

# Introduction to Data Science Immersive (DSI)

Benjamin S. Skrainka

February 8, 2016

# Today's schedule

Time	Activity
8:00 am	Breakfast
8:30 am	Welcome to Galvanize & introductions
9:00 am	Galvanize community & tour
9:30 am	Assessment
11:30 am	Git & first pull request
12:00 pm	Lunch + pro-tips for success
1:15 pm	Lecture: Python fundamentals
2:15 pm	Lecture: Introduction to Pair programming
2:30 pm	Programming exercise 1
5:00 pm-ish	End of day 1

→ We will take your portrait later in the week

# Welcome to Galvanize

# Instructor & DSR introductions

## Instructors:

- Benjamin S. Skrainka: [skrainka@galvanize.com](mailto:skrainka@galvanize.com)
- Brian Mann: [brian.mann@galvanize.com](mailto:brian.mann@galvanize.com)
- Jack Bennetto: [jack.bennetto@galvanize.com](mailto:jack.bennetto@galvanize.com)
- Ming Huang: [ming.huang@galvanize.com](mailto:ming.huang@galvanize.com)

## DSRs:

- Miles Erickson: [miles.erickson@galvanize.com](mailto:miles.erickson@galvanize.com)
- Tetyana Kutsenko: [tetyana.kutsenko@galvanize.com](mailto:tetyana.kutsenko@galvanize.com)

# Typical daily schedule

On most days, class will be a mix of lecture and exercises:

Time	Activity
8:30 am	Mini-quiz (survey on Fridays)
9:00 am	Lecture
10:00 am	Morning exercise (individual)
12:00 pm	Lunch
1:15 pm	Lecture
2:15 pm	Afternoon exercise (pair)
5 pm-ish	End of day

# Logistics

DSI course uses `git` and **GitHub** to manage class materials and exercises.

- `git` is one of the best version control tools
- Fork repo(s) for each lecture and clone to your local machine
- Start with the **Welcome** repository: **view SEA-Feb-2016-Cohort-3 branch**
- Note: for legacy reasons the repo is owned by Zipfian

```
$ git clone https://github.com/joe_student/welcome.git
```

or

```
$ git clone git@github.com:joe_student/welcome.git
```

# Assessment

Assessment consists of:

- Mini quizzes (most mornings)
- Individual exercise – usually in `individual.md`
- Pair programming exercise – usually in `pair.md`
- Roughly weekly assessment – usually in `assessment.md`

Regular feedback helps you track your progress and understanding of the material so that you can focus on the skills you need to improve

# Code of conduct



# Campus tour

# Assessment 1

# Tips for Assessments:

Work smart:

- There is more work than you can do
- Use standard, smart exam taking techniques
- Read the questions first
- Then, solve the easy questions first
- Test your work with the unit tests we have provided

# Fork your repo first!

Fork your repo in GitHub before you start:

- Fork repo on GitHub
- Then, clone locally onto your laptop

# Using make to run unit tests

Use make to test your work:

- Will run `py.test` which exercises unit tests
- Output indicates whether your code passed the test ... or failed:
  - ▶ . → test passes
  - ▶ E → test has an error
  - ▶ F → test fails

## Example: interpreting `py.test`

```
$ make
```

```
py.test test/unittests.py
```

```
===== test session starts =====
```

```
platform darwin -- Python 2.7.11, pytest-2.8.1, py-1.4.30, plu
```

```
rootdir: /Users/bss/sbox/ds_class/class/week1/assessment-day1/
```

```
collected 12 items
```

```
test/unittests.py ....EFFFFFFF
```

Results mean:

- Tests 1-4 passed
- Test 5 has an error
- Tests 6-11 failed

# Submit a pull request

When you are finished, submit a pull request:

- Makes your work available for marking
- If you don't know how to do this, don't panic . . .
- . . . We will introduce git after the assessment

## Lighting fast review of git



# What is git?

Git is one of the best version control tools:

- Collaborate with many users around the Internet
- Keep changes synchronized across machines
- Can manage source, documentation, text files, and more
- A safety net:
  - ▶ Roll back mistakes
  - ▶ Explore new ideas safely (in a branch)
- Backup work!
- Increasingly, the standard tool for version control
- Reproducible research
- Unprofessional not to use `git`...

# Git features

Features that make git awesome:

- SHA-1 hash:
  - ▶ Nigh uniquely identifies a commit across all copies of repo
  - ▶ Provides security from tampering
- Blazingly fast merges and diffs because git works on entire file and not incremental diffs
- Relatively painless merges
- Peer to peer:
  - ▶ Can work without Internet
  - ▶ Distributed
  - ▶ Fault tolerant

# References

Two great references:

- [Pro Git](#)
- [git cheatsheet](#)

Plus, old-school help:

```
$ man git
```

```
$ git clone --help
```

# Components of git

In order to understand git, you must understand how information flows between git's components:

- *Workspace*: current version of your work
- *Stage*:
  - ▶ Temporary container for work before committing to repository
  - ▶ Also known as the *Index* and *Cache*
- *Local repository*:
  - ▶ Local storage of all versions of your work which has been committed
  - ▶ Stored in `.git` directory in root of repo
- *Upstream repository*: remote copy, possibly out of sync with local repo
- *Stash*: 'park' work here when randomized by your boss

See [git cheatsheet](#)

# Install git

Check if git is installed:

```
$ which git  
/usr/local/bin/git
```

If not, install **Homebrew** package manager and git:

```
$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew  
$ brew install git
```

# Create an account on GitHub

Setup up an account on GitHub:

- To access course materials
- To submit your work
- To collaborate with students and colleagues
- To coordinate access to your repos on multiple machines
- To host your portfolio
- To backup your work

# Configure git

Configure ~/.gitconfig and ~/.gitignore\_global for your preferences. At a minimum, set the following:

```
$ git config --global user.name "Eddy Merckx"  
$ git config --global user.email eddy@merckx.com  
$ git config --global core.editor vim  
$ git config --global core.excludesfile ~/.gitignore_global
```

You might prefer a friendlier editor, such as [Atom](#), nano, or pico

**ONLY DO THIS ON YOUR OWN MACHINE**

# git clone *repo*

Create a local copy of the upstream repo:

```
$ git clone https://github.com/joe_student/awesome.git
```

or

```
$ git clone git@github.com:joe_student/awesome.git
```



# Git workflow

Stage changes as you work and commit them to the local repo when you are ready:

```
$ atom genius.py  # Implement your great ideas
$ git status      # Check state of workspace and cache
$ git add genius.py  # Stage genius.py to commit to repo
$ git status
$ git commit -m 'Implement orbital mind control laser'
```

# Caveats

Do not put the following under version control:

- Large files: data, images, binary, .doc, .xls, .pdf, ...
- Derived files: i.e., files built from source code, markdown,  $\text{\LaTeX}$ , ...

Commit message should:

- Use imperative mood to state why you made the change
- Reference JIRA item or bug
- No need to say what you did
- Be nice to your future self!

# Examining changes

You have several tools to compare versions:

- Use `tig` (Install via `brew install tig`)
- `git diff 16d6758 HEAD^^^`
- `git diff master`
- `git diff --stat`
- `git log`: see manual for details
- `git blame fubar.py` to determine who broke `fubar.py`

# Syncing with an upstream repo (1/2)

To copy upstream changes to a local repo:

- Use `git pull`:
  - 1 Performs `git fetch` to create a local copy
  - 2 Attempts to merge changes via `git merge`
- Will require manual intervention if there is a merge conflict
- Can optionally specify the repo and branch to pull

## Syncing with an upstream repo (2/2)

Use `git push` to copy local changes to an upstream repository:

- Must merge upstream changes before pushing your changes!
- May need to set an upstream tracking branch

```
$ git remote -v
origin  git@github.com:zipfian/bss.git (fetch)
origin  git@github.com:zipfian/bss.git (push)
$ git push
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 7.06 KiB | 0 bytes/s, done.
Total 15 (delta 3), reused 0 (delta 0)
To git@github.com:zipfian/bss.git
    16d6758..aac3438  master -> master
```

# Submitting work via git

Typical class workflow:

- 1 Fork repo for lecture and exercise in GitHub
- 2 Clone your version of the repo onto local machine:

```
$ git clone https://github.com/joe/python-intro.git
```

- 3 Commit your work (do this often)

```
$ git add file1.py file2.py ...
```

```
$ git commit -m 'Implement XYZ'
```

```
$ git push origin master
```

- 4 In GitHub,
  - 1 Click **Pull Requests**
  - 2 Click **New pull request**

# Advanced commands

Some helpful commands:

- `git stash`: save changes in workspace so you can work on something else
- `git remote`: create and examine *remotes* to access upstream repositories
- `git init`: create a local repository in current directory
- `git reset`: rollback to previous commit
- `git revert`: revert commit(s)
- `git checkout`:
  - ▶ Revert file to version in local repo
  - ▶ Switch to a branch (create it if necessary)
- `git branch`: manipulate branches
- `git rebase`:
  - ▶ Replay changes from one branch on another
  - ▶ Smash small commits into a single commit

To improve your workflow:

- Use ssh and not https
- Use tig
- Work in a branch
- Rebase your commits before requesting a code review
- Commit and push regularly so that you can roll back changes if you make a mistake
- Never put keys in your repo; e.g., keys for AWS
- Be paranoid: enable dual-factor authentication
- Keep all repositories in ~/repos or equivalent



Lunch

# Introduction to Data Science Immersive (DSI)

# Objectives

Today's lecture focuses on:

- Setup your computer to do data science
- Install/configure core tools
- Adopt a productive workflow for programming
- Review python Essentials
  - ▶ How to write efficient code
  - ▶ How to ensure your code is correct

Will discuss OOP tomorrow...

# Agenda

- ➊ Data science software stack
  - ➊ Overview of programming workflow
  - ➋ Introduction to core tools
- ➋ Essential Python
  - ➊ Writing efficient code
  - ➋  $O(n)$  solution to anagrams problem
  - ➌ Looping efficiently
  - ➍ Using sets & dictionaries

# Data Science Software Stack

# Overview of key tools

We focus on the key tools you need to work on data science projects:

- `ipython`
- `iTerm2`
- Atom or other suitable programming editor (`vi`, `emacs`, Sublime Text)
- Unix

Plus, an overview of typical workflow

# Why Python for data science?

Python is as close to an all purpose language as you can find:

- Strong for programming and integration tasks
- Interpreted  $\Rightarrow$  easy to prototype and explore
- Supports most key data science technologies
- Excellent machine learning syntax
- Simple syntax

But, R often has superior statistics packages . . .

# References

A couple helpful references on Python and the craft of writing software:

- [A Practical Guide to Commands, Editors, and Shell Programming](#)
- [Effective Computation in Physics](#)
- [Python for Data Analysis](#)
- [“Effective Python”](#)
- Skrainka’s [lecture](#) on writing software from [ICE](#)
- [“The Practice of Programming”](#)
- [“Code Complete”](#)



# Installation

To install the data science stack used at Galvanize:

- Do nothing, if on a lab machine
- On your laptop, run commands in `setup.sh`:
  - ▶ Contained in *Welcome* repo  
(`git@github.com:zipfian/welcome.git`)
  - ▶ Run commands by hand because many may time-out and/or require baby-sitting
- Installation uses **Homebrew**, a package manager for OS/X
  - ▶ Brew is incompatible with **MacPorts**
  - ▶ On Linux, use the package manager, e.g., `apt-get` on Ubuntu

Unix's *filters + pipes* paradigm provides a productive platform for doing data science:

- Each command does one thing (only) well
- Commands read from `STD_IN` and write to `STD_OUT`
- Can combine commands via I/O redirection with `|`, `<`, and `>`.
- Easy to write scripts to build bespoke commands from Unix's building blocks

# Navigating in Unix

Basic survival commands for Unix:

Command	Action
<code>pwd</code>	Display current directory
<code>mkdir</code>	Create a directory (folder)
<code>ls</code>	Display contents of folder
<code>cd</code>	Change directory
<code>file</code>	Display file type
<code>man</code>	Get help (manual)

# Essential commands

Some additional essential commands:

Command	Action
<code>ack</code>	Recursively search files in a tree for an RE pattern
<code>grep</code>	Original tool for searching for a pattern in a file
<code>which</code>	Determine which version of command is accessible
<code>less</code>	Page through a file
<code>wc</code>	Count lines, words, and characters
<code>head</code>	Show initial lines of a file

# On OS/X, use iTerm2

iTerm2 is a better version of the Terminal app:

- Provides command line access to bash or other shell
- Decreased eye strain
- Better support for multiple panes and hotkeys
- Improved search facility with highlighting
- Mouse-less copy
- Paste history
- More configurable

```
$ brew cask install iterm
```

# Overview of tactical workflow

To be productive, you want a tight feedback loop:

- Write some code
- Run it/test it to get quick feedback
  - ▶ Easier to fix a bug if you catch it early
  - ▶ Maintenance is the primary cost of building software
- Start small, and get the simplest case to work first
- If working with data, get a shard (subset) to work first
  - ▶ Will provide quick feedback that your code works
  - ▶ Build a shitty first model, then refine
  - ▶ Do not start with an Exabyte of data!

# Unit tests

Unit test everything you can:

- Fastest way to develop code
- Highest quality way to develop code
- Unit test framework will setup and teardown fixtures
- Use Red/Green/Green:
  - ▶ Red: write unit test first; check that it fails
  - ▶ Green: implement code which passes unit test
  - ▶ Green: refactor code to work better
- You can unit test scientific code!

# Become a better programmer

Invest in becoming a better programmer:

- Increased productivity
- Ability to solve bigger and harder problems
- Master the craft as well as grammar:
  - ▶ Write more code
  - ▶ Get code reviews
  - ▶ Read well-written code
- Try to use the best tool for the job:
  - ▶ Learn more languages
  - ▶ R, Scala, Julia, Java, MATLAB, Mathematica, Clojure,...



# Working in Python

Depending what you are doing you will use one of the following workflows, often in conjunction with *Test Driven Development* (TDD):

- Editor + `python` or `ipython`
- `ipython` notebook

TDD emphasizes working with unit tests to ensure a tight feedback loop and avoid the debugger (see below)

# Invest in learning an editor

Master a programming editor, it will save you immense amounts of time:

- Color-syntax highlighting and line numbers
- Can configure PEP8 syntax checking to catch mistakes early (when cheaper to fix)
- Automates much of writing code
- Fancy search and text manipulation
- Quickly navigate a large code base
- Use **Atom** unless you have already mastered vi(m) or emacs

# Working in `ipython`

`ipython` makes Python more friendly for data science:

- `%magic` methods plus `ls`, `cd`, `pwd`, etc. to navigate file-system
- Tab completion
- Enhanced help and introspection via `?`
- Keyboard shortcuts
- Convenience variables:
  - ▶ Access past results with `_`, `__`, `_72`, etc.
  - ▶ Access past inputs with `_i` and number of command
  - ▶ Can re-execute command with `exec _i99`
- History
- Integrated plotting

# Working in `ipython` notebook

But, `ipython` notebook is even better:

- Provides notebook to document work as well as run python commands
  - ▶ Similar to Mathematica ...
  - ▶ Promotes reproducible research by unifying code, documentation, and results
  - ▶ Inline plotting
- Use for exploratory data analysis (EDA) and prototyping
- Do not become sloppy:
  - ▶ Write functions, classes, modules, etc. for production code
  - ▶ ... or to reuse common code
  - ▶ Prefer DRY to WET coding
- Can also run on remote machine and connect via ssh tunnel
- Covered in Thursday's lecture

# Python Essentials

# Review of Python Essentials

Objectives for the rest of lecture:

- ➊ Writing efficient code
- ➋  $O(n)$  solution to anagrams problem
- ➌ Looping efficiently
- ➍ Using sets & dictionaries

# Basic software design

A few basic, old school tips:

- Good programmers are lazy
- Start by listing requirements and/or writing a specification
- Simplify complex problems by 'divide and conquer'
- Decompose problem into functions:
  - ▶ An interface is a contract
  - ▶ Each function does one thing only and does it well
  - ▶ If your code spans a screen or two, break it up!
- Structured programming / Top down / Bottom up / OOP

# More basic design

Algorithms and data structures complement each other:

- Interfaces are a contract
- Choose the right data structure for the job:
  - ▶ E.g., what is the correct container (list, dict) to use?
- Reuse code via libraries wherever possible:
  - ▶ Code is already debugged
  - ▶ Interfaces are battle-tested
- Does it make sense to sort?
- For only a few elements, brute force is fastest
- Fail early



# DRY vs. WET

Be DRY not WET:

- DRY means 'Do not Repeat Yourself'
- WET means 'We Enjoy Typing'

If you are writing code with cut & paste, you really need to write a function:

- Promotes reuse
- Decreases bugs
- Improves maintainability

Don't let `ipython` notebook turn you into a sloppy programmer!

# Scope: LEGB

Questions of scope are resolved using LEGB:

- 1 Locals
- 2 Enclosing
- 3 Globals
- 4 Built-ins

# Deep vs. shallow copy

Python passes object references by value:

- Like passing a pointer to an object in C:  
`status = serialize_data(&data) ;`
- Python uses object reference to refer to an object
  - ▶ Saves memory
  - ▶ Can accidentally modify objects because default copy is shallow
- deep-copy objects when you want a copy of the object and any objects it contains
- 
- Can see object's ID with `id(obj)`

See: [Rob Heaton's post](#)

## Example: deep vs. shallow copy

```
In [1]: x = [ [1, 2, 3], [-4, -5, -6]]
```

```
In [2]: [id(ix) for ix in x]
```

```
Out[2]: [4419005976, 4419004536]
```

```
In [3]: y = x
```

```
In [4]: [id(ix) for ix in y]
```

```
Out[4]: [4419005976, 4419004536]
```

```
In [5]: y[1][0] = 'four'
```

```
In [6]: y
```

```
Out[6]: [[1, 2, 3], ['four', -5, -6]]
```

```
In [7]: x
```

```
Out[7]: [[1, 2, 3], ['four', -5, -6]]
```

# Mutable vs. immutable

Data type can be:

- Mutable:

- ▶ list, set, dict
- ▶ Can be modified after creation

- Immutable:

- ▶ str, int, float, tuple
- ▶ Cannot be modified after creation

# Looping efficiently

To write faster loops:

- Prefer `for word in words:` to direct indexing with `for ix in xrange(len(words)):`
- Prefer `xrange` to `range`
- Prefer `enumerate()` to `xrange()` when you need the index and the value
- Prefer `for k, v in my_dict.iteritems():` to `for k in my_dict:` when you need both the key and value
- Prefer `itertools.izip()` to indexing over multiple lists
- Prefer `itertools.izip()` to `zip()`

# Sets & Dictionaries

Sets and dictionaries provide fast look-up, insertion, and deletion:

- Prefer `for k, v in my_dict.iteritems():` to `for k in my_dict.keys():` if you need both the key and value
- Check membership with `in`
- Prefer `Counter` and `defaultdict` from `collections`
  - ▶ If order matters, prefer `OrderedDict` or `OrderedSet`
- Implemented as hash table so should have roughly  $O(1)$  access speed
- Note: dictionary keys and set items must be immutable

# Solving anagrams in $O(n)$

Recall the interview question: *write a function to return all the words which have anagrams in a list of words*

- Q: what is the algorithmic cost of the double for loop solution?
- Q: can you think of an algorithm which is  $O(n)$ ?
  - ▶ Take a few minutes to ponder and discuss with a partner
  - ▶ Hint: what data structure has fast, random access to elements?
  - ▶ Hint: what does defaultdict do?
  - ▶ Hint: how can you transform each word so that it is easy to find an anagram?



# Anagram timing

How much better is the  $O(n)$  solution?

```
In [18]: %timeit anagrams.anagrams1(anagrams.words)
10000 loops, best of 3: 46.2 µs per loop
```

```
In [19]: %timeit anagrams.anagrams2(anagrams.words)
100000 loops, best of 3: 10.4 µs per loop
```

# List comprehensions (1/2)

List comprehensions provide a compact way to create a list similar to a for loop

- Enclose the for expression in []
- Returns a list with elements:

```
squares = [val**2 for val in xrange(10)]
```

- List comprehension is equivalent to:

```
squares = []  
for val in xrange(10):  
    squares.append(val**2)
```

# List comprehensions (1/2)

List comprehensions are flexible and can be customized:

- Nested for loops:

```
L = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]  
flat_list = [item for row in L for item in row]
```

- Add a condition with if:

```
even_squares = [  
    val**2 for val in xrange(10) if 0 == val % 2]
```

# Generators

Generators are a superior form of an 'iterable' which provide values on demand:

- Decrease memory usage via lazy evaluation
- Often used in a loop, e.g., `xrange()`
- Increase performance with large data because they generate the next value on each iteration:
  - ▶ `range(100000)` will create the entire list of values at once
  - ▶ `xrange(100000)` will create the next value on each iteration
- Prefer generators over classic iterables
- Can create via generator comprehension or with `yield`

See: [David Beazley's talk](#)

# Generator comprehensions

Generator comprehensions resemble list comprehensions and provide a quick way to create an iterator:

- Replace the `[]` with `()` to create a generator
- Will only produce the next value on demand, saving memory
- No need to worry about implementation details such as `.next()`, `.__iter__()`, and `yield`
- But, generator is consumed when you use it, so one use only
- Benefits:
  - ▶ Faster
  - ▶ Uses less memory
  - ▶ Requires less code

# Generator example (From David Beazley)

```
wwwlog = open("my_web_log")
bytecol = (line.rsplit(None,1)[1] for line in wwwlog)
bytes    = (int(x) for x in bytecol if x != '-')
print "Total:", sum(bytes)
```

# Lambda

Provide a handy way to write simple, anonymous functions:

- Useful for simple reductions and aggregations
- Syntax:
  - ▶ `lambda <args>: <body>`
  - ▶ Can have multiple arguments
  - ▶ Do not include a return statement
  - ▶ Should be simple, otherwise write a function
- Lambdas are often used with `map()` and `reduce()`
- Example:

```
df = pd.read_csv('awesome_data.csv')  
df.apply(lambda x: x - x.mean())
```

- Prefer `operator.itemgetter()` for extracting an element from a list/tuple to `lambda + indexing`

# Using exceptions

Python (and your code) raises an exception to signal an exceptional event such as an error:

- Only catch errors you must handle, such as file I/O problems
- Let the error unwind the call stack for everything else:
  - ▶ Easier to read code
  - ▶ Forces user to fix exceptional condition
  - ▶ Improves performance of code
- Keep try blocks small
- Catch only the specific errors you need to handle



# Context management with with

Context managers ensure that you cleanly allocate and deallocate a resource:

- Python handles setup and tear-down behind the scenes by calling object's `__enter__()` and `__exit__()` methods
- Ensures cleanup on an exception
- Often used with file I/O, or when obtaining a lock
- Example:

```
with open('logfile.txt', 'r') as f:  
    lines = f.readlines()
```

PEP8 is the official recommendation for best Python practices:

- Always check your code with `pep8` to help catch bugs early
- Can run from command line
- Better to configure editor to automatically check code
  - ▶ For vim, install `Python-mode`
  - ▶ For Sublime, install *Package Control*, *PEP8 AutoFormat*, and *SublimeLinter*
  - ▶ For Atom, PEP8 support is already installed

# Conclusion

Checks for mastery:

- What is test driven development?
- What does Red Green Green mean?
- Why is DRY better than WET?
- How can collections improve your code?
- When should you use a generator or list comprehension?
- What is LEGB?