Platforms for reliable research

Ben Klemens

23 April 2017

This stems from a Twitter thread about how you can't do replicable science if some package won't be installable five years from now, which is not an unusual case for Python packages. Click on the lower date stamp for more (including another view of the communities I describe here):

Meanwhile, it was observed, C code from the 1980s routinely runs just fine. For example, I work with some large-scale examples that I could expound about. Or, when you type `grep` or `ls`, you are calling up a C program whose core was likely written in the 1980s or early 1990s. Unless it depends on specific hardware features, I think it's hard to find correctly written C code from the 1990s that *doesn't* compile and run today.

I've been using R since 2000, and a lot of my code from even five years ago won't run. We had a project which repeatedly hit problems where package A needed version 2.3.1 of the Matrix package, but not version 2.3.2 or later, but package B relies on version 2.4 of the same package. In the end, it killed our project: every time we had to re-tweak the install, our potential users perceived the project as a little less reliable.

So I honestly believe it's true that C code is more reliable and more likely to run a decade from now, and there's a wash of evidence to demonstrate this. Then, why?

I think there are two reasons: there are cultural differences in a language that has a standard, and the lack of a standard package manager(s) has been an inadvertent blessing here. These features have led to a social system that allows us to build technical systems that don't depend on people.

There's a standard An interpreted language doesn't need a standard because if the interpreter got the right answer, then you've conformed 100%. In the long term, this is terrible. Raymond Chen, a kernel-ready programmer at Microsoft, has a blog that is filled with convoluted stories like this one[1], whose template is that authors of Windows programs found an undocumented/underdocumented part in the workings of Windows, then years later the Windows OS people try to change that part, while fully conforming to documented behavior, then get a million complaints as the undocumented trick breaks. That link is the first example I grabbed, but keep skimming; he's been blogging these stories for a decade or so.

This state is hard to picture in the world of plain C, because having a standard affects culture. I can't count how many people have written to tell me about some piece

[1] `https://blogs.msdn.microsoft.com/oldnewthing/20160822-00/?p=94135`

of code I put out in the public sphere that might compile on every modern compiler, but does not—or even *may* not—conform to the ISO standard. Not writing to the contract is a bug, whether it works for me or not.

Meanwhile, in the world of R or Python, the documentation could be seen as the contract. But do you really think every single R function documents its behavior when it receives a `NaN` or `-Infinity` argument? And with no culture of strict contractual obligation, there is little that prevents the authors from just rewriting the contract. The same holds in greater force with third-party packages, which are sometimes developed by people who have no experience writing contracts-in-documentation so that they are reasonably future-proof.

I liked this Twitter exchange. Each tweet is another variant of the statement that the R community has no coherent contract or contract conformance culture. [Again, click the date stamp for the two replies.]

So, contracts exist in C-world and elsewhere, but from everything I have seen, the cultural norms to take those contracts seriously are far stronger among C authors.

Towers Azer Koçulu has a Javascript package named Left Pad which provides a function to pad a string or number with white space or zeros. That's the whole package: one function, to do something useful that I wouldn't want to get side-tracked into rewriting and testing, but which is not far from a Javascript 101 exercise.

It was a heavily-used micro-package, and not just by fans of left padding, as a data analysis package might use a table-making package, which would depend on Left Pad. When Mr Koçulu unpublished all his Node Package Manager submissions[2], this broke everything.

In practice, R packages tend to have several such microdependencies, while the authors of C packages tend to cut something like a left pad function from the library code base and paste it in to the code base of the project at hand. Authors of the GSL made an effort to write its functions so they could be cut and pasted. SQLite has a version in a single file, whose purpose is to allow authors to copy the file into their code base rather than call an external SQLite library.

I think it is the presence or lack of a standard package manager that led to this difference in culture and behavior. If you have the power to assume users can download any function, so seamlessly it may even happen without their knowledge, why wouldn't you use it all the time?

The logical conclusion of having a single, unified package manager is a tower of left-pad-like dependencies, where A depends on B and C, which depends on D and E, which depends on B as well.

The tower is a more fragile structure than just having four independent package dependencies. What happens when the author of B changes the interface a little, because contracts can change? The author of package A may be able to update to B version 2, but E depends on B version 1. Can you find the author of E, and convince him or her to update to B version 2, and to get it re-posted on CRAN so you can use it in a timely manner?

[2] `http://cryto.net/~joepie91/blog/2016/03/23/reflections-on-npm-gate-one-day-later/`

To get the package on to CRAN, the author of E will have to pass the tests he or she wrote on every machine R runs on. Douglas Bates threw up his hands[3] over the problems this engenders:

> I am the maintainer of the RcppEigen package which apparently also makes me the maintainer of an Eigen port to Solaris. When compilers on Solaris report errors in code from Eigen I am supposed to fix them. This is difficult in that I don't have access to any computers running Solaris .... So I have reached the point of saying "goodbye" to R, Rcpp and lme4 ...

It's good that CRAN wants to provide a guarantee that a package will compile on a given set of platforms. First, it does this because there is no culture of contracts, so the only way to know if an R/C++ package will compile on Solaris is to try it. Second, the guarantee is somewhat limited. Did the package get the right answers on all systems? If it had numeric tests, those tests passed, but the CRAN testing server knows nothing of how to test eigenvector calculations. I've bristled at more than enough people who have told me they trust the numbers a package spits out because it's on CRAN so it must have been tested or peer-reviewed.

In the short run, this is great. We've solved a lot of dependency-mismatch problems.

In the long run, we haven't solved anything. You come back in three years to re-run your analysis, and you find that package B was removed, because the author was unable to reply to requests to fix new issues, because she died. Or the latest version of package B is impossibly different, so you try the old one, and you find that none of the guarantees hold: to the best of my knowledge, CRAN doesn't back-test old versions of a package (what would they do if one broke?), so whether your old version of package B works with the 2020 edition of R and its ecosystem is as much a crap shoot as it ever was.

That was a lot of kvetching about package manager issues, but with the intent of comparing to C-world.

With no official package system, your C library has no means of automatically downloading dependencies, so forget about building a tower where you instruct readers to chase down a sequence of elements. Users will tolerate about two steps, and most won't even go that far. For code reuse, this has not been good, and we see people re-inventing well-established parts over and over again. But for reliability ten years down the road, we get a lemons-out-of-lemonade benefit that building unreliable towers of dependencies is impracticable.

[GUI-based programs sometimes depend on a tower of dependencies, and a package manager like Red Hat's RPM or Debian's APT to install them. Such programs also typically have command-line versions just in case.]

With no package manager, you have to give users the entire source code and let them rebuild. The build tools are standard as well. The norm is the makefile, which is defined in the POSIX standard. If you're using Autoconf, it will run a POSIX-compliant shell script to produce such a standard makefile.

There is nobody to check whether your library/package will build on other peoples' machines, which means that you have to stick to the contract. On the one hand, people

[3] `http://marc.info/?l=r-sig-mixed-models&m=138661192407677&w=2`

screw this up, especially when they have only one compiler on hand to test against. On the other, if they get it right, then you are guaranteed that today's version of the code will compile in a decade.

I'm no Luddite: it's great that package managers exist, and it's great that I can write one line into the metadata to guarantee that all users on an Internet-enable computer [e.g., who don't work on a secured system] with an appropriate configuration can pull the dependencies in a second. It's great that there are systems monitoring basic package sanity.

But for the purposes of long-term research replicability, all of these boons are patches over the lack of a system of contracts or a culture that cares deeply about those contracts. Think back to how often I've mentioned people and social systems in my discussion of interpreters and package managers. Conversely, once you've zipped up code written to contract and a makefile-generator written to contract, you're irrelevant, your wishes are irrelevant (assuming the right licenses), and there is no platform that can deliberately or due to failure prevent a user from installing your package. People and their emotions and evolving preferences are out of the picture. In the end, that's really the secret to making sure that a platform can be used to generate code that is replicable years and decades from now.