Introduction to Python and Git

Lukas Hager

2024-03-27

Python Basics

Why python?

“Python is the second best language for everything.”

Python

- A multiuse programming language that provides interfaces to almost anything you'd want to do
- In recent years, development of tools like `numpy`, `pandas`, and `scikit-learn` have made it a formidable choice for data analysis and cleaning
- Many firms use python as their program of choice
- Interpreted language

Python Editors

- Personal preferences:
  - Jupyter Notebooks/Anaconda
    - To open a Jupyter server (from terminal):

    ```
    jupyter notebook
    ```

    - We (hopefully) have access to a Jupyter instance via UW here.
  - VS Code
    - Nice editor with tons of language-specific extensions (like for `git`, `latex`, `quarto`, `ipython`, `julia`, etc.)

Pylint

- Companies generally want code to confirm to their stylistic standards
- There are general best practices for python code formatting
- `pylint` checks code and ensures that there are no syntax errors or formatting errors
- I will not require linting in this class – but it's good practice

Python Data Structures

Scalar Data Types[1]

- `None`: The Python "null" value (only one instance of the `None` object exists)
- `str`: String type; holds Unicode strings
- `bytes`: Raw binary data
- `float`: Double-precision floating-point number (note there is no separate `double` type)
- `bool`: A Boolean `True` or `False` value
- `int`: Arbitrary precision integer

[1]Taken from Wes McKinney's Book

Strings

We can concatenate strings with +

```
print('Peanut Butter' + 'Jelly')
```

```
Peanut ButterJelly
```

Strings

We can concatenate strings with +

```python
print('Peanut Butter' + 'Jelly')
```

```
Peanut ButterJelly
```

We can also concatenate a list of strings using the `join` method

```python
str_list = ['I', 'love', 'UW']
' '.join(str_list)
```

```
'I love UW'
```

Strings

We can concatenate strings with +

```
print('Peanut Butter' + 'Jelly')
```

```
Peanut ButterJelly
```

We can also concatenate a list of strings using the `join` method

```
str_list = ['I', 'love', 'UW']
' '.join(str_list)
```

```
'I love UW'
```

We can “format” strings as well with `f`, which is hugely convenient

```
month = 'April'
f'The month is {month}.'
```

```
'The month is April.'
```

Testing Types

Sometimes you may want to test the type of a scalar in python. The canonical test for whether a variable is null:

```python
var = 2
var is None
```

```
False
```

Testing Types

Sometimes you may want to test the type of a scalar in python. The canonical test for whether a variable is null:

```python
var = 2
var is None
```

```
False
```

You can always use `type`:

```python
type(2) is str
```

```
False
```

Testing Types

Sometimes you may want to test the type of a scalar in python. The canonical test for whether a variable is null:

```python
var = 2
var is None
```

```
False
```

You can always use `type`:

```python
type(2) is str
```

```
False
```

Can also use `isinstance`:

```python
isinstance(var, int)
```

```
True
```

Lists

Lists are created in brackets, with elements separated by commas:

```python
my_list = ['a', 'b', 'c']
print(my_list)
```

```
['a', 'b', 'c']
```

Lists

Lists are created in brackets, with elements separated by commas:

```python
my_list = ['a', 'b', 'c']
print(my_list)
```

```
['a', 'b', 'c']
```

A powerful tool in python is “list comprehension”, which allows us to perform an operation on every element of a list:

```python
my_new_list = [x + '_new' for x in my_list]
my_new_list
```

```
['a_new', 'b_new', 'c_new']
```

Lists

Lists are created in brackets, with elements separated by commas:

```
my_list = ['a', 'b', 'c']
print(my_list)
```

```
['a', 'b', 'c']
```

A powerful tool in python is “list comprehension”, which allows us to perform an operation on every element of a list:

```
my_new_list = [x + '_new' for x in my_list]
my_new_list
```

```
['a_new', 'b_new', 'c_new']
```

Note that we concatenated individual strings using the + operator.

Exercise: Typing

Use list comprehension to return a list of the types of the elements in the list

```
types_list = ['a', 1, 2., False, None]
```

Solutions: Typing

```
[type(x) for x in types_list]
```

```
[str, int, float, bool, NoneType]
```

Indexing

If we want the first element of a list:

```
my_list[0]
```

```
'a'
```

Indexing

If we want the first element of a list:

```
my_list[0]
```

```
'a'
```

If we want the last element of a list:

```
my_list[-1]
```

```
'c'
```

Indexing

If we want the first element of a list:

```
my_list[0]
```

```
'a'
```

If we want the last element of a list:

```
my_list[-1]
```

```
'c'
```

If we want the first two elements of a list:

```
my_list[:2]
```

```
['a', 'b']
```

Indexing

If we want the first element of a list:

```
my_list[0]
```

```
'a'
```

If we want the last element of a list:

```
my_list[-1]
```

```
'c'
```

If we want the first two elements of a list:

```
my_list[:2]
```

```
['a', 'b']
```

If we want all but the first two elements of a list:

Dictionaries

Dictionaries are similar to lists, except that individual elements are named. They're created as `key:value` pairs within braces, separated by commas.

```
my_dict = {
    'string': 'a',
    'int': 1,
    'list': my_list
}
my_dict
```

```
{'string': 'a', 'int': 1, 'list': ['a', 'b', 'c']}
```

Dictionaries

Dictionaries are similar to lists, except that individual elements are named. They're created as `key:value` pairs within braces, separated by commas.

```
my_dict = {
    'string': 'a',
    'int': 1,
    'list': my_list
}
my_dict
```

```
{'string': 'a', 'int': 1, 'list': ['a', 'b', 'c']}
```

Individual elements can be accessed by their key:

```
my_dict['string']
```

```
'a'
```

Dictionaries

You can access all the keys of a dictionary:

```
my_dict.keys()
```

```
dict_keys(['string', 'int', 'list'])
```

Dictionaries

You can access all the keys of a dictionary:

```
my_dict.keys()
```

```
dict_keys(['string', 'int', 'list'])
```

You can also create dictionaries using list comprehension:

```
list_len = len(my_list)
idx_seq = range(list_len)
my_new_dict = {x: my_list[x] for x in idx_seq}
my_new_dict
```

```
{0: 'a', 1: 'b', 2: 'c'}
```

Dictionaries

You can access all the keys of a dictionary:

```
my_dict.keys()
```

```
dict_keys(['string', 'int', 'list'])
```

You can also create dictionaries using list comprehension:

```
list_len = len(my_list)
idx_seq = range(list_len)
my_new_dict = {x: my_list[x] for x in idx_seq}
my_new_dict
```

```
{0: 'a', 1: 'b', 2: 'c'}
```

Remember that python indexes everything starting from 0, not 1!

```
list_len == max(idx_seq)
```

```
False
```

Dictionaries

We can also update elements of a dictionary, though we should be a little careful doing this.

```
my_dict['string'] = 55
my_dict
```

```
{'string': 55, 'int': 1, 'list': ['a', 'b', 'c']}
```

Tuples

Tuples are arrays of elements, created by separating elements by commas between parentheses. They're very similar to lists, but we cannot change individual elements (immutability).

```
my_tuple = ('a','b','c')
my_tuple[1] = 'z'
```

```
TypeError: 'tuple' object does not support item assignment
```

Tuples

Tuples are arrays of elements, created by separating elements by commas between parentheses. They're very similar to lists, but we cannot change individual elements (immutability).

```
my_tuple = ('a','b','c')
my_tuple[1] = 'z'
```

```
TypeError: 'tuple' object does not support item assignment
```

One reason that they're helpful in that they can allow you to define multiple variables inline simultaneously.

```
a,b = (1,2)
a+b
```

```
3
```

Loops

For Loops

We often want to perform an operation/function/chunk of code on every element of a list. For example, we could have used a for loop instead of list comprehension to create `my_new_list`:

```python
my_newer_list = []
for list_element in my_list:
    my_newer_list.append(list_element + '_new')
my_newer_list
```

```
['a_new', 'b_new', 'c_new']
```

For Loops

We often want to perform an operation/function/chunk of code on every element of a list. For example, we could have used a for loop instead of list comprehension to create `my_new_list`:

```python
my_newer_list = []
for list_element in my_list:
    my_newer_list.append(list_element + '_new')
my_newer_list
```

```
['a_new', 'b_new', 'c_new']
```

What if we wanted to add all the numbers from 1 to 5?

```python
my_sum = 0
for num in range(1,6):
    my_sum += num
my_sum
```

```
15
```

While Loops

Sometimes we don't know how many times we want to loop, so a for loop can't be used. Instead, we want to loop **until** something happens. In these cases, we can use while loops. For example, we can solve the problem

$$\max_n \left\{ \sum_{i=1}^{n} i \leq 2024 \right\}$$

While Loops

```python
my_sum = 0
n = 0
while my_sum <= 2024:
    n += 1
    my_sum += n
n, my_sum
```

```
(64, 2080)
```

While Loops

```python
my_sum = 0
n = 0
while my_sum <= 2024:
    n += 1
    my_sum += n
n, my_sum
```

```
(64, 2080)
```

Why isn't this right? How can we fix it?

While Loops

```python
my_sum = 0
n = 0
while my_sum <= 2024:
    n += 1
    my_sum += n
n, my_sum
```

```
(64, 2080)
```

Why isn't this right? How can we fix it?

```python
my_sum = 0
n = 0
while my_sum + (n+1) <= 2024:
    n += 1
    my_sum += n
n, my_sum
```

```
(63, 2016)
```

if/else

Sometimes we want to have conditional logic in our loops (or in our code more generally)

```python
a_mixed_list = [1, 5, '2', 4, '12', '02']
out = 0
for i in range(len(a_mixed_list)):
    out += a_mixed_list[i]
out
```

```
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

if/else

Sometimes we want to have conditional logic in our loops (or in our code more generally)

```
a_mixed_list = [1, 5, '2', 4, '12', '02']
out = 0
for i in range(len(a_mixed_list)):
    out += a_mixed_list[i]
out
```

```
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

```
for i in range(len(a_mixed_list)):
    if isinstance(a_mixed_list[i], str):
        out += float(a_mixed_list[i])
    else:
        out += a_mixed_list[i]
out
```

```
32.0
```

if/else

Sometimes we want to have conditional logic in our loops (or in our code more generally)

```python
a_mixed_list = [1, 5, '2', 4, '12', '02']
out = 0
for i in range(len(a_mixed_list)):
    out += a_mixed_list[i]
out
```

```
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

```python
for i in range(len(a_mixed_list)):
    if isinstance(a_mixed_list[i], str):
        out += float(a_mixed_list[i])
    else:
        out += a_mixed_list[i]
out
```

```
32.0
```

elif

Before we had an either/or – using `elif`, we can add multiple possibilities:

```
data_list = [10, 20, 30, 15, 25]
label_list = ['red', 'green', 'blue', 'red', 'green']
red_sum, green_sum, blue_sum = 0,0,0
for i in range(len(data_list)):
    if label_list[i] == 'red':
        red_sum += data_list[i]
    elif label_list[i]  == 'green':
        green_sum += data_list[i]
    else:
        blue_sum += data_list[i]
red_sum, green_sum, blue_sum
```

```
(25, 45, 30)
```

pass

Python's "do nothing" statement

```python
x = 0
if x < 0:
    print("negative!")
elif x == 0:
    # TODO: put something smart here
    pass
else:
    print("positive!")
```

break

We can exit for loops entirely using `break` – let's revisit our earlier sum problem:

```python
my_sum = 0
n = 0
for i in range(2024):
    if my_sum > 2024-i:
        n += i-1
        break
    my_sum += i
n, my_sum
```

```
(63, 2016)
```

Exercise: Factorial

Write a loop that takes a natural number $n > 0$ and returns $n!$, where

$$n! = \prod_{i=1}^{n} i$$

If you have time, write a loop that computes the binomial coefficient:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Solution (First Part)

```python
n = 5
n_fac = 1
for i in range(1,n+1):
    n_fac *= i
n_fac
```

```
120
```

Solution (Second Part)

Note that

$$\frac{n!}{k!(n-k)!} = \frac{\prod_{i=k+1}^{n} i}{(n-k)!}$$

```python
k = 3
n_minus_k_fac = 1
for i in range(1,n-k+1):
    n_minus_k_fac *= i
n_choose_k = 1
for i in range(k+1, n+1):
    n_choose_k *= i
n_choose_k /= n_minus_k_fac
n_choose_k
```

```
10.0
```

Functions

Why?

- In the second solution we reused code that we already wrote for the first part
- If we do this enough, we'll make mistakes that are hard to track down
- If we want to repeatedly perform the same operation, we should write a function

DRY (Don't Repeat Yourself)

- If you're copy/pasting code, you could probably do something differently to avoid that
- If you find a way to avoid doing so, your code will be better
- So: try not to "repeat yourself" in this way

Basic Function Syntax

Functions can be defined very generally in python

```python
def square(x):
    return x**2

square(2)
```

```
4
```

Basic Function Syntax

Functions can be defined very generally in python

```python
def square(x):
    return x**2

square(2)
```

```
4
```

There are a few improvements that we can make, both for ensuring that our code runs properly and to make it more readable.

Docstrings

In general, we want other users of our code (or our future selves) to be able to understand what functions do. This is particularly important later when we get to classes.

```
def square(x):
    """
    This function takes as input a scalar and outputs the scal
    """
    return x**2

square(2)
```

```
4
```

Docstrings

In general, we want other users of our code (or our future selves) to be able to understand what functions do. This is particularly important later when we get to classes.

```python
def square(x):
    """
    This function takes as input a scalar and outputs the sca
    """
    return x**2

square(2)
```

```
4
```

For such a simple function, this might be superfluous, but get in the habit of doing it!

Function Annotations

Sometimes, it might not be obvious what types of arguments should be passed to functions. For example, our function `square` cannot take a string as its argument. We can help users out by showing what the input of a function should be and what it will output.

```
def square(x: float) -> float:
    """
    This function takes as input a scalar and outputs the sca
    """
    return x**2

square(2)
```

```
4
```

Function Annotations

Sometimes, it might not be obvious what types of arguments should be passed to functions. For example, our function `square` cannot take a string as its argument. We can help users out by showing what the input of a function should be and what it will output.

```
def square(x: float) -> float:
    """
    This function takes as input a scalar and outputs the sca
    """
    return x**2

square(2)
```

```
4
```

Many code editors can make use of these annotations when your code is being used by others.

Multiple Arguments

Functions can take more than one argument. For example, suppose we wanted to calculate the value y corresponding to point x on the linear function $y = mx + b$.

```
def linear_function(x: float, m: float, b: float) -> float:
    """
    Given the slope (m) and intercept (b) of a line, calculat
    the y coordinate corresponding to x.
    """
    return m*x + b

linear_function(x=3., m=-1., b=0.)
```

```
-3.0
```

Function Defaults

Python makes it very easy to supply default arguments to functions. In the case below, we provide a default slope of -1 and default intercept of 0.

```
def linear_function(x: float, m: float=-1, b: float=0) -> flo
    """
    Given the slope (m) and intercept (b) of a line, calculat
    the y coordinate corresponding to x.
    """
    return m*x + b

linear_function(3.)
```

```
-3.0
```

Function Defaults

Python makes it very easy to supply default arguments to functions. In the case below, we provide a default slope of -1 and default intercept of 0.

```
def linear_function(x: float, m: float=-1, b: float=0) -> floa
    """
    Given the slope (m) and intercept (b) of a line, calculat
    the y coordinate corresponding to x.
    """
    return m*x + b

linear_function(3.)
```

```
-3.0
```

Note that when you define a function, any argument that does not have a default must be listed before arguments with defaults (so `x` must be listed before `m` and `b` above).

Dates

datetime Import

datetime has a module by the same name, so normally it looks something like this

```
from datetime import datetime, date, time, timedelta
```

`datetime` Import

`datetime` has a module by the same name, so normally it looks something like this

```
from datetime import datetime, date, time, timedelta
```

If you don't have access to a package, you can generally install via

```
pip install datetime
```

Creating a `datetime` Object

Passing integers to `datetime` yields a `datetime` object

```
dt = datetime(2011, 10, 29, 20, 30, 21)
dt.day
```

```
29
```

Creating a datetime Object

Passing integers to `datetime` yields a `datetime` object

```
dt = datetime(2011, 10, 29, 20, 30, 21)
dt.day
```

```
29
```

```
dt.minute
```

```
30
```

Useful Methods

```
dt.date()
```

```
datetime.date(2011, 10, 29)
```

Useful Methods

```
dt.date()
```

```
datetime.date(2011, 10, 29)
```

```
dt.time()
```

```
datetime.time(20, 30, 21)
```

Useful Methods

```
dt.date()
```

```
datetime.date(2011, 10, 29)
```

```
dt.time()
```

```
datetime.time(20, 30, 21)
```

We can also format the object into something recognizable:

```
dt.strftime("%Y-%m-%d %H:%M")
```

```
'2011-10-29 20:30'
```

Useful Methods

```
dt.date()
```

```
datetime.date(2011, 10, 29)
```

```
dt.time()
```

```
datetime.time(20, 30, 21)
```

We can also format the object into something recognizable:

```
dt.strftime("%Y-%m-%d %H:%M")
```

```
'2011-10-29 20:30'
```

And we can parse strings into `datetime` objects:

```
datetime.strptime("20091031", "%Y%m%d")
```

```
datetime.datetime(2009, 10, 31, 0, 0)
```

Time Differences

```
dt2 = datetime(2011, 11, 15, 22, 30)
delta = dt2 - dt
delta
```

```
datetime.timedelta(days=17, seconds=7179)
```

Time Differences

```
dt2 = datetime(2011, 11, 15, 22, 30)
delta = dt2 - dt
delta
```

```
datetime.timedelta(days=17, seconds=7179)
```

```
dt + delta
```

```
datetime.datetime(2011, 11, 15, 22, 30)
```

Exercise: `datetime`

Create a list of the date of every day we have class this quarter. You may ignore holidays.

- Hints:
  - The last day of classes is May 31, 2024
  - You can create a `timedelta` object using `timedelta(days=...)`

Solutions: datetime

```
monday = datetime(2024, 3, 25)
wednesday = datetime(2024, 3, 27)
week = timedelta(days=7)
class_dates = []

while wednesday < datetime(2024, 5, 31):
    class_dates.append(monday)
    class_dates.append(wednesday)
    monday += week
    wednesday += week

[x.strftime("%Y-%m-%d") for x in class_dates]
```

```
['2024-03-25',
 '2024-03-27',
 '2024-04-01',
 '2024-04-03',
 '2024-04-08',
 '2024-04-10',
 '2024-04-15'
```

Importing

Module: file with a `.py` extension containing Python code

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

Importing

Module: file with a `.py` extension containing Python code

```
# some_module.py
PI = 3.14159

def f(x):
    return x + 2

def g(a, b):
    return a + b
```

To access these values/functions in another file:

```
import some_module
result = some_module.f(5)
pi = some_module.PI
```

Alternative Import

```python
from some_module import g, PI
result = g(5, PI)
```

Alternative Import

```
from some_module import g, PI
result = g(5, PI)
```

You can import everything using * (you shouldn't generally do this)

```
from some_module import *
```

as

```
import some_module as sm
from some_module import PI as pi, g as gf

r1 = sm.f(pi)
r2 = gf(6, pi)
```

Git

Version Control

- When you use a program like Microsoft Word or Dropbox, you have version control built in
    - Ability to revert document or file to a previous version
- This is *really* important for code as well
    - Imagine you make a bunch of edits to some code to improve readability
    - You rerun the code and your results change
    - Without version control, it'll take a lot of work to figure out what changed!

Collaboration

- You'll often be working with other teammates on projects
- How do you coordinate editing files to make sure that everyone's using the most recent version?
- What do you do if two people edit the same line of code?
- `git` has tools to help you do these things

GitHub

- I highly recommend using `git` with GitHub
- GitHub is a great place to publicly post your code for potential employers or collaborators
- In addition, you can sign up for the Student Developer Pack for free
  - Gives you access to Copilot (careful!)
  - Allows you to make private repositories
  - Plenty of other goodies

Setting Up a Repository[2]

1. Set up a GitHub Account/install git
2. In your profile on GitHub's website, in the upper right-hand corner of the "Repositories" tab is a green "New" button.
3. Choose a name and description.
4. For this class's homework, please keep your repositories public.
5. Check the box for adding a README file, and use the Python .gitignore file.

[2]Code folder for a project.

Setting Up a Repository

6. Click on “Code” and copy the HTTPS or SSH link (based on how you set up your profile)
7. In terminal (Mac or Linux) or Git Bash (Windows), run

```
git clone <the link you copied>
```

within the folder that will store your code files.

Files in Git

- There are two versions of every file – the version that's in the remote repository, and your local version of that file (inclusive of your changes)
- Whenever you want to update the file in the remote repository, you have to do the following:
  1. Stage the file for commit
  2. Commit the files (and write a message)
  3. Pull any edits from the remote
  4. Push your edits to the remote

Step 1: Staging Files

- You created a new file (or edited an existing file) `newfile.py` and want to add it to the remote repository
- Basic syntax:

If we created `newfile.py`, we would stage it for commit via

```
git add newfile.py
```

We can add all files that have changed using `.`:

```
git add .
```

We can check that it's been staged:

```
git status
```

Step 2: Commiting the Files

- Once the files are staged, we want to write a message explaining what edits we made to the files that we've staged
- Basic syntax:

```
git commit -m "I created a new python file for my repository"
```

Step 3: Pulling Remote Changes

- If there have been any changes to the remote, we want to pull those down and resolve any conflicts that may exist
- Resolving those changes: kind of a pain!

```
git pull origin main
```

- Advanced: if you're using branches, replace `main` with your branch name:

```
git pull origin my_branch
```

Step 4: Push Your Changes

- Finally, push your changes to the remote!

```
git push origin main
```

- Advanced: if you're using branches, replace `main` with your branch name:

```
git push origin my_branch
```

Step 3.5: Conflicts

- If the remote's changes and your changes overlap, you need to tell Git how to resolve this

```
 * branch            main       -> FETCH_HEAD
   3f06dd5..81cf6f5  main       -> origin/main
hint: You have divergent branches and need to specify how to r
hint: You can do so by running one of the following commands s
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global"
hint: preference for all repositories. You can also pass --reb
hint: or --ff-only on the command line to override the configu
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Recommendation

I think it's generally good to tell Git to merge by default

```
git config pull.rebase false
```

Then when we run `git pull origin main`, we now get:

```
 * branch            main       -> FETCH_HEAD
Auto-merging my_file.py
CONFLICT (content): Merge conflict in my_file.py
Automatic merge failed; fix conflicts and then commit the resu
```

Then `git status` helpfully tells us:

```
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   my_file.py
```

Fixing the Conflict Manually

If we open my_file.py, Git tells us where the merge issue is and what the two different files say (here the remote has print(greetings) and locally we have print('sup'))

```
<<<<<<< HEAD
print('greetings')
=======
print('sup')
>>>>>>> 81cf6f5ecf31769420f83647af52c870224d4dd2
```

All we have to do is keep whichever option we want (manually delete what we don't want)

Other Options

If we know **for sure** that we want to keep **all** the **local** changes, we can run

```
git pull origin main -X ours
```

And similarly, if we know **for sure** that we want to keep **all** the **remote** changes, we can run

```
git pull origin main -X theirs
```

⚠ Warning

This will not have you proceed difference-by-difference – if there are multiple differences, it will keep all of ours or theirs.

Git Basics (Summary)

- If you're not collaborating, `git` is quite simple.
  - `git add filename` [3] will stage a file for commit
  - `git commit -m "this is my change"` will add a commit message
  - `git pull origin branchname` will pull the remote changes
  - `git push origin branchname` will push your changes to the remote (on branch `branchname`)

[3] If you want to add all files that have changed, use `git add .`

Branches

- Another nice feature of Git is the option to use branches
  - If you branch off of the `main` repository, you'll get a copy of that code that you can use as a sandbox, and then merge back in if you want to later
- Potential Uses:
  - Collaboration
  - When there's a stable version of the code that you don't want to break

Creating a Branch

- To create a new branch off of an existing branch:

```
git checkout -b <new branch name> <old branch name>
```

- To move between branches:

```
git checkout <branch name>
```

Merging a Branch

- Usually done via a “pull request”
- I recommend doing this on GitHub via their GUI