GENERAL ASSEMBLY

INTRO TO DATA SCIENCE:

PYTHON & VERSION CONTROL WITH GIT

AGENDA

I. INTRO TO PYTHON
II. PYTHON STRENGTHS & WEAKNESSES
III. PYTHON DATA STRUCTURES
IV. PYTHON CONTROL FLOW

EXERCISES:
V. LAB: DATA EXPLORATION USING I-PYTHON NOTEBOOK
VI. VERSION CONTROL, GIT, & GITHUB

I. INTRO TO PYTHON

Q: What is Python?

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

open source: *free! (both binaries and source files)*

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

open source*: free! (both binaries and source files)*

high-level*: interpreted (not compiled)*

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

open source*: free! (both binaries and source files)*

high-level*: interpreted (not compiled)*

dynamic*: things that would typically happen at compile time happen at runtime instead (eg, dynamic typing)*

DYNAMIC TYPING

```
>>> x = 1
>>> x
1
>>> x = 'horseshoe'
>>> x
'horseshoe'
>>> _
```

Q: What is Python?

A: An open source, high-level, dynamic scripting language.

open source: *free! (both binaries and source files)*

high-level: *interpreted (not compiled)*

dynamic: *things that would typically happen at compile time happen at runtime instead (eg, dynamic typing)*

scripting language: *"middle-weight"*

Python supports multiple programming paradigms, such as:

Python supports multiple programming paradigms, such as:

- *imperative programming*
- *object oriented programming*
- *functional programming (sort of)*

IMPERATIVE PROGRAMMING IN PYTHON

```
print 'writing publisher counts to file...'
with open(output_file, 'w') as f:
    for k, v in pubs_counter.iteritems():
        try:
            f.write('{0}, {1}\n'.format(k, v))
        except Exception as details:
            print 'error: {0} -- {1}'.format(details, (k, v))
            continue
```

OOP IN PYTHON

```
class MRWordCount(MRJob):

    def mapper(self, _, line):
        # self.set_status('mapper')
        self.increment_counter('mapper_group', 'items_mapped', 1)
        for word in line.split():
            yield word, 1

    def reducer(self, word, counts):
        # self.set_status('reducer')
        self.increment_counter('reducer_group', 'items_reduced', 1)
        yield word, sum(counts)
```

OOP IN PYTHON

```python
class MRWordCount(MRJob):

    def mapper(self, _, line):
        # self.set_status('mapper')
        self.increment_counter('mapper_group', 'items_mapped', 1)
        for word in line.split():
            yield word, 1

    def reducer(self, word, counts):
        # self.set_status('reducer')
        self.increment_counter('reducer_group', 'items_reduced', 1)
        yield word, sum(counts)
```

NOTE

In Python, *everything is an object*.

```
>>> x = range(5)
>>> x
[0, 1, 2, 3, 4]
>>> [k**2 for k in x]
[0, 1, 4, 9, 16]
>>> _
```

NOTE

This is called a *list comprehension*.

Python is an open source project which is maintained by a large and very active community.

Python is an open source project which is maintained by a large and very active community.

*It was originally created by Guido Van Rossum in the 1990s, who currently holds the title of Benevolent Dictator For Life (***BDFL***).*

GUIDO

http://en.wikipedia.org/wiki/Guido_van_Rossum

GUIDO: THE EARLY YEARS

http://pytalent.zandstrasystems.com/meet_py1.html

The presence of a BDFL means that Python has a unified design philosophy.

The presence of a BDFL means that Python has a unified design philosophy.

This design philosophy emphasizes readability and ease of use, and is codified in PEP8 (the Python style guide) and PEP20 (the Zen of Python).

The presence of a BDFL means that Python has a unified design philosophy.

This design philosophy emphasizes readability and ease of use, and is codified in PEP8 (the Python style guide) and PEP20 (the Zen of Python).

NOTE

PEPs (or Python Enhancement Proposals) are the public design specs that the language follows.

II. PYTHON STRENGTHS & WEAKNESSES

Python's popularity comes from the strength of its design.

The syntax looks like pseudocode, and it is explicitly meant to be clear, compact, and easy to read.

This is usually summarized by saying Python is an **expressive** *language.*

Python is also an extremely versatile language, and it attracts fans from many different walks of life:

web development	*https://www.djangoproject.com/*
data analysis	*http://pandas.pydata.org/*
systems admin	*http://docs.fabfile.org/en/1.6/*
(etc)	*https://github.com/languages/Python*

Another great strength is the **Python Standard Library***.*

This is a collection of packages that ships with the standard Python distrubution, and "...covers everything from asynchronous processing to zip files".

The advantages of the PSL are usually described by saying that Python comes with **batteries included***.*

Ultimately, Python's most important strength is that it's easy to learn and easy to use.

Ultimately, Python's most important strength is that it's easy to learn and easy to use.

Because there should be only one way to perform a given task, things frequently work the way you expect them to.

Ultimately, Python's most important strength is that it's easy to learn and easy to use.

Because there should be only one way to perform a given task, things frequently work the way you expect them to.

This is a huge luxury! (Especially when compared with R)

STRENGTHS & WEAKNESSES

Q: Python sounds amazing. What is it bad at?

Q: Python sounds amazing. What is it bad at?

For one thing, Python is slower than a lower-level language (but keep in mind that this is a conscious tradeoff).

Q: Python sounds amazing. What is it bad at?

For one thing, Python is slower than a lower-level language (but keep in mind that this is a conscious tradeoff).

Many people would say that Python's Achilles heel is concurrency. This is a result of the **Global Interpreter Lock** *(again, a conscious design decision).*

STRENGTHS & WEAKNESSES

Q: Python sounds amazing. What is it bad at?

For one thing, Python is slower than a lower-level language (but keep in mind that this is a conscious tradeoff).

Many people would say that Python's Achilles heel is concurrency. This is a result of the **Global Interpreter Lock** *(again, a conscious design decision).*

There are some other subtleties regarding dynamic typing that people occasionally dislike, but again this is intentional (and a matter of opinion).

III. PYTHON DATA STRUCTURES

The most basic data structure is the **None** *type. This is the equivalent of NULL in other languages.*

There are four basic numeric types: **int, float, bool, complex**.

```
>>> type(1)
<type 'int'>
>>> type(2.5)
<type 'float'>
>>> type(True)
<type 'bool'>
>>> type(2+3j)
<type 'complex'>
```

The next basic data type is the array, implemented in Python as a **list***.*

A list is an ordered collection of elements, and these elements can be of arbitrary type. Lists are **mutable***, meaning they can be changed in-place.*

```
>>> k = [1, 'b', True]
>>> k[2]
True
>>> k[1] = 'a'
>>> k
[1, 'a', True]
```

After lists we have **tuples**, *which are immutable arrays of arbitrary elements.*

```
>>> x = (1, 'a', 2.5)
>>> x
(1, 'a', 2.5)
>>> x[0]
1
>>> x[0] = 'b'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Tuples are frequently used behind the scenes in a special type of variable assignment called **tuple packing/unpacking**.

The **string** *type in Python represents an immutable ordered array of characters (note there is no char type).*

Strings support slicing and indexing operations like arrays, and have many other string-specific functions as well.

String processing is one area where Python excels.

Associative arrays (or hash tables) are implemented in Python as the **dictionary** *type. This is a very efficient and useful structure that Python's internal functions use extensively.*

```
>>> this_class = {'subject': 'data science', 'instructor': 'jason', 'time': 1800, 'is_cool': True}
>>> this_class['subject']
'data science'
>>> this_class['is_cool']
True
```

Dictionaries are unordered collections of **key-value pairs**, *and dictionary keys must be immutable.*

Another basic Python data type is the **set**. *Sets are unordered mutable collections of distinct elements.*

```
>>> y = set([1,1,2,3,5,8])
>>> y
set([8, 1, 2, 3, 5])
```

These are particularly useful for checking membership of an element and for ensuring element uniqueness.

Our final example of a data type is the Python **file object**. *This represents an open connection to a file (eg) on your laptop.*

```
>>> with open('output_file.txt', 'w') as f:
...     f.write(my_output)
```

These are particularly easy to use in Python, especially using the with statement context manager, which automatically closes the file handle when it goes out of scope.

IV. PYTHON CONTROL FLOW

CONTROL FLOW

Python has a number of control flow tools that will be familiar from other languages. The first is the **if-else statement**, *whose compound syntax looks like this:*

```
>>> x, y = False, False
>>> if x:
...     print 'apple'
... elif y:
...     print 'banana'
... else:
...     print 'sandwich'
...
sandwich
```

CONTROL FLOW

Next is the **while loop**. *This executes while a given condition evaluates to True.*

```
>>> x = 0
>>> while True:
...   print 'HELLO!'
...   x += 1
...   if x >= 3:
...     break
...
HELLO!
HELLO!
HELLO!
```

CONTROL FLOW

Another familiar (and useful) construct is the **for loop**. *This executes a block of code for a range of values.*

```
>>> for k in range(4):
...   print k**2
...
0
1
4
9
```

The object that a for loop iterates over is called (appropriately) an iterable.

A useful but possibly unfamiliar construct is the **try-except block***:*

```
>>> try:
...     print undef
... except:
...     print 'nice try'
...
nice try
```

This is useful for catching and dealing with errors, also called **exception handling***.*

FUNCTIONS

Python allows you to define custom **functions** *as you would expect:*

```
>>> def x_minus_3(x):
...     return x - 3
...
>>> x_minus_3(12)
9
```

Functions can optionally return a value with a **return statement** *(as this example does).*

Functions can take a number of **arguments** *as inputs, and these arguments can be specified in two ways:*

As **positional arguments***:*

```
>>> def f(x, y):
...     return x - y
...
>>> f(4,2)
2
>>> f(2,4)
-2
```

Functions can take a number of **arguments** *as inputs, and these arguments can be specified in two ways:*

Or as **keyword arguments***:*

```
>>> def g(arg1=x, arg2=y):
...     return arg1 / float(arg2)
...
>>> g(arg1=10, arg2=5)
2.0
>>> g(arg2=100, arg1=10)
0.1
```

Python supports **classes** *with member attributes and functions:*

```
>>> class Circle():
...   def __init__(self, r=1):
...     self.radius = r
...   def area(self):
...     return 3.14 * self.radius * self.radius
...
>>> c = Circle(4)
>>> c.radius
4
>>> c.area
<bound method Circle.area of <__main__.Circle instance at 0x1060778c0>>
>>> c.area()
50.24
>>> 3.14 * 4 * 4
50.24
```

A file with Python code in it is referred to as a **module**. *Modules can be turned into executable scripts in two steps:*

1) *include the* `if __name__ == '__main__'` *block*
2) *specify the interpreter (typically using a Unix shebang)*

The screenshot on the next slide demonstrates both of these.

MODULES

```python
#!/usr/local/bin/python
from mrjob.job import MRJob

class MRHL(MRJob):

    def mapper(self, _, line):
        lat, lon, src, nuid = line.rstrip().split(',')
        if src == 'physical':
            yield nuid, (lon, lat)
        else:
            pass

    def reducer(self, nuid, lonlats):
        unique_lonlats = list(set([tuple(k) for k in lonlats]))
        yield nuid, len(unique_lonlats)

if __name__ == '__main__':
    MRHL.run()
```

IMPORT

The previous slide also demonstrated one use of the **import** *statement.*

The import statement can be used in three ways:

```
>>> import sys
>>>
>>> from operator import itemgetter
>>>
>>> from os import *
```

The differences have to do with how each import statement interacts with the local **namespace***.*

SYNTAX & INDENTATION

Python's syntax is (again) designed with clarity in mind, and good syntax is actually enforced by the interpreter.

This comes from the fact that instead of curly braces or 'begin/end' keywords, code blocks are defined by **indentation***.*

This is unique to Python!

HAVE A LOOK AT THIS AGAIN

```python
#!/usr/local/bin/python
from mrjob.job import MRJob

class MRHL(MRJob):

    def mapper(self, _, line):
        lat, lon, src, nuid = line.rstrip().split(',')
        if src == 'physical':
            yield nuid, (lon, lat)
        else:
            pass

    def reducer(self, nuid, lonlats):
        unique_lonlats = list(set([tuple(k) for k in lonlats]))
        yield nuid, len(unique_lonlats)

if __name__ == '__main__':
    MRHL.run()
```

COMMENTS

Comments in Python are denoted by the ‘#’ character.

```
# break when msg timestamp passes t_end
try:
    if created >= t_end:
        break

# if created DNE, keep going
except Exception as details:
    print details
    pass
```

DOCSTRINGS

There are also special comments called **docstrings** *that immediately follow class and function definitions.*

```
def create_brqfile(queue_object, t_interval):
    """Browses queue, writes all info for a given day to file."""
```

Docstrings are denoted by triple quotes.

V.LAB: DATA EXPLORATION IN I-PYTHON NOTEBOOK

VI: VERSION CONTROL, GIT, & GITHUB

Why do we care about version control?

- *Ever had a hard drive crash?*
- *Ever had a computer stolen?*
- *Ever deleted the wrong file?*
- *Ever modified code and unintentionally broken it?*

Why do we care about version control?

- *Ever had a hard drive crash?*
- *Ever had a computer stolen?*
- *Ever deleted the wrong file?*
- *Ever modified code and unintentionally broken it? (of course not)*

Why do we care about version control?

- *Ever had a hard drive crash?*
- *Ever had a computer stolen?*
- *Ever deleted the wrong file?*
- *Ever modified code and unintentionally broken it? (of course not)*
- *But ever had a co-worker do that...? ;-)*

Why do we care about version control?

- *Ever had a hard drive crash?*
- *Ever had a computer stolen?*
- *Ever deleted the wrong file?*
- *Ever modified code and unintentionally broken it? (of course not)*
- *But ever had a co-worker do that...? ;-)*
- *...and then pushed that code to your production website?!?!*

Why do we care about version control?

- *Ever had a hard drive crash?*
- *Ever had a computer stolen?*
- *Ever deleted the wrong file?*
- *Ever modified code and unintentionally broken it? (of course not)*
- *But ever had a co-worker do that…? ;-)*
- *…and then pushed that code to your production website?!?!*

That's why we care about version control!!

Version control is a system that records changes to a file or set of files over time so that we can recall specific versions later.

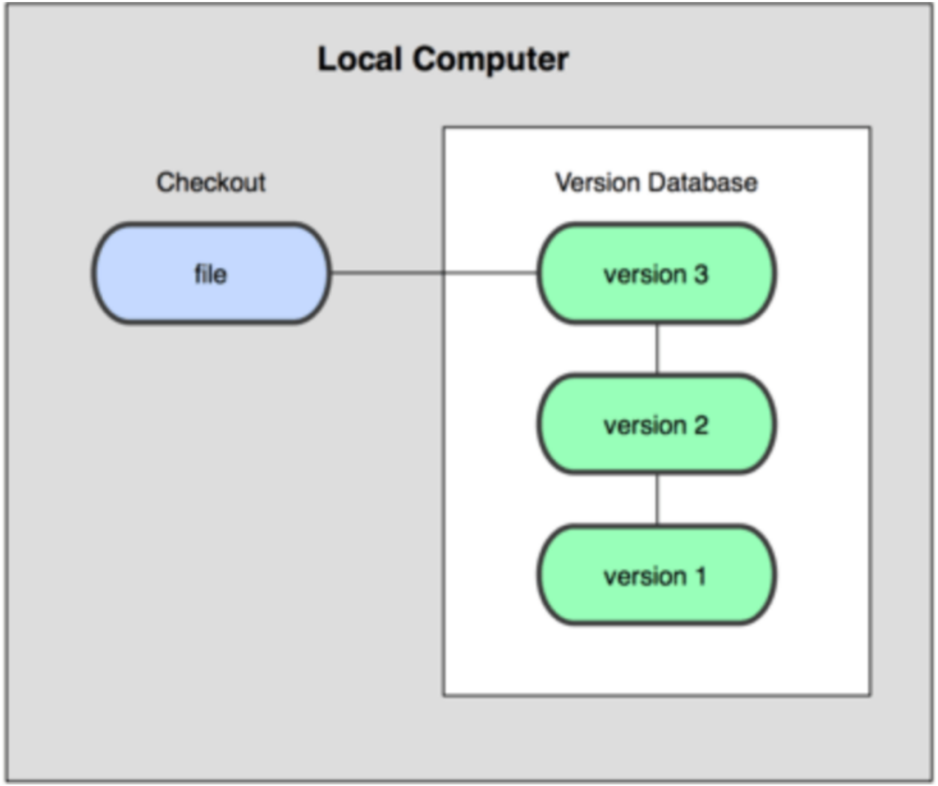
(Think of Time Machine for your Mac)

Version control systems (VCS) can be:

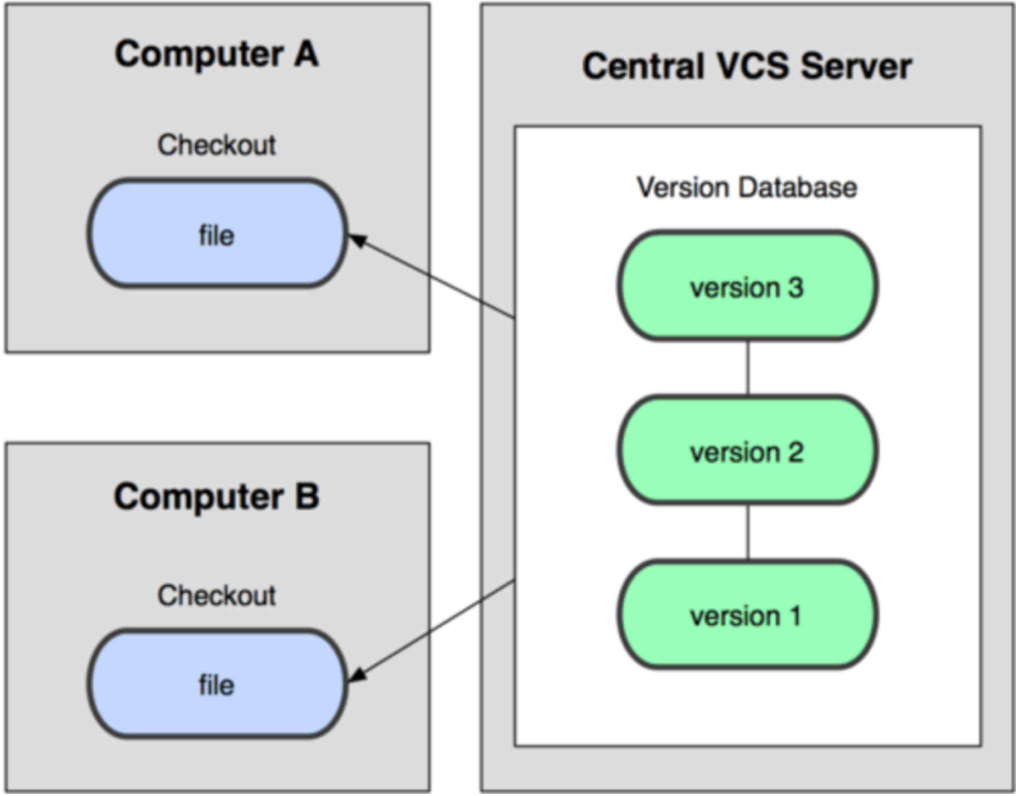
- *Local*
- *Centralized*
- *Distributed*

VERSION CONTROL

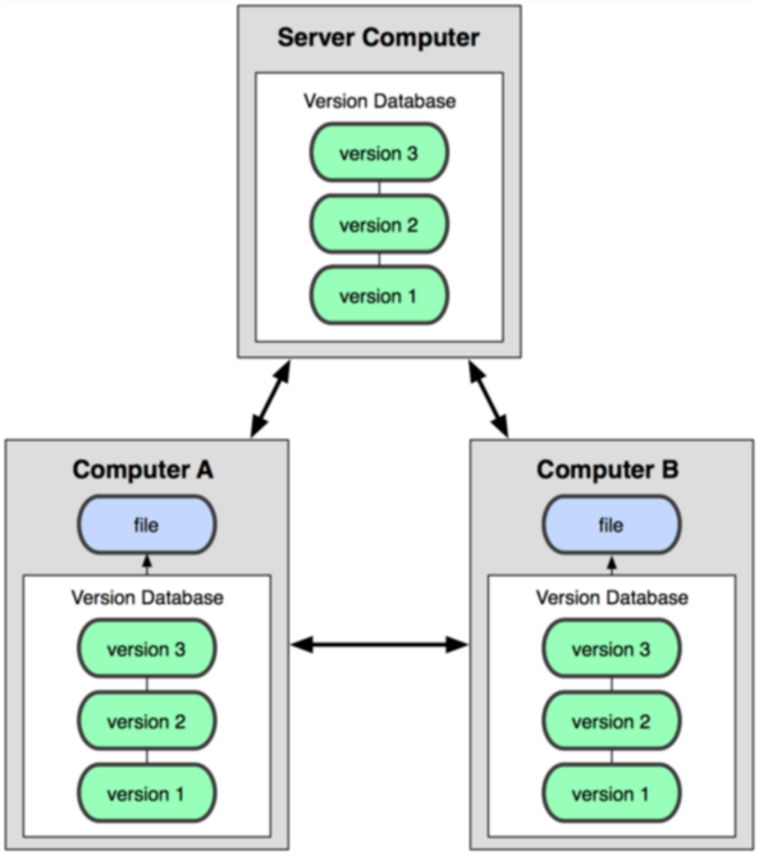
Local version control:



Centralized version control:

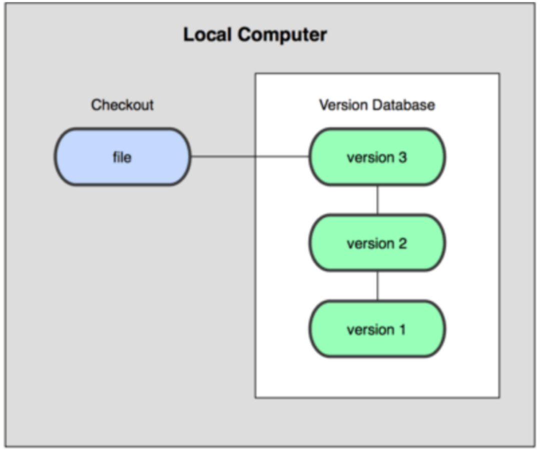


Distributed version control:

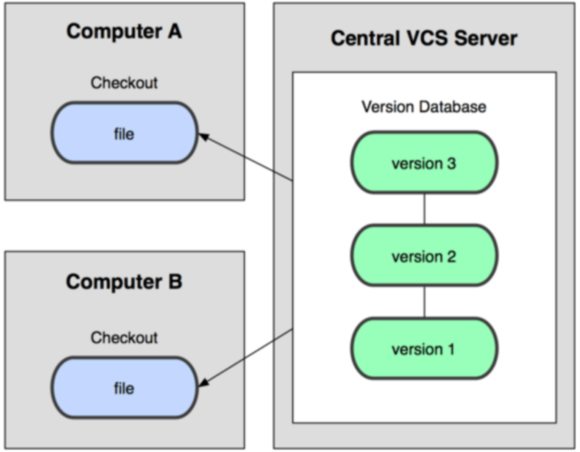


VERSION CONTROL

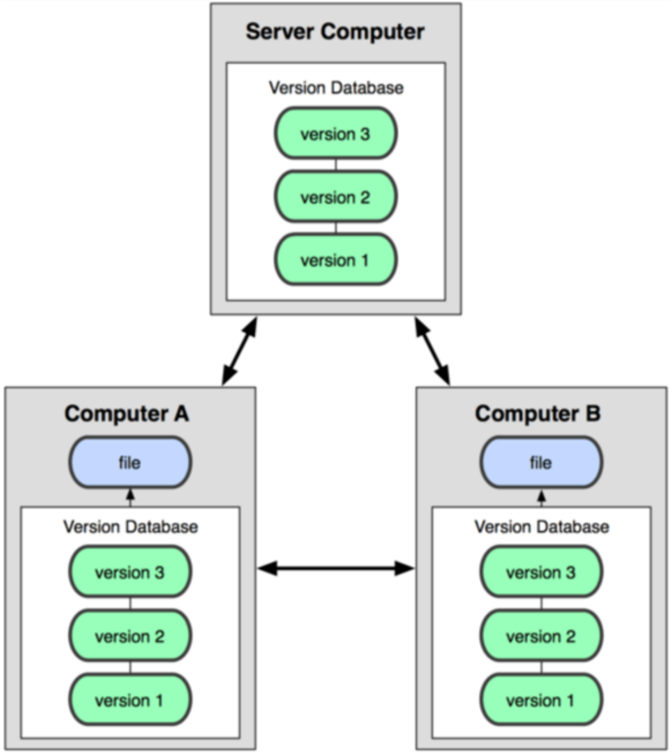
Local



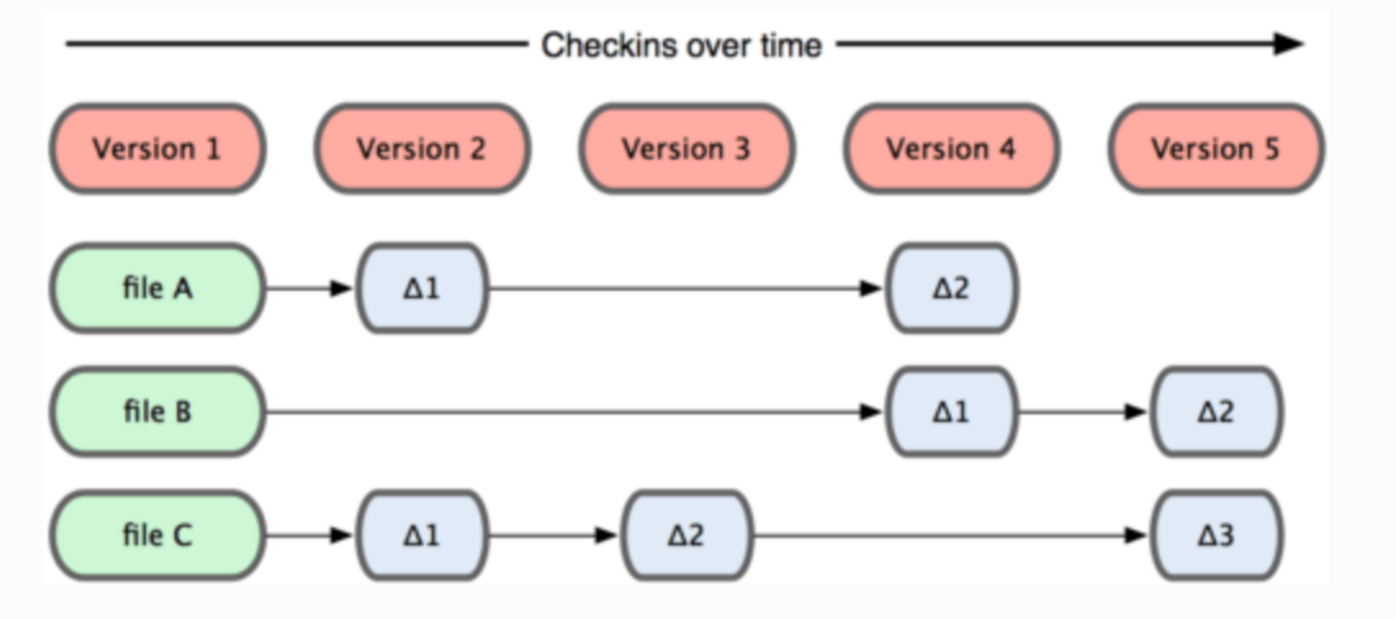
Centralized



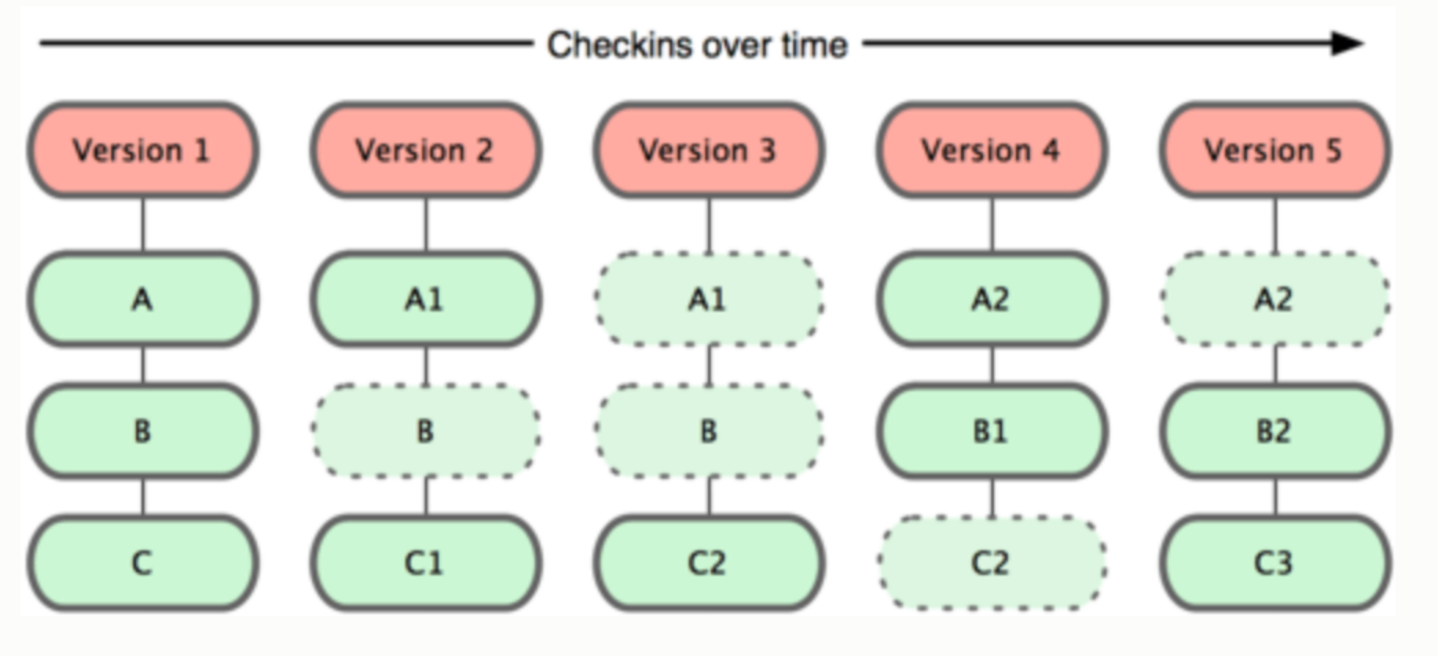
Distributed



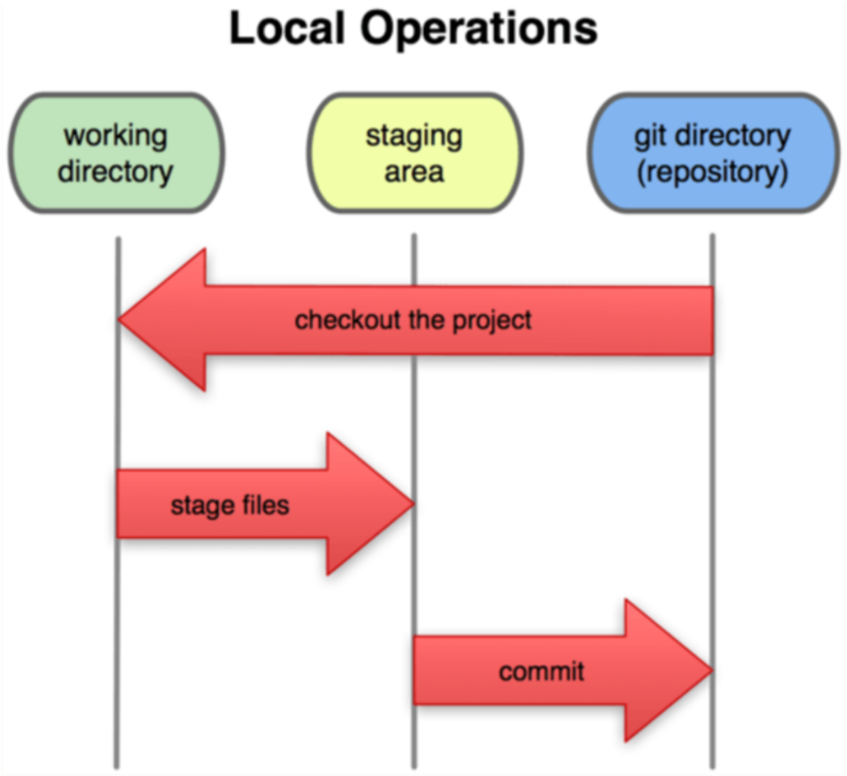
Traditional VCS' work in terms of files and differences



Git stores data as snapshots of a project over time



modified, staged, committed



The basic Git workflow

1. *You modify files in your working directory.*
2. *You stage those files by adding snapshots of the files to your staging area.*
3. *You do a commit, which takes the files in the staging area and stores that snapshot permanently to your Git directory.*

GIT COMMANDS

<u>Main</u>
git clone – clone a repo
git status – get status
git add – add changes to be pushed
git commit – commit the change with a comment
git push – push the change to github
git pull – pull remote changes from github

<u>Others</u>
git branch – see all branches
git checkout – checkout a branch
git merge – merge in another branch
git stash – stash changes
pull request – remote changes requested to be merged in