Scripting and Programming Languages

Jeremy Sanders

October 2011

1 Introduction

Here is a brief guide to available scripting and programming languages. It is not complete (I won't cover IRAF scripting here), and of course I'm sure to have missed out favourite languages.

Scripting languages are programming languages designed to make programming tasks easier. If you write a Perl or Python (or even tcl or Ruby) script, then you won't have to compile the script to run the program.

I'll run down the differences between Perl, Python, a shell script, IDL, C, Fortran 77, Fortran 90/95, C++ and Java. Also I'll include some examples of some of the languages in operation.

1.1 Shell scripts

Shell scripts are useful for programming repetitive Unix tasks and simple manipulation of files. I would not use them for any complex task.

Advantages: Fairly popular, quick to write for simple repetitive tasks.

Disadvantages: Lack required facilities for general programming (e.g. floating point arithmetic). Syntax can be difficult to get used to (e.g. quoting, tests). tcsh shell scripting is broken (see shell scripting notes).

1.2 Python

Python is popular language. Many people agree that it has a simple clear syntax. Its most unusual feature is that it uses alignment of text to mark blocks of code (this is not too difficult to get used to). It is also the language used to drive PyRAF (an easier interface to IRAF).

Advantages: Popular, used in PyRAF, simple syntax, fast development, numerical/scientific library, a large library of code, good object-oriented features, good for writing larger programs.

Disadvantages: Slower than Fortran/C/C++, especially for loops. Lack of static variables may lead to bugs if you do not test programs.

See: `http://www.python.org/`

Note that Python 3 is different to Python 2. The Python 3 language has improvements, but not all libraries work under Python 3 yet.

Books: Learning Python (Mark Lutz, David Ascher), Programming Python (Mark Lutz et al.), Python Cookbook (Alex Martelli (Editor), David Ascher (Editor))

1.3 Perl

Perl is another popular scripting language, used from system administration scripts, to scripts to produce web pages (and astronomical scripts). It has fallen out of popularity a bit because of the slow development of Perl 6. Most people still use Perl 5.

Its philosophy is to provide lots of quick ways to do tasks. It was originally created for quick and easy text processing tasks, but is now being used for a wide range of programs.

Advantages: Popular, fast at certain tasks (like text processing), large library of code to use (CPAN library), fast development, great for processing text files.

Disadvantages: The syntax looks quite messy (in my opinion). If written without care your code will be hard to read in 6 months time. Not fast enough for complex numerical code. Hard to write and maintain large Perl programs.

See: `http://www.perl.com/`

Books: Programming Perl (Larry Wall et al.), Learning Perl (Randal L. Schwartz, Tom Phoenix)

1.4 IDL

Interactive Data Language (IDL) is a proprietary language for manipulating data and plotting. Its plotting is powerful and it can do quite advanced numerical work.

Advantages: Popular in astronomy, powerful facilities (especially astronomy library for FITS files)

Disadvantages: Quite expensive (to buy for yourself), nasty syntax, slow to run for many cases, inherits some bad design descisions (IMHO).

1.5 C

C is a popular standard programming language, used especially in Unix. It is quite a low level language but gives you tremendous control. The main difficulty in learning it is understanding *pointers*. It has many features not found in Fortran 77 (e.g. memory management). The language is relatively simple.

Advantages: Popular (outside astronomy, not as popular as F77 inside astronomy), lots of code available in C (e.g. GSL), standard Unix language, powerful.

Disadvantages: Can be complex to learn and debug pointers (used throughout C), lacks features available in C++, harder to parallelise than Fortran.

Books: C Programming (A modern approach) (K.N. King) [recommended introduction], The C Programming Language (Kerningham and Ritchie) [rather dry].

1.6 Fortran 77

Fortran is a programming language designed for scientists. It is intended for writing numerical code. Fortran 77 is very popular among scientists, but lacks features making certain tasks hard (e.g allocating variable-sized arrays, text handling).

Advantages: Commonly used in science, good for numerics.

Disadvantages: Not common outside science (e.g. for jobs), lacks features, slower development than Python/Perl.

Books: Structured Fortran 77 for Engineers and Scientists (D. Etter), Programming in standard Fortran 77 (Balfour, Balfour, Alexander).

1.7 Fortran 90/95

Fortran 90 and 95 are improvements to the Fortran 77 language, providing modern features (like allocatable arrays). F90/95 is less commonly used than F77, but is far more powerful. F90/95 is easier to learn than C if you have used F77 and provide similar functionality (though I expect scientists would find F90/95 closer to the scientific mindset).

Advantages: More powerful than F77, increasingly common, easy parallel programming. There are some good free compilers now, including gfortran (included in gcc 4.x) and g95.

Disadvantages: Many older supervisors will only have used Fortran 77.

Books: Fortran 90/95 Explained (Metcalf and Reid), Fortran 90/95 for Scientists and Engineers (Chapman) [expensive!].

1.8 C++

C++ was based on the C programming language, providing *object-oriented* features. It is widely used outside science (and increasingly in science, but not very much at the moment). It provides almost everything C does, but adds powerful features like classes, templates, strings and references. It is a very large language (few people understand everything), but you can get things done with much less knowledge.

Advantages: Powerful language, very common outside astronomy (lots of jobs!), can link C code easily to C++ code.

Disadvantages: Is a very complex language in full, probably less popular than Fortran in astronomy.

Books: Accelerated C++ (Practical programming by example) (Koenig and Moo) [very simple introduction], The C++ Programming Language (Bjarne Stroustrup) [the bible, complex].

Online course: `http://www.liv.ac.uk/HPC/F90page.html`

1.9 Java

Java is a modern popular programming language. It is object-oriented, but is not as complex to learn as C++ (but lacks some of its features). Java programs are cross-platform, as they run under a *virtual machine*. Java might be a bit slower than C/C++/Fortran, but is not as slow as Python/Perl.

Advantages: Cross platform, modern, widely used outside astronomy (good for jobs).

Disadvantages: Not so popular in astronomy.

Books: Java in a Nutshell (Flanagan), Learn to Program with Java (Smiley).

2 Numerical programming

You are likely to want to program some numerical code at some point in your PhD. This code might be to solve an equation, minimise a function or integrate it (if you can't solve the problem analytically or generally). The options for writing numerical code are:

1. Use a higher-level language like NumPy (Python with numerical library), IDL or Mathematica to do it. A good idea for many programs.

2. Write the numerical code yourself from scratch. This is often difficult and fraught with difficulties unless you are an expert (I advise you to read "What Every Scientist Should Know About Floating-Point Arithmetic" in my links section).

3. Read Numerical Recipes (Press et al.) and write your own code. Numerical Recipes often has good descriptions for algorithms which you can implement freely (if you don't copy their code).

4. Type code in from Numerical Recipes. This is fine except some of the NR code contains bugs in the printed versions (care!), and you are *not* allowed to give the code to someone else under the licencing terms (e.g. allow your code to be downloaded). NR code is provided in Fortran, C and C++ (though Fortran was the original language it was written in).

5. Use the GSL library (GNU Scientific Library). GSL is a 'free' library and is freely distributable. If you write a program in GSL and give it to people, you need to distribute the source code of your program too (GPL Licence). GSL is a C library, but you can call it from Fortran with a C wrapper function.

6. Use the NAG library (Numerical Algorithms Group). NAG is not free, but the IoA has a licence. The IoA has a set of manuals for NAG. You can't give your code to someone unless they own a copy of NAG. Unfortunately all the function names are difficult to remember.

3 Steps in writing a program

If you are using a compiled programming language (unlike Perl or Python), the steps to get a running program are:

1. Write the source code in a text file (e.g. prog.f, prog.c).

2. Compile the source code with an appropriate compiler, e.g.

   ```
   cc -g test.c -o program.out
   ```

   The C compilers are `cc` (Sun) or `gcc` (GNU). The Fortran compilers `f90` and `f77` (both Sun), or g77/g95 (GNU). The C++ compiler is c++ or g++ (both GNU). If compiling doesn't succeed (check how to switch on extra warnings in your compiler if possible) go back and fix the problem. Otherwise continue.

   The compiler will take options, including optimisation (usually -O), making the program work faster at the expense of compiling speed and ease of debugging. Other options include debugging options, like -g which puts extra debugging information in the program so you can look at a *core file* if it crashes. See the man page of your compiler to find out more (or info for gcc/g77)

3. Run the program.

4. If the program crashes, insert print statements every so often. This will enable you to see where it crashes. If you get a core file, you can use this to find the line it crashed on using a debugger (`gdb` or `dbx`). Use the commands:

   ```
   dbx program.out core    (Sun) /or/
   gdb program.out core    (GNU)
   ```

   to start a debugger (read man pages to understand them, but the command `backtrace` in gdb is the most useful to print out where the program crashed, or `where` in dbx).

5. Look at its results, check them if possible against what you would expect for simple cases.

6. Go back to the text editor and the source code if there is a bug or the program needs improvements.

4 Different compilers

There are several different compilers available for the compiled languages C/C++/Fortran. They differ in their ability to optimise your code, do debugging and give good warnings and diagnostics about your code.

The compilers include

1. Gcc (gcc, g++, g95) - standard GNU compiler collection used on Linux

2. Solaris compilers on Linux - the Sun Studio Compiler, which comes with the SUN Performance Library, a set of optimized, high-speed mathematical subroutines for solving linear algebra and other numerically intensive problems. Includes the Sun graphical debugger.

3. Intel compilers for Linux.

You can switch between the compilers easily using the 'module environment' - see the IoA computing users' guide or my guide 'Useful Unix Topics'.

If you code takes a long time to run I would recommend that you test the compilers to see which is the fastest. Remember to switch on optimisation when using the GNU compiler (`-O2`).

5 Programming style tips

Good programming style will make it much easier to write your code. Here are some tips to make your life easier.

1. Think how you could get your result without writing another program (don't write another `sort`!). Think how you can reuse the code you (or someone else) already has.

2. Plan how you are going to write your program before you start typing, even if you only plan in your head. Think carefully about the bits that will be tricky.

3. Unless you plan to write in terms of objects, divide your code into functions or subroutines that have a particular task (i.e. modular programming).

4. Start from the smallest functions, write them, compile and test them one by one (or together for very simple functions). *Do not write all your code in one go and expect it to work!*.

5. Keep unrelated code in separate source files, and compile them together (or link `.o` files). If you have multiple files you can do things like

```
f90 -g -c test1.f90                <-- makes test1.o
f90 -g -c test2.f90                <-- makes test2.o
f90 test1.o test2.o -o testprogram <-- makes testprogram
```

a *Makefile* is very useful to keep a program up to date. If you don't want to learn Makefiles write a little script to compile your program.

6. *Assertions* can help with tricky code. If you expect variables to be in a particular state (e.g. positive) at a particular point in the program, test whether they are and stop the program if they aren't. If the program stops then you know there is a bug somewhere else is in the code.

7. *Comment your code*. Do not write anything but the simplest program without something to document your code. However, don't waste time documenting obvious things:

   ```
   a=1;  /* this sets a to 1 */
   ```

   Keeping code in small chunks (functions or subroutines), with an explanation for each, is often enough commenting, except for particularly non-obvious code.

8. Backup your code (pretty obvious!).

9. Avoid arbitrary limits in your code (e.g. size of arrays). Work out how big your arrays should be. If the sizes need to change on running the program use `allocate` in Fortran 90, `malloc` in C, or `new` in C++ to allocate it.

10. In C and C++ code, use `const` as often as possible. Avoid casting unless necessary.

11. In Fortran, explicitly declare all variables (`implicit none`). Write in standard F90 or F77, not a mixture of both! Don't use proprietary compiler extensions.

12. Avoid global variables. Keep data in subroutines or functions (or hidden in modules). Doing this enforces the robustness of code and makes it easier to reuse later.