**Python Programming Tools and Workflow**

**This is an introduction to the tools and workflow we use during the course. Please note, there are no assignments you need to hand in for this chapter.**

**Tools and Workflow**

Having a structured and effective workflow is foundational to your success at this programming course and other data science courses at this institute. This document will cover the core best practices that you will use over the next 6 months.

**Overview**

1. [Toolchain](https://github.com/alexseong/dsy_python_programming/blob/master/notes/workflow.md#toolchain): know the tools, use the tools, love the tools

* iTerm2
* Atom (IDEs)
* iPython

1. [Keyboard Shortcuts](https://github.com/alexseong/dsy_python_programming/blob/master/notes/workflow.md#keyboard-shortcuts)

* Don't use the mouse!

1. [Interactive Development Workflow](https://github.com/alexseong/dsy_python_programming/blob/master/notes/workflow.md#interactive-development-workflow)

* Keep the feedback loop tight when writing code

1. [Version Control with git](https://github.com/alexseong/dsy_python_programming/blob/master/notes/workflow.md#version-control-with-git)

* Always be committing

1. [Recap](https://github.com/alexseong/dsy_python_programming/blob/master/notes/workflow.md#recap)

**Toolchain**

Everyday here you'll be writing programs in Python. You have 2 options when you're developing:

1. Develop in a text editor (Atom) and run the code with the Python interpreter (iPython in iTerm2)
2. Develop in an interactive repl (read-eval-print loop, iPython in iTerm2)

For the most part, you'll use option 1. Option 2 (developing in a repl) is best when you're trying out small bits of code.

You can significantly increase your productivity by mastering your tools and continuing to invest in the craft of building software. Learning and using a programming editor is a foundational skill. If you know vi or emacs, continue to use them. If you haven't mastered them, then Atom is a good option. Editors are a personal choice and the source of bitter religious disputes. Choose a professional editor which works for you.

**Atom, iTerm2, and iPython will be your workhorses. Get to know them well.**

**Hello World in 2 Acts**

**Act 1: Text Editor**

*In IDE:*

# hello.py

def hello\_world():

print "hello, world!"

hello\_world()

*In iTerm2:*

$ ipython hello.py

hello, world!

**Act 2: REPL**

*In iTerm2:*

$ ipython

Python 2.7.6 (default, Apr 9 2014, 11:48:52)

Type "copyright", "credits" or "license" for more information.

IPython 2.0.0 -- An enhanced Interactive Python.

? -> Introduction and overview of IPython's features.

%quickref -> Quick reference.

help -> Python's own help system.

object? -> Details about 'object', use 'object??' for extra details.

In [1]: def hello\_world():

...: print "hello, world!"

...:

In [2]: hello\_world()

hello, world!

Even when developing in a text editor, it's important to keep a tight feedback loop, which means running your code frequently. **We'll see how to make programming in a text editor as interactive as possible later in this document.**

**Other Tools**

* git

git is used for version control. It will be a critical part of your development workflow. Being able to rollback changes and create branches will enable you to be more confident in developing programs because you can change code and try new approaches without worrying about losing a currently working version. git also enables collaboration with other developers.

* GitHub

To make it perfectly clear, git is independent of GitHub. Each copy of a git repository is independent of all the others (hence, *distributed* version control system, or dvcs). GitHub is just a place to put a copy of a repository; the benefit is that because GitHub is web-based, anybody can access that repository at anytime, which makes it an ideal place to host a master version of shared repositories. Many companies use GitHub in just that way.

Here at Galvanize, we have our curriculum in git repositories hosted on GitHub. You'll be viewing, cloning, and forking those repositories quite a bit.

* Anaconda Python and Packages

We use the Anaconda scientific python stack which is just a vanilla version of Python 2.7 along with all the packages that a data scientist would need, including **NumPy**, **SciPy**, **SciKit-Learn**, **Pandas**, and **matplotlib**. Anaconda manages the Python environment for us. If you need to install other Python packages (unlikely), do so with the condacommand-line utility (i.e. conda install some-cool-package). Use conda list to see what's installed.

* Google Chrome

Our browser of choice here Galvanize. Use it.

* Homebrew

Homebrew is a Mac package utility. To install a package: brew install package-name. To see which packages are installed: brew list. **DO NOT INSTALL MACPORTS ON THE WORKSTATIONS.**

**Keyboard Shortcuts**

You should make a concerted effort to use the mouse as little as possible while you are developing.

*Note:****META****=****CMD***

**System Shortcuts**

* To open an application, use Spotlight: **META** + **SPACE**
* To switch between applications: **META** + **TAB**
* To switch between open windows of a single application: **META** + **`**
* To quit an application: **META** + **q**
* To close a window of an application: **META** + **W**

**Atom Shortcuts**

* To open Atom from the command line: $ atom file-or-directory
* To open a new file: **META** + **n**
* To close a tab: **META** + **w**
* To save a file: **META** + **s**

**Terminal/iTerm2 Shortcuts**

**Window Management**

* To open a new window: **META** + **n**
* To open a new tab: **META** + **t**
* To move left and right between tabs: **META** + **LEFT ARROW** / **RIGHT ARROW**
* To split a pane vertically: **META** + **D**
* To split a pane horizontally: **SHIFT** + **META** + **D**
* To move between panes: **META** + **[** / **]** (left or right bracket)
* To close a split pane or tab: **META** + **w**
* To clear the terminal screen: **META** + **k**

**Command Line Basic Commands**

* ls: list files in current directory
* cd directory: change directories to directory
* cd ..: navigate up one directory
* mkdir new-dir: create a directory called new-dir
* rm some-file: remove some-file
* man some-cmd: pull up the manual for some-cmd
* pwd: find the path of the current directory
* mv path/to/file new/path/to/file: move a file or directory (also used for renaming)
* find . -name blah: find files in the current directory (and children) that have blah in their name

**Command Line Navigation**

* To jump to beginning of line: **CTRL** + **a**
* To jump to end of line: **CTRL** + **e**
* To cycle through previous commands: **UP ARROW** / **DOWN ARROW**

**Interactive Development Workflow**

**Having a tight feedback loop between writing and testing/playing with code ensures that you build your programs incrementally and efficiently.**

The ideal workflow is to write a little bit of code, then ensure that the code is doing what you expect by inspecting some output or playing with it in an interactive environment. Plus, having a tight feedback loop is more fun.

You will most often be writing code in an IDE and playing with it in iPython (in iTerm2).

*IDE:*

# hello.py

def hello\_world():

print "hello, world!"

def add\_em\_up(a, b, c):

return a + b + c

Now you'd like to test out some of the code. The most straightforward way of doing so would be to insert some printstatements into your file and run the file in the terminal.

*IDE:*

# hello.py

def hello\_world():

print "hello, world!"

def add\_em\_up(a, b, c):

return a + b + c

if \_\_name\_\_ == "\_\_main\_\_":

hello\_world()

print add\_em\_up(3, 4, 5)

*iTerm2:*

$ ipython hello.py

hello, world!

12

As you continue to add to and modify your code, you'd rerun the file in the terminal each time to see the output of your print statments. That's fine, but there are better, more interactive ways.

*Note on if \_\_name\_\_ == "\_\_main\_\_":: This basically says, only run this code if the file is being directly run from the command line, as opposed to being imported as a module. If it's being imported, we just want the function and class definitions to be available; we don't want to run anything. This guard ensures that that's the case. Anytime you're writing code on the top level (i.e. outside of a function or class definition), it should be within this guard.*

**Modules and Autoreload**

You can instead import the file as a module in iPython, and as you make modifications to the file, iPython will automagically reload the module (This is a setting that has been enabled on all the Galvanize workstations). Let's take a look.

*IDE:*

# hello.py

def hello\_world():

print "hello, world!"

def add\_em\_up(a, b, c):

return a + b + c

if \_\_name\_\_ == "\_\_main\_\_":

hello\_world()

print add\_em\_up(3, 4, 5)

*iTerm2:*

$ ls

hello.py

$ ipython

In [1]: import hello as lib

# Notice how none of the print statments happened thanks to the

# if \_\_name\_\_ == "\_\_main\_\_" guard.

In [2]: lib.hello\_world()

hello, world!

In [3]: lib.add\_em\_up(3, 4, 5)

Out[3]: 12

Note that you can import your files into iPython as modules. Here, hello.py was imported and aliased as lib. All the functions and classes defined in that file are available in the imported module.

**When developing using the module pattern, it's important to write all your code in functions and classes.** Don't just have code hanging out on the top level unless it's a very short script (and it should always have the \_\_name\_\_ guard).

Now let's see how autoreload makes our life easier.

*IDE:*

# hello.py

def hello\_world():

print "hello, cruel world!"

def add\_em\_up(a, b, c):

return a + b + c

def power\_up(b, e):

return b \*\* e

if \_\_name\_\_ == "\_\_main\_\_":

hello\_world()

print add\_em\_up(3, 4, 5)

Here we made two changes: we modified hello\_world and we added a method power\_up.

*iTerm2*

# A continuation of the above iPython session

In [4]: lib.hello\_world()

hello, cruel world!

In [5]: lib.power\_up(5, 2)

Out[5]: 25

The file has been automatically reloaded for us! We can interact with all the functions and classes interactively without any fuss. And no need to hop back and forth adding additional print statments and whatnot; we can just go ahead and play with all of our code interactively.

*Note on global imports: You'll sometimes see a global import from somelib import \*. This is very bad practice. It pollutes the global namespace (i.e. all the variable names declared on the top level) and it also won't work withautoreload in iPython, so don't do it.*

To add autoreload functionality, feel free to inspect this [document](https://gist.github.com/rsepassi/2cdde6c6d4b36916cb37) and copy it to your own machine in the filepath ~/.ipython/profile\_default/startup/autoreload\_startup.ipy

**Interactive Debugging**

Sometimes you'd like to drop into your code on a specific line and explore what's going on. The Python interactive debugger allows you to do just that and more.

An interactive debugger allows you to step through your code line by line and inspect the local scope and the value of variables. Here's how it's used:

*IDE:*

# hello.py

import ipdb # The interactive Python debugger

def hello\_world():

print "hello, cruel world!"

def add\_em\_up(a, b, c):

return a + b + c

def power\_up(b, e):

return b \*\* e

if \_\_name\_\_ == "\_\_main\_\_":

hello\_world()

a = 22

ipdb.set\_trace()

b = 33

print add\_em\_up(3, 4, 5)

*iTerm2:*

$ ipython hello.py

hello, cruel world!

> ~/hello.py(14)<module>()

13 ipdb.set\_trace()

---> 14 b = 33

15 print add\_em\_up(3, 4, 5)

ipdb> print a

22

ipdb> print b

\*\*\* NameError: name 'b' is not defined

ipdb>

Note that ipdb.set\_trace() opens up an interactive debugger just after it is called; the code is paused right at that line. ais defined and has the value 22. b is not defined yet since we have yet to evaluate this line and so we get an error. To go to the next line we use n.

*iTerm2:*

# continued from above

ipdb> print b

\*\*\* NameError: name 'b' is not defined

ipdb> n

> /Users/Ryan/Dropbox/DataScience/Zipfian/dsr/assessment-day1/code/hello.py(15)<module>()

14 b = 33

---> 15 print add\_em\_up(3, 4, 5)

16

ipdb> print b

33

ipdb>

**Debugger Commands**

* n: next line
* c: continue to end (or next breakpoint)
* s: step into function call
* b 25: set a breakpoint at line 25
* print a: print the value of a
* list: see where you are

**Workflow**

**Keep the feedback loop tight.**

1. Create a file
2. Import the file as a module into iPython (autoreload takes care of the rest)
3. Write some code
4. Play with the code in iPython
5. Write some more code
6. Use ipdb (interactive debugger) as necessary
7. Repeat until done

**Version Control with git**

If you remember one thing and one thing only about version control it should be this: **always be committing**.

Each commit is taking a snapshot of your work so far which enables you to go back in time to older versions of your program. You will most certainly find yourself in a situation where you had some working code, modified it to add a new feature or work out some kink, only to find that you've hopelessly ruined everything and would give your left index finger just to get back to what you had before. Enter git.

**Key concepts**

* Repository (a folder managed by git)
* Workspace (current state)
* Index (staged for commit)
* Commit (take a snapshot)
* Branch (a series of commits)
* Remote (a remote repository that you can push to or pull from)

Any folder can be turned into a git repository with git init. Your **workspace** is the current state of all your files. Some of them will be different from what was last committed. You can see what's different by running git status. From your workspace, you can use the git add command to add files to the index, which is a sort of staging area for commits. When you run git commit, the files in your index are included in the commit snapshot. You can use git reset to roll back to prior commits and you can use git log to see the history of commits.

Here's a [visual cheatsheet](http://ndpsoftware.com/git-cheatsheet.html#loc=workspace) that covers all this and more.

**Key commands**

* git status: see the status of the workspace, index, and what branch you're on
* git add: add files to the index (commit staging area)
* git commit: take a snapshot of the project, committing the files in the index
* git checkout: switch to a different branch (use the -b option to switch to a new branch)
* git branch: list the branches
* git reset: rollback to a previous commit
* git push: push up the changes in a local repository to a remote repository
* git pull: pull down the changes from a remote repository to the local repository
* git clone: copy a remote repository to the local machine

**git Workflow**

1. Choose a feature/segment/thing to work on next
2. Write some code
3. Play with the code
4. Rewrite, play some more, etc.
5. git add .: add all your changes to the index
6. git commit -m "Describe the work you just did"
7. Repeat

**DO NOT commit large files to a Github repo (anything larger than ~20mb). In case you have accidentally committed a large file (or dataset) use this**[**tutorial**](http://blog.jessitron.com/2013/08/finding-and-removing-large-files-in-git.html)**or this**[**commandline tool**](http://rtyley.github.io/bfg-repo-cleaner/)**to clean up your repo**

**Recap**

1. Know the tools. Use the tools.

* iTerm2
* IDE - Integrated Development Environment (Atom)
* iPython

1. Use the keyboard. Don't use the mouse. Know your shortcuts.
2. Keep a tight feedback loop when writing code.

* Write code in IDE
* Import file into iPython
* Write, run, repeat

1. Use git. Always be committing (ABC).