Managing Bioinformatics Software

Version 2018-10

Licence

Table of Contents

Introduction

This course is intended to be a general introduction to the topics you need to understand in order to be able to install and configure new software in a Unix or Linux environment. Nothing in the course is inherently specific to bioinformatics, but we intend to focus on the types of software and environments most frequently used in bioinformatics. This course is not a comprehensive coverage of every possible scenario you will encounter but should hopefully cover the most common use cases and types of failure you will see.

Depending on the system you are working on you may or may not have systems administrator privileges on your machine. Some of the operations described in this course will require root (system level) access, but a lot of what you will commonly need to do can also be accomplished as a normal user. Where possible we will try to make clear exactly which parts of the instructions would require root privileges, and what alternatives might be available if you are running as a normal user.

Basic Concepts

For the purposes of this course we start from the assumption that you have some basic knowledge of how to use a Unix environment. Specifically we won't cover how to connect to your system or basic file system operations (changing directory, moving, copying or linking to files etc.). If you are not familiar with Unix to this level then it would be worth attending an introductory Unix course before going into the material here.

Before we get into the specifics of software installation though we should go through some basic concepts which will be important to understand for many of the topics later covered in this course.

Shells

A shell in Unix is the piece of software you use to interact with the system. It is the program which reads the commands you type and launches the appropriate software. As well as being command interpreters, shells also provide a simplified programming environment and can be used for basic programming and automation of repetitive tasks.

Whilst some aspects of software installation and usage are generic and apply to all shells, other commands used will differ based on the shell you are using. In this course we are going to assume that you are using the "bash" shell, which is the default on most Unix systems you are likely to encounter. If you are using a shell other than bash you will need to look up the equivalent functions in your own shell. We will aim to point out when functions discussed will differ between shells.

Where to install software

From a technical perspective you can install software pretty much anywhere you like on a Unix system, some locations will take a bit more configuration than others, but there is no technical limit to where you can put software. Having said that there are a number of locations in which it is more common to find software which has been installed.

- `/bin` and `/usr/bin` are the directories where most software installed and managed by the operating system and intended to be run by normal users reside
- `/sbin` and `/usr/sbin` are for system managed software which is primarily intended to be run by administrators, and isn't generally useful to normal users
- `/usr/local/bin` or `/opt/bin` are the normal location for additional software installed by administrators which isn't managed by the operating system and is available to all users

On some systems you will have other locations for system wide software, but there is no standard for these locations – you will have to ask the administrators of your system about these.

As a normal user you can install software in your home directory* but this will generally only be available to you. Likewise you can often install extensions or packages to other languages in your home directory, and your local copies will normally take precedence over any copies installed at the system level.

*The caveat here is that it is possible for administrators to designate certain areas of the file system as 'noexec', meaning that you can't run programs installed in those areas. Normally if home directories have been set up with this restriction then you should have been informed, but you can always check with your administrator.

The search path

One key concept to understand when installing software is the search path. This is the mechanism by which your shell will decide which executable to run when you type in the name of a program.

The search path is simply an ordered list of directories on your system within which executable files might reside. When you enter the name of a program to run your system will systematically step through the directories in the path until it finds a match to the program. Once it's found a match it will launch the program and will stop looking for further matches.

The search path allows us to solve a number of common problems:

1. How do I tell the system about a new program I've just installed
2. How does the system decide which program to run when two or more copies of a program are present in different directories

When you launch a program you can do it in one of two ways;

You can just type the name of the program – if you do this then the system will work through the path to find a match and will stop at the first hit it finds. If no hit is found then you will get a "Command not found" error.

You can type the path to the program. If you want to run a program which isn't in the path, or you want to run a copy which isn't the first within the path then instead of just giving the program's name you can provide a path to the program. This can be an absolute path like `/home/me/runme` or a relative path such as `../../bin/runme`. If a path is provided then no search is done. It is worth noting that in general the directory you are in is NOT in the search path, so if you want to run a program which is in the current directory you can't generally just run `runme`, you need to do `./runme` to tell the shell that you want to run the copy in the current directory.

You can see your current path by running (in bash) `echo $PATH`

You should see something like this:

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin
```

It can also be useful to check where in the path a particular program was found (it's easy to not be running the copy of a program you thought you were). You can do this using the `which` command followed by the command name you want to search for.

```
$ which nano
/usr/bin/nano
```

Your path is set up by a series of different configuration files in your system, some available only to your sysadmin, and some which you can modify. The path is what is termed an environment variable – a named piece of data stored by your shell. You can temporarily modify an environment variable in bash using the export command;

```
echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin
$ export PATH=/home/andrewss/bin
$ echo $PATH
/home/andrewss/bin
```

Rather than replacing the existing path it is more common to simply append new directories to it. Since it is an ordered structure you normally add new directories to the front so they get found before the ones which were there already.

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin
$ export PATH=/home/andrewss/bin:$PATH
$ echo $PATH
/home/andrewss/bin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bi
n
```

NB: Note that when you are using the path variable you need to use `$PATH`, but when you are setting it you omit the `$` and just use `PATH`.

Changes made in this way will affect only the shell you are currently using and their effect will disappear as soon as the shell exits. If you want to make this change permanent you can put the commands into a file called `.bashrc` in your home directory. Commands in this file will be run every time a new bash shell is started so changes made there will be effectively permanent.

Types of programs

There are generally two types of executable file on any Unix system.

1. Binary executables
2. Scripts

Ultimately a platform specific binary executable is what you need. These executables are tied specifically to the type of processor you have x86 vs ARM, 32bit vs 64bit etc. The same source code can often be used to compile executables for different platforms. You can move executable files between different machines but they will only work if the new machine has the same CPU architecture as the one for which the executable was originally compiled.

Scripts are files of text based source code which are not intended to be run directly, but instead are designed to be passed to a separate executable interpreter. Although you appear to run the script, what you are actually doing is running the interpreter and passing the contents of the script to it as data.

On Unix scripts work because of a convention in their first line (the header line). If a script file adheres to this format then the system will locate the interpreter and run that, passing the script as an argument.

This first line is often called the 'shebang' line, and has the structure:

```
#![interpreter location]
```

..so it could be:

```
#!/usr/bin/perl
```

```
#!/usr/bin/python
```

```
#!/bin/bash
```

..in each case the appropriate listed interpreter will be located and run when the script is executed. The location of the interpreter will be taken from the script, even if a different location for the same interpreter is found in the path.

Since the location of a specific interpreter might not be the same on all systems (my Perl might be in `/usr/local/bin` for example) then another common structure for the first line of a script file is to use the `/usr/bin/env` program which can then be passed a program name, and this will then use the first instance of that program in the search path to specify the interpreter for the rest of the script.

So, for example, if I have a script which starts with

```
#!/usr/bin/perl
```

Then the Perl interpreter in /usr/bin will be used to execute the script. If, however, I use

```
#!/usr/bin/env perl
```

Then the first instance of Perl in the PATH directories will be used instead.

Tar files and Zip

Most software will be distributed in some kind of archive file format. The most common formats you will encounter will be the tar and zip formats. Knowing how to manipulate these formats will be important to being able to work with software installations.

Tar files

Tar files are a format designed simply to bundle together a file and directory structure in a single file in order to make them easier to deal with. Originally tar was used to save data to tape (tar = tape archive), but these days it's mostly used to bundle files together. Tar is a really useful format for Unix files since it supports pretty much all of the extended attributes associated with these files so it's easy to distribute files which are all configured correctly as soon as they are extracted.

The tar file format doesn't itself have any compression associated with it, but a couple of different compression schemes are now routinely used with tar so that you gain the benefits of compression on top of the bundling. Compressed tar files are indicated by their file extension

- .tar = plain uncompressed tar file
- .tar.gz = gzip compressed tar file
- .tar.bz2 = bzip2 compressed tar file

Because the compression is added after the tar file is created you can remove it separately from unbundling the tar file contents, but most of the time people will do both operations in a single step.

The main things you need to specify when working with a tar file are:

1. Whether you are creating a new tar file or extracting from an existing one
2. Whether you want to use compression, and if so, which scheme
3. What the name of the tar file is
4. The names of the files you want to add or extract

A tar command looks very similar whether you are adding or extracting data. Let's go through the options:

1. To create a new tar file use the `c` flag. To extract use the `x` flag. You can also list the contents of a tar file without extracting by using the `t` flag.
2. To use gzip compression use the `z` flag. To use bzip2 use the `j` flag
3. To specify the tar file use the `f` flag immediately followed by the file name. If putting the `f` flag as part of a group of flags make sure it's at the end.
4. Specify the names of the files you want to add or extract. If you want to extract everything you can omit this step.
5. If you are extracting you finally want to say where to extract to. If you don't specify this then it will extract to the current working directory.

Zip files

Whilst tar files are mostly considered to be a Unix file format, zip files are more commonly found on desktop operating systems. They fulfil a similar role to tar files and many software packages are distributed in zip format.

Unlike tar files, zip files have somewhat reduced support for the full range of permissions and attributes which can be present on files in a Unix file system. You might therefore find that you need to run some additional commands to set permissions after extracting from a zip file.

The basic operation for a zip archive is to use the `unzip` command to decompress it into the current folder. Normally no additional arguments are necessary beyond the path to the zip archive itself.

Installing Software

System Package Managers

Virtually all modern Linux systems come with a suite of pre-built software which the user can opt to install. This is distributed as a set of 'packages' which are simply bundles of files which contain the executables and data files for a particular application, plus some metadata to describe the functionality they provide, and to list any dependencies they have in terms of other packages which need to be installed for this package to work.

To make it easier to work with packages, these systems also provide package managers which are programs which automate the process of fetching and installing packages and their dependencies. Most of the time when you're working with packages you will interact with the package manager, and not with the lower levels of the system.

Package managers will inherently require admin privileges so they are of no use to normal users. However, they are the easiest way to install new programs or libraries onto a system.

On a system which uses a package manager it is usually the case that only files installed through the package manger will reside in the main system directories `/usr /bin /lib` etc. Files coming from software installed system-wide, but not through the package manager, would generally go into `/usr/local` or `/opt`.

Package managers are a closed system which means that they only recognise files installed by the package manager. This has consequences for the resolution of dependencies, so that program fred lists program bob as a dependency, then even if bob has been manually installed it won't be recognised since it wasn't installed through the package manager.

Whilst you would normally use these systems for working with the suite of software provided by the operating system, sometimes people will distribute new software in these packaged formats. In these cases you can install these new files through the same system you use to manipulate your system software.

Yum and RPM

On any RedHat or CentOS based systems the packaging system is based around RPM (Redhat Package Manager) files. These have the `rpm` program to manipulate them, but generally you interact with this system through the `yum` (yum updater modified – Unix does love a recursive acronym) program, which is how you'd normally install and remove packages.

Basic operations using yum are:

To search for a package to install you do `yum search [package name]`

To install a package you would do `yum install [package name]`

To update a package to the latest version you can do `yum update [package name]`

You can do the search as a normal user, but to install a new package or update an existing package will require root privileges so you'd normally run those commands through `sudo`.

If you need to install some external software which has been distributed as an rpm file then you can do this through the lower level rpm system (and often the install instructions will tell you to do this), but this won't install any dependencies if they exist, so it's much easier to do this through yum by using.

```
yum localinstall new_software.rpm
```

Apt and DPKG

On Debian based systems such as Debian, Ubuntu, Mint etc., the equivalent package managers are `dpkg` which interacts with the deb (Debian package) file format and `apt`, which is the program you would interact with to perform most operations.

To search for a package to install you do `apt search [package name]`

To install a package you would do `apt install [package name]`

To update a package to the latest version you can do `apt upgrade [package name]`

To remove a package you would do `apt purge [package name]`

If you have a manually distributed .deb file then it is best to install it using apt since it will be able to resolve any additional dependencies and install those as well as the package you downloaded. To do this you simply do:

```
apt install ./downloaded.deb
```

As with many of these systems level operations, the installation or removal of software is not something a normal user can do, so you'd therefore need to use `sudo` for all of the commands above to elevate your privileges to root and allow you to perform the installation.

Binary Distributions

For software sourced outside your operating systems package manager the simplest type will be a binary installation. In this case the provider of the software will have already done the compilation to platform specific binary executables so all you need to do is to download these to your system. This would also apply to software which consisted only of scripts such that the interpreters would already be on your system.

To install binary software you will first need to locate the correct binary package for the system you're working on. The important parameters will be your operating system and type of CPU. For the CPU the most important distinction is usually whether you have a 32 bit or a 64 bit system. Most modern systems are 64 bit and you should always aim to run 64 bit software on a 64 bit system. If you're not sure what your system is you can run `uname -m` which should tell you the hardware platform you're using.

Once you've downloaded the correct binary package you will need to extract it so you can see the directory structure within it. If you're installing for just yourself you can extract to your home directory, but if you're doing this for all users of a system then you would normally extract it to a folder within the `/opt` directory.

Inside the directory you extracted you may well find files called README or INSTALL, which may give you specific installation instructions. If these are present then you should follow them rather than doing a generic installation.

If no specific install instructions are visible then you should be able to see the executable files for the program you're installing – these may be at the top level, or they could be in a 'bin' directory within the installation. You should at this point be able to run the software by providing a direct path to these binary files – if this doesn't work then there's no point going any further, but if this does work then your final job is to put the location of the executable files into your PATH. Instructions for doing this were in the earlier section describing how the PATH works.

Below is an example of installing a binary distribution of the bowtie2 read mapper onto a Linux x86_64 system.

The downloads page offers a few different files, but the one we want is the Linux 64 bit file.

We could download this through a browser, but in our case we copied the URL and use the command line `wget` program to fetch the file.

```
$wget http://downloads.sourceforge.net/project/bowtie-
bio/bowtie2/2.2.9/bowtie2-2.2.9-Linux-x86_64.zip

$ ls
bowtie2-2.2.9-Linux-x86_64.zip
```

This is a zip file so we need to unzip it

$ unzip bowtie2-2.2.9-Linux-x86_64.zip

```
$ ls -l
drwxrwxr-x 5 andrewss bioinf     4096 Apr 21 23:40 bowtie2-2.2.9
-rw-r--r-- 1 andrewss bioinf 27073243 Apr 21 23:22 bowtie2-2.2.9-Linux-x86_64.zip
```

We can now move into the newly created bowtie2-2.2.9 folder and see what's in there.

```
$ cd bowtie2-2.2.9
$ ls
AUTHORS                 bowtie2-build-s           LICENSE
bowtie2                 bowtie2-build-s-debug     MANUAL
bowtie2-align-l         bowtie2-inspect           MANUAL.markdown
bowtie2-align-l-debug   bowtie2-inspect-l         NEWS
bowtie2-align-s         bowtie2-inspect-l-debug   scripts
bowtie2-align-s-debug   bowtie2-inspect-s         TUTORIAL
bowtie2-build           bowtie2-inspect-s-debug   VERSION
bowtie2-build-l         doc
```

```
bowtie2-build-l-debug  example
```

In this case there is no README or INSTALL file, but we can see the bowtie2 executable file. We should check that the file is flagged as executable.

```
$ ls -l bowtie2
-rwxrwxr-x 1 andrewss bioinf 18489 Nov 11 17:03 bowtie2
```

It is already executable so we should be able to run it. If it wasn't we'd have needed to run `chmod 755 bowtie2` to set the executable flags for all users.

We can now try to run that to make sure it works.

```
$ ./bowtie2
Bowtie 2 version 2.2.9 by Ben Langmead (langmea@cs.jhu.edu,
www.cs.jhu.edu/~langmea)
Usage:
  bowtie2 [options]* -x <bt2-idx> {-1 <m1> -2 <m2> | -U <r>} [-S <sam>]
```

So this works, the last part would be to add this to our path to make it work everywhere. We'll do this temporarily as an example, but you could put this into `~/.bashrc` or `/etc/bash_profile` to make the change permanent.

```
$ cd

$ bowtie2
-bash: bowtie2: command not found

$ export PATH=/home/andrewss/bowtie2-2.2.9/:$PATH
$ bowtie2
Bowtie 2 version 2.2.9 by Ben Langmead (langmea@cs.jhu.edu,
www.cs.jhu.edu/~langmea)
Usage:
  bowtie2 [options]* -x <bt2-idx> {-1 <m1> -2 <m2> | -U <r>} [-S <sam>]
```

Dynamic libraries

One possible consideration for binary installations, which will again be relevant for source code packages are dynamic libraries on your system.

In general there are two types of binary executable which can run on a Linux system:

1. Statically compiled executables
2. Dynamically compiled executables

In a statically compiled executable all of the code needed to run the program is contained within the executable itself. There is no dependency on any other part of the system. This is a very safe option because it doesn't rely on anything else, but it's not very efficient, and it can make updating difficult.

In a dynamically compiled executable, some parts of the code are pulled from other binaries elsewhere on the system. In addition to program files, Linux systems also contain a large number of 'libraries' which are binary files containing executable code, but which just provide isolated snippets of functionality designed to be taken up by other programs. They aren't designed to be run as a program directly.

By using libraries you can easily bring complex functionality into a new program. By referring to a central copy of the library providing that functionality you can also save space by keeping just a single copy of commonly used code. If bugs or security problems are found with code in a library it is also easy to update it, without having to do anything to the set of programs which dynamically call it.

You can tell whether a binary program is statically or dynamically compiled by using the file program.

```
$ file /sbin/grub
/sbin/grub: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.18, stripped
```

This is a statically linked program which is completely self-contained.

```
$ file /usr/bin/ssh
/usr/bin/ssh: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.18, stripped
```

This is dynamically linked and will pull additional functionality from library files.

A dynamically linked executable file will not run if any of the libraries it links to are missing, broken or the wrong version. In the same way that the `PATH` variable determines which folders contain executables, the `LD_LIBRARY_PATH` variable says which folders to look for libraries in. In addition to the folders listed in `LD_LIBRARY_PATH`, those listed in `/etc/ld.so.conf` will also be searched.

To see which libraries will be used to satisfy the requirements for a program you can use the ldd program.

```
$ ldd /usr/bin/ssh
        Linux-vdso.so.1 =>  (0x00007ffc0998c000)
        libfipscheck.so.1 => /lib64/libfipscheck.so.1 (0x00007f01e3675000)
        libseLinux.so.1 => /lib64/libseLinux.so.1 (0x00007f01e3455000)
        libcrypto.so.10 => /usr/lib64/libcrypto.so.10 (0x00007f01e3072000)
        libutil.so.1 => /lib64/libutil.so.1 (0x00007f01e2e6f000)
        libz.so.1 => /lib64/libz.so.1 (0x00007f01e2c58000)
        libnsl.so.1 => /lib64/libnsl.so.1 (0x00007f01e2a3f000)
        libcrypt.so.1 => /lib64/libcrypt.so.1 (0x00007f01e2808000)
```

```
        libresolv.so.2 => /lib64/libresolv.so.2 (0x00007f01e25ed000)
        libkrb5.so.3 => /lib64/libkrb5.so.3 (0x00007f01e20cc000)
        libk5crypto.so.3 => /lib64/libk5crypto.so.3 (0x00007f01e1e9f000)
        libcom_err.so.2 => /lib64/libcom_err.so.2 (0x00007f01e1c9b000)
        libnss3.so => /usr/lib64/libnss3.so (0x00007f01e1960000)
        libc.so.6 => /lib64/libc.so.6 (0x00007f01e15cb000)
        libplc4.so => /lib64/libplc4.so (0x00007f01e13c6000)
        libdl.so.2 => /lib64/libdl.so.2 (0x00007f01e11c2000)
        /lib64/ld-Linux-x86-64.so.2 (0x000000376a400000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x00007f01e0933000)
        libnssutil3.so => /usr/lib64/libnssutil3.so (0x00007f01e070d000)
        libplds4.so => /lib64/libplds4.so (0x00007f01e0509000)
        libnspr4.so => /lib64/libnspr4.so (0x00007f01e02cb000)
```

Each line lists a dependent library for that program. The requirement is satisfied if it lists a library path, along with a memory location where it is loaded (eg `/lib64/libz.so.1 (0x00007f01e2c58000)`), or if it just has a memory location – which means it's satisfied by code in the kernel, eg (`Linux-vdso.so.1 => (0x00007ffc0998c000)`)

If you find a dependency which isn't satisfied then you will see the right-hand side say "Not found"

```
$ ldd jellyfish
        Linux-vdso.so.1 =>  (0x00007ffd4a352000)
        libpthread.so.0 => /lib64/libpthread.so.0 (0x000000376b400000)
        libjellyfish-2.0.so.2 => not found
        libstdc++.so.6 => /usr/lib64/libstdc++.so.6 (0x0000003770400000)
        libm.so.6 => /lib64/libm.so.6 (0x000000376ac00000)
        libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x000000376f000000)
        libc.so.6 => /lib64/libc.so.6 (0x000000376a800000)
        /lib64/ld-Linux-x86-64.so.2 (0x000000376a400000)
```

This means that you will need to either install the missing library into a standard location, or install it somewhere else and add that new folder to `LD_LIBRARY_PATH`.

Source Code Distributions

If a pre-compiled binary distribution isn't available for the platform you wish to run on then you may need to download a source code distribution and compile the software locally.

Downloading Source Packages

The first step in this installation will be to download the source code you need. Depending on the project you're working with you may either have to pull the code straight from their version control repository, or you might download an archive file of code. Be careful when downloading that you've pulled down the appropriate code. You may well find that the most obvious download link you find is the current development version of the software and not the last stable release. You should generally work with the most recent stable release unless you have a specific reason to select another branch of the code.

Pulling code from source code repositories

One of the most common ways to get hold of a project's source code is to pull it directly from their version control system. There are many different systems but the most commonly used ones are git, subversion and mercurial. All of these require you to have installed the client application for the version control system on your machine first – these will normally be available from your system's software packages, but if not then binary downloads for all of these are available.

In this course we're not going to go into the intricacies of how to use these version control systems, we're going to do the simplest operation which is to use them to pull down a copy of the code for a project.

In each case you will need to find the URL for the branch of the code you want to check out. Depending on how the repository is set up this might be a simple `http://` or `https://` web url, it might be an `ssh://` url direct into their server or it might be using a custom protocol such as `svn://` or `git://`. The project page should give you the details of this.

To clone a repository branch in git you would use:

```
git clone [repository url]
```

For subversion you would use

```
svn co [repository url]
```

For mercurial you would use

```
hg clone [repository url]
```

Each of these commands will bring down all of the source code, but will also configure the repository in such a way that if you later want to update to the latest version of the code you can simply go back into the folder you created and run one of:

```
git pull
svn update
hg pull && hg update
```

Compiling

For some projects the code you download is functional without an additional compilation step – this would apply to projects which distribute code in scripted languages such as Perl or Python. For projects

written in compiled languages such as C or C++ you will need to compile the source code to produce executables which you can then install in the system.

As with the binary distributions the easiest thing to do is to look within the files you've downloaded to see if there is an INSTALL or README file which details the correct compilation and installation procedure for this software. If there is then you should follow those instructions, if not then you can see if one of the common compilation routines will work.

For C and C++ programs there are a common set of tools which you will need to have available in order to compile new software. The most important parts will be the compiler itself, which is the program which interprets the source code and produces platform specific binary files. The most common compilers are the GNU standard compiler `gcc` (and `gcc++` for C++) or `clang` which is most commonly found on OSX, but also works on other operating systems. Well written C or C++ can normally be compiled by any valid compiler, but you will often find that in practice the project may be much easier to compile with one compiler than another.

In addition to the compiler you will normally need a 'make' system, which is a piece of software used to coordinate the somewhat complex process of building a large software project. The most common make systems are the gnu make program (just called `make`) or `cmake`. The make system to use is determined by how the project has been set up – the authors get to choose this, not you.

Finally, most C/C++ programs will also rely on the presence of a number of different libraries. These are separate packages which provide functionality not in the core language, and which are used so that the program authors don't have to implement everything themselves within their code. Some libraries, such as the standard C library (libc or glibc) are used in pretty much every compiled program, some are used fairly commonly (the boost libraries for example) and some are more specialised and will only be required by a smaller subset of software (libraries dealing with network communication or compression for example). We will talk more about libraries in the debugging section below.

In some cases the software will be distributed with a fixed make file, so that all you need to do to run the build is to type 'make' and it will do the rest.

More commonly though, software will have been configured with a system called autotools. This is a configuration system which can check for the presence of pre-requisites and can modify the way the compilation works based on the capabilities of the system the build is running on. An autotools installation is the most common type of standard compilation, so we'll look at this in more detail.

Autotools compilations have four steps to them:

1. Configuration
2. Compilation
3. Testing
4. Installation

The configuration step is started by moving into the top level directory of the project to compile and running:

```
./configure
```

This can be modified by passing additional arguments to configure to change how it behaves. Many of these options will be project specific to add or remove functionality from the program, or to manually specify the location of specific dependencies the program might have. You can in many cases see what options you have by running:

```
./configure -help
```

One option which is also set here though is the final installation location of the compiled program. The normal behaviour of autotools is to install the final binaries under `/usr/local`. If you want to install the program somewhere else you need to specify `prefix=[folder to install to]` in the configure options.

Configure will run a large batch of tests to assess the functionality of your system, compiler and libraries. It will check this against the requirements of the program. If all of the dependencies aren't met then it should hopefully give you an intelligible error message to say what it requires which you are missing.

If configure completes successfully then it will write out a customised make file (called Makefile) which allows you to move onto the compilation stage.

Once you have a valid Makefile you can run the compilation by running:

```
make
```

This will run the compilation in a single thread on your machine. If you have multiple cores on your machine you can allow make to use these by instead running:

```
make -j [number of cores to use]
```

This should make your build complete substantially faster since many parts of the build process can normally be run in parallel.

After make has completed you should test your installation before installing it. Most projects will come with a built-in test suite which can be run by running:

```
make test
```

Hopefully these will all pass, which means you can proceed to the final installation of the compiled binaries by running:

```
make install
```

This will write the compiled objects to either the directory you specified at the configure stage, or to `/usr/local` if you didn't specify anything.

The configuration, building and testing phases of this process do not require administrator privileges, and should always be run as a normal user. For the final installation stage, if you are installing system-wide you should use elevated privileges for the final `make install`. If you are installing into a directory you own already though, you can run this as a normal user too.

Here is an example of downloading and compiling the latest ncbi-blast codebase. This is distributed as a source tarball so we'll download that rather than pulling from a source repository.

```
$ wget ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/LATEST/ncbi-blast-2.3.0+-src.tar.gz
$ tar -xzvf ncbi-blast-2.3.0+-src.tar.gz
```

We can now move into the ncbi-blast-2.3.0+-src folder which was created. In there we find just a single folder called c++, which we also have to move into. In this inner folder we find a standard autotools configure script.

```
$ ls
configure.orig
configure
scripts
```

```
include
src
compilers
```

We can now run the configure script, opting to install in a non-standard directory.

```
$ ./configure --prefix=/home/andrewss/blast
[cut lots of stuff]
To build everything:  cd /home/andrewss/ncbi-blast-2.3.0+-
src/c++/ReleaseMT/build && make all_r
or simply run make in the current directory
******* CONFIGURATION SUCCESSFUL *******
```

We can now run the compilation and spread this over 16 cores.

```
$ make -j 16
```

This will take a while to complete and will generate a lot of output. It is fairly common to see warnings generated at this stage, but you shouldn't get any errors.

We can see if this package has a test suite:

```
$ make test
make: *** No rule to make target `test'.  Stop.
```

So in this case there isn't one, so we can move ahead to install it.

```
$ make install
```

If we now look in the directory we specified as the install target we should see the newly added files.

```
$ ls /home/andrewss/blast/
bin  include  lib
$ ls /home/andrewss/blast/bin/
aalookup_unit_test        delta_unit_test                queryinfo_unit_test
aascan_unit_test          dustmasker                     redoalignment_unit_test
align_format_unit_test    gapinfo_unit_test              remote_blast_unit_test
bdbloader_unit_test       gencode_singleton_unit_test    rpsblast
bl2seq_unit_test          gene_info_reader               rpstblastn
blastdb_aliastool         gene_info_unit_test            rps_unit_test
blastdbcheck              hspfilter_besthit_unit_test    scoreblk_unit_test
blastdbcmd                hspfilter_culling_unit_test    search_strategy_unit_test
blastdbcp                 hspstream_unit_test            seedtop
blastdb_format_unit_test  legacy_blast.pl                segmasker
blastdiag_unit_test       linkhsp_unit_test              seqdb_demo
blastengine_unit_test     makeblastdb                    seqdb_perf
blastextend_unit_test     makembindex                    seqdb_unit_test
blastfilter_unit_test     makeprofiledb                  seqinfosrc_unit_test
blast_formatter           msa2pssm_unit_test             seqsrc_unit_test
blast_format_unit_test    ntlookup_unit_test             setupfactory_unit_test
blasthits_unit_test       ntscan_unit_test               split_query_unit_test
blastinput_unit_test      optionshandle_unit_test        subj_ranges_unit_test
blastn                    phiblast_unit_test             tblastn
blastoptions_unit_test    prelimsearch_unit_test         tblastx
blastp                    project_tree_builder           tracebacksearch_unit_test
blast_services_unit_test  psibl2seq_unit_test            traceback_unit_test
blastsetup_unit_test      psiblast                       uniform_search_unit_test
blast_unit_test           psiblast_iteration_unit_test   update_blastdb.pl
```

```
blastx                          psiblast_unit_test              version_reference_unit_test
convert2blastmask               pssmcreate_unit_test            windowmasker
datatool                        pssmenginefreqratios_unit_test  windowmasker_2.2.22_adapter.py
deltablast                      querydata_unit_test             writedb_unit_test
```

We can now run the programs using an absolute path, and by adding the bin directory to our PATH we could make them accessible anywhere.

```
$ /home/andrewss/blast/bin/blastn
BLAST query/options error: Either a BLAST database or subject sequence(s)
must be specified
Please refer to the BLAST+ user manual.
```

Compilation troubleshooting

The most common point of failure when compiling software happens at the configuration stage and is the result of missing dependencies. This could be dependencies which are completely absent, those which are partially present but parts are missing, and those which are present but are not the specific version the program requires.

For example, the text below is the end of a configure session which failed. In this case one of the libraries required was found, but the version found was too old to use:

```
checking zlib.h usability... yes
checking zlib.h presence... yes
checking for zlib.h... yes
checking if zlib version >= 1.2.5... no
checking whether zlib support suffices... configure: error: zlib library
and headers are required
```

To work out how to fix this we need to talk a bit about how shared libraries work. A shared library (which is what you pretty much always use in this type of compilation) is simply an independently packaged set of code which is designed to be used by many different programs on a system. When you compile a program which relies on a shared library the code from the library isn't incorporated into your executable, but instead your executable 'links' to it, so that it can refer to the code sitting in a separate file rather than having to duplicate it. This means that you can update a shared library on a system without having to recompile all of the programs which use it.

There are two parts to a shared library. The main part is a shared object file (usually with a .so file extension), which is the compiled code the library provides. When running a dynamically linked program this binary file is all that is needed.

The other part to a library is the 'header' file. This is a text file (usually with a .h file extension) which lists the functions and variables the library provides so that the program knows how to use them. These header files are only required when compiling programs against the shared library, and often aren't installed by default when the object files are installed.

If a dependency is missing or out of date you have a few options therefore:

The library is installed but the headers are not

If you are using a system installed library you will often find that only the object files are installed. For each library package there is usually a corresponding header package which you'll need to install to be able to compile against it.

On RedHat/CentOS systems the header package appends `-devel` to the end of the normal library name, so the header package for `zlib-1.2.3-29` would be `zlib-devel-1.2.3-29`.

On Debian/Ubuntu/Mint systems the header package appends `-dev` to the end of the normal library name, so the header package for `zlib1g_1.2.8` would be `zlib1g-dev_1.2.8`.

If you are missing the header package for a system installed library you can therefore simply use your system's package manager to install the missing headers.

Neither the library nor the headers are installed, but they are in the system package repository

If the library and its associated headers are missing then the easiest way to install them is from your OS's package repository, you just need to make sure that you install both the binary and header packages.

The OS doesn't have the library you need, or the OS library version is too old

If you can't install the dependency you need via your OS package manager then you'll need to install it manually. Installing a library is exactly the same as installing a source distribution of a software package with the same download, unpack, `configure`, `make`, `make test`, `make install` routine. By default libraries will install under `/usr/local` where they will be automatically detected by the system ahead of any system copies of the library and things will just work.

Sometimes, however, you may need to install the library somewhere else on the system (for example, if you don't have permission to write into `/usr/local`. For the compilation of the library you'd do this by setting the `prefix` option during configure, but you'll need to take some extra steps to have the library recognised by the compiler when you come back to compiling your original software package.

There are two stages at which your new library needs to be recognised, during compilation and when the compiled program is being run. All of these steps can be achieved by modifying environment variables.

For the compilation step you need to tell the compiler where both the header and object files are for your library. You tell it about the header files by modifying the `CPPFLAGS` environment variable, and the object files using the `LDFLAGS` variable.

For the `CPPFLAGS` you're adding options to the standard compilation so you need to put in `-I/include/dir/location` for each additional directory you want to include.

For the `LDFLAGS` it's the same idea but you need `-L/lib/dir/location`.

If you have multiple entries you separate them by spaces. The example below shows how to set the environment to add 2 non-standard library locations.

```
export CPPFLAGS="-I/bi/apps/zlib/1.2.8/include/ -I/bi/apps/bzip2/1.0.6/include/"
export LDFLAGS="-L/bi/apps/zlib/1.2.8/lib -L/bi/apps/bzip2/1.0.6/lib"
```

Since these variables are only needed at compile time you don't normally need to add them to your permanent environment.

When the compiled program is running it will still need to find the object files for these libraries so you need to make a permanent change to your environment so they are visible. In this case the variable you need to change is the `LD_LIBRARY_PATH`. You should append the directories you want to include to the front of the existing value for this variable. Since you'll need this addition to be permanent you should make this change in your `~/.bashrc` file where it will be applied every time you start a new session.

```
export LD_LIBRARY_PATH=/bi/apps/zlib/1.2.8/lib: /bi/apps/bzip2/1.0.6/lib:$LD_LIBRARY_PATH
```

Extending other languages

As well as installing new software, the other common installation you will need to do is to add extensions to an existing package, normally a programming language. Sometimes these can add functionality you can use when writing scripts in that language, but in some cases they will actually add complete new software packages.

Here we'll go through the most common types of installation in some of the most common software environments you're likely to encounter.

Installing R packages

R has become one of the core technologies which bioinformaticians rely on for their day to day work. The core R language is relatively compact and most of the scientific analysis routines are provided by additional packages which you can use to extend the language. We'll look at how to install these packages from the most common sources.

Core packages from CRAN

The R developers have an official package repository which is directly integrated into the R language and can be used to install a large number of additional packages. This repository is called the Comprehensive R Archive Network (CRAN) and can be viewed at https://cran.r-project.org/.

Installing packages from CRAN can be done within any R session. You use the `install.packages` function to install a package from CRAN. For example:

```
install.packages("RColorBewer")
```

…will install the RColorBrewer package. If you have sufficient privileges to write into the main R installation then the package will be installed for all users. If not then R will make a local package repository for you and install the package just for you in there.

When you first run this on a system you will be asked to select a mirror to use for the downloads. You can select a mirror which is geographically close. If you have a slightly older version of either R or the curl libraries on which R depends you may find that your R doesn't support https transport. In this case you'll need to look for the option to use a normal http mirror (option 29 in the example below) and then select from that list.

```
--- Please select a CRAN mirror for use in this session ---
HTTPS CRAN mirror

 1: 0-Cloud [https]                2: Austria [https]
 3: Belgium (Ghent) [https]        4: Chile [https]
 5: China (Beijing 4) [https]      6: Colombia (Cali) [https]
 7: France (Lyon 1) [https]        8: France (Lyon 2) [https]
 9: France (Paris 2) [https]      10: Germany (Münster) [https]
11: Iceland [https]               12: Italy (Padua) [https]
13: Japan (Tokyo) [https]         14: Mexico (Mexico City) [https]
15: New Zealand [https]           16: Russia (Moscow) [https]
17: Serbia [https]                18: Spain (A Coruña) [https]
19: Spain (Madrid) [https]        20: Switzerland [https]
21: UK (Bristol) [https]          22: UK (Cambridge) [https]
23: USA (CA 1) [https]            24: USA (KS) [https]
25: USA (MI 1) [https]            26: USA (TN) [https]
```

```
27: USA (TX) [https]                28: USA (WA) [https]
29: (HTTP mirrors)
```

Many R packages are distributed as binary packages, but a lot will require you to have a minimal build environment on your machine (`make`, `gcc` etc). Some of the packages might also need additional header packages to be available for their compilation (the `XML` package would require the `libxml2` headers for example).

Bioconductor packages

Bioconductor is an independent package repository dedicated to packages performing biological analyses. It enforces coding and documentation standards on the packages it supports as well as using some common data structure standards to make inter-operability between packages better. It can be found at http://bioconductor.org/.

Bioconductor is not integrated into the R language in the same way that CRAN is, so to use it you need to install the code for the bioconductor package manager, called biocLite. To do this you need to run some code direct from the bioconductor web site.

```
source("https://bioconductor.org/biocLite.R")
```

As with CRAN mirror selection it's possible that your version of R doesn't support https, in which case you'd just modify the URL to use http.

The first time you use bioconductor you need to install all of the base code by running:

```
biocLite()
```

After this new packages can be added by running:

```
biocLite("package_name")
```

For example, to install DESeq2 you would run:

```
biocLite("DESeq2")
```

Installation from Github

For new packages which are still in development and haven't yet been submitted to CRAN or Bioconductor it is fairly common to host these on github. Whilst it is possible to do a manual installation by downloading the latest code release from the site, there is an easier way to pull a package from github straight into R.

This method is provided by an R package called `devtools`. This package is in CRAN so you can install it from there. It provides wrappers for installing from github, bitbucket and arbitrary git repositories. Github is probably the most commonly used of these, but they all work in a similar way.

To use `devtools` to install from github you simply load the `devtools` package and then call `install_github`, passing a string which incorporates the username of the user hosting the repository, and the repository name.

For example, to install the atSNP package from https://github.com/chandlerzuo/atSNP you would do:

```
library(devtools)
install_github("chandlerzuo/atSNP")
```

Manual installation

All of the methods above are wrappers around the core R offline installation method. In this method of installation you need to download or create a tar file for the package you want to install. This is a specially structured file with a directory and file structure R will recognise.

Unlike in other tar based installations, in this case you do NOT expand the tar file yourself. Instead you need to pass the entre tar file as an argument to R. The command you need is:

```
R CMD INSTALL [tar file location]
```

You would normally want to install this for all users, so you will probably want to run this via `sudo` to run it as root.

For example, if I download the `vioplot` distribution from https://cran.r-project.org/src/contrib/vioplot_0.2.tar.gz I can install it using:

```
R CMD INSTALL vioplot_0.2.tar.gz
* installing to library '/bi/apps/R/3.3.0/lib64/R/library'
* installing *source* package 'vioplot' ...
** R
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* DONE (vioplot)
```

Installing Perl Modules

Perl has a central repository of modules called CPAN (the comprehensive Perl Archive Network) and most Perl modules will be installed from here. As with other languages there are easier and harder ways to achieve this depending on the level of control you need.

Perl modules sit in a structured file hierarchy based on the name of the module. Each Perl installation comes with a default set of base folders into which modules will be installed, and if possible it's a good idea to put new modules into one of these since they will then be recognised by all users of the system.

You can see where the global folders for Perl module installation are by running `perl -V`. The folders are stored in a structure called `@INC`.

```
@INC:
    /bi/apps/perl_modules/lib/perl5/x86_64-Linux-thread-multi
    /bi/apps/perl_modules/lib/perl5
    /bi/apps/perl_modules/share/perl5/
    /bi/apps/perl_modules/lib/perl5/x86_64-Linux-thread-multi
    /bi/apps/perl_modules/lib64/perl5
    /usr/local/lib64/perl5
    /usr/local/share/perl5
    /usr/lib64/perl5/vendor_perl
    /usr/share/perl5/vendor_perl
    /usr/lib64/perl5
    /usr/share/perl5
    .
```

You can see that the current directory (a single dot) is included in this list, so one option is that you can install a module in your current directory if it's only needed for one script, however most of the time you want to put them in a more permanent location.

You can also set up local repositories for Perl if you do not have permission to write to the global directories. There is no standard location for personal repositories so you can put it wherever you like. You will however need to tell Perl where to find it.

When you've set up a new personal repository you can't directly add it to @INC (not without recompiling Perl at any rate), but there are two other options to making it be recognised. One of these is to add an environment variable called PERL5LIB to your environment. This can contain a set of directories which will be added to @INC dynamically at run time. The other option is that within your Perl script itself you can manually add directories to the search path by including one or more lines like:

```
use lib "/dir/with/modules/in/it";
```

This will change the module search path just for that script.

Installing from OS repositories

Possibly the simplest way to install new Perl modules is to use bundled modules which are part of your OS package repository. Any OS package which relies on a Perl module will have had to either incorporate that code into the package, or more normally will have made a package just for that Perl module. For example, on Redhat/Centos the name of the modules is simply the Perl module name with the `::` replaced by `-` and with `perl-` prepended to the name, so `Net::IP` would be `perl-Net-IP`. Installation of these packages would be the same as for any other OS package.

Automatic installation with the CPAN module

The next easiest way to install a Perl module is – use a Perl module. Specifically the CPAN module is designed to make it easy to install modules and their dependencies from CPAN. Perl ships with a set of 'core' modules – modules which are guaranteed to be present with any installation of Perl, and the CPAN module is one of these so if you have Perl, then you can use it.

The easiest way to use the CPAN module is to invoke it from a command line.

```
perl –MCPAN –e 'shell'
```

The first time you do this you will be asked a bunch of questions about where you want to store and build your modules, any parameters you want to add to the default options, which mirror you want to use, and whether you want to automatically install dependencies. Generally the default answers to these questions are all sensible so you can just press return a bunch of times.

This will get you a prompt which looks like this:

```
cpan>
```

Which is the CPAN shell. To install a module you simply run:

```
install Module::name
```

You will see a lot of text scroll past as the module downloads, configures, compiles and installs all of the modules and dependencies you need.

One thing which sometimes gets set strangely is the policy for installing dependencies. The default for this is to ask you whether you want to add the new package to the list to install which gets really annoying for big installs. To change this so dependencies will be automatically installed, start the CPAN shell and then use:

```
cpan> o conf prerequisites_policy follow
cpan> o conf commit
```

To make dependency installation automatic for future installs.

If you want to change the options the CPAN shell is using then you can run the following command in a CPAN shell.

```
o conf init
```

..and it will re-run the initial setup (your existing settings will be the defaults so just change what you need to).

Manual installation

The manual installation of a Perl module requires that you first locate and download a tar.gz file for the module you want to install. You can get these direct from the cpan.org website or any other source.

Once you have the tar.gz file you need to uncompress it. This should generate a folder with the same name as the module. To start the installation you would then move into the newly created folder.

There are two different systems used to build Perl modules. Older modules used a system called `ExtUtils::MakeMaker` but some newer modules use a system called `Module::Build`. The two systems have slightly different commands to use, but both follow the traditional compilation steps:

1. Configure
2. Make
3. Test
4. Install

In each case there are no commands which are specific to an individual module – all modules use the same commands.

The way to tell which system a given module uses is to look at the files in the top level of the installation folder. If you see a file called `Makefile.PL` then this is a MakeMaker module. If you see a file called `Build.PL` then it's a `Module::Build` module.

For make maker the commands you need are:

```
perl Makefile.PL
```

If you want to install in a custom location then you can change this to:

```
perl Makefile.PL INSTALL_BASE=/my/perl/dir
```

This is then followed by

```
make
make test
make install
```

If you want to install globally then the make install command will need to be run from an account with sufficient privileges to write into the global installation folders. This could be by switching accounts, or by using `sudo`.

For a module build module the equivalent commands are:

```
perl Build.PL
```

To customise the install location you'd use:

```
perl Build.PL --install_base /my/perl/dir
```

This is then followed by:

```
./Build
./Build test
./Build install
```

Again, the final command will need to be run with suitable privileges.

Installing Python Packages

Python too has the ability to install extensions to the core language, called packages. As with R and Perl it comes with tools to automate the installation and dependency resolution within the package set.

One complication with Python is that it is very common to have two separate python installations on your system, one for python2 and one for python3 so you need to be careful to check which version you are using. Where both versions of Python are installed, the python 3 installation can normally be unambiguously addressed using programs such as `python3` and `pip3` rather than just `python` and `pip`, but if you're not sure you can use which Python to double check which installation you're using.

Python will search for packages defined in a set of directories on your system. Some of these will be locations hard linked to the location of the python installation, but you can add folders to the PYTHONPATH environment variable to make Python search in other locations for installed packages.

To see the full list of searched directories you can use the `sys.path()` function in Python.

```
$ python
Python 2.7.3 (default, Feb 15 2013, 11:09:45)
[GCC 4.4.6 20110731 (Red Hat 4.4.6-3)] on Linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> print ("\n".join(sys.path))

/bi/apps/python/2.7.3/lib/python27.zip
/bi/apps/python/2.7.3/lib/python2.7
/bi/apps/python/2.7.3/lib/python2.7/lib-dynload
/bi/apps/python/2.7.3/lib/python2.7/site-packages
/bi/apps/python/2.7.3/lib/python2.7/site-packages/ipython-0.13.1-py2.7.egg
/bi/apps/python/2.7.3/lib/python2.7/site-packages/ply-3.4-py2.7.egg
/bi/apps/python/2.7.3/lib/python2.7/site-packages
```

Automatic installation with pip

Python actually has more than one system which can potentially manage its packages, but the most commonly supported package manager is pip, and this is probably the best one to use for most purposes.

Although pip will often be shipped along with python it is not required to be, so it's possible that you'll find a Python installation which doesn't have pip available. It's also worth checking to see if pip might be present but not in the PATH. In some Python installations pip is included in a Scripts sub-directory of the main installation, and that may not be in the PATH.

To check for the presence of pip you can run:

```
pip --version
```

This will give an error if pip is not present. If it's not available then there are a few ways to get it. If you're using your operating system's version of Python then there may well be a separate package for pip and that will be the best way to install it. For example something like:

```
sudo apt install python-pip
```

```
sudo yum install python-pip
```

Would work for the Python installs on Debian/Ubuntu or RedHat/CentOS.

If you're not using an OS provided version of Python then you should be able to install pip using a package which should ship with python.

```
python -m ensurepip --default-pip
```

If even that doesn't work then the last resort is to download the `get-pip.py` script from https://bootstrap.pypa.io/get-pip.py and run it. This should install pip and all dependencies.

Once you have pip available then installing packages from the main Python Package Index (PyPI) is easy:

```
pip install vectormath
```

By default pip will try to install the package into the global package repository on your system so that it is available to all users. This will normally require admin privileges to work so you will need to run the command through `sudo`.

If you want to use pip to install a package but you either don't have admin privileges, or you only want to install for you then you can use:

```
pip install --user vectormath
```

Which will use a Python package repository in your home directory.

In addition to installation you can also use pip to upgrade a package which is already installed to the latest version.

```
pip install --upgrade vectormath
```

..and you can even remove packages (as long as they were originally installed using pip)

```
pip uninstall vectormath
```

Manual installation

You can also do a more manual installation of Python packages following a very similar recipe to the one we saw previously with Perl. The source for packages is distributed as a tar.gz file so you'll need to download that from PyPI or wherever the source of the package is. Once you have that then the actual installation is relatively straight forward. The package itself should contain a configuration and installation script called setup.py, so all you need to do is to run:

```
python setup.py install
```

This will do the configuration checking, compilation (if needed) and installation of the files to their final location. Other package dependencies will **not** be automatically resolved in this mode of installation so you'll need to install them manually. Other compilation issues, particularly in C components may also fail and you'll need to resolve these in the same way you would for a C program installation before retrying the `setup.py` command.

By default the packages will install into the standard system library folder, which should already be in your PYTHONPATH. You can specify an alternate installation location by running:

```
python3 setup.py install --prefix=/my/custom/install/folder
```

What you will probably find is that the install will then fail because the destination folder (which will be a sub-directory of the folder you actually specify) is not in your PYTHONPATH.

```
$ python setup.py install --prefix=/bi/scratch/andrewss
running install
error: bad install directory or PYTHONPATH

You are attempting to install a package to a directory that is not
on PYTHONPATH and which Python does not read ".pth" files from.  The
installation directory you specified (via --install-dir, --prefix, or
the distutils default setting) was:

    /bi/scratch/andrewss/lib/python2.7/site-packages/

and your PYTHONPATH environment variable currently contains:

    ''

Here are some of your options for correcting the problem:

* You can choose a different installation directory, i.e., one that is
  on PYTHONPATH or supports .pth files

* You can add the installation directory to the PYTHONPATH environment
  variable.  (It must then also be on PYTHONPATH whenever you run
  Python and want to use the package(s) you are installing.)

* You can set up the installation directory to support ".pth" files by
  using one of the approaches described here:

  https://setuptools.readthedocs.io/en/latest/easy_install.html#custom-
installation-locations


Please make the appropriate changes for your system and try again.
```

The fix is to then add this folder to your PYTHONPATH (preferably in a way which makes the change permanent), and then run the command again, where it should then succeed.

Software package installation with CONDA

Earlier in the manual we talked about using your operating system's package manager to install software packages which have been prepared to run on your system. This is a very convenient way to install core software and libraries and is the basis for software maintenance in most modern Linux distributions.

There is now a project called CONDA which provides a similar service to your operating system's package manager, but in a distribution agnostic manner, meaning that it can be used on a variety of different operating systems. This is an open repository which allows people to submit recipes for new applications which weren't supported before, and it now contains an impressively large number of applications.

Whilst CONDA is not tied to any particular type of software, there is a sub-project called BioCONDA which has a focus on the packaging of bioinformatics software. This can be a very quick and easy way to get the software you want to be available easily.

CONDA Usage

Installation

To start using CONDA you need to install the base system on your machine. This actually encompasses a complete python installation which contains the Anaconda installer. The simplest way to get this is to use the Minconda Python distribution.

You can install Miniconda into your home directory by using the auto-install bash script distributed by the project. Simply download and run:

https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh

..and the conda command will be available in your shell.

Setting up channels

If you want to be able to use the BioCONDA packages then you'll need to configure your installation to know about the BioCONDA repository. To do this you need to run:

```
conda config --add channels bioconda
conda config --add channels conda-forge
```

You only need to do this once and the settings will be remembered.

Installing software

Once `conda` is installed and the channels are configured then you can install new software packages as follows:

```
conda install bwa
```

This will install the `bwa` package, plus any dependencies and will make it immediately (and permanently) available in your shell

One other issue which conda can address is what to do when you need more than one version of an application to be installed, or if you have multiple applications you want to install which are fundamentally incompatible with each other (conflicting dependencies). The solution for this in conda

is to create a custom environment. An environment is an isolated set of package installations and you can switch between environments whenever you like.

To install packages into a new environment you can do:

```
conda create -n aligners bwa bowtie2 hisat2
```

This will create a new environment called “aligners” and will install bwa, bowtie2 and hisat2 into it. Once the installation is complete you can run:

```
source activate aligners
```

..to active that environment to gain access to the programs installed into it. You can later run:

```
source decativate
```

..to turn the environment off and get back to your standard set of packages.

Just use conda for everything then?

Given that `conda` is so simple compared to some of the other methods of software installation it’s fair to ask why should we not just use this for everything? In many cases conda can be an excellent solution to quickly gain access to a range of applications, but there are some caveats to using a system like this.

Applications need to be supported in conda

Kind of an obvious one, but in order to use conda your application needs to be supported by an existing recipe. If there isn’t a recipe then you can’t use it to install the application. You’re also tied to whatever version of the software they support in the recipe – if there is an update it may not appear in conda immediately.

Conda should be treated as a closed system

In exactly the same way that we saw for the operating system’s package manager, conda only knows about applications that it has installed itself. In order to guarantee compatibility it won’t rely on copies of applications which may be available already on your system, but will install and use its own versions.

This firstly means that you can end up with duplication on your system. Applications you have installed by other mechanisms will be duplicated within conda. Also, if you are maintaining a system for others to use then the model conda uses means that you are likely to get duplication between different users on the same system, which means everyone gets their own copy of an application, which can cause issues when troubleshooting. If you use environments within conda then you will also get duplication between these – the same application installed into different environments will cause a new copy of the files for that application to be downloaded and installed.

Finally, the closed nature of conda can cause problems if you use programs installed within conda to interact with other parts of your system. For example we’ve seen problems if you try to install R or Python packages from a conda version of R or Python using the libraries and compiler available on your system. Some packages can be installed directly through conda, but incompatibilities can arise if you start to mix and match parts of your operating system with software installed through conda.

The bottom line is that conda can be great as long as everything you need for a given application is directly available in the conda repositories already.

Running Containerised applications

A major development in the distribution of applications or pipelines in recent years has been the use of containers. These are a mechanism through which a controlled environment containing one or more applications and associated data. The container can then be used to either provide an interactive environment where you can use the applications and data defined in it, or by including a master program in the container you can treat it as if it was an application and use it as a black box to process new data or reproduce previous analyses.

What types of container are there

The mostly commonly used container format is Docker (https://www.docker.com/), however Singularity (https://www.sylabs.io/singularity/) containers are now becoming increasingly popular.

Both of these systems can be used to package up applications in a portable fashion and both are widely used. Singularity is more directly focused on scientific reproducibility and some of the containers it offers are fairly heavy as they bundle large datasets and pipelines as well as the application which are run in them.

On a practical level, one big difference between docker and singularity is that docker containers are designed to be installed by root, and using docker requires admin privileges. Singularity on the other hand can be used by conventional users if they just want to re-use an existing container (creating new containers locally requires root). Since singularity can also transparently use docker containers we are going to focus on the use of singularity to run containerised environments on our Linux systems.

Finding a container

Whilst it is possible to create containers locally and use them in a similarly local fashion, the point of containers is to be able to share them between users and systems. You will therefore most commonly obtain your container from a public repository.

- DockerHub (https://hub.docker.com/) is the main repository of docker images.
- SingularityHub (https://singularity-hub.org/) is the main repository of singularity images.

Both of these sites have search engines which are generally a bit rubbish, but if you know the name of a given pipeline or application then you can normally find an image which contains what you want. For popular applications you should be aware that there are likely to be multiple containers which have that application and you may want to look at what else is in the container, and if there are any user comments which highlight problems.

Containers should have some documentation with them to say what they are designed to be used for, but you may well find that this is either terse or completely absent in many cases.

What you need to obtain is the name of the container within the system, eg "`qiime2/core`" or "`dleehr/qiime2-singularity`". You'll also need to know whether the repository was on docker or singularity hub.

Installing a container with singularity

Installing Singularity

Before you can start to use a container you will need to install the actual singularity application. If you are on Debian/Ubuntu then singularity is in the standard package repository and you can install it with;

```
sudo apt install singularity-container
```

[Note that 'singularity' is a different package (it's a game) – you need 'singularity-container']

On other systems you can install the 2.x.x. versions of singularity by downloading the latest release from the github project page. It comes as a standard autotools distribution so you can use the standard `configure`/`make`/`make install` routines to install it.

For singularity v3.x.x the project has switched from writing in C to writing in Go, so the install is a bit more complicated. Instructions for doing this can be found in the singularity documentation.

Installing a container

Once you have singularity installed then you can download a container from either docker or singularity hub.

Eg:

```
singularity pull shub://dleehr/qiime2-singularity
```

..or from docker..

```
singularity pull docker://qiime2/core
```

This will generate a local .simg file which you can use to run the container. The file will be given a name based around the URL from which it was downloaded, eg dleer-qiime2-singularity-master-latest.simg, which can be a bit of a mouthful. If you want to give the local file a shorter name then you can do:

```
singularity pull --name qiime.simg shub://dleehr/qiime2-singularity
```

..which will allow you to specify the filename you want to use.

Running a container

Once you have the container image file on your local system then you can run it to be able to access the functionality it contains.

There are two main ways to run a container. The most flexible is to start a shell session within the container. This will allow you to access the software contained in the container.

```
singularity shell qiime.simg
```

This will move your shell into the container. You will still be able to see the files you had in your home directory, but the rest of the system will be replaced with the contents of the container system. You won't be able to mix and match between running applications in your base OS and those in the container.

The other method of execution is to run the container as an application. You can do this either by explicitly calling the run method for the container.

```
singularity run qiime.simg
```

..or, more simply, you can simple execute the image file directly.

```
./qiime.simg
```

In either case what this does is to open a shell into the container and then run the /singularity program. The person building the container can configure what actions this actually performs, but you can treat the whole thing as if it's a conventional application, even if it's doing something way more complex in the background.

Debugging

To finish with we're going to trying to give some ideas of how to diagnose, debug and fix problems when trying to run a program in Linux.

The program won't start at all

Here we're assuming that the error you get when you try to launch the program doesn't come from the program itself (ie to say that the options you've set are wrong or something like that), but are a more generic error from the operating system.

Check you're definitely running the program you think you are

Start simply – assuming you're just running something like `myprogram` from a shell then make sure that you're launching the file you think you are:

```
which myprogram
```

In case there is some ambiguity about this you can check for other instances of `myprogram` later in the PATH using.

```
which -a myprogram
```

If this doesn't find anything then either the program isn't installed, or it's installed somewhere which isn't in the PATH, or it's installed into the PATH, but doesn't have execute permissions.

Check the permissions on the file

In order to be able to run a program you need certain permissions.

1. You need to be able to read the program file and any other files it needs to read to run (configurations, libraries etc).
2. You need to have the execute bit set on the program.
3. You should check for any unnecessary permissions, particularly SUID or SGID

In general, if you're not sure then `chmod 755 myprogram` is probably the correct permissions for it to have.

If it's a script, check the shebang line

Look at the first line of the script, make sure that it starts with the correct structure of shebang line, eg something like:

```
#!/usr/bin/python
```

Check a few things. Stop when it works!;

1. Check the shebang is definitely on line 1 (no previous blank lines) and has the correct structure
2. Check the location of the interpreter is the same on your system. If it's not either update it to the new location or use the `/usr/bin/env` program to set the interpreter from the PATH
3. Check if you can run the script if you run `/usr/bin/python [scriptname]` rather than relying on the shebang line at all
4. Try putting `--` after the interpreter, eg `#!/usr/bin/python --` Sometimes odd line endings can mess up the ability of the shell to find the interpreter.

If it's a binary file check for broken links

Run `ldd [program name]` and look for anything where a link is required but not found. If the corresponding library isn't on your system then you'll need to install it. If it is on the system, but not being found, then modify your `LD_LIBRARY_PATH` to include the directory where the library can be found and then try again. It can also be helpful to run `file [program name]` just to double check that the file you're trying to execute actually is a program file!

The program gives an error when you run it

Errors from within a piece of software are much more varied, since they will often be the result of how the program is launched. Some generic pieces of advice though:

Read the error messages!!

Software authors usually go to great efforts to try to tell you (as best they can) why something isn't working. Reading the error they give is the quickest way to try to understand what's gone wrong.

One thing to remember is that the most relevant error may not be the last thing printed by the program. Often when things die, finding the most relevant error is the biggest problem.

Some things to remember:

Many command line applications will respond to incorrect launch options by printing an error and then printing the help page for the program. This can push the actual error off the screen so you need to scroll up to find it.

In general you want to find the earliest error in a large set of errors, because when something starts to go wrong it can trigger a cascade of failures. Often the total amount of error output can be large, so you might be better off to save the output from the launch to a file (remembering that errors go out on STDERR so you need to use 2>[somewhere] to save them, and then looking for error in a program like less to find the initial problem.

In some languages (eg Python and java) an error will often come with a 'stack trace' which is the set of internal calls which were being followed when the error was triggered. It's worth noting that different languages structure these differently. For example, java writes a stack trace with the most recent call at the top and older calls further down, so the more informative error is at the top. Python, on the other hand, does this the other way around and puts the newest call at the bottom. Figuring out which end of the trace to look at can help to focus on the actual cause of the error.

Check arguments and file paths

For command line programs, by far the two most common mistakes made are:

1. People type the name of an option wrongly, or mess up the spacing between options, eg:
    a. `--gnome` instead of `--genome` (check your spelling!)
    b. `--genome=Human` instead of `--genome Human` (some programs will accept either, but it varies)
    c. `--quiet--genome Human` instead of `--quiet --genome Human` (spaces matter!)
    d. `--genome Homo sapiens` instead of `--genome "Homo sapiens"` (can't have spaces in arguments without using quotes)

2. People get file paths wrong, and spell folders or file names incorrectly. Always use command line completion to fill in the names of files – even if they're really short. You can't make a spelling error if you used tab to complete the file name.