Package Managers

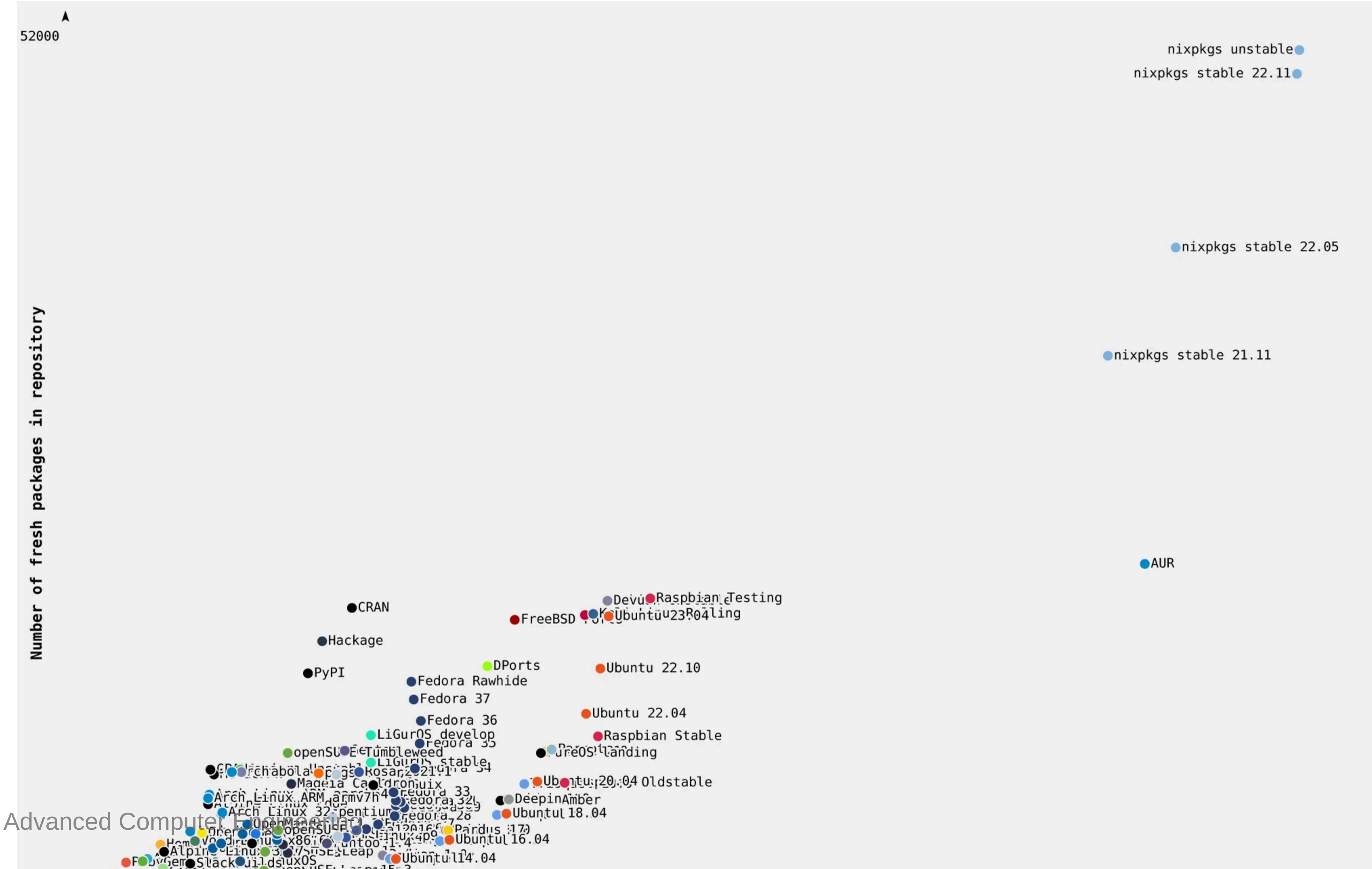
```
Module Code: ELEE1119

Module Name: Advanced Computer Engineering

Credits: 30

Module Leader: Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA
```

Download as a PDF


52000
Number of fresh packages in repository
nixpkgs unstable
nixpkgs stable 22.11
nixpkgs stable 22.05
nixpkgs stable 21.11
AUR
CRAN
Hackage
PyPI
FreeBSD
Raspbian Testing
Ubuntu 23.04
Ubuntu 22.10
Ubuntu 22.04
DPorts
Fedora Rawhide
Fedora 37
Fedora 36
LiGurOS develop
Raspbian Stable
openSUSE Tumbleweed
LiGurOS stable
Mageia Cauldron
Ubuntu 20.04
Oldstable
Deepin
Amber
Arch Linux ARM armv7h
Fedora 33
Arch Linux 32 pentium
Ubuntu 18.04
Pardus 17
Ubuntu 16.04
Ubuntu 14.04
Alpine
Slackbuilds

Package Management Overview

- Unix systems are superior to Windows: Package Management.
- Can install almost anything with ease of from your terminal.
- Update to the latest version with one command.
    - No more download the latest installer nonsense!
- Various tools can be installed by installing a package.
    - A package contains the files and other instructions to setup a piece of software.
    - Many packages depend on each other.
    - High-level package managers download packages, figure out the dependencies for you, and deal with groups of packages.
    - Low-level managers unpack individual packages, run scripts,
      and get the software installed correctly

Many different philosophies

- Monolithic binary packages: one big "app" with everything bundled together
  - docker containers, most windows programs
- Small binary packages: separate common code into independently-installed "libraries"
  - MSI files, Ubuntu, most of linux
- Source-based packages: no installers at all! Compile all your programs
  - language-based package managers, brew, portage
- Benefits to all approaches
  - monolithic binary: fastish install, very independent programs
  - small binary: very fast install, less wasted space
  - source-based: fastest code, smallest install, easy to use open-source

Package Managers in the Wild

- GNU/Linux:
  - Low-level: two general families of binary packages exist: `deb`, and `rpm`.
  - High-level package managers you are likely to encounter:
    - **Debian/Ubuntu**: `apt-get`, `apt`, `aptitude`.
    - **SUSE/OpenSUSE**: `zypper`.
    - **Fedora**: `dnf` (Fedora 22+) / `yum`.
    - **RHEL/CentOS**: `yum` (until they adopt `dnf`).
    - **Arch**: `pacman`
    - **Gentoo**: `Portage`, `emerge`
- **Mac OSX**:
  - Others exist, but the only one you should ever use is `brew`.
  - Don’t user others (e.g. `port`), they are outdated / EOSL.

Using Package Managers

- Though the syntax for each package manager is different, the concepts are all the same:
  - This lecture will focus on `apt`, `dnf`, `emerge`, and `brew`.
  - The `dnf` commands are almost entirely interchangeable with `yum`, by design.
  - Note that `brew` is a "special snowflake", more on this later.
- What does your package manager give you? The ability to
  - install new packages you do not have.
  - remove packages you have installed.
  - update installed packages.
  - update the lists to search for files / updates from.
  - view dependencies of a given package.
  - a whole lot more!!!

A Note on update

- These "subcommands" are by category, not name: `update` is not always called `update`
- The `update` command has importantly different meanings in different package managers.
- Most do not default to system (read linux kernel) updates.
  - **Fedora** does; most others do not.
- It depends on your operating system, and package manager.
  - Know your operating system, and look up what the default behavior is.
- If your program needs a specific version of the linux kernel, you need to be very careful!
  - very, very few programs care about your kernel version.

A Note on Names and their Meanings

- Package names sometimes specify architecture:
  - [`3456x`]`86` (e.g. `.i386` or `.i686` or `x86`): these are the **32-bit** packages.
  - `x86_64` or `amd64`: these are the **64-bit** packages.
  - `noarch`: these are independent of the architecture.
- Ubuntu / fedora often splits packages into smaller pieces:
  - The header files are usually called something like:
    - `deb`: usually `<package>-dev`
    - `rpm`: usually `<package>-devel`
  - The library you will need to link against:
    - If applicable, `lib<package>` or something similar.
  - The binaries (executables), often provided by just `<package>`.
  - Most relevant for `C` and `C++`, but also Python and others.
  - Use the `search` functionality of your package manager.

Example Development Tool Installation

- To compile and link against `Xrandr` (X.Org X11 `libXrandr` runtime library) on **ubuntu**, you would have to install:
    - `libxrandr2`: the library.
    - `libxrandr-dev`: the header files.
    - Usually don't explicitly include the architecture (e.g. `.x86_64`), it's inferred
    - If you're getting link errors, try installing explicit 32/64-bit version.
        - just google your error
- Splitting devel files more common for binary package managers, less for source-based ones.

System Specific Package Managers

RHEL / Fedora Package Managers (yum and dnf)

- Installing and uninstalling:
    - Install a package:
        - `dnf` `install` `<pkg1>` `<pkg2>` ... `<pkgN>`
    - Remove a package:
        - `dnf` `remove` `<pkg1>` `<pkg2>` ... `<pkgN>`
    - Only **one** pkg required, but can specify many.
    - "Group" packages are available, but different command:
        - `dnf` `groupinstall` `'Package Group Name'`
- Updating components:
    - Update EVERYTHING: `dnf` `upgrade`.
    - `update` exists, but is essentially `upgrade`.
        - Specify a **package** name to only upgrade that package.
    - Updating repository lists: `dnf` `check-update`
- Searching for packages

Gentoo package manager (`portage` with `emerge`)

- Source-based package manager: compiles your packages
  - just runs a special **bash** script to compile
  - very, very fine-grained control over dependencies and features
  - use the latest software specialized to your hardware!
- **USE** flags control special "optional" features
  - would be separate packages on ubuntu
  - Want **java** or **emacs** integration? `USE="java emacs…"`
- Installing, uninstalling, and updating
  - `emerge` `<pkg>` to install
  - `emerge` `-v` `--depclean` to remove
    - explicitly checks to ensure other packages don’t need it first
  - `emerge` `-uND` `@world` to upgrade everything
    - flags are "update", "newuse" (if you turned on a feature), "deep"(also check

Cautionary Tales

- **WARNING**: if you install package **Y**, which installs **X** as a dependency, and later remove **Y**
  - Sometimes **X** will be removed immediately!
  - Sometimes **X** will be removed during a cleanup operation later
- Solution?
  - Basically, **pay attention to your package manager**.
  - Install packages explicitly that you need
  - Check lists of packages when removing things
- Why does this happen at all?
  - Linux splits things into dependencies: avoids lots of extra copies
  - Side effect: dependencies are visible to you; you can use directly
  - In windows: dependencies are hidden

Package Management is a core Philosophy

- Most of what makes a Linux distribution is its package manager
- Reflects Distribution’s philosophy
  - Ubuntu: "just work" and don’t think too hard
  - Fedora: "latest everything" but keep it stable+not too hard
  - Arch: I want to understand how my distro works.
  - Gentoo: I do understand how my distro works.

If you’re thinking of installing Linux, by the way…

- **Ubuntu**
  - Benefits: easy install, out-of-the-box setup, common things "just work"
  - Drawbacks: too much magic; system "just work" scripts break if you need to do too many uncommon things and aren’t really careful
- **Fedora**
  - Benefits: still pretty easy to install, lots of good "get started quick" stuff. Good in a VM too
  - Drawbacks: a little less stable; can change deep system things but also not hard to break your system that way.

If you’re thinking of installing Linux, by the way…

- **Arch**
  - Benefits: wealth of knowledge, really helps you understand why your system works and what makes it work
  - Drawbacks: limited automagic. Takes real time to set things up, or change important things.
- **Gentoo**
  - Benefits: similar to Arch, plus the source-based Portage package manager is pure gold. Great if you’re doing serious programming/systems work, or if you really need a thing from github that was released last week, or you have a limited environment. Great way to really learn Linux.
  - Drawbacks: absolutely no automagic. Takes real time to set things up, compiling is time-consuming, all the docs think you know what you’re doing.

OSX Package Management: Install brew on your own

- Sitting in class right now with a Mac?
- **DON’T DO THIS IN CLASS**. You will want to make sure you do not have to interrupt the process.
- Make sure you have the "Command Line Tools" installed.
  - Visit http://brew.sh/
  - Copy-paste the given instructions in the terminal as a regular user (not root!).
- **VERY IMPORTANT**: READ WHAT THE OUTPUT IS!!!! It will tell you to do things, and you have to do them. Specifically
  You should run '`brew doctor`' **BEFORE** you install anything.

OSX Package Management (brew)

- Installing and uninstalling:
    - Install a formula:

      `brew` `install` `<fmla1>` `<fmla2>` ... `<fmla2>`

      Remove a formula:

      `brew` `uninstall` `<fmla1>` `<fmla2>` ... `<fmlaN>`
    - Only one `fmla` required, but can specify many.
    - "Group" packages have no meaning in brew.
- Updating components:
    - Update `brew`, all taps, and installed formulae listings. This does not update the actual software you have installed with `brew`, just the definitions: brew update.
    - Update just installed formulae: `brew` `upgrade`.
        - Specify a formula name to only upgrade that formula.
- Searching for packages:
    - Same command: brew search

OSX: One of These Kids is Not Like the Others (Part I)

- Safe: confines itself (by default) in `/usr/local/Cellar`:
  - common feature of "non-system" package managers
  - No `sudo`, plays nicely with OSX (e.g. Applications, python3).
  - Non-linking by default. If a conflict is detected, it will tell you.
  - Really important to read what `brew` tells you!!!
- `brew` is modular. Additional repositories ("taps") available:
  - This concept exists for all package managers
- Common taps people use:
  - `brew tap homebrew/science`
    - Various "scientific computing" tools, e.g. opencv.
  - `brew tap caskroom/cask`
    - Install `.app` applications! Safe: installs in the "Cellar", symlinks to `~/Applications`, but now these update with brew all on their own when you brew update!

OSX: One of These Kids is Not Like the Others (Part II)

- `brew` installs **formulas**.
    - A `ruby` script that provides rules for where to download something from `/` how to compile it. Similar concept to portage’s bash files
- Sometimes the packager creates a "Bottle":
    - If a bottle for your version of OSX exists, you don’t have to compile locally.
    - The bottle just gets downloaded and then "poured".
- Otherwise, `brew` downloads the source and compiles locally.
- Though more time consuming, can be quite convenient!
    - `brew options opencv`
    - `brew install --with-cuda --c++11 opencv`
    - It really really really is magical. Just like USE flags in Gentoo!
    - `brew reinstall --with-missed-option formula`

OSX: One of These Kids is Not Like the Others (Part III)

- Reiteration: pay attention to `brew` and what it says. Seriously.
- Example: after installing `opencv`, it will return:

```
==> Caveats
Python modules have been installed and Homebrews site-packages
is not in your Python sys.path, so you will not be able to
import the modules this formula installed. If you plan to
develop with these modules, please run:
mkdir -p /Users/sven/.local/lib/python2.7/site-packages
echo 'import site; site.addsitedir( \
"/usr/local/lib/python2.7/site-packages")' >> \
/Users/sven/.local/lib/python2.7/site-packages/homebrew.pth
```

- brew gives copy-paste format, above is just so you can read.
- I want to use `opencv` in `Python`, so I do what brew tells me.

Language-specific package management

- Modern programming language environments have their own package managers
  - **Haskell**: `cabal`
  - **Ocaml**: `opam`
  - **Python**: `conda` / `pip` / `pip3`
  - **Ruby**: `bundler` / `gem`
  - **Rust**: `cargo`
- Works basically exactly like `brew`
  - separate, user-specific install directory
  - preferred to system packages but does not replace them
- Be careful when using these!
  - system packages are not preferred, but sometimes get used anyway
  - when languages rely on external packages, things get really hairy

Other Managers

- There are so many package managers out there for different things, too many to list them all!:
  - **Ruby**: `gem`
  - **Anaconda Python**: `conda`
  - **Python**: `pip`
  - **Python**: `easy_install` (but really, just use `pip`)
  - **Python3**: `pip3`
  - **LaTeX**: `tlmgr` (uses the CTAN database)
    - Must install `TeX` from source to get `tlmgr`
  - **Perl**: `cpan`
  - **Sublime Text**: `Package Control`
  - Many many others…

Some notes and warnings about Python package management.

- **Notes:**
    - If you want X in Python 2 and 3:
        - `pip install` **X** and `pip3 install` **X**
        - **OSX** Specifically: advise only using `brew` or **Anaconda Python**. The system Python can get really damaged if you modify it, you- are better off leaving it alone.
        - So even if you want to use `python2` on Mac, strongly encourage you to install it with `brew`.
- Warnings:
    - Don’t mix `easy_install` and `pip`. Choose one, stick with it.
        - But the internet told me if I want `pip` on Mac, I should `easy_install pip`
        - NO! Because this `pip` will modify your system `python`, USE BREW.
- Don’t mix `pip` with `conda`. If you have **Anaconda python**, just stick to using `conda`

Concepts in language-specific (per-user) package management

- Packages do not require **root** to install
- Packages installed to per-user directory
  - normall a "dotfile" directory in your home
  - better-behaved things in `~/.local/share`
- need to change your environment variables to use correctly
  - usually at least `$PATH` and `$LD_LIBRARY_PATH`
  - sometimes also `$JAVA_HOME`, `$PYTHON_PATH`, etc
- can control selection of package managers with edits to `$PATH`

Useful Resources:

- https://distrowatch.com/dwres.php?resource=package-management
- https://repology.org/