

Documentation,
version control, &
collaborating on code

BILD 62



Goals for today

- Define guidelines for writing and documenting good code
- Provide information on collaborating on code via Git
- Describe & practice code review



Writing good code
and documenting it

How do we write good code for humans?

- **Use good structure**
 - If you design your program using separate functions for each task, avoid copying + pasting (functions and loops instead), and consider structure beforehand, you'll be set up for success
- **Use good naming**
- **Use code comments and include documentation**

What does this code do?

```
def ff(jj):  
    oo = list(); jj = list(jj)  
    for ii in jj: oo.append(str(ord(ii)))  
    return '+'.join(oo)  
ff('Hello World.')
```

separate lines

better
names!

clear function
name

some
comments

```
def return_unicode(input_list):  
    string = list() # []  
    input_list = list(input_list)  
  
    for character in input_list:  
        string.append(str(ord(character)))  
  
    output_string = '+'.join(string)  
    return output_string  
  
return_unicode('Hello World.')
```

Writing useful comments

- Good code has good documentation - but code documentation should *not* be used to try and fix unclear names, or bad structure.
- Rather, comments should add any additional context and information that helps explain what the code is, how it works, and why it works that way.
 - focus on the how and why, over literal ‘what is the code’
 - explain any context needed to understand the task at hand
 - give a broad overview of what approach you are taking to perform the task
 - if you’re using any unusual approaches, explain what they are, and why you’re using them

Types of comments

Block comments

```
# this box describes block  
# comments and the best way  
# to write them
```


- apply to some (or all) code that follows them
- are indented to the same level as that code.
- Each line of a block comment starts with a `#` and a single space

Inline comments `# inline`

- to be used sparingly
- to be separated by at least two spaces from the statement
- start with a `#` and a single space

Docstrings are in-code text that describe modules, classes and functions. They describe the operation of the code.

- Numpy style docs are a particular type of dogstring
- available to you outside of the source code using `help()`
- get stored as the `__doc__` attribute and can be accessed from there too
- Common structure:
 - starts and ends with triple quotes: `"""`
 - one sentence overview at the top - the task/goal of function
 - **Parameters** : description of function arguments, keywords & respective types
 - **Returns** : explanation of returned values and their types



```
# Let's fix this code!
def convert_to_unicode(input_string):
    """Converts an input string into a string containing the unicode code points.

    Parameters
    -----
    input_string : string
        String to convert to code points

    Returns
    -----
    output_string : string
        String containing the code points for the input string.
    """

    output = list()
    # Converting a string to a list, to split up the characters of the string
    input_list = list(input_string)

    for character in input_list:
        temp = ord(character)
        output.append(temp)

    output_string = '+'.join(output)

    return output_string
```

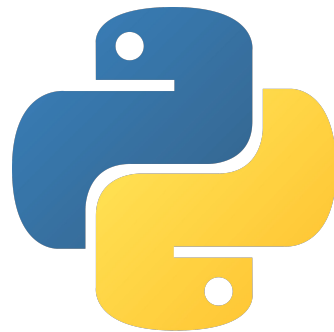
Now with a docstring!

What should be included in a docstring?

- ❑ **A brief overview sentence about the code**
 - ❑ **Input arguments/parameters and their types**
 - ❑ **Returned variables and their types**
-

Style guides

- **Coding style** refers to a set of conventions for how to write good code.
- Python Enhancement Proposals (PEPs) are written by the people responsible for the Python Programming language and propose standards for writing Python code
- PEP8 is an accepted proposal that outlines the style guide for Python.



<https://peps.python.org/pep-0008/>

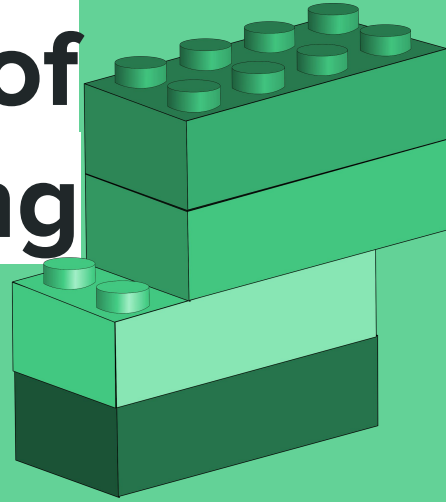
Naming conventions

- CapWords (leading capitals, no separation) for Classes
- snake_case (all lowercase, underscore separator) for variables, functions, and modules

Spacing conventions

- Put one (and only one) space between each element
- Index and assignment don't have a space between opening & closing ' () ' or ' [] '
- One statement per line
- Blank line conventions:
 - Use 2 blank lines between functions & classes, and 1 between methods
 - Use 1 blank line between segments to indicate logical structure

Let's illustrate some of these principles, along with modular code organization...



Collaborating on code

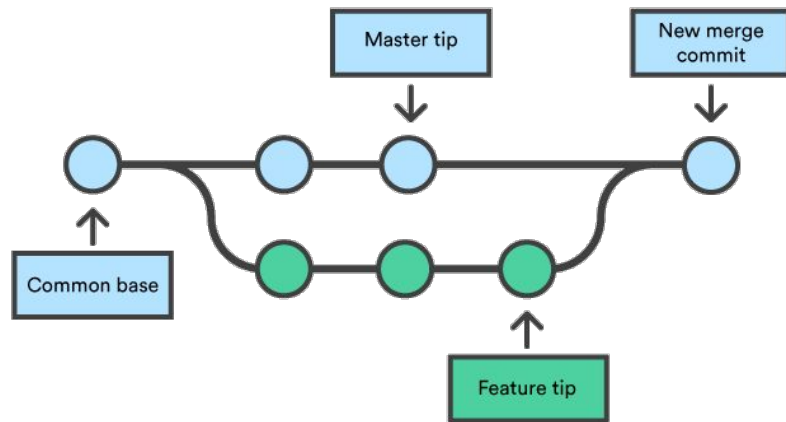
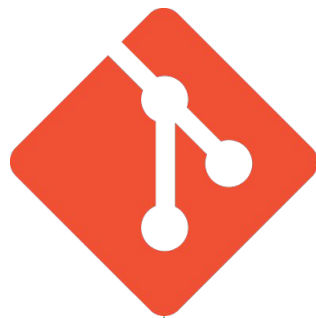
Options for collaboration

- **DataHub Container** — automatically set up once group is on Canvas
- **Google Colab** (<https://research.google.com/colaboratory/>)
 - Notebook can be shared with multiple people and viewed simultaneously
 - *However* multiple people cannot edit simultaneously (unlike Google docs)
- **Git & Github**
 - Github is a cloud storage service for the files that you maintain with Git.
 - Professional way to share code & implement version control!
 - First step: set up a GitHub account and *one repository for your project*



Version Control

- **Version control** is conceptually similar to track changes in Word — it keeps track of who makes changes, and what they did.
- A really common way to do this with code is a program called **git**.
 - GitHub is an online code hosting service that uses git and has a friendly user interface and a couple other features
- In theory, git can be used to track any type of text.
- Git can be used locally and/or with a remote repository of code



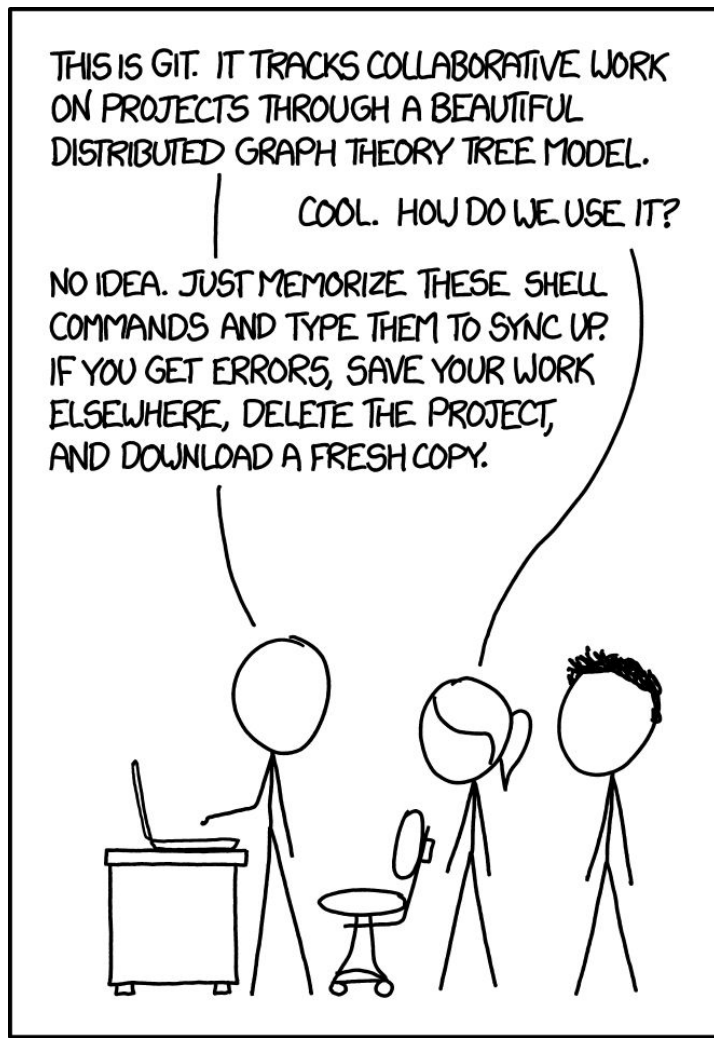
Using git

git helps you collaborate on code & track changes


In other words, it implements **version control**.

More information: [Git/Github Introduction - VOYTEKlab](#)

<https://xkcd.com/1597/>



Steps for creating your project repository

1. Make an account on GitHub
2. **Team Member #1** creates a new **public** repository 
3. **Team Member #1** adds initial code to the repository (by uploading on Github)
4. **Other team members** **clone** repository via terminal in DataHub
5. **Other team members** **add, commit, and push** their code (either as separate files, when that makes sense, or to scripts/notebooks)
6. **Team Member #1** **pulls** changes
7. BEFORE ANYONE EDITS CODE use **git pull** to get changes

Typical git workflow

1. **clone** code from a remote source to your local computer
(or in our case, to your server space)

terminal prompt name

```
ajuavine@dsm1p-jupyter-ajuavine:~$ git clone http://www.github.com/ajuavinett/CellTypesLesson.git
```

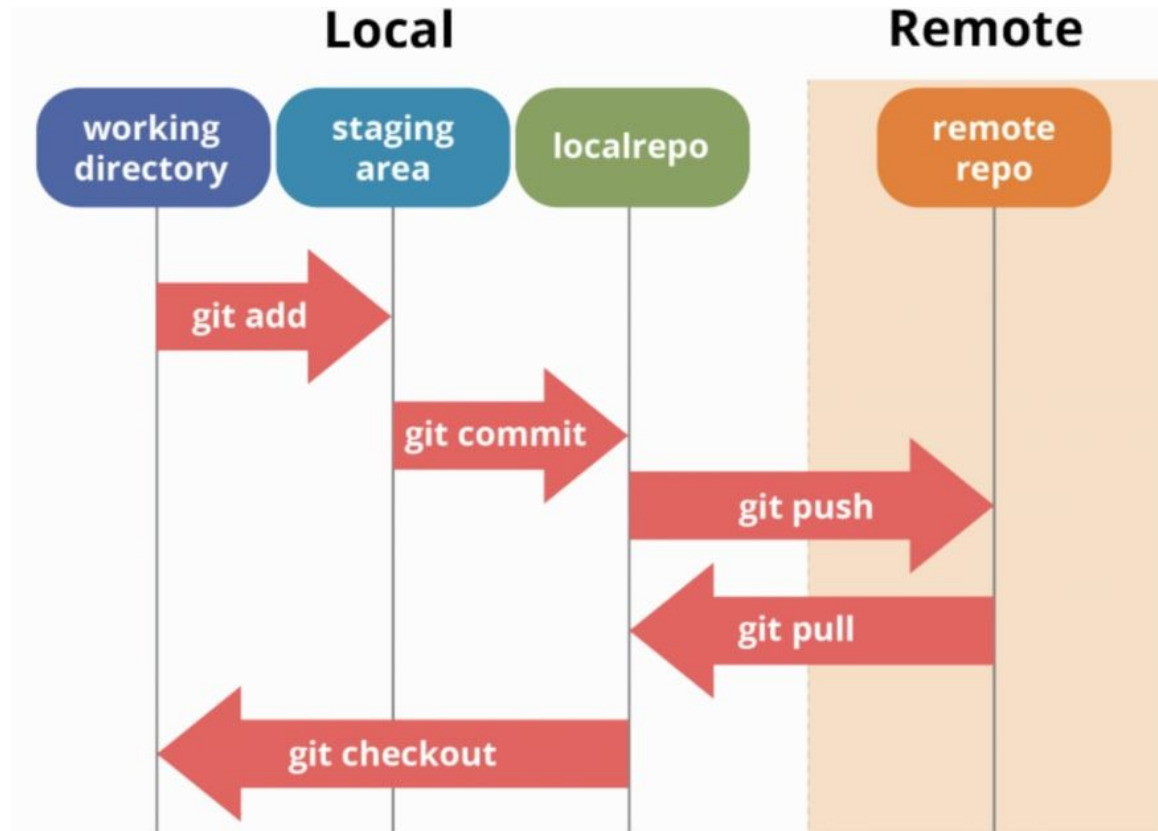
2. Edit code locally on your machine (**add & commit**)
3. **push** code back to a remote source

git status

Tells you whether
you're up to date

git checkout <filename>

Discard your changes (only use
this if necessary!)




Git flow for a simple project (from [here](#))

Goals for today

- Define guidelines for writing and documenting good code
 - Provide information on collaborating on code via Git
 - Describe & practice code review
-

Code review is a
process for
systematically reviewing
someone else's code.



Negative criticism usually fails in one or more the following ways

1. **It isn't strategic.** The critic does not think about what specifically they want to change or what goals and solutions they can offer.
2. **It isn't improvement oriented.** The critic doesn't make suggestions as to how to improve.
3. **It attacks self-esteem.** The critic uses labels (such as "lazy"), speaks in absolutes, and does not allow the recipient to save face.
4. **It uses the wrong words.** The critic uses negative statements and words like "should" instead of "could."
5. **It comes with no supporting evidence.** Critic does not support comments with evidence or fair comparisons.

Categories for code review

- 1. Functionality:** Does the code work as intended? Is it robust? If not, where is the issue?
- 2. Documentation & Style:** Is the code properly documented and commented? Where should the documentation be better? Does spacing & capitalization follow PEP-8 guidelines?
- 3. Error handling:** Does the code handle errors properly? Where could the error-handling be better?
- 4. Other suggestions:** Are there areas where the code could be more concise? Functions that could have been used, but weren't?

Today: review our “CodeReview” notebook in the four categories & submit on Canvas.

(As you work on your project, review each other’s code!)

Final Projects reminders

- Rubric is on Canvas
- Sign up for a slot on the project spreadsheet
 - 30 minute presentation slot
 - Plan on attending
for that 30-minute slot



Regardless of how you collaborate, in the end, everyone needs to submit via **DataHub** AND **Canvas**

so that we can
test your code

so that we can
leave comments

No need for
data files;

**ONLY ONE PER
GROUP!**

PLEASE INCLUDE YOUR DATA FILES!