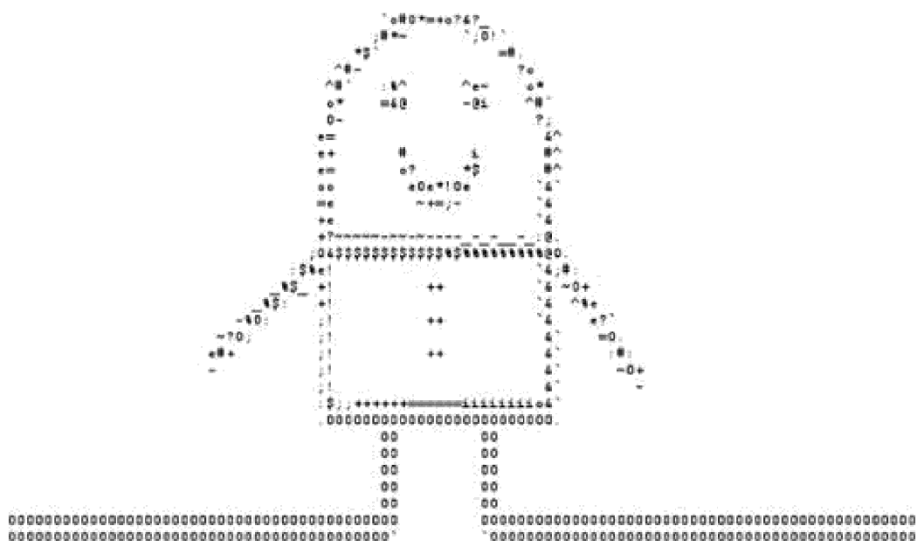
Historical

Computing systems are like an onion, with layers upon layers of abstraction developed over the years in order to facilitate the expression of intent. It is important to know what the inner layers really entail. The first three programming styles illustrate what programming was like several decades ago, and to some extent, what it still is – because ideas keep getting reinvented.



## Good Old Times

---



### 1.1 CONSTRAINTS

---

- ▶ Very small amount of primary memory, typically orders of magnitude smaller than the data that needs to be processed/generated.
- ▶ No identifiers – i.e. no variable names or tagged memory addresses. All we have is memory that is addressable with numbers.

## 1.2 A PROGRAM IN THIS STYLE

---

```
1 #!/usr/bin/env python
2 import sys, os, string
3
4 # Utility for handling the intermediate 'secondary memory'
5 def touchopen(filename, *args, **kwargs):
6     try:
7         os.remove(filename)
8     except OSError:
9         pass
10    open(filename, "a").close() # "touch" file
11    return open(filename, *args, **kwargs)
12
13 # The constrained memory should have no more than 1024 cells
14 data = []
15 # We're lucky:
16 # The stop words are only 556 characters and the lines are all
17 # less than 80 characters, so we can use that knowledge to
18 # simplify the problem: we can have the stop words loaded in
19 # memory while processing one line of the input at a time.
20 # If these two assumptions didn't hold, the algorithm would
21 # need to be changed considerably.
22
23 # Overall strategy: (PART 1) read the input file, count the
24 # words, increment/store counts in secondary memory (a file)
25 # (PART 2) find the 25 most frequent words in secondary memory
26
27 # PART 1:
28 # - read the input file one line at a time
29 # - filter the characters, normalize to lower case
30 # - identify words, increment corresponding counts in file
31
32 # Load the list of stop words
33 f = open('../stop_words.txt')
34 data = [f.read(1024).split(',')] # data[0] holds the stop words
35 f.close()
36
37 data.append([]) # data[1] is line (max 80 characters)
38 data.append(None) # data[2] is index of the start_char of word
39 data.append(0) # data[3] is index on characters, i = 0
40 data.append(False) # data[4] is flag indicating if word was found
41 data.append('') # data[5] is the word
42 data.append('') # data[6] is word,NNNN
43 data.append(0) # data[7] is frequency
44
45 # Open the secondary memory
46 word_freqs = touchopen('word_freqs', 'rb+')
47 # Open the input file
48 f = open(sys.argv[1], 'r')
49 # Loop over input file's lines
50 while True:
51     data[1] = [f.readline()]
52     if data[1] == ['']: # end of input file
53         break
```

```

54     if data[1][0][len(data[1][0])-1] != '\n': # If it does not end
55         with \n
56         data[1][0] = data[1][0] + '\n' # Add \n
57     data[2] = None
58     data[3] = 0
59     # Loop over characters in the line
60     for c in data[1][0]: # elimination of symbol c is exercise
61         if data[2] == None:
62             if c.isalnum():
63                 # We found the start of a word
64                 data[2] = data[3]
65             else:
66                 if not c.isalnum():
67                     # We found the end of a word. Process it
68                     data[4] = False
69                     data[5] = data[1][0][data[2]:data[3]].lower()
70                     # Ignore words with len < 2, and stop words
71                     if len(data[5]) >= 2 and data[5] not in data[0]:
72                         # Let's see if it already exists
73                         while True:
74                             data[6] = str(word_freqs.readline().strip
75                                 (), 'utf-8')
76                             if data[6] == '':
77                                 break;
78                             data[7] = int(data[6].split(',')[1])
79                             # word, no white space
80                             data[6] = data[6].split(',')[0].strip()
81                             if data[5] == data[6]:
82                                 data[7] += 1
83                                 data[4] = True
84                                 break
85                             if not data[4]:
86                                 word_freqs.seek(0, 1) # Needed in Windows
87                                 word_freqs.write(bytes("%20s,%04d\n" % (
88                                     data[5], 1), 'utf-8'))
89                             else:
90                                 word_freqs.seek(-26, 1)
91                                 word_freqs.write(bytes("%20s,%04d\n" % (
92                                     data[5], data[7]), 'utf-8'))
93                             word_freqs.seek(0,0)
94                             # Let's reset
95                             data[2] = None
96                             data[3] += 1
97     # We're done with the input file.
98     f.close()
99     word_freqs.flush()
100
101 # PART 2
102 # Now we need to find the 25 most frequently occurring words.
103 # We don't need anything from the previous values in memory
104 del data[:]
105
106 # Let's use the first 25 entries for the top 25 words
107 data = data + [[]]*(25 - len(data))
108 data.append('') # data[25] is word, freq from file
109 data.append(0) # data[26] is freq
110

```

```

107 # Loop over secondary memory file
108 while True:
109     data[25] = str(word_freqs.readline().strip(), 'utf-8')
110     if data[25] == '': # EOF
111         break
112     data[26] = int(data[25].split(',')[1]) # Read it as integer
113     data[25] = data[25].split(',')[0].strip() # word
114     # Check if this word has more counts than the ones in memory
115     for i in range(25): # elimination of symbol i is exercise
116         if data[i] == [] or data[i][1] < data[26]:
117             data.insert(i, [data[25], data[26]])
118             del data[26] # delete the last element
119         break
120
121 for tf in data[0:25]: # elimination of symbol tf is exercise
122     if len(tf) == 2:
123         print(tf[0], '-', tf[1])
124 # We're done
125 word_freqs.close()

```

---

**Note:** If not familiar with Python, please refer to the Prologue (Pythonisms) for an explanation of lists, indexes, and bounds.

## 1.3 COMMENTARY

---

IN THIS STYLE, the program reflects the constrained computing environment where it executes. The memory limitations force the programmer to come up with ways of rotating data through the available memory, adding complexity to the computational task at hand. Additionally, the absence of identifiers results in programs where the natural terminology of the problem is absent from the program text, and, instead, is added through comments and documentation. This is what programming was all about in the early 1950s. This style of programming, however, is not extinct; it is still in use today when dealing directly with hardware and when optimizing the use of memory.

The example program may look quite foreign to programmers not used to these kinds of constraints. While this is certainly a program that one doesn't associate with Python or with any of the modern programming languages, it embodies the theme of this book quite well: programming styles emerge from *constraints*. Very often, the constraints are imposed externally – maybe the hardware has limited memory, maybe the

assembly language doesn't support identifiers, maybe performance is critical and one must deal directly with the machine, etc.; other times the constraints are self-imposed: the programmer, or the entire development team, decides to adhere to certain ways of thinking about the problems and of writing the code, for many different reasons – maintainability, readability, extensibility, adequacy for the problem domain, past experiences on the part of the developers; or simply, as is the case here, to teach what low-level programming looks like without having to learn new syntax. Indeed, it is possible to write low-level, Good Old Times style programs in just about any programming language!

Having explained the reason for this unusual implementation of term frequency, let's dive into this program. The memory limitations are such that we can't ignore the size of the data to be processed. In the example, we have self-imposed a size of 1024 memory cells (line #15). The term "memory cell" is used here in a somewhat fuzzy manner to denote, roughly, a piece of simple data, such as a character or a number. Given that books like *Pride and Prejudice* contain much more than 1024 characters, we need to come up with ways to read and process the data in small chunks, making heavy use of "secondary memory" (a file) to store the data that doesn't fit in primary memory. Before we start coding, we need to do some back-of-the-envelope calculations about the different options regarding what to hold in primary memory and what to dump to secondary memory, and when (see comments in lines #16 through #26). Then as now, access to primary memory is orders of magnitude faster than access to secondary memory, so these calculations are about optimizing for performance.

Many options could have been pursued, and the reader is encouraged to explore the solution space within this style. The example program is divided into two distinct parts: the first part (lines #28 through #98) processes the input file, counting word occurrences and writing that data into a word-frequency file; the second part (lines #100 through #128) processes the intermediate word-frequency file in order to discover the 25 most frequently occurring words, printing them at the end.

The first part of the program works as follows:

- Hold the stop words, roughly 500 characters, in primary memory (lines #33 through #36)
- Read the input file one line at a time; each line is only 80 characters maximum (lines #50 through #95)
- For each line (lines #60 through #95), filter the characters, identify the words, and normalize them to lowercase
- Retrieve/Write the words and their frequencies from/to secondary memory (lines #73 through #90)

After processing the entire input file like this, we then turn our attention to the word frequencies that have been accumulated in the intermediate file. We need a sorted list of the most frequently occurring words, so the program does the following:

- Keep an ordered list in memory holding the current 25 most frequently occurring words, and their frequencies (line #104)
- Read one line at a time from the file. Each line contains a word and its corresponding frequency (lines #108 through #120)
- If the new word has higher frequency than any of the words in memory, insert it at the appropriate place in the list and remove the word at the end of the list (lines #116 through #120)
- Finally, print the 25 top words and their frequencies (lines #122 through #124) and close the intermediate file (line #126)

As seen, the memory constraint has a strong effect on the algorithm employed, as we must be mindful of how much data there is in memory at any given point in time.

The second self-imposed constraint of this style is the absence of identifiers. This second constraint also has a strong effect on

the program, but this effect is of a different nature: readability. There are no variables, as such; there is only a data memory that is accessed by indexing it with numbers. The problem's natural concepts (words, frequencies, counts, sorting, etc.) are completely absent from the program text, and are, instead, indirectly represented as indexes over memory. The only way we can bring those concepts back in is by adding comments that explain what kinds of data the memory cells hold (e.g. see comments in lines #38 through #44 and #103 through #106, among others). When reading through the program, we often need to go back to those comments to remind ourselves what high-level concept a certain memory index corresponds to.

## 1.4 THIS STYLE IN SYSTEMS DESIGN

---

In the age of computers with multi-gigabyte RAM, constrained memory such as that shown here is mostly a vague memory from the past. Nevertheless, with modern programming languages that encourage obliviousness with respect to memory management, and with the ever-growing amounts of data that modern programs handle, it is very easy to let memory consumption of programs run out of control, with negative consequences on run-time performance. A certain amount of awareness regarding the consequences that the different programming styles carry for memory usage is always a good thing.

Many applications these days – namely those falling in what's known as *Big Data* – have brought the complexities of dealing with small memory back to the center of attention. In this case, although memory is not scarce in absolute terms, it is much smaller than the size of the data to be processed. For example, if instead of just *Pride and Prejudice* we would apply term frequency to the entire Gutenberg Collection, we would likely not be able to hold all the books in memory at the same time; we might not even be able to hold the list of word frequencies all in memory either. Once the data can't fit in memory all at once, developers must devise smart schemes for (1) representing data so that more of it can fit in memory at any given time;

and (2) rotating the data through primary and secondary memory. Programming with these constraints tends to make programs feel like the Good Old Times style.

Regarding the absence of names, one of the main drivers behind programming language evolution throughout the 1950s and 1960s was precisely the elimination of cognitive indirections such as those shown in the example: we want the program texts to reflect the high-level concepts of the domain as much as possible, instead of reflecting the low-level machine concepts and relying on external documentation to do the mapping between the two. But even though programming languages have provided for user-defined named abstractions for a long time, it is not unusual for programmers to fail to name their program elements, Application Programming Interfaces (APIs) and entire components appropriately, resulting in programs, libraries and systems as obscure as the one shown here.

Let this Good Old Times style be a reminder of how thankful we should be for being able to hold so much data in memory and for being able to give appropriate names to each and every one of our program elements!

## 1.5 HISTORICAL NOTES

---

This style of programming came directly from the first viable model of computation, the Turing Machine. A Turing Machine consists of an unbounded modifiable state “tape” (the data memory) and a state machine that reads and modifies that state. The Turing Machine had a tremendous influence in the development of computers and how they were programmed. Turing’s ideas influenced von Neumann’s design of the first computer with stored programs. Turing himself also wrote the specifications of a computing machine known as the Automatic Computing Engine (ACE), which was, in many ways, more advanced than von Neumann’s. Because that report was classified by the British government, and also because of the politics following the Second World War, Turing’s design was not acted upon until several years later, and still in secrecy. Both von Neumann’s architecture and Turing’s machines led to



the first programming languages in the 1950s, which enforced the concept of programming by reusing and changing state in memory over time.

## 1.6 FURTHER READING

---

---

Bashe, C., Johnson, L., Palmer, J. and Pugh, E. (1986). *IBM's Early Computers: A Technical History* (History of Computing), MIT Press, Cambridge, MA.

*Synopsis:* IBM was the major commercial player in the early days of computing machines. This book tells the story of IBM's transition from manufacturer of electromechanical machines to a powerhouse of computing machines.

Carpenter, B.E. and Doran, R.W. (1977). The other Turing Machine. *Computer Journal* **20**(3): 269–279.

*Synopsis:* An account of one of Turing's technical reports describing a complete architecture for a computing machine based on von Neumann's, but including subroutines, stacks and much more. The original report can be found at <http://www.npl.co.uk/about/history/notable-individuals/turing/ace-proposal>

Turing, A. (1936). On computable numbers, with an application to the Entscheidungs problem. *Proceedings of the London Mathematical Society* **2**(42): 230–265.

*Synopsis:* The original “Turing Machine.” In the context of this book, this paper is suggested not for its mathematics but for the programming model of a Turing Machine: a tape with symbols, a tape reader/writer that moves left and right, and the overwriting of symbols on the tape.

von Neumann, J. (1945). First draft of a report on the EDVAC. Reprinted in *IEEE Annals of the History of Computing* **15**(4): 27–43, 1993.

*Synopsis:* The original “von Neumann architecture.” As with Turing's paper, suggested for the programming model.

## 1.7 GLOSSARY

---

---

**Main memory:** Often referred to simply as *memory*, this data storage is directly accessible by the CPU. Most data in this storage is volatile in the sense that it does not persist beyond the execution of the programs that use it and also does not persist upon machine power downs. These days, the main memory is random access memory (RAM), meaning that the CPU can address any cell in it quickly, as opposed to having to scan sequentially.

**Secondary memory:** In contrast to primary memory, secondary memory refers to any storage facility that is not directly accessible by the CPU and that, instead, is indirectly accessed via input/output channels. Data in secondary memory persists in the device through power downs and until it is explicitly deleted. In modern computers, hard disk drives and “pen” drives are the most common secondary storage forms. Access to secondary memory is several orders of magnitude slower than access to primary memory.

## 1.8 EXERCISES

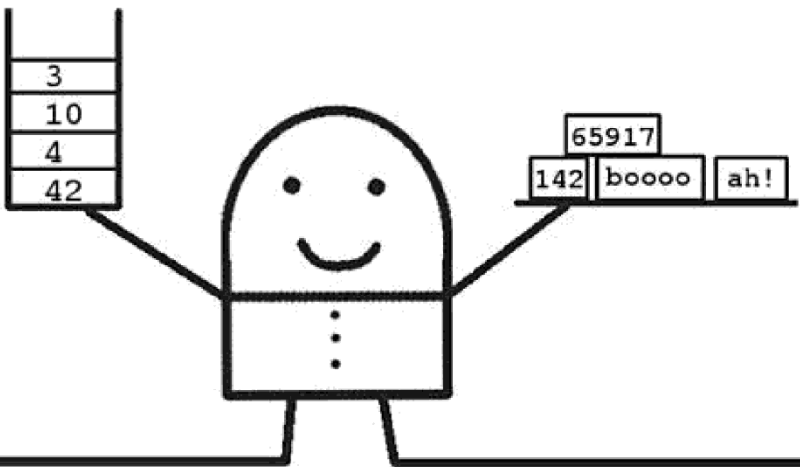
---

---

- 1.1 *Another language.* Implement the example program in another language, but preserve the style.
- 1.2 *No identifiers.* The example program still has a few identifiers left, namely in lines #60 (*c*), #116 (*i*) and #122 (*tf*). Change the program so that these identifiers are also eliminated.
- 1.3 *More lines.* The example program reads one line at a time into memory. In doing so it is underutilizing the main memory. Modify the program so that it loads as many lines as possible into memory without going over the established limit of 1024 memory cells. Justify the number of lines you chose. Check if your version runs faster than the original example program, and explain the result.
- 1.4 *A different task.* Write one of the tasks proposed in the Prologue using the Good Old Times style.

## Go Forth

---



### 2.1 CONSTRAINTS

---

- ▷ Existence of a data stack. All operations (conditionals, arithmetic, etc.) are done over data on the stack.
- ▷ Existence of a heap for storing data that's needed for later operations. The heap data can be associated with names (i.e. variables). As said above, all operations are done over data on the stack, so any heap data that needs to be operated upon needs to be moved first to the stack and eventually back to the heap.
- ▷ Abstraction in the form of user-defined “procedures” (i.e.

names bound to a set of instructions), which may be called something else entirely.

## **2.2 A PROGRAM IN THIS STYLE**

---

---

```

1  #!/usr/bin/env python
2  import sys, re, operator, string
3
4  #
5  # The all-important data stack
6  #
7  stack = []
8
9  #
10 # The heap. Maps names to data (i.e. variables)
11 #
12 heap = {}
13
14 #
15 # The new "words" (procedures) of our program
16 #
17 def read_file():
18     """
19     Takes a path to a file on the stack and places the entire
20     contents of the file back on the stack.
21     """
22     f = open(stack.pop())
23     # Push the result onto the stack
24     stack.append([f.read()])
25     f.close()
26
27 def filter_chars():
28     """
29     Takes data on the stack and places back a copy with all
30     nonalphanumeric chars replaced by white space.
31     """
32     # This is not in style. RE is too high-level, but using it
33     # for doing this fast and short. Push the pattern onto stack
34     stack.append(re.compile('[\W_]+'))
35     # Push the result onto the stack
36     stack.append([stack.pop().sub(' ', stack.pop()[0]).lower()])
37
38 def scan():
39     """
40     Takes a string on the stack and scans for words, placing
41     the list of words back on the stack
42     """
43     # Again, split() is too high-level for this style, but using
44     # it for doing this fast and short. Left as exercise.
45     stack.extend(stack.pop()[0].split())
46
47 def remove_stop_words():
48     """
49     Takes a list of words on the stack and removes stop words.
50     """
51     f = open('./stop_words.txt')
52     stack.append(f.read().split(','))
53     f.close()
54     # add single-letter words

```

```

55     stack[-1].extend(list(string.ascii_lowercase))
56     heap['stop_words'] = stack.pop()
57     # Again, this is too high-level for this style, but using it
58     # for doing this fast and short. Left as exercise.
59     heap['words'] = []
60     while len(stack) > 0:
61         if stack[-1] in heap['stop_words']:
62             stack.pop() # pop it and drop it
63         else:
64             heap['words'].append(stack.pop()) # pop it, store it
65             stack.extend(heap['words']) # Load the words onto the stack
66     del heap['stop_words']; del heap['words'] # Not needed
67
68 def frequencies():
69     """
70     Takes a list of words and returns a dictionary associating
71     words with frequencies of occurrence.
72     """
73     heap['word_freqs'] = {}
74     # A little flavour of the real Forth style here...
75     while len(stack) > 0:
76         # ... but the following line is not in style, because the
77         # naive implementation would be too slow.
78         if stack[-1] in heap['word_freqs']:
79             # Increment the frequency, postfix style: f 1 +
80             stack.append(heap['word_freqs'][stack[-1]]) # push f
81             stack.append(1) # push 1
82             stack.append(stack.pop() + stack.pop()) # add
83         else:
84             stack.append(1) # Push 1 in stack[2]
85             # Load the updated freq back onto the heap
86             heap['word_freqs'][stack.pop()] = stack.pop()
87
88     # Push the result onto the stack
89     stack.append(heap['word_freqs'])
90     del heap['word_freqs'] # We don't need this variable anymore
91
92 def sort():
93     # Not in style, left as exercise
94     stack.extend(sorted(stack.pop().items(), key=operator.
95                          itemgetter(1)))
96
97 # The main function
98 #
99 stack.append(sys.argv[1])
100 read_file(); filter_chars(); scan(); remove_stop_words()
101 frequencies(); sort()
102
103 stack.append(0)
104 # Check stack length against 1, because after we process
105 # the last word there will be one item left
106 while stack[-1] < 25 and len(stack) > 1:
107     heap['i'] = stack.pop()
108     (w, f) = stack.pop(); print(w, '-', f)
109     stack.append(heap['i']); stack.append(1)
110     stack.append(stack.pop() + stack.pop())

```

---

**Note:** If not familiar with Python, please refer to the Prologue (Pythonisms) for an explanation of lists, indexes, and bounds.

## 2.3 COMMENTARY

---

**T**HIS STYLE takes inspiration in Forth, a small programming language first developed as a personal programming system in the late 1950s by Charles Moore, a programmer working at the time for the Smithsonian Astrophysical Laboratory. This programming system – an interpreter for a simple language – supported the need to handle different equations without having to recompile the program – a time-consuming activity at the time.

This curious little language has at its heart the concept of a stack. Equations are entered in postfix notation, e.g. “3 4 +”. Operands are pushed onto the stack one at a time; operators take the operands on the stack and replace them with the result. When data is not immediately needed, it can be placed on a part of the memory called the heap. Besides the stack machine, Forth supports the definition of procedures (called “words” in Forth); these procedures, like the built-in ones, operate over data on the stack.

The syntax of Forth may be hard to understand due to its postfix nature and several special symbols that aren’t used in any other language. But the language model is surprisingly simple once we understand the constraints – the stack, the heap, procedures and names. Rather than using a Forth interpreter written in Python, this chapter shows how the constraints underlying Forth can be codified in Python programs, resulting, roughly, in a Forth style of programming. Let’s analyze the example program.

To start with, to support the style, we first define the stack (line #7) and the heap (line #12).<sup>1</sup> Next we define a set of procedures (“words” in Forth terminology). These procedures help us divide the problem in smaller sub-steps, such as reading a file (lines #17 through #25), filtering the characters (lines #27 through #36), scanning for words (lines #38 through

#45), removing stop words (lines #47 through #66), computing the frequencies (lines #68 through #90), and sorting the result (lines #92 through #94). We'll look into some of these in more detail next. But one thing to notice is that all these procedures use (pop) data from the stack (e.g. lines #22, #36, #45) and end by pushing data onto the stack (e.g. lines #24, #36, #45).

Forth's heap supports the allocation of data blocks that can be – and usually are – bound to names. In other words, variables. The mechanism is relatively low level, as the programmer needs to define the size of the data. In our emulation of Forth's style, we simply use a dictionary (line #12). So for example, in line #56, we are popping the stop words on the stack directly into a variable on the heap named `stop words`.

Many parts of the example program are not written in Forth style, but some parts are true to it, so let's focus on those. The procedure `remove stop words` (starting in line #47), as the name suggests, removes the stop words. When that procedure is called, the stack contains all the words of the input file, properly normalized. The first few words of *Pride and Prejudice* are:

```
['the', 'project', 'gutenberg', 'ebook', 'of', 'pride', 'and', 'prejudice', ...]
```

That is how the stack looks like at that point, for that book. Next, we open the stop words file and push the list of stop words onto the stack (lines #51 through #55). To make things simple, we keep them in a list of their own instead of merging them with the rest of the data on the stack. The stack now looks like this:

```
['the', 'project', 'gutenberg', 'ebook', 'of', 'pride', 'and', 'prejudice', ..., ['a', 'able', 'about', 'across', ...]]
```

After reading all the stop words from the file and placing them onto the stack, we then pop them out to the heap (line #56), in preparation to process the words of the book that are still on the stack. Lines #60 through #64 iterate through the words on the stack in the following way. Until the stack is empty (test in line #60), we check if the word at the top of the stack (`stack[-1]` in Python) is in the list of stop words (line



#61). In real Forth, this test would be much more low level than what is shown here, as we would need to explicitly iterate through the list of stop words too. In any case, if the word is in the stop words list, we simply pop it out of the stack and ignore it. If the word is not in the list of stop words (line #63), we pop it out of the stack onto another variable in the heap called `words` – the list accumulates the non-stop words (line #64). When the iteration is over, we take the variable `words` and place its entire contents back on the stack (line #65). We end up with the stack containing all non-stop words, like this:

```
['project', 'gutenberg', 'ebook', 'pride', 'prejudice', ...]
```

At that point, we don't need the variables on the heap anymore, so we discard them (line #66). Forth supports deletion of variables from the heap in the spirit of what is being done here.

The `frequencies` procedure (starting in line #68) shows one more stylistic detail related to arithmetic operations. That procedure starts with the non-stop words on the stack (as shown above) and ends with placing a dictionary of word frequencies onto the stack (line #89). It works as follows. First, it allocates a variable on the heap called `word_freqs` that stores the word-frequency pairs (line #73) – it starts with the empty dictionary. It then iterates through the words on the stack. For each word at the top of the stack, it checks whether that word has been seen before (line #78). Again, this test is expressed at a much higher level than it would be in Forth, for performance reasons. If the word has been seen before, we need to increment its frequency count. That is done by pushing the current frequency count onto the stack (line #80), then pushing the value 1 onto the stack (line #81), and then adding those 2 top-most operands on the stack and placing the result on the stack (line #82). If the word had not been seen before (line #83), we simply push the value 1 onto the stack. Finally, we pop both the frequency count (right side of assignment in line #86) and the word itself (left side of assignment in line #86) out of the stack and into the variable on the heap, and move to the next word on top of the stack, until the stack is empty (back to line #75). At the end, as stated before, we push the entire content of the heap variable onto the stack, and delete

that variable.

The main function starting in line #98 is the beginning of the program. We start by pushing the name of the input file onto the stack (line #98), and then invoke the procedures sequentially. Note that these procedures are not completely independent of each other: each of them relies on strong assumptions about the data that is left on the stack by the previous one.

Once the counting and sorting is done, we then print out the result (lines #105 through #109). This block of code shows a final stylistic detail related to what Forth calls “indefinite loops,” or loops that run until a condition is true. In our case, we want to iterate through the dictionary of word frequencies until we count 25 iterations. So we do the following. We start by pushing the number 0 onto the stack (line #102), on top of the data that is already there (word frequencies), and proceed to an indefinite loop until the top of the stack reaches the number 25. In each iteration, we pop the count out of the stack into a variable (line #106), then pop the next word and frequency out of the stack and print them (line #107); then, we push the count in the variable back to the stack, followed by the value 1, and add them, effectively incrementing the count. The loop, and the program, terminate when the top of the stack has the value 25. The second clause for termination (`len(stack) > 1`) is there in the case of small test files that might not even have 25 words.

Many options could have been pursued, and the reader is encouraged to explore the solution space within this style.

## 2.4 HISTORICAL NOTES

---

Early computing machines did not have stacks. The earliest reference for the idea of using a stack in computers can be found in Alan Turing’s Automatic Computing Engine (ACE) report in 1945. Unfortunately, that report was classified for many years, so not many knew about it.

Stacks were invented again in the late 1950s by several people independently. It was several more years before