Next Steps in Learning and Working With Python

Python 3

home (../handouts.html)

Entering the wider world

This section is for gaining a broader perspective of Python, the coder's job and responsibilities, and the industry.

- Being "Pythonic": Python's coding philosophy
- Proper program form and conventions
- Docstrings and documentation
- Github and git

Pythonic coding: The Zen of Python

One of the advantages of Python is its emphasis on design elegance. The non-backwards-compatible **Python 3** is a clear indication that doing things "the right way" (however it may be defined) is more important than popularity.

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one -- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

PEP8: the Python design standard

A **PEP** or **Python Enhancement Proposal** is a series of numbered discussion documents that propose new features to be added to Python. A significant discussion over whether or not an addition is useful, meaningful, worthwhile, and "Pythonic" accompanies each proposal. Proposals are accepted or rejected by common consensus (with special weight given to the opinion of **Guido van Rossum**, the proclaimed **Benevolent Dictator for Life**).

PEP8 (https://www.python.org/dev/peps/pep-0008/) is often cited in online discussions as the standard guide for Python style and design. It treats both specific style decisions (such as whether to include a space before an operator; the case and format of variable names; whether to include one blank line or two after a function definition) as well as design considerations (how to design for inheritance, when/how to write return statements, etc.).

While the proposal itself proclaims "A Foolish Consistency is the Hobgoblin of Little Minds" it is generally accepted as the standard to follow.

"A universal convention supplies all of maintainability, clarity, consistency, and a foundation for good programming habits too. What it doesn't do is insist that you follow it against your will. That's Python!"

-- Tim Peters on comp.lang.python, 2001-06-16

important conventions in **PEP8**

- 4-space indents
- maximum line length: 79 characters

- wrapping long lines with parentheses, and using "implicit string concatenation", for example:

```
var = ( 'this is a string I want to tell you about, '
        'but it is so long I figured I would split it up.' )
```

- 2 blank lines before and after functions
- imports on separate lines, 'absolute' imports recommended
- single space on either sides of operators
- naming style is not recommended, but general convention is lowercase_with_underscores

Adhering to the 79-character limit

The original CRT terminal screens had an 80-character width; this is how the 79-character limit was born (wrapped lines are difficult to read).

Many text editors and IDEs can be set to display the 80-character limit onscreen and/or warn the user when the limit exceeded.

Even though modern screens allow for pretty much any length (since you can size the text to any size), the 79-character limit is still considered an important discipline, since it will allow multiple files to be displayed side-by-side.

When attempting to adhere to this limit, we must learn how to wrap our lines. Wrapping long lines is usually done with parentheses, for example:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

I have found that my preference for explicit variable names makes respecting this limit to make life more difficult.

Some developers hold an opinion (acknowledged by PEP8) that the 79-character limit is antiquated (since it was based in the old terminal width) and that another limit, up to 100 characters, should be adopted. PEP8 acknowledges that with agreement among developers, it should be permissible to extend the limit. You can see some of what is said regarding the debate in this spirited discussion (http://stackoverflow.com/questions/4841226/how-do-i-keep-python-code-under-80-chars-without-making-it-ugly).

PEP257: docstrings

PEP257 (https://www.python.org/dev/peps/pep-0257/) defines the overall spec for docstrings.

If you violate these conventions, the worst you'll get is some dirty looks. But some software (such as the Docutils docstring processing system) will be aware of the conventions, so following them will get you the best results.

-- David Goodger, Guido van Rossum

The actual format is discretionary; below I provide a basic format.

A **docstring** is a string that appears, *standalone*, as the first string in one of the following:

- a script/module
- a class
- a function or method

The presence of this string *as the very first statement* in the module/class/function/method populates the __doc__ attribute of that object.

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

"""myprog.py:  show a sample of docstrings"""

def myfunc():
    """this is myfunc"""

print(__doc__)               # 'myprog.py:  show a sample of docstrings'
print(myfunc.__doc__)        # 'this is myfunc'
```

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 also have docstrings. A package may be documented in the module docstring of the __init__.py file in the package directory.

It is also allowable to add a docstring after each variable assignment inside a module, function or class. Everything except for the format of the initial docstring (which came from a module I wrote for AppNexus) is an example from PEP257:

```
"""
NAME
    populate_account_tables.py

DESCRIPTION
    select data from SalesForce, vertica and mysql
    make selected changes
    insert into fiba mysql tables

SYNOPSIS
    populate_account_tables.py
    populate_account_tables.py table1 table2     # process some tables, not others

VERSION
    2.0 (using Casserole)

AUTHOR
    David Blaikie (dblaikie@appnexus.com)
"""

num_tried = 3
"""Number of times to try the database before failing out."""

class Process(object):

    """ Class to process data. """

    c = 'class attribute'
    """This is Process.c's docstring."""

    def __init__(self):
        """Method __init__'s docstring."""

        self.i = 'instance attribute'
        """This is self.i's docstring."""

def split(x, delim):
    """Split strings according to a supplied delimeter."""

    splitlist = x.split(delim)
    """the resulting list to be returned"""

    return splitlist
```

These docstrings can be automatically read and formatted by a docstring reader. **docutils** is a built-in processer for docs, although the documentation for *this* module is a bit involved. We'll discuss the more-easy-to-use **Sphinx** next.

Sphinx: easy docs from docstrings

In order to generate docs for a project, you must set up a separate Sphinx project, which includes a config file and a build file. The Sphinx docs are a bit obscure, but this tutorial (https://codeandchaos.wordpress.com/2012/07/30/sphinx-autodoc-tutorial-for-dummies/) covers the steps pretty clearly.

I built some Sphinix docs against a test lib - they look very pretty (http://www.davidbpython.com/intermediate_python/slides/testdoc/outputdir/build/html/index.html) and suspiciously similar to documentation for Flask (http://flask.pocoo.org/docs/0.10/).

git: version control system

Version control refers to a system that keeps track of all changes to a set of documents (i.e., the revisions, or *versions*). Its principal advantages are:

1) the ability to restore a team's codebase to any point in history
2) the ability to *branch* a parallel codebase in order to do development without disturbing the **main** branch, and then *merge* a development branch back into the **main** branch.

Git was developed by Guido van Rossum to create an open-source version of **BitKeeper**, which is a distributed, lightweight and very fast VCS. Other popular systems include **CSV**, **SVN** and **ClearCase**.

Git's docs (https://git-scm.com/) are very clear and extensive. Here is a quick rundown of the highlights:

Creating or cloning a new repository (https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository)

- **git init** to create a brand-new repository
- **git clone** to create a local copy of a repository on the remote server (a *github*) **Adding, changing and committing files to the local repository (https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository)**
- **git add** to add a new (*or changed*) file to the *staging area*
- **git status** to see what files have been changed but not added, or changed and added
- **git diff** to see what changes have been made but not committed, or changes between commits
- **git commit** to commit the added changes to the local repository **Reviewing commit history (https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History) with git log**
- **git log** to review individual commits

github: the home of open-source and team collaboration

As discussed, **git** is a *change management system* or *version control system*, basically a program that keeps track of software as it goes through changes. The place where the code is stored is called a *code repository*.

github is a publicly hosted server that stores git repositories on a free and also on a commercial basis. Free repositories are expected to be publicly available (unless they are for non-profit or educational institutions) and are generally used to host *open-source* projects.

open-source refers to software projects that expose the code *source* (i.e., the code itself) to the public. Such projects often are licensed in such a way that the public can create a copy of the project, make development changes to the project copy, and then merge the changes back to the original project. The step in which a copy of the repository is made is known as *forking the repo*.

Scott Lowe provides a pretty straightforward description of the process in Using the fork-and-branch git workflow (http://blog.scottlowe.org/2015/01/27/using-fork-branch-git-workflow).

Creating a project to add to github

1. sign up for free github.com (http://github.com) account and/or login to github
2. on your main github page, click the "+" and then click "New Repository" - if needed, select an 'Owner' account (only necessary if you have multiple github accounts) - type a name for your repository, and an optional description - the default permissions will be 'public'; in order to create a 'private' repository, you must have a paid or educational github account - click 'Create Repository'
3. at the command line, find a directory location where you will keep your git repos. One possible directory name is 'git-repos' - **cd** to where you want the repos to live (your home directory is a good place) - **mkdir git-repos** (or whatever you decided to name your directory) - **cd git-repos**
4. create a directory for your new project. name the directory after the project name, although it doesn't have to be exact - **mkdir testproj** (or whatever you decide to name your project dir) - **cd testproj**
5. initialize the local directory as a git repo. This sets up files and directories that git looks for to base a repository in this location. - **git init** - git responds with **Initialized empty Git repository in [pathname]**
6. back on the new github project page, copy the URL of the project to the clipboard (Ctrl-C or Cmd-C)

7. back in your terminal, use the URL to connect your local repo with the github repo - **git remote add origin [URL]** (where [URL] is the URL you copied) - **git remote -v** (verifies the new URL) - git responds with **origin https://github.com/dervid/testproj (fetch)** and **origin https://github.com/dervid/testproj (push)**
8. create a new text file and save it in the project directory
9. add the file to the local repo - **git add newfile.txt** (where **newfile.txt** is the name of the new file)
10. commit the file to the local repo - **git commit -m 'this is a new file for the new project'** - git responds with lines indicating the adding of the new file
11. sync the local repo with the remote repo - **git push origin master** - git responds with messages indicating that the repo is syncing with remote
12. check to see that the file has been added - return to the web page and refresh it or click on the project name in the upper left hand corner: the new file has been added

The git commit cycle: add file, commit, push

git is unlike other repository schemes (**csv**, **subversion**) in that each user has his or her own *local repository*, which is kept in sync with the *remote repository*. This allows a user to make commits to a repo without a network connection (one will be needed to eventually sync the commits).

The process of committing changes to a team repository is 3-fold: stage a file, commit the file locally, then push the changes to remote.

add or change a file (in this case I modified **test-1.py**)

git status to see that the file has changed

This command shows us the status of all files on our system: modified but not staged, staged but committed and committed but not pushed.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add ..." to update what will be committed)
  (use "git checkout -- ..." to discard changes in working directory)

        modified:   test-1.py

no changes added to commit (use "git add" and/or "git commit -a")
$
```

git add the file (whether added or changed) to the staging area

```
$ git add test-1.py
$
```

git status to see that the file has been added

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD ..." to unstage)

        modified:   test-1.py

$
```

git commit the file to the *local repository*

```
$ git commit -m 'made a trivial change'
[master e6309c9] made a trivial change
 1 file changed, 4 insertions(+)
$
```

git status to see that the file has been committed, and that our *local* repository is now "one commit ahead" of the *remote* repository (known as *origin*

```
$ git commit -m 'made a trivial change'
[master e6309c9] made a trivial change
 1 file changed, 4 insertions(+)
$
```

git pull to pull down any changes that have been made by other contributors

```
$ git pull
Already up-to-date.
$
```

git push origin master to push local commit(s) to the remote repo

The *remote repo* in our case is **github.com**, although many companies choose to host their own private remote repository.

```
$ git push
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 318 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/NYU-Python/david-blaikie-solutions
   2ce8e49..e6309c9  master -> master
$
```

Once we've pushed changes, we should be able to see the changes on **github.com** (or our company's private remote repo).

Forking a git repo to contribute to a project

Each project (containing code files, associated library files, documentation, and everything needed to support the project) is stored in its own repository on github, which we refer to as a **git repo** or a **git repository**.

If we are interested in a particular project, we can view the code for it online, or we can download all the files in the project

Steps to clone a repository

1. sign up for free github.com (github.com) account and/or login to github
2. travel to the github page of the project you want -- here are two choices - **Spoon-Knife**, a test repo (https://github.com/octocat/Spoon-Knife) - **link**, a db/api wrapper we use at AppNexus (https://github.com/dhrod5/link)
3. click on the 'fork' button on upper right hand corner. The project is forked to your own account, and the git web page page travels to the forked repository.
4. at the command line, if you have not already done so, find a directory location where you will keep your git repos. One possible directory name is 'git-repos' - **cd** to where you want the repos to live (home dir is a good place) - **mkdir git-repos** - **cd git-repos**

5. 'clone' your forked repo so it is a local project: **git clone https://github.com/dervid/Spoon-Knife** (where "dervid" is the name of your own github directory)

```
$ git clone https://github.com/dervid/Spoon-Knife
Cloning into 'Spoon-Knife'...
remote: Counting objects: 16, done.
remote: Total 16 (delta 0), reused 0 (delta 0), pack-reused 16
Unpacking objects: 100% (16/16), done.
Checking connectivity... done.
$
```

6. a folder appears containing the project. **cd** into that project and have a look around.

git branching basics

see what branches are available	**git branch**
create a new branch	**git branch *newfeature***
switch to the new branch	**git checkout *newfeature***
add and/or change file(s)	
commit and push to the branch	
if needed, push the branch to the remote repo	**git push origin HEAD**
switch back to master branch	**git checkout master**
merge the changed branch back to master	**git merge *newfeature***
push the branch to remote	**git push origin master**
delete the *newfeature* branch locally	**git branch -d *newfeature***
delete the *newfeature* branch remotely	**git push origin :*newfeature***

git's tutorial on branches (https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell) is an extremely clear tutorial on this sometimes challenging topic.

Links

Open Hatch: a resource for finding open source projects to contribute to, sortable by level
https://openhatch.org/

Code 52: launching a new open-source project each week
http://code52.org/contributing.html

writing documentation: volunteers wanted
https://sourceforge.net/p/forge/helpwanted/documenters/

Python user groups
https://wiki.python.org/moin/LocalUserGroups

NYC Python user group
http://www.meetup.com/nycpython/