**Ten simple rules for biologists using R**

Rule 1: Begin with the end in mind

Focus on your goal. What are you trying to achieve? Are you trying to just conduct one analysis? Do you want to use R to save time (e.g., combine many steps together or repeat an analysis many times)? Or do you want to become a programmer and do you want to design bioinformatic tools? You should be developing your scripts and pipelines accordingly.

Rule 2: Baby steps are steps

Once you’ve begun, focus on one task at a time and apply your critical thinking and problem solving skills. This requires breaking a problem down into steps. Analyzing complex data may sound challenging, but the individual steps do not: e.g., read in your data, decide how to interpret missing values, scale as needed, identify comparison conditions, calculate significance, correct for multiple testing, plot your results. Break a large problem into modular tasks and implement one task at a time. Iteratively edit for efficiency, flow, and succinctness. Mistakes will happen. That’s ok; what matters is that you find them, correct them, and learn from them.

Rule 3: Immersion is the best learning tool

Don’t stitch together an analysis by switching between or among languages and/or point and click environments (Excel, ArcGIS, etc.). While learning, if a job can be done in one language or environment, do it all there. This will ultimately let you conduct the entire analysis by one simple click or copy-paste.

Rule 4: Ask for help

There are numerous online resources: tutorials, documentation, and sites intended for community Q and A (StackOverflow, StackExchange, Biostars, etc.). You can also find a community of programmers, ranging from beginning to experienced users, to ask for help. Once you find a community, ask for help. At the beginning stages, help from colleagues is invaluable. Additionally, ask a friend for code. You wouldn’t write a paper without first reading a lot of papers or begin a new project without shadowing a few experimenters. First, read their code. Implement and interpret, trying to understand each line. If you get stuck, ask them questions.

Rule 5: Learn how to ask questions

There’s an answer to almost anything online, but you have to know what to ask to get help. In order to know what to ask, you have to understand the problem. Start by interpreting an error message. Watch for generic errors and learn from them. Identify which component of your error message indicates what the issue is and which component indicates where the issue is. Understanding the problem is essential; this process is called “debugging.” Without truly understanding the problem, any “solution” will ultimately propagate and escalate the mistake, making harder-to-interpret errors down the road. Once you understand the problem, look for answers. Looking for answers requires effective googling. Learn the vocabulary (and meta-vocabulary) of the language and its users. Once you understand the problem and have identified that there is no obvious (and publicly available) solution, ask for answers in programming communities. When asking, paraphrase the fundamental problem. Include error messages and enough information to reproduce the problem (include packages, versions, data or sample data, code, etc.). Present a brief summary of what was done, what was intended, how you interpret the problem, what troubleshooting steps were already taken, and whether you have searched other posts for the answer. See the following website for suggestions: <http://codereview.stackexchange.com/help/how-to-ask> . End with a “thank you” and wait for the help to arrive.

Rule 6: Don’t reinvent the wheel

Use all resources available to you, including online tutorials, examples in the language’s documentation, published code, cool snippets of code your lab-mate shared, and, yes, your own work. Read widely to identify these resources. Copy-and-paste is your friend. Provide credit if appropriate (i.e., comment “adapted from so-n-so’s X script”) or necessary (e.g., read through details on software licenses). Document your scripts by commenting in notes to yourself so that you can use old code as a template for future work. These comments will help you remember what each line of code intends to do, accelerating your ability to find mistakes.

Rule 7: Develop good habits early on

Computational research is research, so use your best practices. This includes maintaining a computational lab notebook and documenting your code. A computational lab notebook is by definition a lab notebook: your lab notebook includes protocols, so your computational lab notebook should include protocols, too. Computational protocols are scripts, and these should include the code itself and how to access everything needed to implement the code. Include input (raw data) and output (results), too. Figures and interpretation can be included if that’s how you organize your lab notebook. Develop computational “place habits” (file-saving strategies). It is easier to organize one drawer than it is to organize a whole lab, so start as soon as you begin to learn to program. If you can find that experiment you did on June 12, 2011—its protocol and results—in under five minutes, you should be able to find that figure you generated for lab meeting three weeks ago, complete with code and data, in under five minutes as well. This requires good version control or documentation of your work. Like with protocols, each time you run a script, you should note any modifications that are made. Document all changes in experimental and computational protocols. These habits will make you more efficient by enhancing your work’s reproducibility.

Rule 8: Practice makes perfect

Use toy datasets to practice a problem or analysis. Biological data get big, fast. It’s hard to find the computational needle-in-a-haystack, so set yourself up to succeed by practicing in controlled environments with simpler examples. Generate small toy datasets that use the same structure as your data. Make the toy data simple enough to predict how the numbers, text, etc., should react in your analysis. Test to ensure they do react as expected. This will help you understand what is being done in each step and troubleshoot errors, preparing you to scale up to large, unpredictable datasets. Use these datasets to test your approach, your implementation, and your interpretation. Toy datasets are your negative control, allowing you to differentiate between negative results and simulation failure.

Rule 9: Teach yourself

How would you teach you if you were another person? You would teach with a little more patience and a bit more empathy than you are practicing now. You are not alone in your occasional frustration. Learning takes time, so plan accordingly. Introductory courses are helpful to learn the basics because the basics are easy to neglect in self-study. Articulate clear expectations for yourself and benchmarks for success. Apply some of the structure (deadlines, assignments, etc.) you would provide a student to help motivate and evaluate your progress. If something isn’t working, adjust; not everyone learns best by any one approach. Explore tutorials, online classes, workshops, books, local programming meetups, etc., to find your preferred approach.

Rule 10: Just do it

Just start coding. You can’t edit a blank page. Learning to program can be intimidating. The power and freedom provided in conducting your own computational analyses bring many decisions points, and each decision brings more room for mistakes. Furthermore, evaluating your work is less black-and-white than for some experiments. However, coding has the benefit that failure is risk free. No resources are wasted—not money, time (a student’s job is to learn!), or a scientific reputation. In silico, the playing field is leveled by hard work and conscientiousness. So, while programming can be intimidating, the most intimidating step is starting.