Designing command-line interfaces (CLIs) for scientific software

Daniel S. Standage <daniel.standage@gmail.com>

March 4, 2013

Abstract

It is hard to estimate how much time is lost and how much frustration results from trying to use software written by others to conduct your research. Often, the biggest problem is that the scientist writing the software has given little or no consideration to how other scientists will interact with it: how inputs and outputs are supplied, how to list and adjust settings, and so on. You do not need to become a software engineering expert to write simple, easy-to-use command line interfaces (CLIs) for your research software. This tutorial, intended for scientists with beginning- to mid-level programming skills, provides a basic overview of common CLI design patterns.

Contents

1 Overview

One very important consideration when writing scientific software is how the user will interact with it: expected inputs and outputs, how to configure settings for a particular analysis, where to look for results/output, etc. A lot of scientific software is prototypical, and often you don't need (or want) to invest much time upfront worrying about such design issues when you may never use that particular script or program ever again. However, if you find yourself frequently tweaking and reusing a piece of software, or more importantly if you are planning to distribute your code for general use (within your lab, within the community, etc), then these considerations are crucial and can save yourself and others a lot of wasted time and headache.

Unfortunately, there is little investment in teaching even the practical basics of software design in academic science since there is such intense competition over resources and so few incentives to actually write good software. Scientists must recognize that investing *that little bit of extra effort* can make a huge difference in someone else's ability to actually use the software. That *someone else* could be your advisor, your lab mate, a reviewer on a paper you have submitted, that one professor in the department on your tenure review committee, or even a future version of yourself after having worked on a different project for 6 months and trying to remember what the heck the "cutoff" setting on your old script does. Taking the time to design and document a simple, usable command-line interface for your software can save hours and hours of frustration for yourself and others in the future.

In this tutorial I provide a brief introduction to CLI design patterns commonly used in scientific software.

2 An example

Imagine for a moment that you have spent several weeks writing a program to analyze your data, and that the code for your program sits in a file named `runanalysis`. When you enter the command

```
./runanalysis
```

in your terminal, this program loads data from two different files (`reads.bam` and `genes.bed`), filters the data based on some cutoff, analyzes the data, and prints a short report to a new text file called `report.txt`.

For this example, what is the command-line interface for the `runanalysis` program? Essentially, there is none—all you can do from the command line is execute the program. *Changing how the program works*, however (such as loading different input files, using a different cutoff value, or printing the report to a different output file) requires you to modify the source code of your program.

This is a hypothetical situation, but a lot of research software begins this way. There is nothing necessarily *wrong* with implementing software as described above, but there are a few simple things you can do with its CLI that will drastically improve the program's usability, as will be discussed hereafter.

3 Arguments, options, and configuration files

3.1 Command-line arguments

Going back to our example, imagine if instead of using the command `./runanalysis` to execute the software, you used something like the following.

```
./runanalysis reads.bam genes.bed 0.5 report.txt
```

That way if you want to analyze different input files (`uni-reads.bam` and `genes-hq.bed` for example) or use a different cutoff value (`0.75` for example), you would not have to change the program's source code—instead, you would change the command you run on the command line like so.

```
./runanalysis uni-reads.bam genes-hq.bed 0.75 report.txt
```

The values on the command line that follow the name of the script or program that you run are called **arguments**. Every programming language provides a way to accessing these arguments from within your script or program. One very simple thing you can do to improve the usability and re-usability of your program is to use arguments to store the value of your input and output files, rather than "hard-coding" those filenames directly in your source code.

3.2 Options and option parsing

In the previous example, we used a command line argument to set a cutoff value to be used by the program. Imagine if you later decide that there are 5 or 6 additional program parameters that you would like to be able to set from the command line. Are you going to have to provide 9 or 10 arguments to the `runanalysis` program every time you run it? How will you remember the correct order of the arguments?

One common approach to addressing this issue is the use of **options** and **option parsing**. Options are simply command-line arguments (typically grouped in key/value pairs), while option parsing refers to the process of reading the command-line arguments and adjusting program settings as needed. Using option parsing for your program means that you set default values for certain settings in your program, and you only need to provide values for those settings on the command line if you want to use something different from the default. For example, if we used option parsing in our hypothetical script, we could set `0.5` as the default value for the cutoff, and `report.txt` as the default value for the output file. Executing the command

```
./runanalysis --cutoff=0.75 --output=new-report.txt reads.bam genes.bed
```

would override the defaults for both the cutoff and the output file, the command

```
./runanalysis --cutoff=0.75 reads.bam genes.bed
```

would override only the cutoff, the command

```
./runanalysis --output=new-report.txt reads.bam genes.bed
```

would override only the output file, whereas the command

```
./runanalysis reads.bam genes.bed
```

would use the defaults for both. Note that in each case, the names of the input files had to be provided explicitly, and in the correct order. In the context of option parsing, these are called **positional arguments**[1]. A good (and commonly used) interface design is to use a mix of options and positional arguments: positional arguments for values that *must* be provided and for which no suitable default can be used (such as the name of a data file), and options for program parameters for which a reasonable default can be used, which any given user may or may not want to override.

All common programming languages include libraries for option parsing, and they are usually pretty comprehensive in their support of different formatting conventions for option syntax. Here are a few examples of the different conventions.

[1]Some option parsers allow positional arguments to occur before, between, and/or after options, but some require that all options be placed before positional arguments.

```
# Short options: option 'b' is set to true/on, option 'i' is set to true/on, and
# option 'm' is set to 'update'; 'input.dat' is the positional argument
yourprogram -b -i -mupdate input.dat

# Same as previous example, but with a space between the option key and the
# option value
yourprogram -b -i -m update input.dat

# Same as first example, but with flags (options requiring no associated value)
# condensed
yourprogram -bimupdate input.dat

# Long options: same as first example, but with longer more descriptive option
# keys/labels
yourprogram --bin --increasing --mode update input.dat

# Same as previous example, but with a '=' sign between the option key and the
# option value
yourprogram --bin --increasing --mode=update input.dat

# Mixing and matching the different option formats; all different ways of
# calling the program with the same settings
yourprogram -i --mode update -b input.dat
yourprogram -bi --mode=update input.dat
yourprogram -m update --increasing -b input.dat
```

Appendix A provides examples for how to use readily available option parsers in a variety of common programming languages.

3.3 Configuration files

Command-line options and arguments can provide a usable and flexible interface, but only to a point. There is an inverse relationship between the number of arguments as user is expected to provide and the usability of the interface—that is, more arguments, less usability (more frustration, more chances for mistakes, etc). If your software expects the user to explicitly provide values for tens of program parameters, then arguments and options may not provide the best interface. Instead, you may consider using a configuration file: a plain text file in which the user sets values for program parameters.

Consider another hypothetical program, which is run using the following command.

```
./simulation 1000 10000 5000 0.6666 2.5 0.2 0.0001 -1.37 25000 bayes obs.dat spec.dat
```

With 12 arguments, this command is beginning to get pretty bulky. It is already hard to read, and typing out the command on the command line is error-prone. Imagine instead if we modified the program so that it read parameter values from a configuration file...

```
xlim:      1000
ylim:      10000
zlim:      5000
rate:      0.6666
magnitude: 2.5
gamma:     0.2
delta:     0.0001
mu:        -1.37
duration:  25000
method:    bayes
observe:   obs.dat
spectra:   spec.dat
```

...and was executed on the command line like this.

```
./simulation sim.conf
```

You could even implement a hybrid approach, where most parameters are set in the config file but some options/arguments are still passed via the command line.

```
./simulation --conf sim.conf --out results.sim obs.dat spec.dat
```

The config file approach removes the complexity of parameter configuration from the command line and delegates it primarily to a configuration file. Of course this has its pros and cons. On one hand, it is less likely you will be able to run the program on multiple data sets without editing the config file or creating a new one (very similar to one of the original motivating use cases). On the other hand, complex parameter configurations can be easier to manage in a text editor than on the command line, and a configuration file provides a record of the particular analysis performed which is readily available for future reference.

How many arguments/options is too many for the command line? At what point is it better to use a configuration file? There is no one right answer to this question, as it depends on a variety of factors (not the least of which is your preference as the scientist writing the software). It is safe to say that a complex program with hundreds of parameters should definitely use a configuration file, whereas for a relatively simple program with 3-5 parameters a config file is probably overkill. As you evaluate the alternatives, consider the parameters for which reasonable default values can be determined, as this can drastically reduce the complexity of the interface as well as the user's configuration task.

4 Standard input (stdin), standard output (stdout), and standard error (stderr)

Many data processing tasks can be broken down into a sequence of smaller tasks. The output of one data processing subtask serves as the input for the next subtask. Many programs and UNIX commands write their output to the terminal by default. The `print` function and its relatives found in all common programming languages allow you to print to the terminal in two ways: the standard output (`stdout`) and the standard error (`stderr`). When running a program on the command line, you can use the `>` symbol to redirect any output written to `stdout` or `stderr` to a file for subsequent processing (see this quick demo at `http://ascii.io/a/2235`).

However, an alternative to writing output to intermediate files for subsequent processing tasks is to direct the output of one program or command directly into another program or command using the `|` symbol (pipe). This paradigm is commonly used by UNIX commands. Consider the following set of commands.

```
cut -f 3 input.txt > output1.txt
sort output1.txt > output2.txt
uniq -c output2.txt > output3.txt
sort -rn output3.txt > finaloutput.txt
```

Four processing commands are run, and 3 intermediate files are created in the process. Now consider the alternative approach to performing the same processing task.

```
cut -f 3 input.txt | sort | uniq -c | sort -rn > finaloutput.txt
```

This single command produces the same output as the original 4 commands, without creating any intermediate files.

Implementing your software so that the user can choose whether output is written to a file or to the terminal is one way to improve a program's command-line interface, especially if the output can be processed line-by-line. This enables users to easily stitch your program together with other programs or shell commands. Along those same lines, you can further improve the program's CLI by enabling the user to choose whether to read input from a file or from the standard input (data redirected from another command using the `|` symbol or from a file using the `<` symbol).

5 Usage statement

When distributing software, it is imperative to provide documentation along with the code. Of course this is not just an interface issue—providing a description of the software's intended purpose, expected inputs, and how to interpret the results is all very important. One common best practice is to provide a brief comment (no more than a phrase/sentence) summarizing the program's purpose at the top of the source code file, and additional notes in `README` files and/or manuals distributed with the code.

The most common way of documenting a program's command-line interface is to provide a usage statement. A usage statement is a concise statement issued by the program itself, showing by example how to execute the program and providing a brief description of the command-line options. Typically, the usage statement should be issued if the program is executed with the wrong number of arguments, or if the user includes a special flag (typically an option such as `-h` or `--help`). The usage statement serves simultaneously as a reminder for experienced users of a software package, and as a quick reference for new users.

Below is an example of what a program's usage statement might look like on the command line.

```
standage@lappy$ talesf --help
Usage: talesf [options] genomeseq "rvdseq"
  Options:
    -f|--forwardonly         only search the forward strand of the genomic
                             sequence
    -h|--help                print this help message and exit
    -n|--numprocs: INT       the number of processors to use; default is 1
    -o|--outfile: FILE       file to which output will be written; default is
                             terminal (stdout)
    -s|--source: STRING      text for the third column of the GFF3 output;
                             default is 'TALESF'
    -w|--weight: REAL        user-defined weight; default is 0.9
    -x|--cutoffmult: REAL    multiple of best score at which potential sites
                             will be filtered; default is 3.0
standage@lappy$
```

6 Appendix A: CLI templates

The following examples show the same command-line interface implemented in several programming languages. The programs don't actually do anything other than read options in from the command line. They are intended primarily to serve as a template or reference for implementing a CLI for your own program.

Note that these examples use the canonical module for option parsing for that given language. Some languages have alternative modules (such as `argparse` and `docopt` for Python and `optparse` for R) that can simplify the process of coding and documenting your CLI. However, the price of convenience is the flexibility and control you will sacrifice when you use these alternative packages. Additionally, these alternatives are not as stable as the canonical package and require separate installation. You should consider that each external dependency you add to your software makes it more likely that someone will give up trying to use your software simply because they don't want to deal with the hassle of installing prerequisites.

6.1 Perl CLI

```
#!/usr/bin/env perl
# demo.pl: template for parsing command-line options in Perl
use strict;
use Getopt::Long;

sub print_usage
{
  my $OUT = shift(@_);
  print $OUT "Usage: demo.pl [options] data.txt
  Options:
    -f|--filter          apply strict filtering
    -h|--help            print this help message and exit
    -o|--out: FILE       file to which output will be written;
                         default is terminal (stdout)
    -s|--strand: INT     strand to search; provide a positive number for the
                         forward strand, a negative number for the reverse
                         strand, or 0 for both strands; default is 0
    -w|--weight: REAL    user-defined weight; default is 0.9
"
}

my $filter    = 0;
my $outfile   = "";
my $outstream = \*STDOUT;
my $strand    = 0;
my $weight    = 0.9;
GetOptions
(
  "f|filter"   => \$filter,
  "h|help"     => sub{ print_usage(\*STDOUT); exit(0) },
  "o|out=s"    => \$outfile,
  "s|strand=i" => \$strand,
  "w|weight=f" => \$weight,
);

if($outfile ne "")
{
  open($outstream, ">", $outfile) or die("error opening output file $outfile");
}
```

```
my $instream;
my $infile = "";
if(scalar(@ARGV) > 0)
{
  $infile = shift(@ARGV);
  open($instream, "<", $infile) or die("error opening input file $infile");
}
elsif(not -t STDIN)
{
  $instream = \*STDIN;
}
else
{
  print STDERR "error: please provide input with argument or standard input\n";
  print_usage(\*STDERR);
  exit(1);
}

while(my $line = <$instream>)
{
  chomp($line);
  # process your input here
  # use 'foutstream' file handle when printing program output
}
close($instream);
close($outstream);
```

6.2 C CLI

```
// demo.c: template for parsing command-line options in C
// compile with 'gcc -Wall -o demo demo.c'
#import <getopt.h>
#import <stdio.h>
#import <stdlib.h>

typedef int bool;
#define true 1
#define false 0
#define MAX_LINE_WIDTH 1024

void print_usage(FILE *outstream)
{
  fprintf( outstream,
"Usage: demo [options] data.txt\n"
"  Options:\n"
"    -f|--filter          apply strict filtering\n"
"    -h|--help            print this help message and exit\n"
"    -o|--out: FILE       file to which output will be written;\n"
"                         default is terminal (stdout)\n"
"    -s|--strand: INT     strand to search; provide a positive number for the\n"
"                         forward strand, a negative number for the reverse\n"
"                         strand, or 0 for both strands; default is 0\n"
"    -w|--weight: REAL    user-defined weight; default is 0.9\n" );
```

```
}

int main(int argc, char **argv)
{
  bool filter = false;
  const char *outfile = NULL;
  FILE *outstream = stdout;
  int strand = 0;
  float weight = 0.9;

  int opt = 0;
  int optindex = 0;
  const char *optstr = "fho:s:w:";
  const struct option demo_options[] =
  {
    { "filter", no_argument,       NULL, 'f' },
    { "help",   no_argument,       NULL, 'h' },
    { "out",    required_argument, NULL, 'o' },
    { "strand", required_argument, NULL, 's' },
    { "weight", required_argument, NULL, 'w' },
    { NULL, no_argument, NULL, 0 },
  };

  for( opt = getopt_long(argc, argv, optstr, demo_options, &optindex);
       opt != -1;
       opt = getopt_long(argc, argv, optstr, demo_options, &optindex) )
  {
    switch(opt)
    {
      case 'f':
        filter = true;
        break;

      case 'h':
        print_usage(stdout);
        return 0;
        break;

      case 'o':
        outfile = optarg;
        outstream = fopen(outfile, "w");
        if(outstream == NULL)
        {
          fprintf(stderr, "error opening output file '%s'\n", outfile);
          return 1;
        }
        break;

      case 's':
        strand = atoi(optarg);
        break;

      case 'w':
        weight = atof(optarg);
```

```
        break;

      default:
        break;
    }
  }

  int numargs = argc - optind;
  const char *infile = NULL;
  FILE *instream;
  if(numargs > 0)
  {
    infile = argv[optind];
    instream = fopen(infile, "r");
    if(instream == NULL)
    {
      fprintf(stderr, "error opening input file '%s'\n", infile);
      return 1;
    }
  }
  else if(!isatty(fileno(stdin)))
  {
    instream = stdin;
  }
  else
  {
    fprintf(stderr, "error: please provide input with argument or standard input\n");
    print_usage(stderr);
    return 1;
  }

  char buffer[MAX_LINE_WIDTH];
  while(fgets(buffer, MAX_LINE_WIDTH, instream) != NULL)
  {
    // process your input here
    // use 'outstream' file handle when printing program output
  }
  fclose(instream);
  fclose(outstream);

  return 0;
}
```

6.3 Python CLI

```
#!/usr/bin/env python
# demo.py: template for parsing command-line options in Python
import getopt
import sys

def print_usage(outstream):
  usage = ("Usage: demo.py [options] data.txt\n"
           "  Options:\n"
           "    -f|--filter          apply strict filtering\n"
```

```
            "    -h|--help            print this help message and exit\n"
            "    -o|--out: FILE       file to which output will be written;\n"
            "                         default is terminal (stdout)\n"
            "    -s|--strand: INT     strand to search; provide a positive number for the\n"
            "                         forward strand, a negative number for the reverse\n"
            "                         strand, or 0 for both strands; default is 0\n"
            "    -w|--weight: REAL    user-defined weight; default is 0.9")
  print >> outstream, usage

# Option defaults
dofilter = False
outfile = None
outstream = sys.stdout
strand = 0
weight = 0.9

# Parse options
optstr   = "fho:s:w:"
longopts = ["filter", "help", "out=", "strand=", "weight="]
(options, args) = getopt.getopt(sys.argv[1:], optstr, longopts)
for key, value in options:
  if key in ("-f", "--filter"):
    dofilter = True
  elif key in ("-h", "--help"):
    print_usage(sys.stdout)
    sys.exit(0)
  elif key in ("-o", "--out"):
    outfile = value
    try:
      outstream = open(outfile, "w")
    except IOError as e:
      print >> sys.stderr, "error opening output file %s" % options.outfile
      print >> sys.stderr, e
      sys.exit(1)
  elif key in ("-s", "--strand"):
    strand = int(value)
  elif key in ("-w", "--weight"):
    weight = float(value)
  else:
    assert False, "unsupported option '%s'" % key

infile = None
instream = None
if len(args) > 0:
  infile = args[0]
  try:
    instream = open(infile, "r")
  except IOError as e:
    print >> sys.stderr, "error opening input file %s" % infile
    print >> sys.stderr, e
    sys.exit(1)
elif not sys.stdin.isatty():
  instream = sys.stdin
else:
```

```
  print >> sys.stderr, "error: please provide input with file or standard input"
  print_usage(sys.stderr)
  sys.exit(1)

for line in instream:
  line.rstrip()
  # process your input file here
  # use 'outstream' file handle when printing program output

instream.close()
outstream.close()
```

6.4 Bash CLI

```
#!/usr/bin/env bash
# demo.sh: template for parsing command-line options in Bash

print_usage()
{
  cat <<EOF
Usage: demo.sh [options] data.txt
  Options:
    -f    apply strict filtering
    -h    print this help message and exit
    -o    file to which output will be written; default is terminal (stdout)
    -s    strand to search; provide a positive number for the forward strand,
          a negative number for the reverse strand, or 0 for both strands;
          default is 0
    -w    user-defined weight; default is 0.9
EOF
}

FILTER=0
STRAND=0
WEIGHT=0.9
while getopts "fho:s:w:" OPTION
do
  case $OPTION in
    f)
      FILTER=1
      ;;
    h)
      print_usage
      exit 0
      ;;
    o)
      OUTFILE=$OPTARG
      ;;
    s)
      STRAND=$OPTARG
      ;;
    w)
      WEIGHT=$OPTARG
      ;;
```

```
  esac
done
shift $((OPTIND-1))
INFILE=$1

# process your input here;
# shell scripts aren't typically used to process files line-by-line;
# rather, they call system commands that in turn do the line-by-line processing;
# if your bash script needs to accept input from stdin, you can use the system
# file '/dev/stdin' to redirect the contents of the stdin to other commands
```

6.5 R CLI

```
#!/usr/bin/env Rscript
# demo.R: template for parsing command-line options in R
library('getopt');

print_usage <- function(file=stderr())
{
  cat("Usage: demo.R [options] --in data.txt
  Options:
    -f|--filter          apply strict filtering
    -h|--help            print this help message and exit
    -o|--out: FILE       file to which output will be written;
                         default is terminal (stdout)
    -s|--strand: INT     strand to search; provide a positive number for the
                         forward strand, a negative number for the reverse
                         strand, or 0 for both strands; default is 0
    -w|--weight: REAL    user-defined weight; default is 0.9\n")
}

spec = matrix(
  c(
    "filter", 'f', 0, "logical",
    "help",   'h', 0, "logical",
    "in",     'i', 1, "character",
    "out",    'o', 1, "character",
    "strand", 's', 1, "integer",
    "weight", 'w', 1, "double"
  ),
  byrow=TRUE, ncol=4
);
opt = getopt(spec);
if( !is.null(opt$help) )
{
  print_usage(file=stdout());
  q(status=1);
}
if( is.null(opt$in) )     { opt$in     = "/dev/stdin" }
if( is.null(opt$strand) ) { opt$strand = 0 }
if( is.null(opt$weight) ) { opt$weight = 0.9 }

opt.outstream <- stdout()
if( !is.null(opt$out) )
```

```
{
  opt.outstream <- tryCatch(
    file(opt$out, "w", blocking=FALSE),
    error = function(e)
    {
      cat(sprintf("error opening output file %s\n", opt$out), file=stderr());
      quit(save="no",status=1)
    }
  )
}

# it is very uncommon to parse input line-by-line in R, so we'll use a common
# import function; if you do want/need to parse input line-by-line, you may want
# to consider whether R is the right tool for the job
data <- read.table(opt$in, header=FALSE, sep=',')

# process your input here
# use opt£out file handle when printing program output

q(save="no", status=0);
```

7 Appendix B: Revisions

- 2013-03-04: Improvements based on BioStar feedback (see `http://www.biostars.org/p/65487/`)
  - More concise abstract with an intro/overview section
  - Added R example
  - Replaced deprecated python option parsing module with the canonical module
  - Integrated a variety of other suggestions
  - Made some minor revisions to the text
- 2013-03-01: Initial version