Reusable vs. re-editable code

Konrad Hinsen

DEPARTMENT: Scientific Programming

Reusable vs. re-editable code

Konrad Hinsen *Centre de Biophysique Moléculaire, Orléans*

One of my current projects is a MOOC (Massively Open Online Course) on reproducible research that I am working on within a multidisciplinary team. If all goes well it will go on-line in autumn. French-speaking readers might want to watch out for it on the FUN platform (https://www.fun-mooc.fr/). But my topic for today is not the MOOC, it's a lesson learned during its preparation. To sum it up in a keyword, it's about the good and bad sides of reusable code.

One module of the MOOC shows how to prepare a simple data analysis as a reproducible computational document. We decided to present this example in several variants, using different languages and technologies. One variant uses a Jupyter notebook and the Python 3 language. The second one is done in R, using RStudio as an authoring tool for the R Markdown format. The final variant, for power users, mixes several programming languages using the org-mode extension to the Emacs editor. Our goal is to demonstrate the current best practices for each platform. Rather than developing a single strategy for the data analysis and translating it into three dialects, we start from scratch three times, asking ourselves how a competent user of each platform would approach the problem. An additional benefit of this approach is the possibility to compare the strategies that the three platforms encourage, and that is what I will do in the following.

Since we concentrate on reproducibility rather than on mathematics or statistics, the chosen analysis is simple: find the distribution of the severity of the annual flu epidemics in France from data on the number of diagnosed cases for each week since 1985. The data is publicly available from the "Réseau Sentinelles" (http://www.sentiweb.fr/). The essential steps are (1) download the data as a CSV file, (2) read the CSV file, (3) check for potential problems, in particular missing data points, (4) transform the weekly sampling into an annual sampling, and (5) produce a histogram of the annual number of flu victims.

Python and pandas

Since I am the Python guy of the team, my first assignment was to prepare the analysis as a Jupyter notebook using the Python 3 language. The standard tool in the Python ecosystem for working with tabular data is the pandas library (http://pandas.pydata.org/), so that's what I based my code on. It starts by downloading and parsing the data, which looks like this:

```
# reseau Sentinelles, INSERM, UPMC, http://www.sentiweb.fr
week,indicator,inc,inc_low,inc_up,inc100,inc100_low, ...
199501,3,26556,21916,31196,46,38,54,FR,France
199452,3,22036,17495,26577,38,30,46,FR,France
199451,3,26912,21750,32074,47,38,56,FR,France
199450,3,28831,23470,34192,50,41,59,FR,France
...
```

I have shortened the column name line a to make it fit into the page. The meaning of the columns is documented on the "Réseau Sentinelles" Web site. We only need two of them: `week` and `inc`. The former encodes a reference to each week in a somewhat uncommon format: it's a six-digit string with the first four digits indicating the year and the last two the week number according to the ISO-8601 standard. The `inc` column contains the number of reported flu cases during each week.

Reading such a dataset using pandas isn't difficult:

```
import pandas as pd

def data_url(year1, week1, year2, week2):
    return "http://websenti.u707.jussieu.fr/sentiweb/" + \
           "api/data/rest/getIncidenceFlat?indicator=3" + \
           ("&wstart=%4d%02d&wend=%4d%02d"
            % (year1, week1, year2, week2)) + \
           "&geo=PAY1&$format=csv"

data = pd.read_csv(data_url(1990, 1, 1995, 1), skiprows=1)
```

The argument `skiprows=1` tells pandas to ignore the first line of the downloaded file, which is a comment that doesn't fit into the CSV format. What we get back as `data` is a `DataFrame` object with 262 rows and 10 columns. When reading a CSV file, pandas tries to guess appropriate data types for each column. In this case, it decides on "int64" for both of the columns we need, although this isn't quite appropriate for `week`. An excellent opportunity to explain the subtleties of data representations and their conversions to the students in our MOOC!

Unfortunately, there is another subtlety to come. After testing the analysis code on the rather small five-year sample, we want to run it on the full dataset, which starts in 1985 and is continually updated. Using `data_url(1985, 1, 2018, 1)`, we get data for 33 years. On the screen it looks much the same. But a more careful inspection shows that the data type for `inc` has changed from "int64" to "object". Before going

through the 1723 lines to check for an anomaly that could explain this change, I took a look at the pandas documentation for CSV import (http://pandas.pydata.org/pandas-docs/stable/io.html). It mentions "intelligent conversion of tabular data" but the rules for this intelligent action are not given. In view of the length of the documentation, which spreads over several pages, I concluded that going through the 1723 lines of data is the lesser evil. And indeed the offending line clearly stands out visually:

```
198919,3,-,,,-,,,FR,France
```

A missing data point for week 19 of the year 1989 is indicated by a dash instead of a number in the `inc` column. Faced with a column containing many integers and one string, pandas apparently decided on the "object" type.

I won't discuss the many options of dealing with such nuisances here. They are the rule rather than the exception when dealing with data coming from the real world. In fact, the pandas documentation I just cited proposes a few optional arguments I could pass to the function `read_csv` in order to deal with the irregularity in the input file. What makes this problem unpleasant is not that it's difficult to deal with, but that it's difficult to identify. A data type chosen as a function of the data values means that the subsequent analysis code can never be trusted to work in general. All you can say is that it works for the data ranges that it has been explicitly tested on.

Emacs and org-mode

I won't say much about the R version of the data analysis, since it was prepared by my R-savvy colleagues. It isn't very different in spirit from the Python 3/pandas version, and requires similar decisions for how to deal with irregularities. What I will focus on is the multi-language version in the Emacs org-mode environment, which has been presented in the Scientific Programming department before [3]. In org-mode, so-called "source code blocks" contain executable pieces of code, much like the cells in a Jupyter notebook. However, org-mode offers much more flexibility in combining these source code blocks into a computational document. Their execution can be in sequential order, as for a Jupyter notebook, but also in arbitrary order, defined by naming the inputs and outputs of each block and thereby defining a a data-flow graph. But the main difference for this story is that each source code block can be written in a different language, which makes exchanging data between them much more challenging. Org-mode defines a common data representation that it translates to and from for each language. This common representation is a tree structure made from nested lists, with the leaves being numbers or strings. Such tree structures are well-known to be very flexible, and therefore widely used in data interchange formats, the best-known one probably being XML.

The requirement to use this tree data structure at the interface between source code blocks discourages the use of language-specific data structures such as pandas' `DataFrames`, which would require a conversion layer at the beginning and end of each snippet of code. What is encouraged instead is to keep all data in the tree structure as much as possible. This corresponds to a long tradition of data representation in the Lisp family of programming languages, which is no coincidence because most of Emacs, and in particular all of org-mode, is written in Emacs' own Lisp dialect called Emacs Lisp. This emphasis on

a single universal data structure, in stark contrast with the ideas of object-oriented programming, is often justified using a quote from computer science pioneer Alan J. Perlis [2] dating back to 1982: “It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.”

Emacs being a text editor, it has excellent support for manipulating text, and therefore it seems natural to start the flu data analysis using Emacs Lisp. A side benefit is the possibility of loading the data into an Emacs buffer, which allows direct inspection using all of Emacs’ text editing functions. The first step thus is to download the data into a read-only buffer named `*data*`:

```
(require 'url)

(defun data-url (year1 week1 year2 week2)
  (concat "http://websenti.u707.jussieu.fr/sentiweb/"
          "api/data/rest/getIncidenceFlat?indicator=3"
          (format "&wstart=%4d%02d&wend=%4d%02d"
                  year1 week1 year2 week2)
          "&geo=PAY1&$format=csv"))

(with-current-buffer (get-buffer-create "*data*")
  (unless buffer-read-only
    (url-handler-mode)
    (insert-file-contents (data-url 1990 1 1995 1))
    (setq buffer-read-only t)))
```

Making the buffer read-only provides a protection against accidental manipulation. Since the data format is actually very simple, I wrote my own special-purpose parser for this particular file rather than searching for a CSV parser library for Emacs Lisp. All it takes is (1) split the file into lines, (2) discard the first line, (3) split each of the remaining lines at the commas, yielding the columns of the table, and (4) select the first and third columns. In Emacs Lisp this can be written as

```
(require 'dash)
(with-current-buffer "*data*"
  (let* ((lines (split-string (buffer-string) "\n" t))
         (data-lines (rest lines))
         (table (--map (split-string it ",") data-lines)))
    (-insert-at 1 'hline
                (-select-columns '(0 2) table))))
```

The last operation, inserting a `hline` between the column names and the data, puts the parsed data into the format of an org-mode table, which Emacs then formats nicely before inserting it into the computational document.

For the benefit of readers not familiar with Emacs Lisp, and for a better comparison with the pandas-based code I have shown earlier, here is an implementation of the same method in Python 3. It leads to exactly the same result, but doesn’t permit inspection of the data file in an Emacs buffer.

```
from urllib.request import urlopen

handle = urlopen(data_url(1990, 1, 1991, 1))
data_as_bytes =  handle.read()
data = data_as_bytes.decode('ascii')
lines = data.strip().split('\n')
data_lines = lines[1:]
table = [line.split(',') for line in data_lines]
selected_columns = [(row[0], row[2]) for row in table]
selected_columns.insert(1, None)
return selected_columns
```

Note that the final `return` is a particularity of Python embedding in org-mode: a Python source code block is treated as the body of a function. The `None` being inserted after the column header is the Python equivalent of the `hline` in Emacs Lisp.

Not counting the shared function `data_url`, this back-to-the-basics implementation consists of 10 lines of code, compared to two lines for reading the data using pandas. However, and that is the main message of this story, it is immediately clear to anyone with a minimal knowledge of Python 3 how the code works, which assumptions it makes about the format of the data, and that all the data is always returned as text strings. There is no hidden intelligence in these 10 lines of basic Python 3 code.

Another way to compare the two Python 3 implementations is to compare the effort that a reader must invest to understand what is going on. The pandas version provides an immediate "executive summary": we are reading a CSV file. Moving on to a more detailed understanding of what we get in return, and under which conditions we can expect the code to work, is almost a research project, since even the multi-page documentation of the `read_csv` function does not provide the answers. The basic Python 3 version, on the other hand, requires more effort to reach a first level of understanding, because the only superficially available information is that the code processes a text file. However, after a two-minute inspection of the code, a reader knows everything there is to know, without reading lengthy documentation or doing experiments.

As I was working on this example for the MOOC, I stumbled upon a ten-year-old interview with Donald Knuth (http://www.informit.com/articles/article.aspx?p=1193856) that I saw quoted in the context of literate programming, one of Knuth's major inventions. Since time spent reading Donald Knuth is rarely wasted, I read through the complete interview and found the following statement:

> I also must confess to a strong bias against the fashion for reusable code. To me, "re-editable code" is much, much better than an untouchable black box or toolkit. I could go on and on about this. If you're totally convinced that reusable code is wonderful, I probably won't be able to sway you anyway, but you'll never convince me that reusable code isn't mostly a menace.

This is a nice summary of my experience preparing two versions of my data analysis example. The `read_csv` function of pandas is designed to be reusable in a wide range of situations, but that also makes it a nearly impenetrable black box. In contrast, the

much more basic approach that the org-mode conventions encouraged me to adopt is not reusable but re-editable. People who need to process a somewhat different dialect of the CSV format can read it, understand it, and adapt it to their own situation.

And the winner is...

It may seem at this point that I am fully adhering to Donald Knuth's preference for re-editable over reusable code, but it is not as simple as that. If I have explained the advantages of re-editable code in so much detail, it is mainly because I have rarely seen them exposed. It is much easier to find discussions of the virtues of reusable code. The two main reasons given are economy and reliability. The economy argument is based on development costs being shared among many users. This does, however, silently suppose that the development investment leads to savings for the users, which isn't obvious as my CSV reading example illustrates. The reliability argument says that heavily used code is better tested, an idea often referred to as "Linus's law" in the open-source developer community and summarized as "given enough eyeballs, all bugs are shallow". It sounds plausible but, like most statements about software engineering, is hard to back up by solid evidence. Heavy usage has certainly not prevented the existence of a severe security bug in OpenSSL for two years [1].

What, then, is the better approach: reusable or re-editable code? As so often, it depends. Let's look at the other end of my reproducible data analysis example: plotting a histogram of the annual flu incidence. I wouldn't even think of doing this in basic Python, or whatever other language, because of the complexity of the task. Even the people who write reusable plotting libraries don't limit themselves to basic Python or C, but build on reusable graphics libraries. So what's the difference between reading a CSV file and plotting a histogram? It's the localization of complexity.

In the CSV reading case, all the complexity is at the surface. In fact, it's the CSV format itself that is the source of complexity. Or perhaps one should better say that it is the habit of referring to a diverse family of file formats by a single name, leading to the idea that a single piece of code should deal with all of them. This complexity cannot be hidden from the user, which is why the documentation of pandas' `read_csv` function is so long. In the histogram plotting case, most of the complexity is in the depth of the implementation. The user of a plotting library does not need to know how graphics libraries work or how the PDF format is defined, and would in general prefer not to be exposed to such details.

As a rough guide to whether packaging a piece of code as a reusable library makes sense, or whether looking for reusable code is preferable to rolling your own code, compare the length of the code to the length of documentation that users would consider sufficient. Reusable code makes sense when the former clearly exceeds the latter. Another way to look at this is that well-written code is its own best documentation. Hiding the code from its users and replacing it by documentation makes sense only when this clearly reduces the users' mental load.

A corollary of this principle is that if you publish your code, make it either re-editable, i.e. self-explaining, or reusable, i.e. well documented. Scientists often complain that

writing documentation is a lot of work that is not well rewarded. Going for re-editable and keeping documentation to a minimum is thus worth considering as an alternative, keeping in mind that the audience for reusable and re-editable code is not quite the same. And that is perhaps the ultimate criterion for choosing the right approach: the one that best suits your audience.

In the context of scientific programming, there is one more criterion to keep in mind. If your code implements a scientific method, rather than a technical detail such as reading CSV files or plotting histograms, the decision to expose or to hide the code implies the decision to expose or hide the method itself. Well-known and frequently used methods are the perfect candidates for implementation in reusable and documented libraries. Recent methods, and in particular methods still under development, are better proposed as re-editable implementations. Otherwise, readers might feel cheated when presented with a computational solution to a scientific problem that consists only of a call to a black-box library - for an illustrative example, see the public review of a submission to the ReScience journal at https://github.com/ReScience/ReScience-submission/pull/46#issuecomment-366257844. The solution in that case was to include the library code into the review, which also shows that reusable and re-editable code are just the extremes of a continuum.

References

[1] Zakir Durumeric et al. “The Matter of Heartbleed”. en. In: *IMC’14*. Vancouver, BC, Canada: ACM Press, 2014, pp. 475–488. ISBN: 978-1-4503-3213-2. DOI: 10.1145/2663716.2663755.

[2] Alan J. Perlis. “Epigrams on Programming”. In: *ACM Sigplan Notices* 17.9 (Sept. 1982), pp. 7–13. ISSN: 0362-1340.

[3] Eric Schulte and Dan Davison. “Active Documents with Org-Mode”. In: *Computing in Science & Engineering* 13.3 (May 2011), pp. 66–73. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.41.

About the author

Konrad Hinsen is a researcher at the Centre de Biophysique Moléculaire in Orléans and at the Synchrotron SOLEIL in Saint Aubin. His research interests include protein structure and dynamics and scientific computing. Hinsen has a PhD in theoretical physics from RWTH Aachen University. Contact him at konrad.hinsen@cnrs.fr.