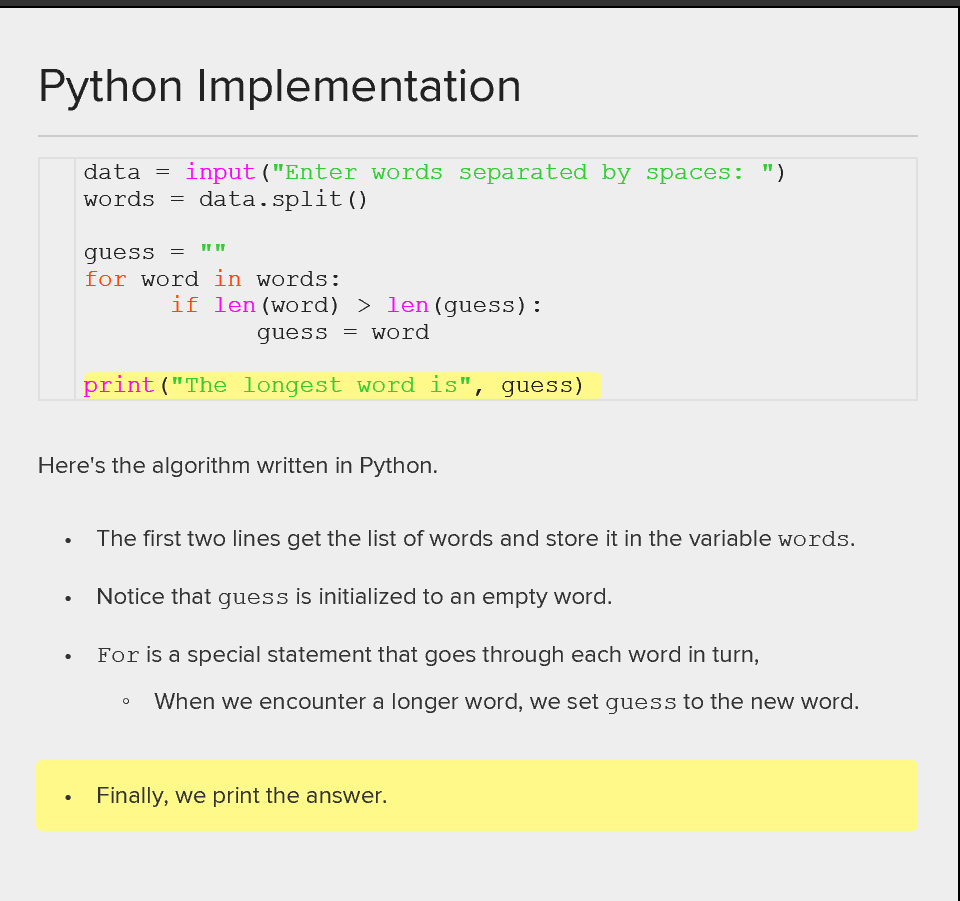
Reading Chapter 1

1.3

A program is a set of instructions we’ll pass to the computer to tell it what to do.

The plane for solving a problem is what we call an algorithm (abstract set of steps to solve a problem )



1.4

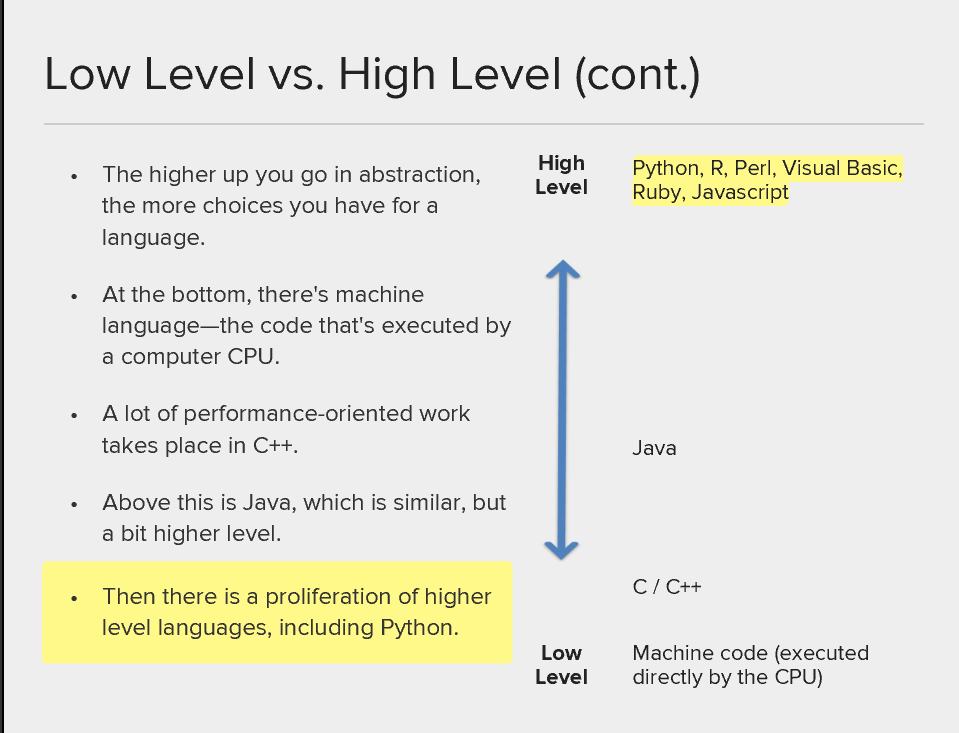
How to choose a language

1. Low level V.S High level language

Low level languages give you control over the detail of a computer. E.g memory allocation, memory addresses, processor sharing, computer hardware.

These languages run fast !!!

High level languages abstract away from these details, making many tasks automatic

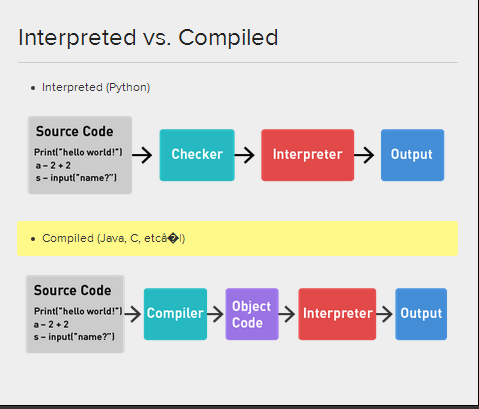


1. Machine code is the lowest level language. It is just sequences of bits. Usually, a programmer would program in a higher level language, like assembly language, that’s then compiled into machine language.

Higher level languages are easier to understand. They automate low level tasks for the programmer.

E.g memory is allocated automatically; you don’t need to know what memory address you’re using; when it is not needed any more, it’s automatically “garbage collected” so it can be used again (this means if you can use that memory again to store new data). High level languages also abstract away from the computer hardware. You don’t need to worry about different types of memory, different processors, where components are located (this is important because computer hardware is always changing, and your code may have to run on many different types of hardware.)

1. A computer is something that executes statements in a programming language. No matter what the underlying hardware, you can expect a computer to execute python predictably. The computer will have a program called the python interpreter, this program takes python statements and translates them into lower level instructions that may be specific to the hardware it is running on. The interpreter ensures that whatever hardware we are running on looks the same to us as programmers. The interpreter is the face of the computer that we see.
2. Interpreted and compiled.



Because of the object code, the complied languages and be a lot more faster

While interpreted languages can be more flexible

1.5 The world of programming languages

1. Shell script:

Is a specialized language that lets you interact with your operating system by typing commands into a terminal.

You can do basically everything you can do on your normal computer e.g. move files around, create directories, start other programs, etc. this was once he only wat of doing these tasks. You may be used to doing these things with a graphical windowed interface but the shell is still around and is useful for many things. We can be very precise about what we want and automate some low level tasks.

Depending on your operating system, there may be variations in the commands and format that you will use.

1. Data oriented languages.

Specialized languages including R Stata SPSS SAS. These languages treat data as a principal unit of analysis. They do what they do well but struggle to do more. Besides data analysis, other tasks may be difficult to perform (or slow or messy…)

1. C++(and its predecessor, C) is one of the most popular general purpose languages. It is used in many places where speed is very important. There is a lot more code than in Shell Script or R. (We have to declare a variable to allocate memory before we can use it. It is statically typed which means once the variable type is declared, it can never hold another type of data). This helps performance but reduces flexibility.
2. Java

is a higher level than C++ but is also statically typed. It is very popular in commercial and enterprise applications. It is very verbose, you must explicitly state how different program components interact with each other. This is restrictive but makes for organized code.

1. Higher level languages (perl, ruby, (web development in Ruby and Rails), python…)

Dynamically typed, flexible and make for fast development

1. Lisps

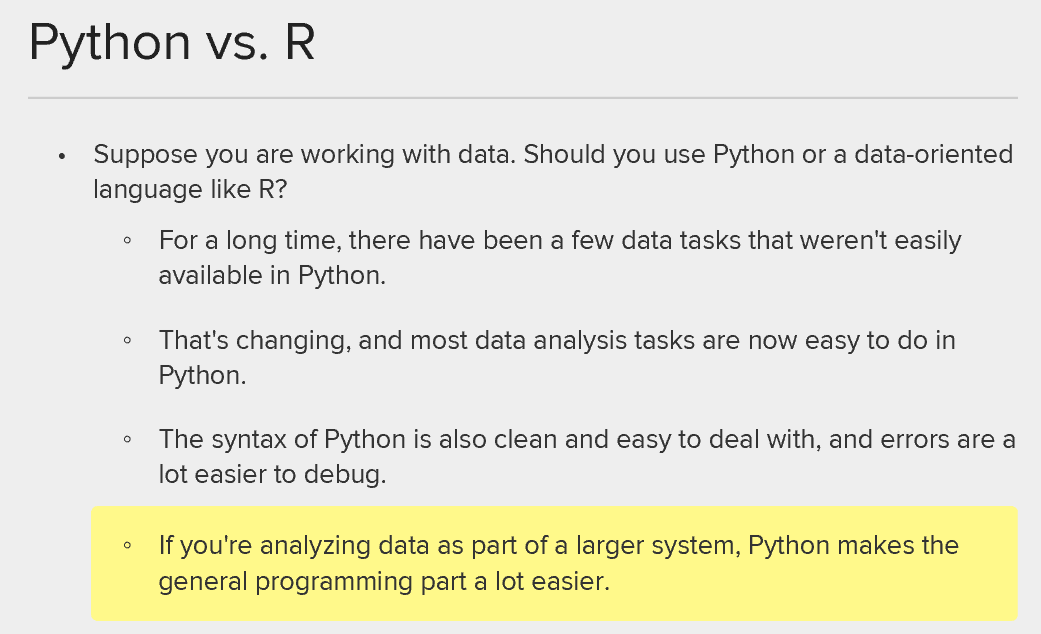
Is the second oldest high level programming language used today. It uses lots of parentheses and Polish notation. Lisp stands for LISTt Processing. It has a steep learning curve but is very expressive.

1. Python

Is among the highest level languages. It has gained popularity rapidly since it is so fast to develop in and programmers generally like using it.

Easy to learn, fast to develop, used in many commercial applications, simple syntax that “gets out of the way”

However, worse performance than C or Java, but the interpreter has been optimized a lot, and the gap is not significant for most applications



The answer is unknown!

1.7 the command line

pwd : print working directory

ls : list all of the files and folders in the directory

man ls : instructions on what is available to you

type q to quite

/F perform a search like ctrl F

^F means you need to hit ctrl to activate this command

**clear** : clear the page

**cd ~/Desktop** : change to working directory to the desk top (~ stands fro the home folder )

**mkdir test\_creation :** create a folder under desktop named test\_creaton

**cd test\_creation** : change into this working directory

**touch my\_file.txt** : create a .txt file named my\_file under the test creation folder

**echo “Hello From the Command line” >> my\_file.txt** : put this sentence into my\_file.txt

the >> means append

**mkdir test\_internal** : create another folder called test\_internal inside test\_creation

**mv .\my\_file .txt .\test\_internal** : this is to move the my\_file.txt into the folder test\_internal

**echo “Hello, this is my second echo command” >>.\test\_internal\ my\_file.txt** : put this sentence into my\_file.txt

**cat .\test\_internal\my\_file.txt** : this print out the content in my\_file.txt

**cd ~\Desktop :** change to working directory to the desk top

**echo "second call file ">> C:\Users\hrli1\Desktop\test\_creation\test\_internal\my\_second\_file.txt**  : put this sentence into my\_second\_file.txt using absolute directory . Note, even though my\_second\_file.txt does not exist, now, the computer will create one and then write the file

**cat .\my\_file.txt .\my\_second\_file.txt > third.txt**  : overwrite the two files to the third.txt

the > means overwrite

**cat .\third.txt > .\my\_second\_file.txt :** overwrite my\_second\_file.txt such that it is now the same as the third.txt

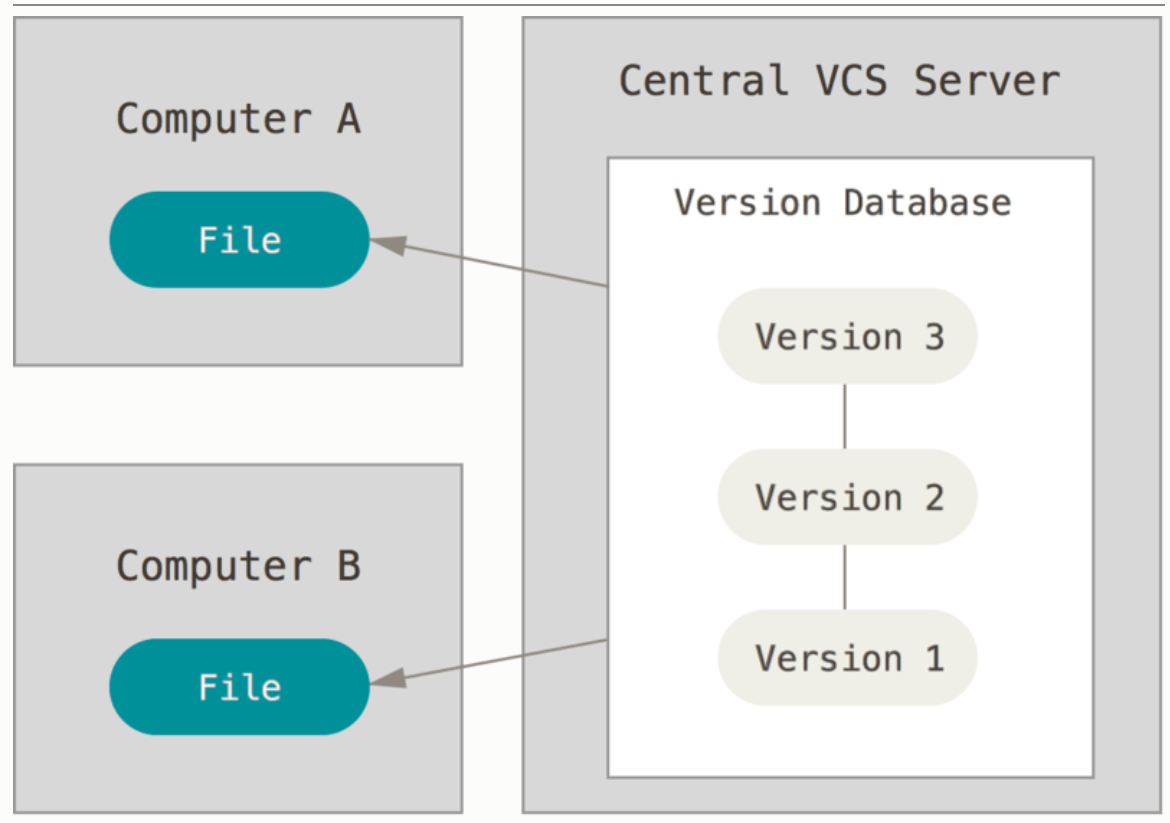
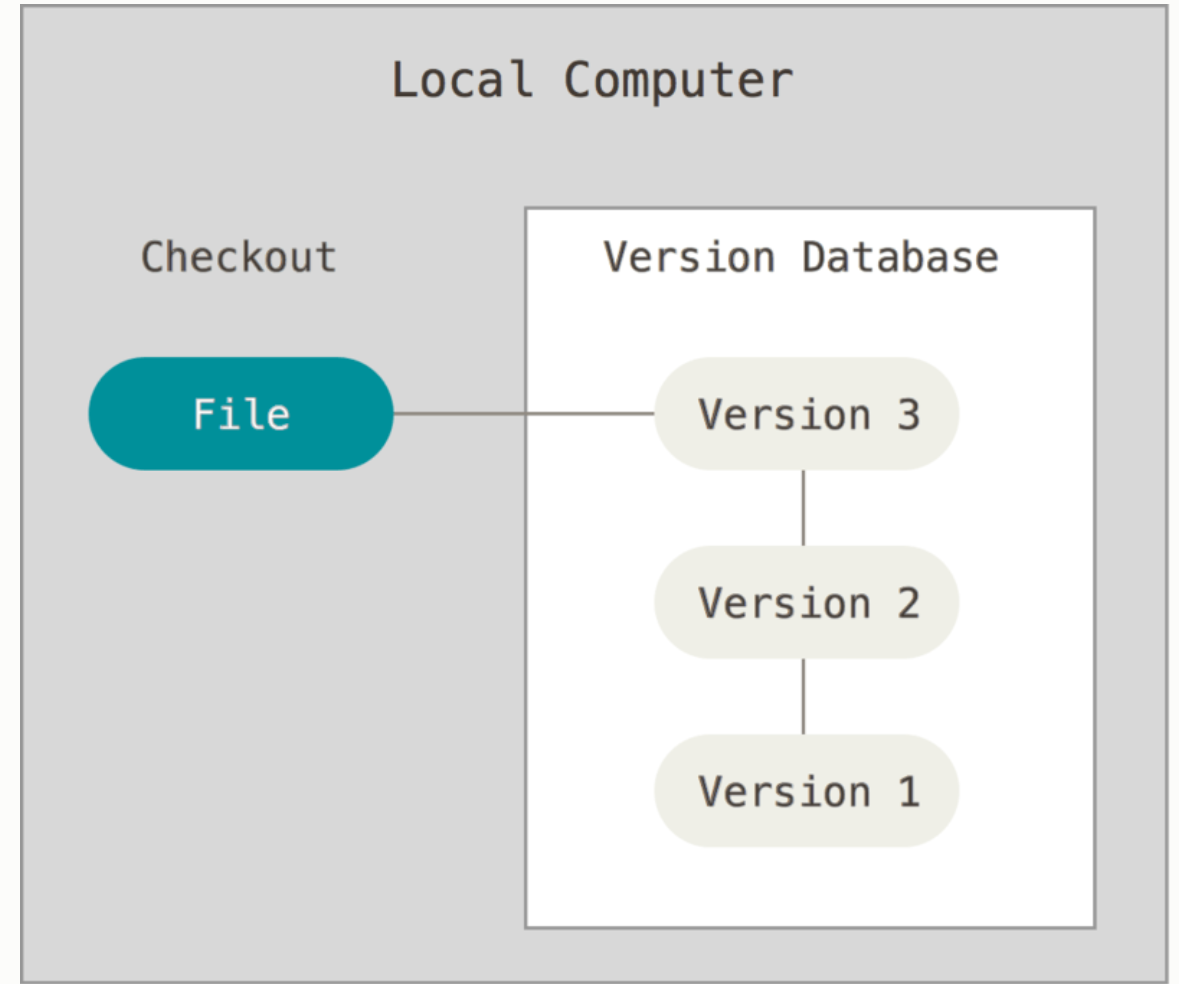
**cp**  **.\third.txt fourth.txt :** copy the third.txt and name is fourth.txt

**cd .**. : go to one folder up (parent folder)

1.8 source control

Source control management or version control is a system for organizing different versions of a project under development

Local version control system: To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control.



Centralized version control systems

**These store all project files in one central location, and allow different people to check out files from that one place to work on them, the most common example is SVN (Subversion)** This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it’s far easier to administer a CVCS than it is to deal with local databases on every client.

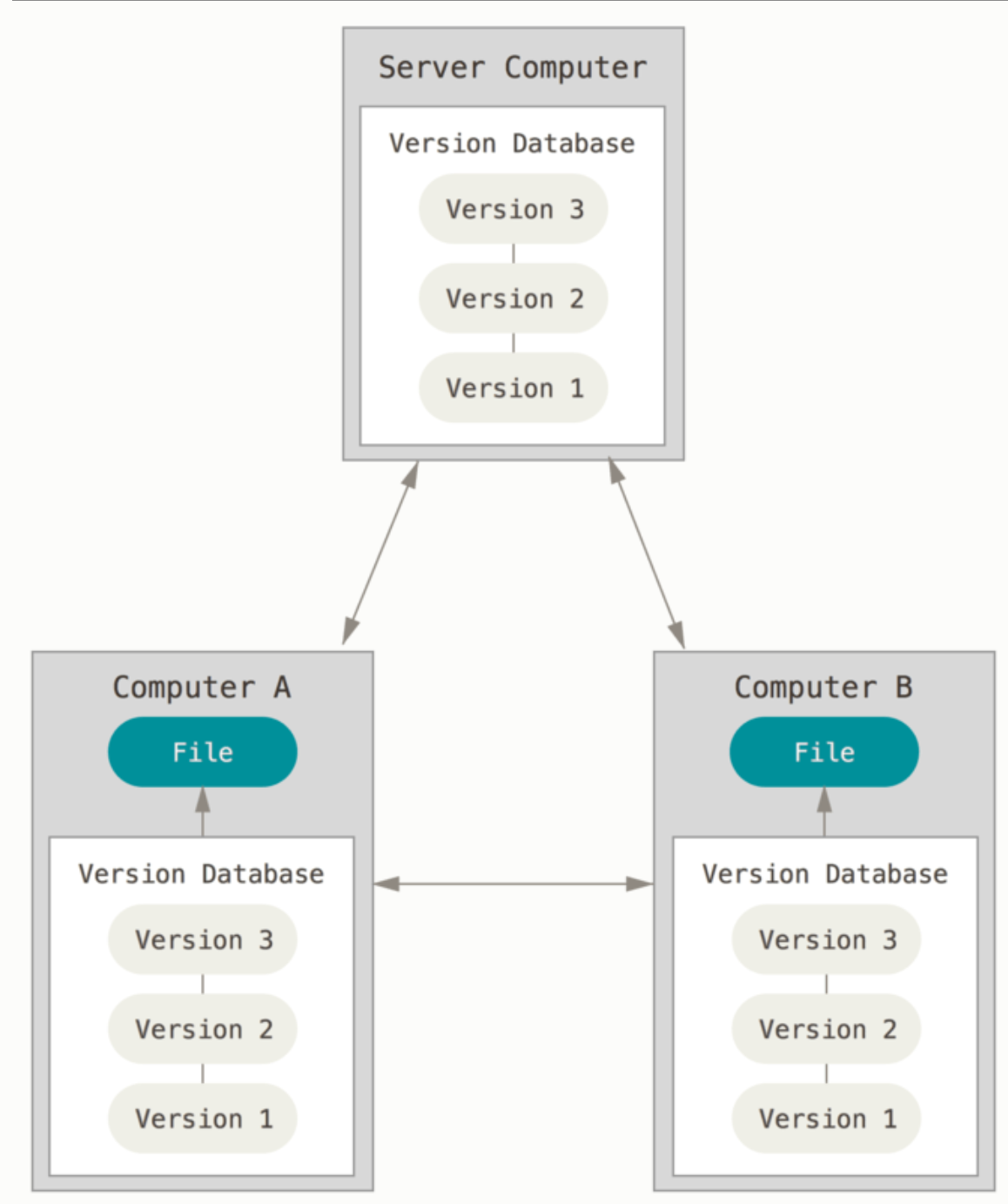
However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines. Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

Distributed Version control systems

**This means that everybody working on the project has a full copy of every version of every file. This is a relatively robust system because there is no one single point of failure. E.g Git and Mercurial.**

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files: they fully mirror the repository. Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data.

Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.



1.12 Starting out with Git

**cd C:\Users\hrli1\Desktop\**

**mkdir git\_test**

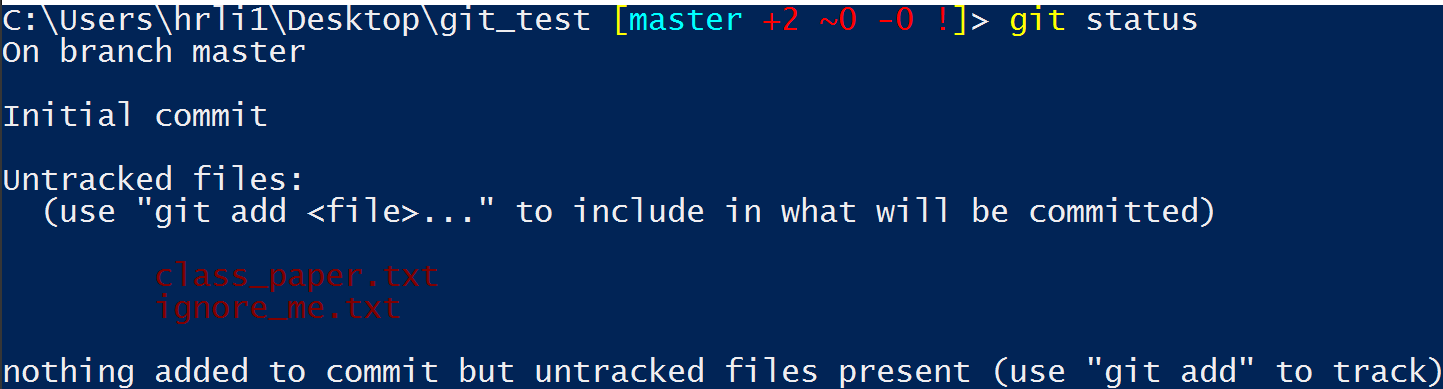
**cd .\git\_test\**

**git init** : initialized empty git repository in the working directory, this create a .git folder (hidden) in the git\_test folder

**touch class\_paper.txt**

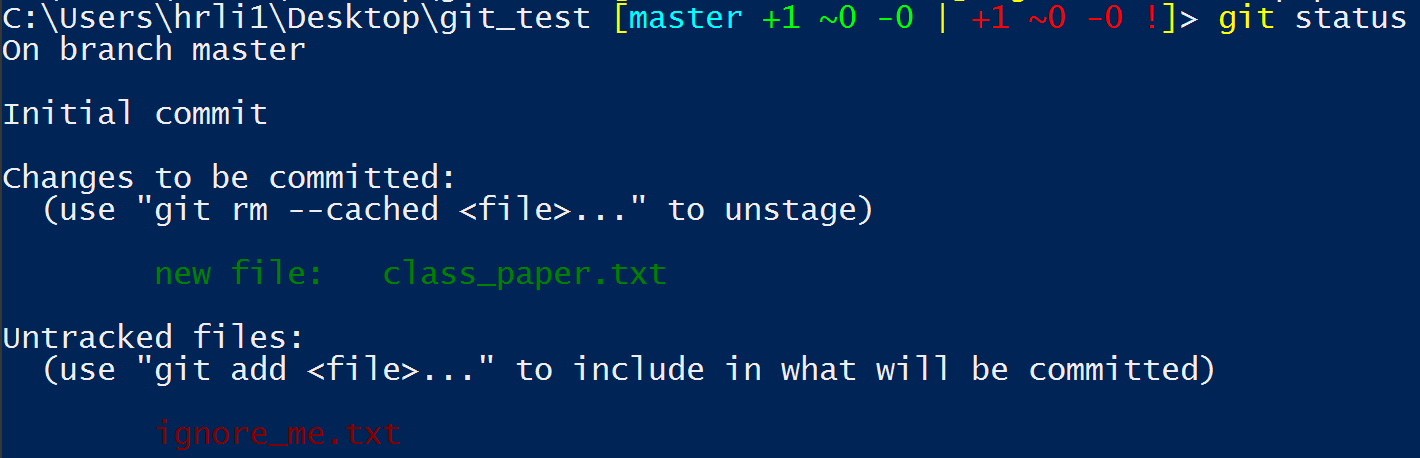
**touch ignore\_me.txt**

**git status** : show the status

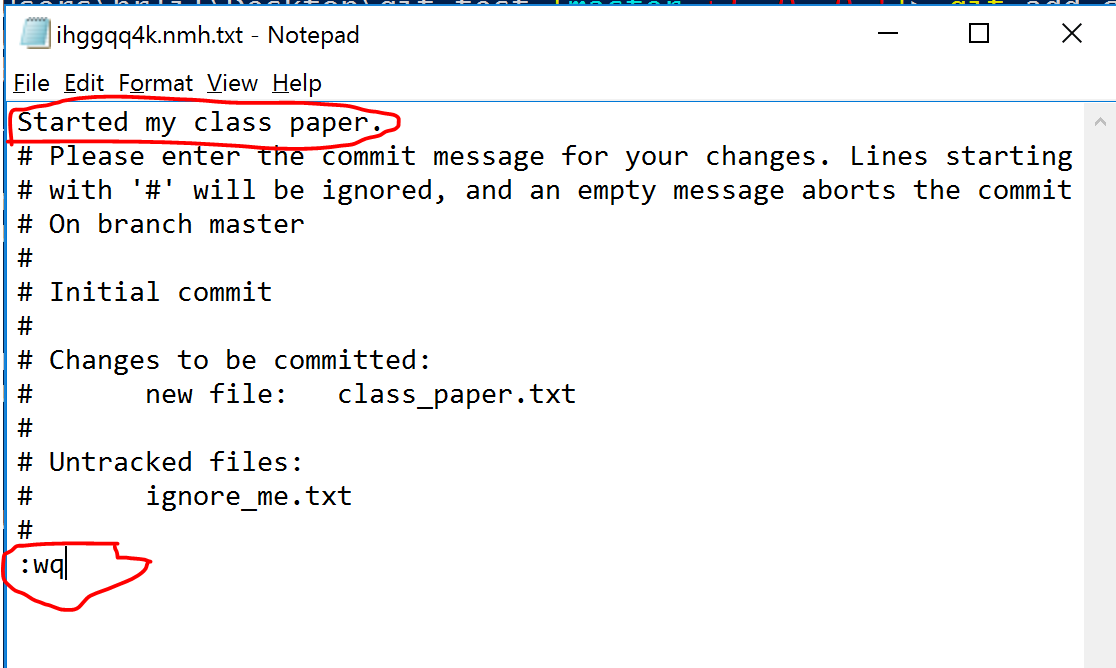


**git add class\_paper.txt** : in order to commit the changes made to the class\_paper.txt, we need to first add it to git so that git is aware of it.

**git status** : show the status



**git commit** : this commit the change and opens up a editor so that you can put some commitment message. “Started my class paper” was added to the commitment message and remember to put **:wq** before save and exit



Now if I type **git status**, it will only give me ignore\_me.txt as an untracked file

We have permanently saved the new version of class\_paper.txt, so we can always go back and take a look at this version if we have to.

If we open class\_paper.txt and type some new things and save it and we run **git status,** it will show that the class\_paper.txt is modified

**git add class\_paper.txt**

**git commit -m "Added second line of paper"** :this is another way to add this commitment message without opening up the vim editor.

If we want git to ignore or not track the change of some file we can do:

**touch .gitignore**

**echo "ignore\_me.txt" >> .gitignore**

We want to keep track of what file is ignored so we commit .gitignore as well

**git add .gitignore**

**git commit -m "added .gitignore"**