COMP1005 Week 11 Cheat Sheet

Lisa Luff

10/27/2020

Contents

Programming Environments

- Other command line interfaces (CLI) (other than Vim/Vi)
  - nano
  - gedit
  - atom
  - emacs
  - IDLE
  - Jupyter Notebook
  - Spyder
  - Pycharm
  - Many more

IDLE

- Integrated Development and Learning Environment
- Included in Python
- Uses tkinter GUI toolkit
- Features:
  - 100% python
  - Cross-platform (windows, mac, linux, etc.)
  - Python shell window (interactive interpreter) with colourised code input, output and error messages
  - Multi-window text editor will multiple undo, smart indent, call tips, auto-completion, and more
  - Debugger with persistent breakpoints, stepping and viewing global and local namespaces
- To use:
  import tkinter as tk

Jupyter Notebook

- Features:
  - Combine code and text, formatting, equations
  - Save as PDF
  - Share easily
  - Environments and kernels for selecting software versions
  - Inline images and plots
  - %run to run Python scripts
  - Auto-indent, auto-brackets, syntax highlighting

Spyder

- Powerful scientific environment written in Python for Python
- Designed by and for scientists, engineers and data analysts
- Included with most Python installers
- Features:
  - Advanced editing, analysis, debugging, and profiling functionality for code development
  - Data exploration, interaction execution, deep inspection, and visiualistion capabilities
  - Abilities can be extended further via it's plugin system and API
- To use
  $ spyder&
  - Press F8 to run through the code

Pycharm

- "The Python IDE for Professional Developers"
- Features:
  - Intelligent Python editor
  - Grpahical debugger and test runner
  - Navigation and refactorings
  - Code inspectors
  - VCS support
- Paid version:
  - Scientific tools
  - Web development
  - Python web frameworks
  - Python Profiler
  - Remote development capabilities
  - Database and SQL support
- To use - actual program

Debugging and Code Inspection Tools

- Automated tools to help debug code
  - Some can do as you go
- Code checkers:
  - Pyflakes -
    - Parses code instead of importing it, so doesn't detect as many errors, but is safer to use
    - Wont execute broken code that would do permanent damage to the system
      - To use Pyflakes: pyflakes *file* in command line, prints issues
  - Pylint and PyChecker -
    - Import the code, so produce more extensive lists of errors and warnings
    - Used for greater functionality
  - pycodestyle -
    - Specifically looks for bad coding style against PEP8
    - To use: pycodestyle *file* in command line, prints issues
    - **Note - pep8 deprecated**
- Debuggers:
  - pdb -
    - Module provides an interactive source code debugger
    - Supports:
      - Setting (conditional) breakpoints
      - Single stepping at the source of line level
      - Inspection of stack frames
      - Source code listing
      - Evaluation of arbitrary Python code in the context of any stack frame
      - Post-mortem debugging (after a crash) called under program control
  - IDLE, Spyder and Pycharm provide debugging support
    - Options include:
      - Setting (conditional) breakpoints
      - Single stepping at the source line level
      - Stack trace inspection

Version Control

- Also known as Revision Control or Source Control
- Lets you track your files over time
- Can do it manually with save as, but gets messy on large projects
- Programs to do it are Version Control Systems (VCS's)
- Features:
  - Backup and restore - Files saved as they're edited, and you can jump to them any any time
  - Synchronisation - Lets people share files and stay up-to-date with the latest version
  - Short-term undo - Throw away changes and go back to "last known good" version in the database
  - Long-term undo - Jump back to the old (working) version to see very old changes
  - Track changes - All files are updates, and you can leave messages explaining why the change happened. Easy to see how and why file is evolving over time
  - Track ownership - A VCS tags every change with the name os the person who made it. Good for blamestorming or giving credit
  - Sandboxing - Make temporary changes in an isolated area, test and work out the kinks before "checking in" the changes
  - Braching and merging - Branch a copy of yoru code into a separate area and modify it in isolation (tracking changes). Later, you can merge your work back into the common area
- Tracking changes:
  - VCS start with a base versin of the document and then record changed you make each step of the way
  - Like a recording of your progress, you can rewind to start at the case, and play back each change made, eventually arriving at the current version
- Terminology:
  - Basic setup
    - Repository (repo) - The database storing the files
    - Server - The computer storing the repo
    - Client - The computer connecting to the repo
    - Working set/working copy - The local directory of files, where you make changes
    - Trunk/main - Primary location for code in the repo. Like a family tree, the trunk is the main line.
- Merging:
  - You can change a section of code, and it will slot it into the main code
- Conflicts:
  - If trying to merge but doesn't align with current trunk code (eg. merging two things and they double up on removing something, one will work, one will have a conflict)
- VCS's:
  - git
  - mercurial (hg)
  - bazaar
  - subversion (svn)
  - version control
  - concurrent version system (cvs)
  - perforce
  - visual source safe
  - Bitbucket

Git

- Advantages:
  - Resiliance - No one repository has more data than any other
  - Speed - Very fast operations compared to other VCS (especially CVS and Subversion)
  - Space - Compression can be done across repository not just per file. Minimises local size as well as push/pull data transfers
  - Simplicity - Object model is very simple
  - Large userbase with robust tools
- Disadvantages:
  - Difficult learning curve, especially for those used to centralised systems
  - Can seem overwhelming to learn
  - Conceptual difference
  - Huge amounts of commands
- Uses snapshot storage
- Three trees of Git:
  - The HEAD - last commit snapshot, next parent
  - Index - Proposed next commit snapshot
  - Working directory - Sandbox
- Basic workflow:
  - Init a repo (possible init of clone)
    $ git init
  - Tell git who you are
    $ git config –global user.name "*your name*"
    $ git config –global user.email "*your email*"
  - Edit files
  - Stage the changes
    $ git status
  - Review your changes
    $ git add *filename*
    $ git status
  - Commit the changes of directory with a comment
    $ git commit -m "*comment*"
- Checking changes and history:
  - git diff - Show the difference between working directory and staged
  - git diff –cached - Show the difference between staged and the HEAD
  - git log - View history
- Using backups:
  - git checkout *commit hash*
    - Commit hash is the first 4 numbers of commit when looking at the log
- Using remote repository:
  - Get changes
    - git fetch
    - git pull (fetches and merges)
  - Propagate changes
    - git push
  - Protocols
    - Local filesystem - (file:////)
    - SSH - (ssh:////)
    - HTTP - (http://// or https:////)
    - Git protocol (git:////)
- Github:
  - An online collaborative CVS

Writing Packages

- PyPI - https:////pypi.python.org/
- Need to consider structure
  - Need to use guides beyond PEP8
  - Eg. PEP257 - docstring conventions
    - All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings
    - Public methods (including the ___init___ constructor) should have docstrings
    - A package may be documented in the module docstring of the ___init___.py file in the package directory
- Package guidelines should make it easy to;
  - Install with pip or easy_install
  - To specify as a dependency for another package
  - For other users to download and run tests
  - For other users to work on and have immediate familiarity with the basic directory structure
  - To add and distribute documentation
- Package name constraints:
  - All lowercase
  - Unique on pypi, even if you don't want to make your package publicly available
  - Underscore-separated or no work separators at all (no hyphens)
- Directory structure
  - Top level directory is the root of the SCN repo, eg *package*.git
  - Sub-directory of the same name is the actual python module, holds:
    - ___init___.py
    - setup.py -
      from setuptools import setup
      setup(name = '*package*',
      version = '*n*',
      description = '*string*',
      url = '*url*',
      author = '*me*',
      author_email = '*email*',
      license = 'MIT',
      packages = ['*package*'],
      zip_safe = False)
      - Then the package can be downloaded with pip locally
- To register the package to PyPI
  $ python setup.py register
  - If you haven't registered anything before you will need an account
  - Then anyone can download it with pip, and it can be made a dependency of other packages, and be automatically installed when that package is installed
- Ignoring files
  - Don't want to include all files in the package (Eg. intermediaary files made automatically by Python during development)
  - Use .gitignore to automate (or equivalent for other SCM/VCS's)
    - Compiled Python modules:
      *.pyc
    - Setuptools distribution folder:
      /dist/
    - Python egg metadata, regenerated from source by setuptools
      /*.egg-info

Package Risks

- Any code you haven't written yourself (and your own does too) presents a risks:
  - Errors in the code
  - Slow or no support for updates
  - Becoming unsupported
  - Dependencies on other packages
- What to consider:
  - Is it developed by an individual or community?
  - How responsive are the developers?
  - How recently has it been updated?
  - Does it depend on other packages that are neglected?

Agent-Based Models

- ABM combines simulation and object-oriented models to simulate the behaviour of autonomous objects over time
- A simple behavioural model can generate complex results over time
- Each simulation will give different results as there is a random factor in the behaviour/position/environment for each agent

Games and Graphics

- Range of packages for Python to provide graphics and games for development capability
  - Pyglet
  - Pygame
  - Many more

Pyglet

- A pure python cross-platform application frameowrk intended for hame development
- Supports windowing, user interface event handling, OpenGL graphics, loading images and videos and playign sounds and music
- It works on Windows, OS X and Linux
- Features:
  - No external dependencies or installation requirements
  - Takes advantage of multiple windows and multi-monitor desktops
  - Load images, sound, music, and video in almost any format
  - Provided under the BSD open-source license, allowing you to use it for both commercial and other open-source projects with very little restriction