How to learn a new programming language

Ben Klemens

20 March 2012

This is a sequence of a dozen or so steps for getting to know a new programming language.

It came to mind a few weeks ago, during an informal job-type interview. There's always that point in the interview where the interviewer asks *So what programming languages do you know?* I know she was expecting something like 'Yeah, I took some SAS classes'; I told her I had reasonable facility and had done at least some nontrivial work in C, C++, Java, S-PLUS and R, Scheme, Matlab and Octave, Perl, Python, Ruby, FORTRAN 77, and I forget a few.

I find some people like my interviewer, who get by on what they learned in school; and others who find my list of a dozen languages to be average, who see that it's the same thing over and over but with the parens in different places. The intent of this paper is to point out the commonalities, and to help you make the jump to being a computing badass who sees across languages.

This paper is a checklist, a series of exercises, and a series of suggestions of where to look when you first find out that you're going to have to do work in some platform that you never thought about until somebody told you it's going to be your best friend for the duration of the next project. You'll need to work with the documentation and tutorials for your target language to answer all the questions I ask here—and you'll really have to delve. Introductory tutorials tend to be advertisements for the easiest features of a language, but you can't seriously work until you've gotten to know the ugly and the missing parts as well.

In writing this, I contend that this checklist applies to any mainstream programming language (thus excluding specialized languages like TEX, sed, gnuplot, or SAS). When you discover inevitable exceptions, that's great, because revealing those exceptions will reveal the character of the language and what makes it different from (and hopefully better than) all the others. So the questions are easy and lowest-common-demoninator, but once you find the right part of the manual that describes how to do the easy things, you're in the right place to see if your language does more.

OK, on to the checklist. I'll start with the basic platform, then move on to data types, functions, and larger blobs of code.

How do I print "Hello world" to the screen? This was originally an exercise from Kernighan and Ritchie's 1978 C book, and the question it is really asking is the same

in all languages: how do I set up the environment in which I have to work?

It is the question I can offer the least guidance on, because there are just so many ways in which these things happen: there are compilers, interpreters, integrated development environments (IDEs), a few languages that primarily run in your browser, and who knows what else.

ℚ. Get *Hello world* to print on your screen.

How do I insert comments? As essential as this is, some systems don't have an explicit comment mechanism, but just expect you to write a string or some other expression that evaluates to something innocuous.

Python.`"""This part prints Hello World:"""`

Where's the documentation? Your language may have a clever means of built-in documentation, which you will naturally want to get to know immediately. There may be manual pages, like the `man perl` set of pages or the POSIX-standard man pages for C libraries (try `man 3 printf` or `man operator`).

But the answer to this question also lies with your Web browser. The built-in and official documentation is probably a reference work, and learning from reference works is a bad idea: you also want didactic works that point out what's important in the reference, and you'll find those by checking in with your favorite search engine.

You will almost certainly be using external libraries or packages of some sort, so take some time to find the reference and/or tutorial documentation for those. They may or may not be related to the official documentation.

There's no explicit exercise for this one, because one of the goals of this tutorial is to get you to find and poke around the documentation, but if you want one, how about: ℚ. Find the documentation for reading from/writing to a text file. The instructions may not be clear at this point, but you know where they are.

What are numbers like? Your language has a data type that represents plain text (i.e. *strings*) and a type or two that represents numbers. We'll get to strings later, but there isn't much variation in number systems. The only serious point of variation is whether an operation on two integers always produces an integer—that is, does 8/3 turn out to be 2.66666 or just the truncated value of 2?

ℚ. Check whether your language keeps a wall between integers and real numbers. What is `8/3`? What about `8/3.` or `8/(3+0.0)`?

Python 2. 8/3 $\rightarrow$ 2
Python 3. 8/3 $\rightarrow$ 2.6666

What are the lists like? I say *list*, but this could also be called an *array* or a *vector*. The big difference is that some of these types tend to have fixed length and some

variable, but in all cases, they are an ordered collection of homogeneous items. Some languages don't even hold that requirement that all elements of the list/array/vector have the same type, but we'll stick to that for now.

ℚ. Create a list of five numbers, 1, 2, 3, 4, 5 (don't bother with a `for` loop or such cleverness yet; just type it out). Print them to the screen.

ℚ. Double all of the elements of your list; print.

Were you able to double the elements in place, or did you have to make a copy (perhaps a copy with the same name)? Did you have to double each element individually, or were you able to write something like `2*my_list`?

How do I declare a new variable? Your language may make some smug claims that it doesn't need you to declare variables, but you sometimes do need a means of indicating the type of an item. Here is a snippet of code for you to implement, which starts with an empty list, and then grows it by appending one item at a time:

```
my_list = [ ]  #an empty list
for i=1 to 10:
   my_list = [my_list, i]
```

That first line is a declaration, even if you don't want to call it that.

The problem here is that all languages have some set of rules to automtically convert one type to another. The more types you have, and the more *do what I mean* the language tries to be, the more conversion rules you need (and every language has at least a few). On the first step in the loop, when you append 1 to an empty list, you need to know if you can refer to a not-yet-existent list, whether that empty list will somehow get typecast away, and whether your one-element list `[1]` remains that way or gets cast into an integer.

ℚ. Let a variable `x` be an arbitrary integer. Write code like the sample above to incrementally build the list, [1, 2, 3, ..., x].

ℚ. Casting: assign the number ten to a text string (`"10"`), perhaps via a print-to-string function or a cast-to-string-type; convert the text string `"10"` to an integer variable.

C. char *str; asprintf(&str, "%s", 10); int n=atoi(str);

How are references handled? Every operation that takes in a value and returns a value can either modify its input where it is, or make a copy of the input and mangle the copy to produce output. It is imperative that you know at every step of your program which is happening.

Some languages lack pointers/references/aliases, and so copy every single time. If your langauge does that, bear in mind that it will be slow for operations on large data sets, and when we get to function calls, remember that modifying the variable you passed in to a function is really just modifying a copy of the passed-in variable. You can

sometimes use this to your advantage to write shorter functions that don't have to take care to prevent side-effects.

On the other end, some languages mostly lack copying, in the sense that they work with aliases/references/pointers by default. After you assign `x = y`, if you double `x`, then `y` will double along with it. But there is going to be some way to specify `x = copy(y)`, which you should take note of for occasional cases when you'll need it.

ℚ. [Impossible in some languages] You have a list `[1 2 3 ...9 10]`; give the list an alias as per the `x = y` example above. Verify that the alias works by changing the second element in the aliased list to 100, then print the original list.

ℚ. Copy your list to a new variable with a new name. Change the second element in your new variable to 100; verify that the original list didn't change.

How do I handle strings of text? If you stop to think about how they are handled in memory, you quicky realize that strings are *hard*. A number has a fixed memory footprint: you don't need twice as much memory to write 20 or 4 as you need to write 2. But `Hello` needs five slots in memory while `Hi` only needs two. If you wrote `HHello`, then fixing your typo means moving every item in the string over by one in the little array that is the sequence of letters.

C is famously terrible about hiding these details from you, to the point that you might want to check GLib for its smarter string library. Many languages (C family included) treat them like arrays, so you can do array-like operations. Some languages make heavy use of regular expressions for string manipulation, and if you're already good with regexes, then great—you can reduce your problem count with them.

ℚ. Fix the typo: put `HHello` in a string and replace it with the string with the extra H lopped off. Were you able to do it in place, or did you have to copy to a new variable?

Perl. $x = "Hhello"; $x =~ s/Hh/h/;

What are the structs like? Arrays are for homogeneous items; structures, dictionaries, or hashes can be used for heterogeneous collections, where each item is a named element of the whole.

If your preferred language uses formally declared structures, you may one day find yourself in a language that uses a dictionary or hash—an array with names instead of numeric indices—to serve as your structure. Conversely, if you're used to dictionaries or hashes, bear in mind that some languages require small collections of heterogeneous items be declared in structures. For a long list of homogeneous items that happen to have the same type, you may have to just use a numeric index, generate a hash-type device using an array of key/value structs, or find a key/value system in the libraries.[1]

ℚ. Write a structure, which we'll call the `rational` structure, with three parts representing a fraction: an integer numerator, an integer denominator, and a text name (like

[1] Dictionary, hash, or struct—they're all about as good, but there are some older languages that have absolutely no way to bundle a set of heterogeneous variables. They are obsolete, and should be used only when the situation really gives you no other choice.

`"5/6"`). For now, just declare the type if necessary, fill an instance of such a structure for 5/6, and print the elements.

How do I write and call a function? Are function arguments copied in or pointed to? The form of a function call doesn't change much from language to language. You define the language in one place, and call it with a form like, `new_fn(x)` (or (for LISP-inspired languages) a form like `(new_fn x)`).

$\mathbb{Q}$. Reusing the code you wrote above, write a function that takes in an integer `x`, and returns a list [1, 2, ..., x].

Some languages allow you to write inline functions—nameless little routines for throw-away transformations of a list $[a, b, c]$ into $[f(a), f(b), f(c)]$. This may be in the index under *list comprehension*, *lambda functions*, or *anonymous functions*. Those languages that allow you to do this kind of thing tend to rely heavily on the facility, so check that it's possible, and if it is, redo the above example about doubling your list using that feature.

Scheme.
```
(define L (list 1 2 3 4 5))
(map (lambda (x) (* 2 x)) L)
```

How do I debug a function? Here are the sort of things I mean by debugging:

- pausing your program at a certain point,
- getting the current value of any variables that exist in that function;
- jumping to a parent function and checking variable values there;
- stepping past the point where you paused, one line of code at a time.

There's diversity here: for C, you might use the GNU debugger or your IDE might have a built-in hook, some interpreted languages have the facility built in to the interpreter, some have a debugging library that you import like any other library, javascript has some browser plugins, bash has a verbose mode which doesn't do all of the above tasks but is at least a start.

$\mathbb{Q}$. Write a program/script that calls your function to produce a list 100 elements long. Set a breakpoint that stops when your list is 34 items long and print the list as it looks at that moment.

The worst case is inserting print statements where you need to know a variable's value. It takes a few seconds to set up each print statement, and is annoying when you run and then find out that you needed one more variable's value or the same variable's value three lines down. If the debugging facilities I enumerated above really are not available for for your language of choice, bear in mind that large projects may be a pain relative to how they'd work in other languages.

What are the scoping rules? At the least, you need to know how to declare variables that are global to the entire program, and variables that are local to a function. Some languages go crazy from there; I count four different scoping systems that you'd have to bear in mind for C++ (file, curly brace delimited, object, namespace).

ℚ. Rewrite your function to have an explicit iterator (i.e., use a `for i=1 to N` sort of loop). After calling the function, try to print the value of your iterator (`i`), and verify that it is not defined outside the function's context.

ℚ. Make the iterator global, so that its value is `N` after the function is called. [Then revert to the prior version, because having an iterator as a global variable is absurdly bad form.]

In lexical scoping, variables that are not explicitly defined in a function, and so would normally be looked up in the global environment, are looked up in the environment as it looked when the function was first called. If you're in a lexically-scoped language, you can do some tricks to generate on-the-fly specialized functions, but well before you get creative with those methods you'll need to make sure you don't get confused and presume that you are looking at a global variable as it is now when you are actually looking at the version that was bound to the function on first call.

How do I maintain continuity across function calls? To fix the idea, I'll start with the exercise:

ℚ. Write a function so that on the first call, `count()` returns one, on the next call, `count()` returns two, and so on to infinity.

The cheap way of making this work is to use a global variable. C goes one step further with the `static` keyword. Some have an explicit syntax for continuations.

C. int count(){static int i=0; return i++;}

ℚ. Write a function that takes in a list or `NULL`/`nil`/`0`/whatever. If it gets a list, it returns the first item in the list and stores the list internally; if it gets a blank marker, then it returns the next item in the stored list; use the mod (or %) operator to cycle back to the beginning of the list if you hit the end.

ℚ. If your language has lexical scoping, write a function that takes in a list and returns a function. The returned function is as above: on each call, it will return the next item in the list, cycling back to the beginning as needed. Here's a sample use of the function you're going to write:

```
next_prime = generate_list_step_function([1 2 3 5 7 11 13])
non_prime = generate_list_step_function([4 6 8 9 10 12 14])
next_prime()
next_prime()
next_prime()
non_prime()
```

which will print `1 2 3 4`. In some languages without lexical scoping, implementing this requires some creativity; in some it is impossible (in which case you'll probably

wind up sending in the list every time).

Can I do text substitutions (i.e., macros)? Your typical language mostly focuses on functions that generate their own space in which to work (as per the scope section); the text substitution simply replaces a blob of text with another blob of text.

There are some languages that have no macro processing abilities, some that have a preprocessor that takes the text that is your program file and convert pieces of text into other pieces of text, some that are sufficiently self-aware to convert a text string to operational code in real time via an `eval`-type function, and some that use lazy evaluation to leave your text as text until you want it to be evaluated.

ℚ. Write a macro you'd call like this: `call_function(your_function, 10)` or like this: `call_function("your_function", 10)`. The macro would then print `Calling the your_function function` to the screen and then execute `your_function(10)`. Use this to call the list-generating function you wrote above.

OK, to this point, everything has been about the details of the language: how do I deal with types? Can I do clever tricks with scope and persistent variables? The rest of this is going to be about design: how can I introduce new nouns, and the verbs that those nouns can do? How do I package them so I can easily use them for the next project, and how do I use already packaged elements for today's project?

How do I load libraries/packages so I don't have to reinvent wheels? The most basic sort of library inclusion is to simply have a mechanism for including a text file with some code verbatim at the head of the text file you're working with now.

Let me give you a few examples, because the point of the package concept is that not everybody has similar needs.

ℚ. A: Make a hundred draws from a standard Normal distribution.

ℚ. B: Load an XML document into memory, and print all of one type of element.

ℚ. C: Open an empty SQLite database and create a table.

How do I set up functions that act on a specific structure? So very much of the code in the world involves a single specialized data structure, and a set of functions that manipulate that struture. This, of course, is the basis of object-oriented coding, wherein those functions to manipulate a structure are a part of the structure itself. But before that happened, there were simple naming conventions, which worked just fine but weren't as attractive. Some languages have no discernable naming convention, in which case there's a null answer to this question: just put the functions that act on a structure wherever.

ℚ. Write a `rational_set` function that takes in two integers and outputs a `rational` structure, a `get_value` function that returns the value of the fraction, a `get_name` function that returns the fraction as a text string, and an `add` or `+` function that adds two rationals. If appropriate, also define a `free` or `destroy` function. Add those functions to the object itself if appropriate.

How do I get an auxiliary structure that builds upon a base structure? OK, so you wrote a structure above that has a fixed list of elements, but would now like to extend the structure in some way. As above, some systems will require that you declare the structure beforehand, so you will need to extend either via a structure-extending mechanism (new struct is a version of old struct), or a new structure that consists of some elements and an embedded old structure (new struct has a version of old struct). The wrapping-around approach works everywhere; the object-inheritance method of directly extending a structure is pretty common, and when it exists people will expect you to make some use of it.[2] Even in languages where extending a structure just means tacking another element onto the list, there are reasons for an object-extension grammar.

$\mathbb{Q}$. Use accepted custom to create a new structure, a signed rational, where all of the elements are positive, but there is another element `sign`, which is 1 if the fraction is positive, -1 if it is negative. Write new `set/get/add` functions that makes use of the new structure. If you are comfortable with irrational numbers, make this exercise more interesting by extending your rationals to complex rationals.

How do I package my own stuff? Since you already loaded a package/library/file above, you have some idea of how packages work in your new language. There may or may not be extra steps to packaging your own.

$\mathbb{Q}$. Package the rational structures' declarations (if any) and the get/set/add functions in a separate file (or files if you need a header or manifest or what-have-you). To make all this work, you wrote tests that made sure that the functions worked as planned; make sure those tests still work when you import/include the structure and functions as a package or library.

[2]This is a claim about custom across communities, and therefore can only be roughly true. Some languages have a bolted-on object model that never received wide acceptance—some even have two bolted-on object models.