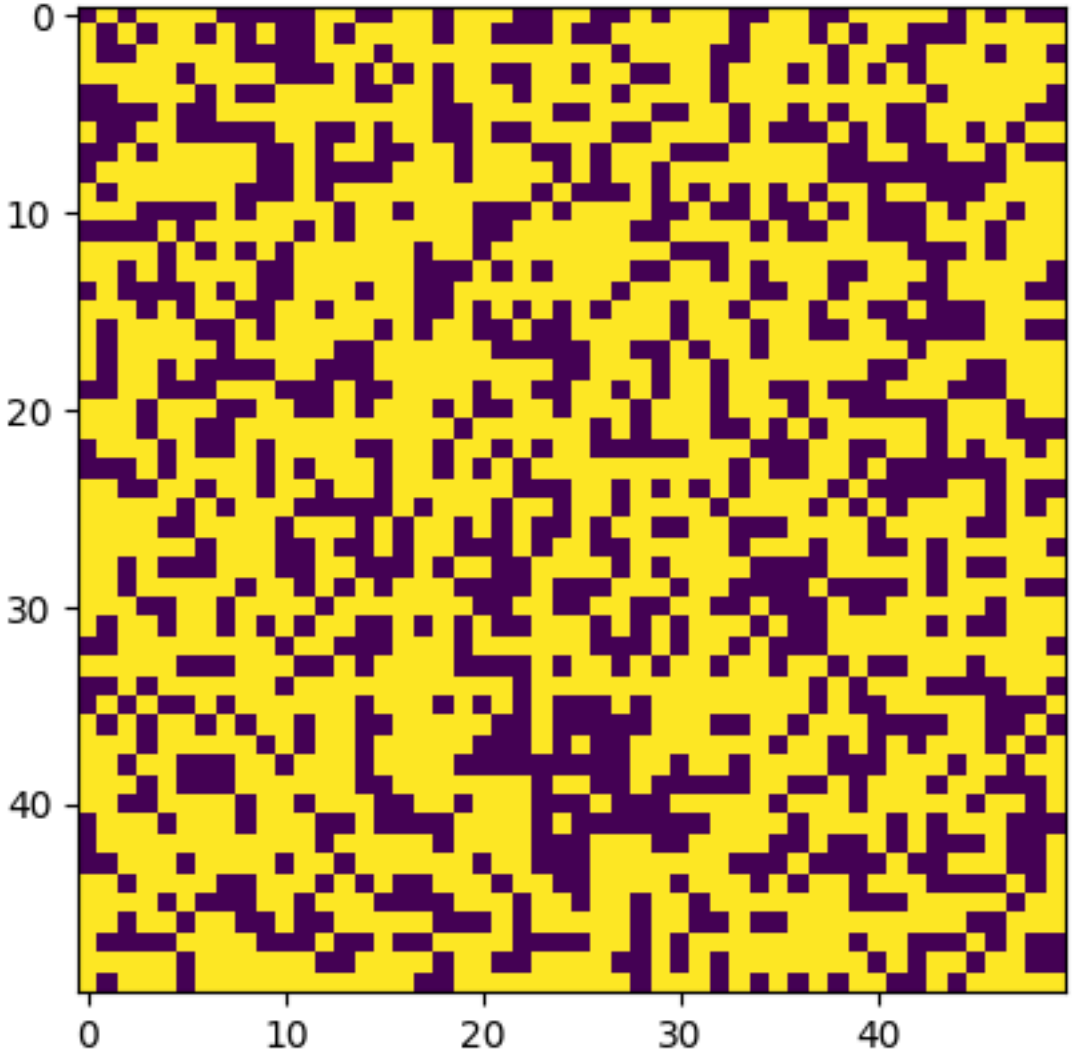
One last topic

We are small enough - time to introduce yourselves (and apologies in advance when I forget your name)

Any questions?



Where you will do your work?

Ideally you would be working on a UNIX/LINUX command line. Some of you likely use Windows. This is fine, and I'm not going to make you dual-boot your computers, and as such, not going to teach you any Linux shell commands

How will you do your work?

Your assignments can be handed in using **Python** or **C++**. If you are only comfortable with a different programming language, come talk to me, since C++ is a pre-requisite for this course. Matlab is not an option. Note that **Python is the recommendation for what you use!**

How will you submit your work?

We will use github classroom for assignments. We will briefly cover how to use github, which anyway is very useful to know these days

If you have a github account

Great!

If you do not have a github account

Ungraded homework for today (but do it today!) is to go to github.com and create an account. It's pretty straightforward

How will you do your work?

If you want to work in C++, by all means, please do so. That is great. We will often discuss things in pseudo-code in this course, but examples will be in Python.

If you want to submit C++ code for assignments, each submission then has to have a Makefile that compiles on my laptop, and you are responsible for finding any needed visualization packages. It's your choice.

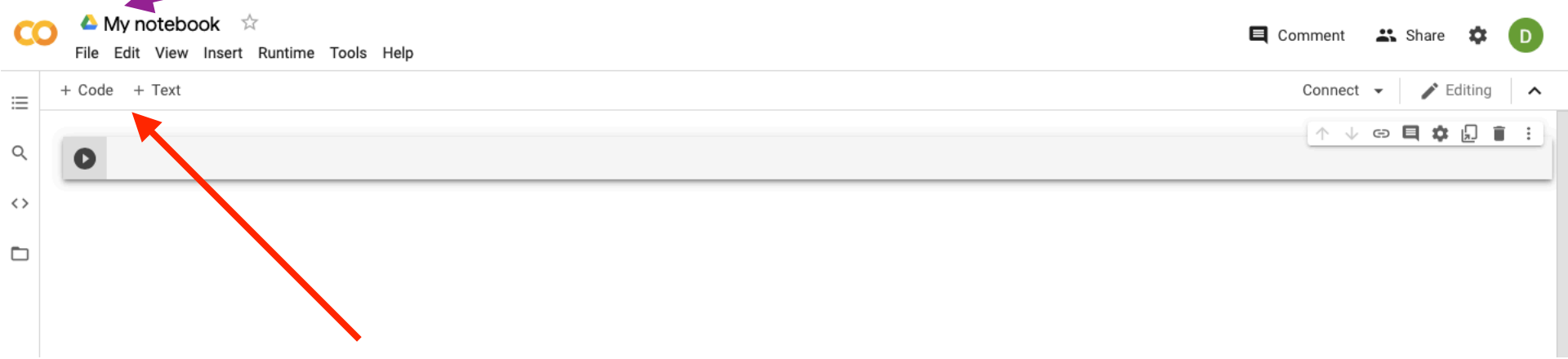
Let's all now visit https://colab.research.google.com/notebooks/intro.ipynb

You need a google account to use this. If you don't have one, please create one (and feel free to then use gmail and complain like I do about O365 and Outlook)

Please review the material there before the next course. It's straightforward (there's also a nice video). These are "Jupyter notebooks" that are hosted on Google Colab servers

Our colab notebook

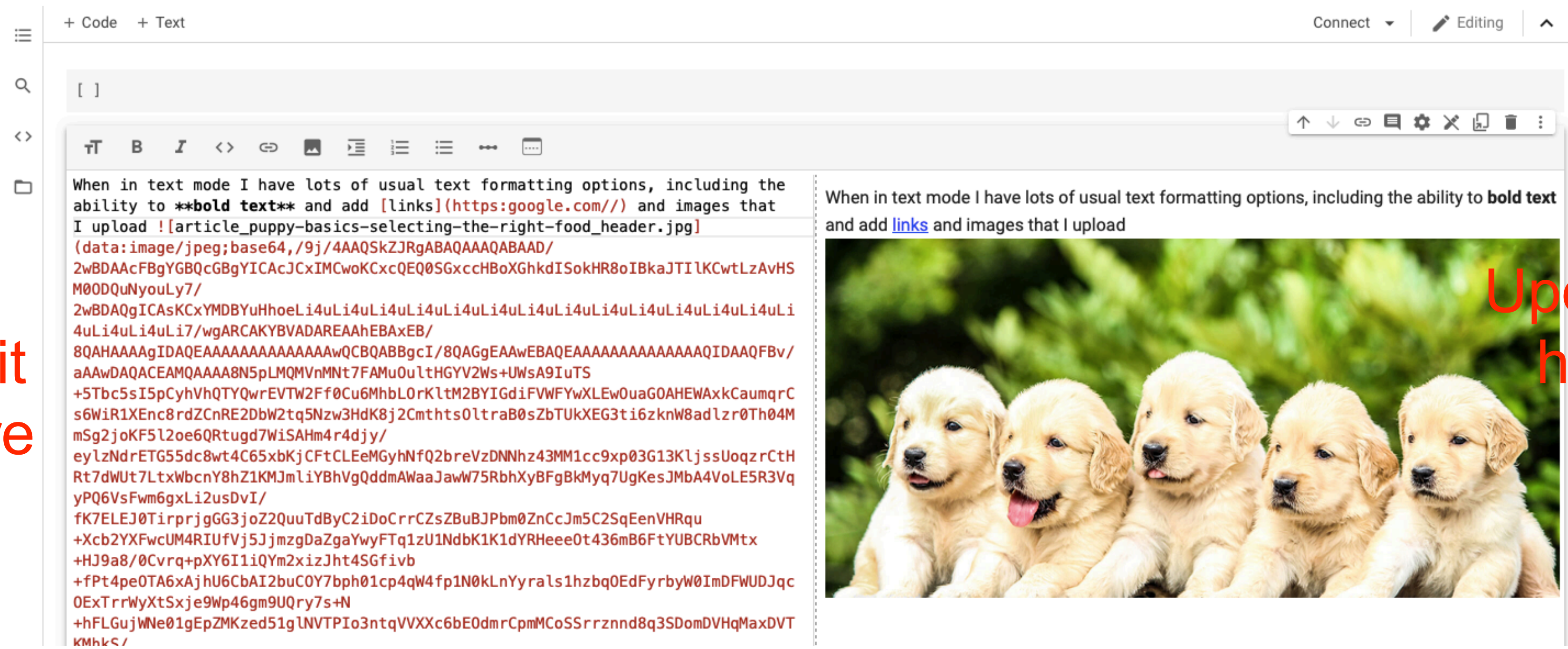
Name of my notebook (can be changed), and you can see from the logo that it is currently stored on my google cloud drive



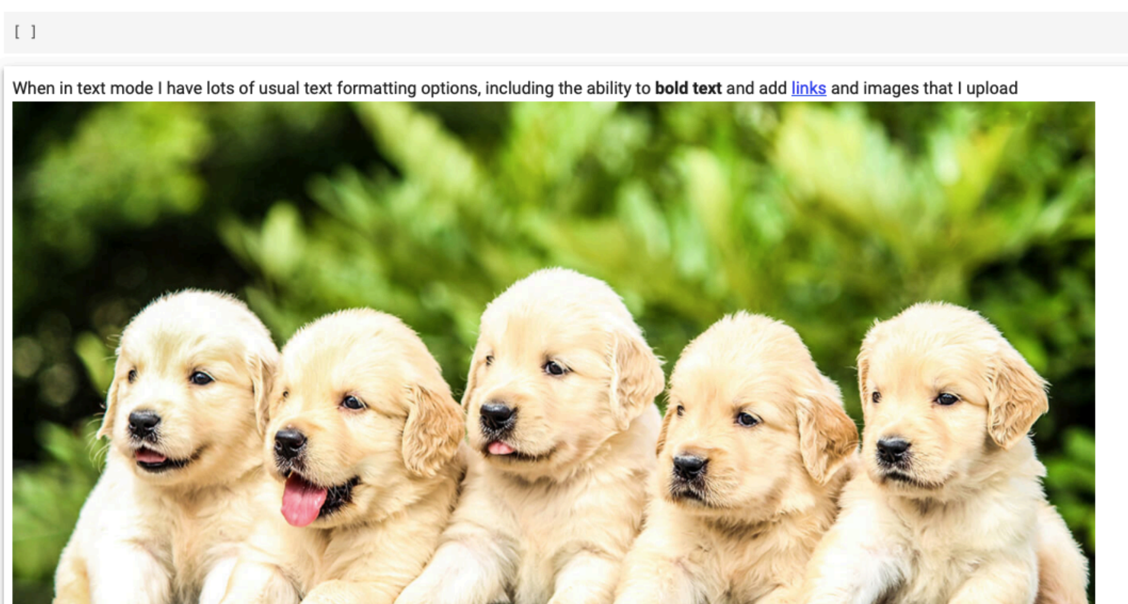
I can choose whether I am inserting text or code (so that you can nicely document code with formatting!)

Colab text

Edit here

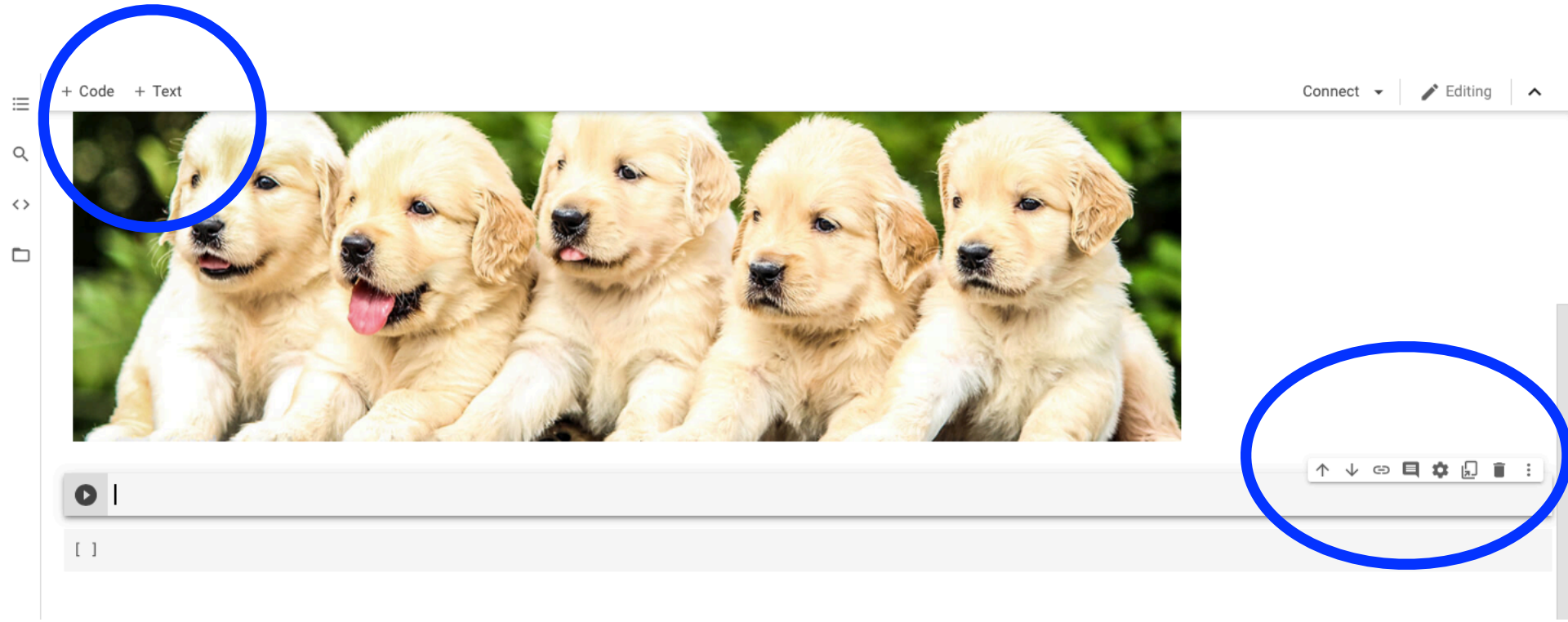


Updates here



When we're all done (we can always edit it again later)

Colab code

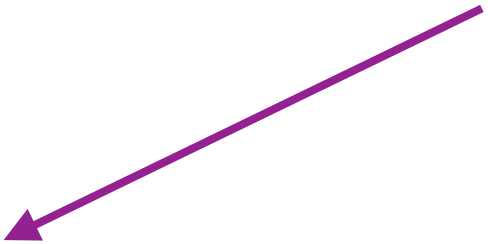


Next I clicked on “+ Code” to add some code. The **cell** gets placed below the last one. But note that I can delete these cells, you can have multiple cells of the same type in a row, or you can move them around. And you can type whatever code you want here

```
[6]  This is not good code, it should not work
```

How to run code

Hit SHIFT-Enter or else click the “Play” button and then wait



```
[6]  This is not good code, it should not work
```

```
This is not good code, it should not work

  File "<ipython-input-6-a7b6366c138d>", line 1
    This is not good code, it should not work
                  ^
SyntaxError: invalid syntax

SEARCH STACK OVERFLOW
```

Not surprising!!

If your code doesn't run, edit the same cell and try again


```
[7]  print("This is now good code, it should work")
```

```
This is now good code, it should work
```

```
[3]  x=5
     print(x)
```

```
5
```

```
[4]  print(x)
```

```
5
```

When code runs it moves to the next cell. Note that print(x) works in the next cell. Why?! Only one cell at a time runs but anything you import and load, and any variables and functions that have been defined, remain in memory!

```
[8]  def func(val):
       print(val*val)
```

```
[9]  print(func(4))
```

```
16
None
```

Makes it easy to "split up" your code and make small changes one piece at a time

Great use of colab

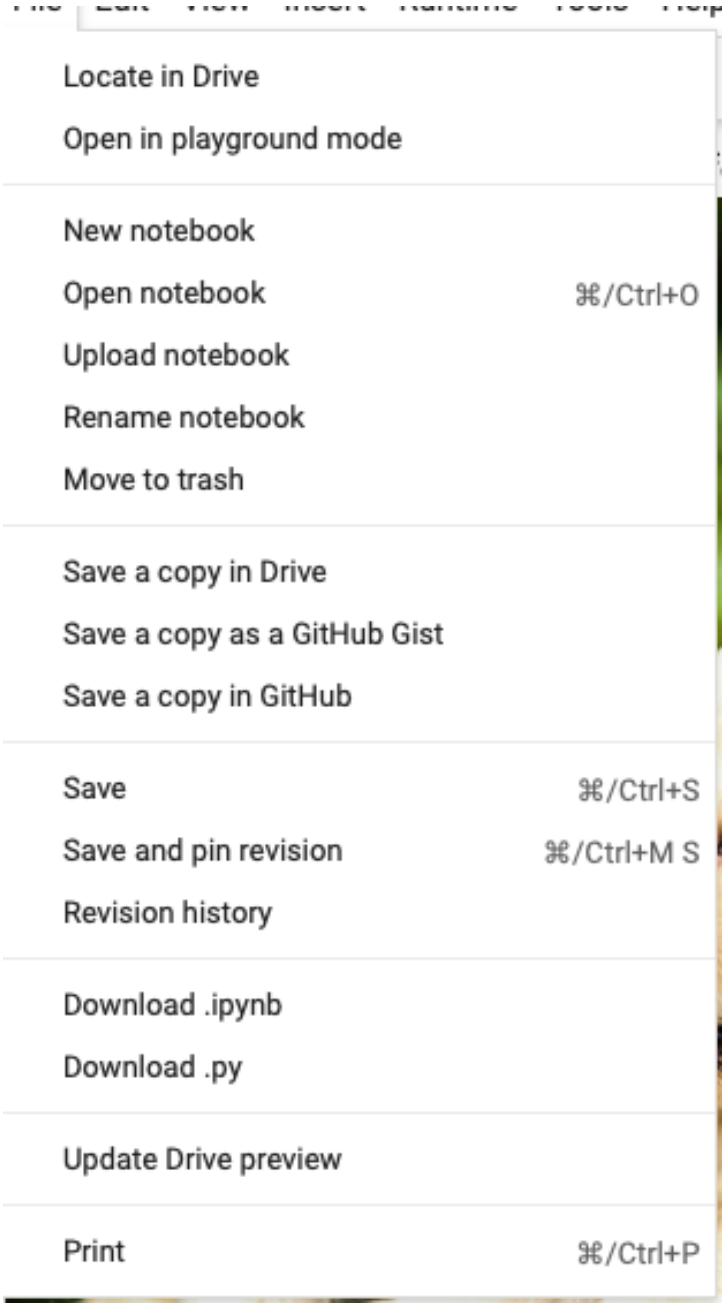
Tired of doom-scrolling? Just load this up and hit play (let's try):
https://colab.research.google.com/github/NIUCompPhys/Spring2024/blob/main/2025.ipynb

```
import datetime
import time
import IPython

def increase_font():
  from IPython.display import Javascript
  display(Javascript('''
  for (rule of document.styleSheets[0].cssRules){
    if (rule.selectorText=='body') {
      rule.style.fontSize = '30px'
      break
    }
  }
  '''))

IPython.display.clear_output()
increase_font()
out = display(IPython.display.Pretty('Calculating'), display_id=True)
while (1):
  time_now = datetime.datetime.now()
  time_2025 = datetime.datetime(2025, 1, 1)
  diff = time_2025-time_now
  seconds = int(diff.total_seconds())
  text="Terrible news! There are still "+str(seconds)+" seconds left until 2025"
  out.update(IPython.display.Pretty(text))
  time.sleep(1)
```

And if you are submitting C++?

Play along with colab, and submit C++ code with a makefile as mentioned in github. Though I strongly urge you to learn Python, too

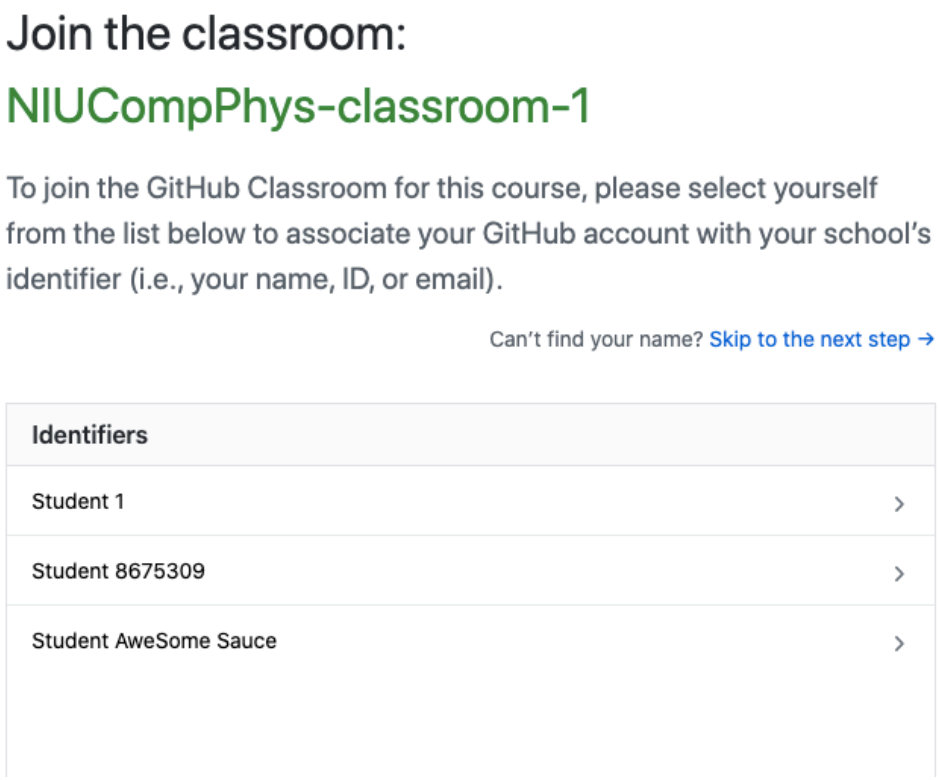
Don’t forget to save your code



Under the file menu you have the usual list of options for opening new notebooks, downloading or printing them, saving them to Google Drive …. or saving to GitHub!

OK, let's get back to github

https://classroom.github.com/classrooms/67593652-computational-physics-2024

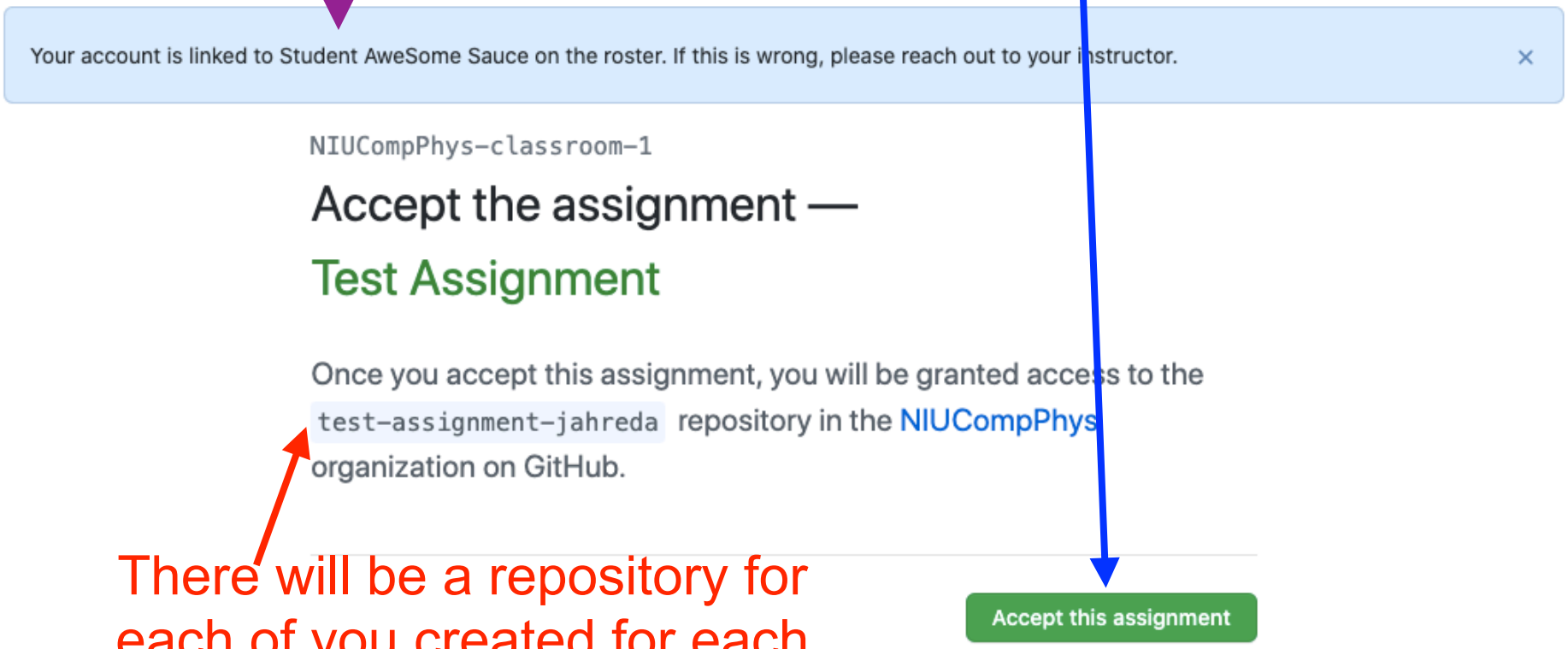


Once we log in to our github accounts we need to link it to a student identifier. Pick your name! I added you all before the start of class, but if you aren't here, skip the step and let me know

OK, let's get back to github

This should be your name!

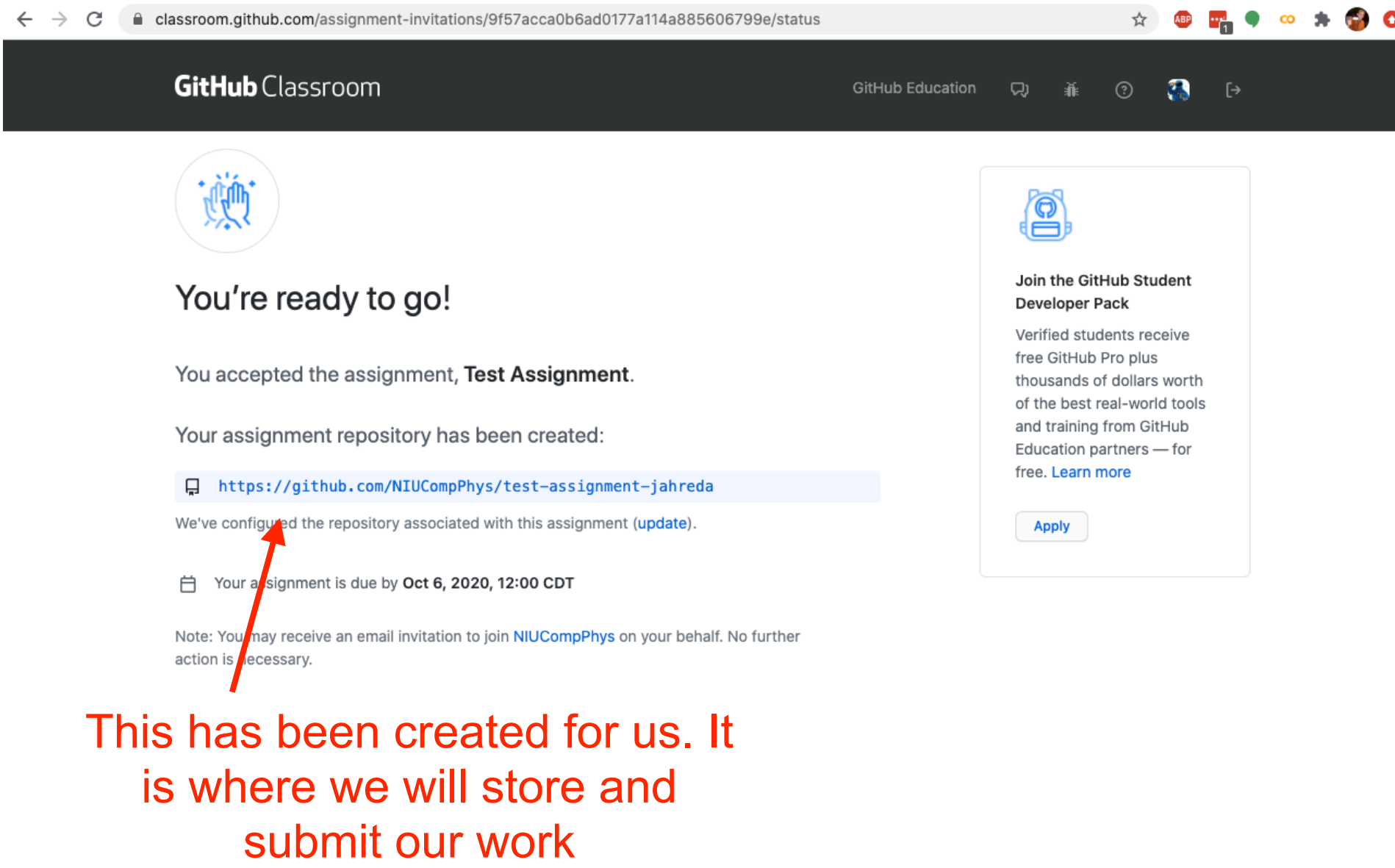
Accept this assignment!



There will be a repository for each of you created for each assignment

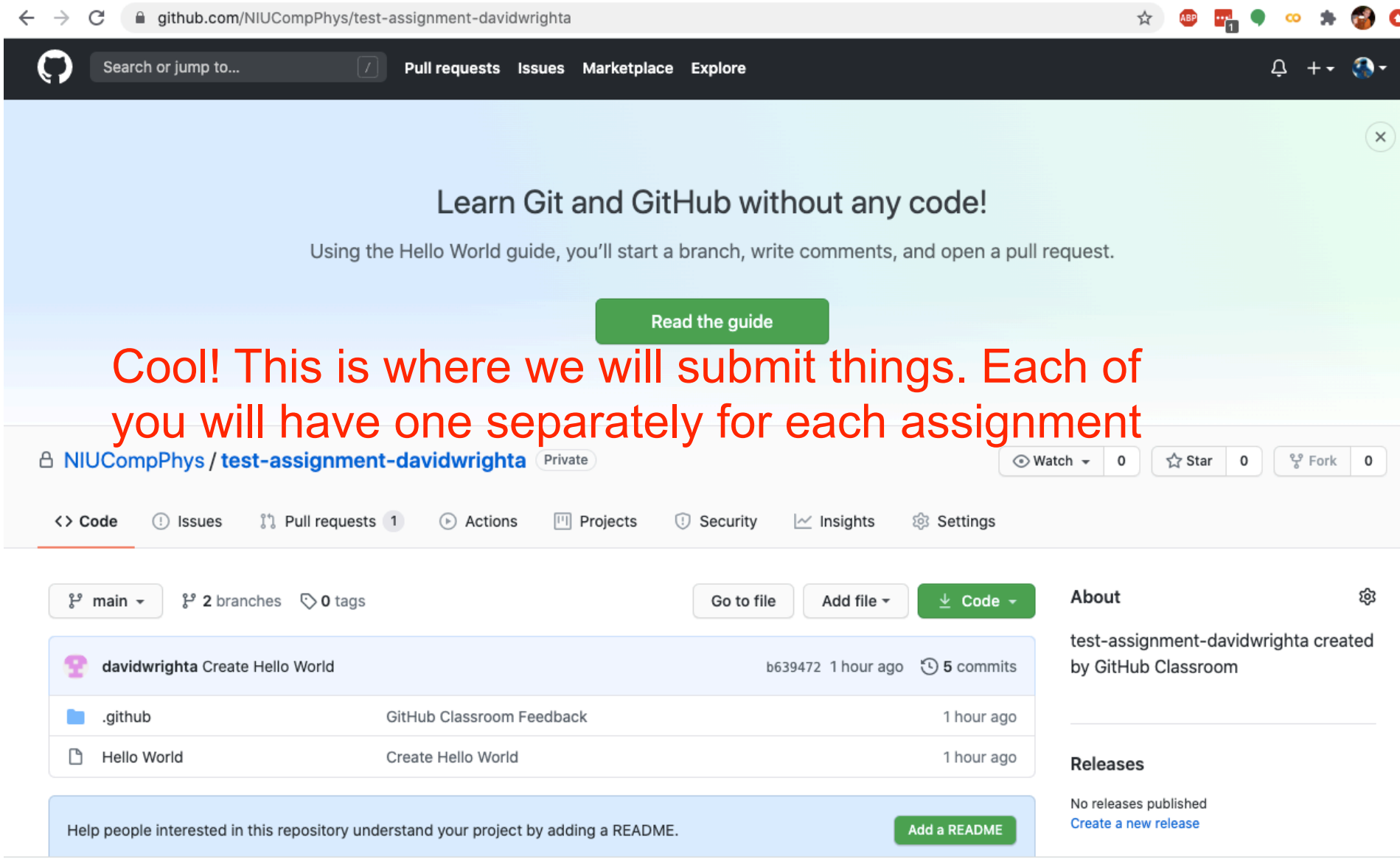
Accepting the assignment is not as bad as this

We accepted it!

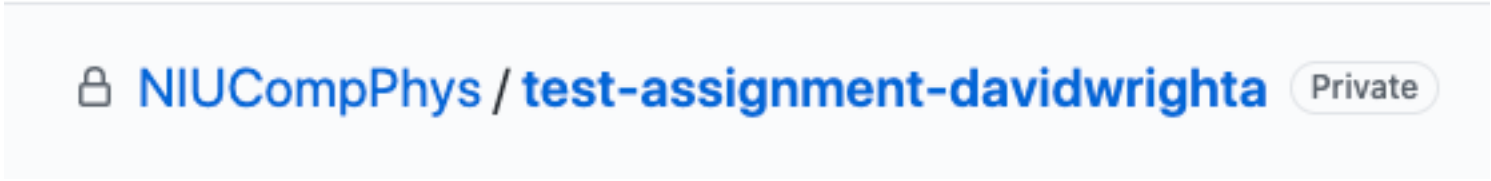


This has been created for us. It is where we will store and submit our work

OK, we can now view our assignment



We can make a copy of it locally (clone it) if we want



Cursor over test-assignment-davidwrighta (another example student) gives you the link you want for “git clone”

```
~/Desktop % git clone https://github.com/NIUCompPhys/test-assignment-davidwrighta
Cloning into 'test-assignment-davidwrighta'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 10 (delta 3), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (10/10), done.
~/Desktop % cd test-assignment-davidwrighta
~/Desktop/test-assignment-davidwrighta % ls
Hello World
~/Desktop/test-assignment-davidwrighta %
```

Workflow on git

```
~/Desktop/test-assignment-davidwrighta % more Hello\ World
Hello world and NIU
```

I've edited my file to include some important updates (you will update with code). Check that there are changes to update with `git status`

```
~/Desktop/test-assignment-davidwrighta % git status
On branch main
Your branch is up-to-date with 'origin/main'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   Hello World

no changes added to commit (use "git add" and/or "git commit -a")
~/Desktop/test-assignment-davidwrighta %
```

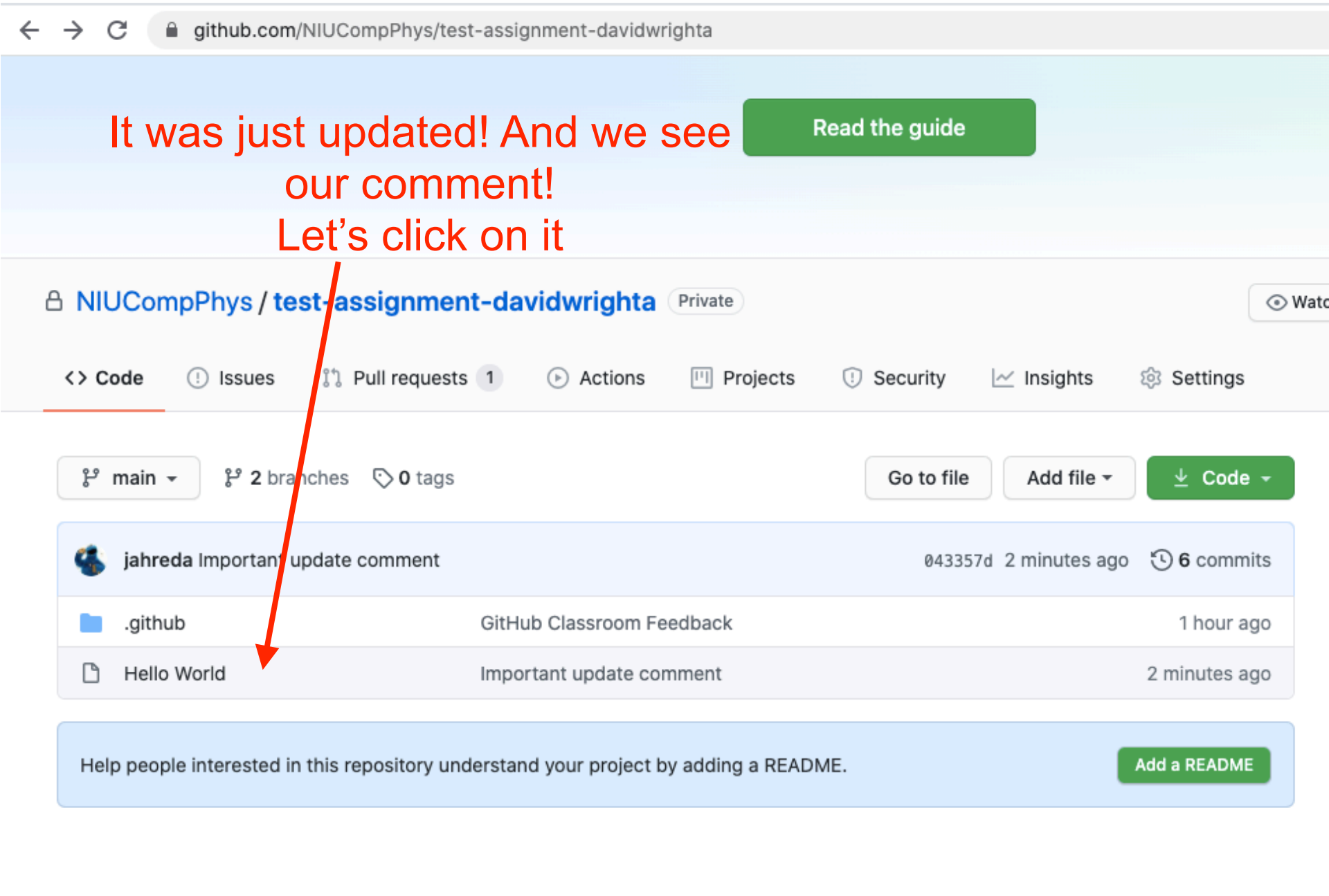
Add/commit/Push

Want to add our files to change
Then commit them
Then push them

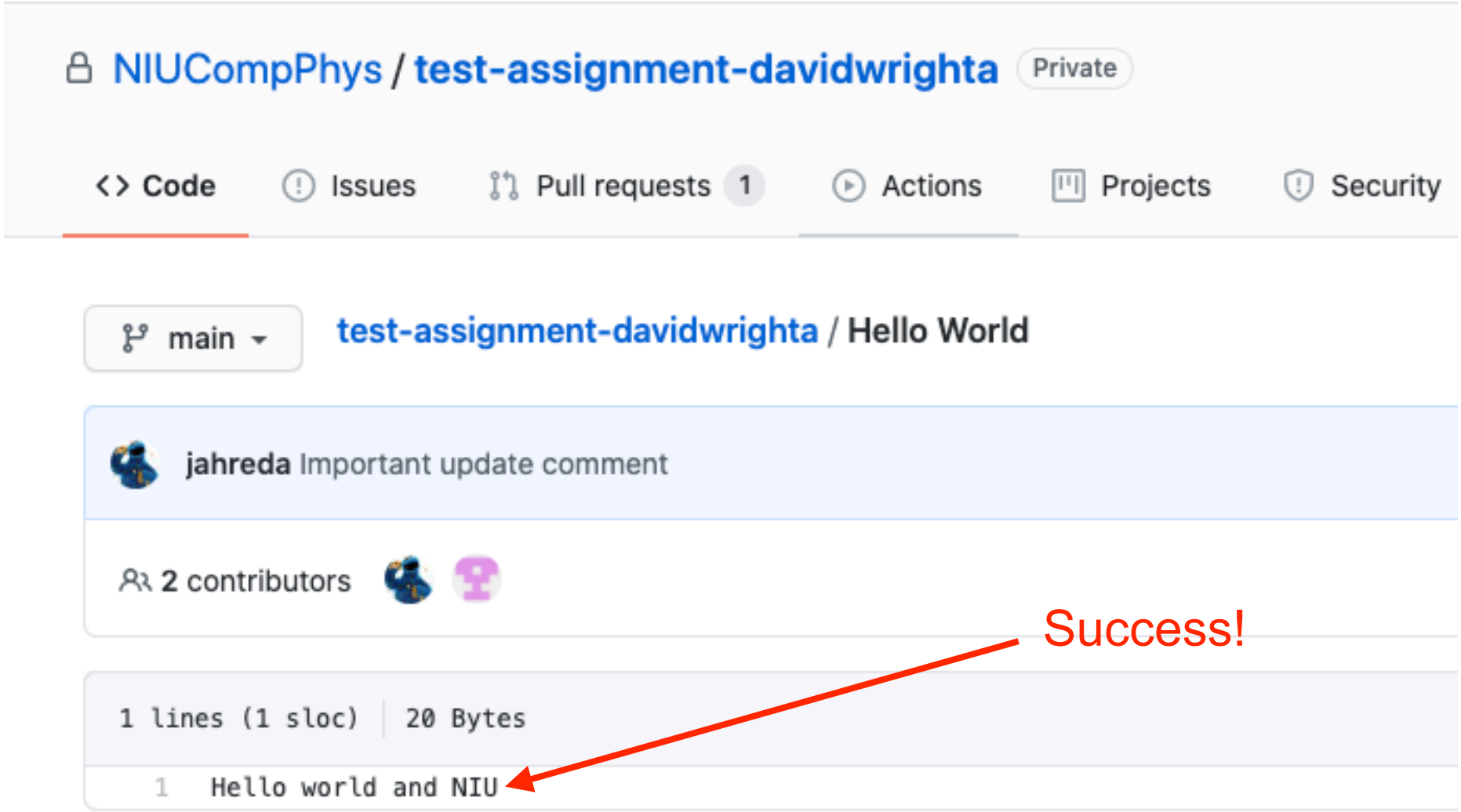
```
~/Desktop/test-assignment-davidwrighta % git add Hello\ World
```

```
~/Desktop/test-assignment-davidwrighta % git commit -m "Important update comment"
[main 043357d] Important update comment
 1 file changed, 1 insertion(+), 1 deletion(-)
~/Desktop/test-assignment-davidwrighta % git push
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 308 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/NIUCompPhys/test-assignment-davidwrighta
   b639472..043357d  main -> main
~/Desktop/test-assignment-davidwrighta %
```

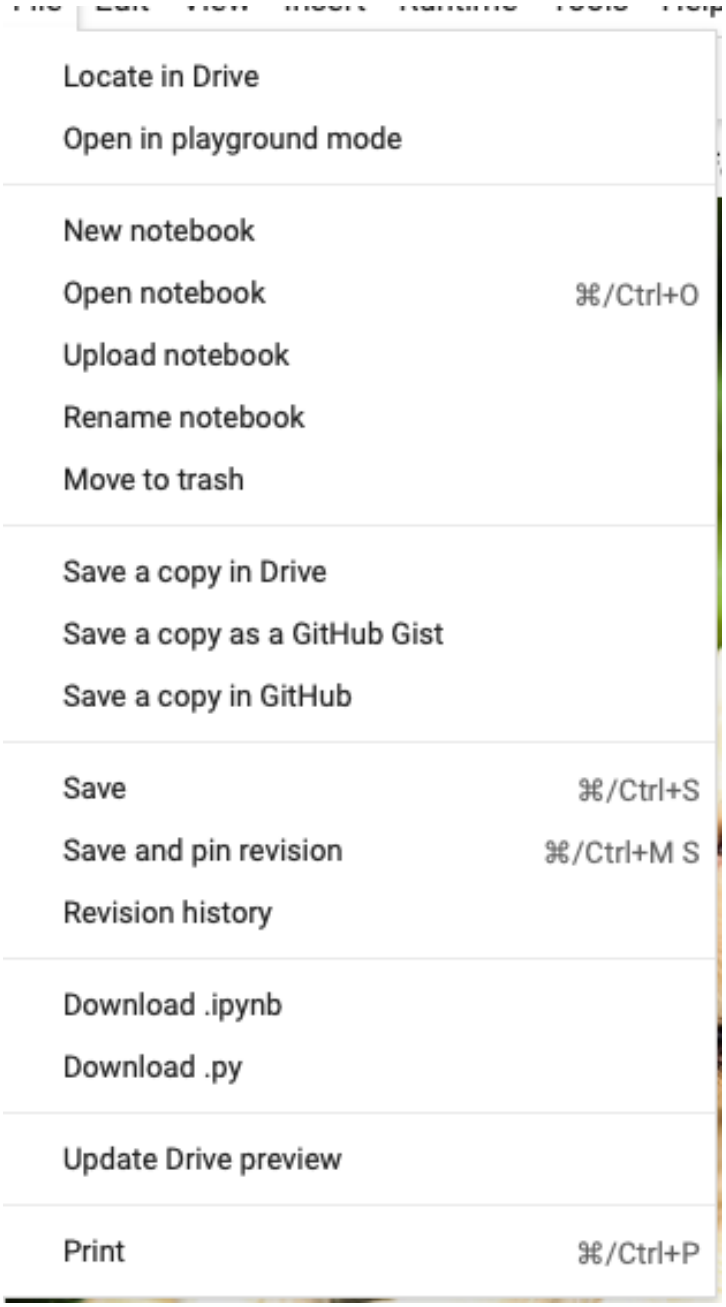
Back to the repo on the web



Success!

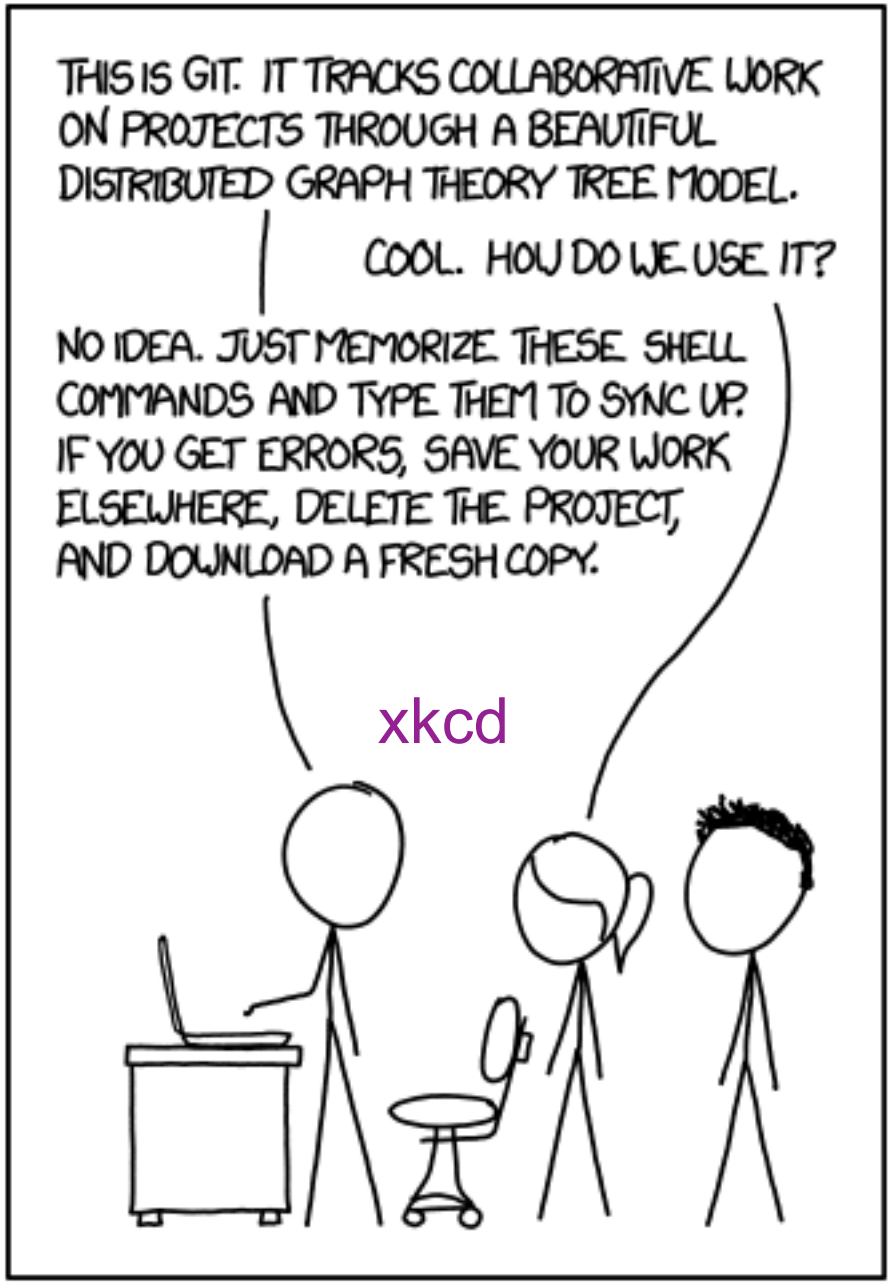

NIUCompPhys / test-assignment-davidwrighta
Private
Code
Issues
Pull requests 1
Actions
Projects
Security
main
test-assignment-davidwrighta / Hello World
jahreda Important update comment
2 contributors
Success!
1 lines (1 sloc) 20 Bytes
1 Hello world and NIU

Nice integration of two systems



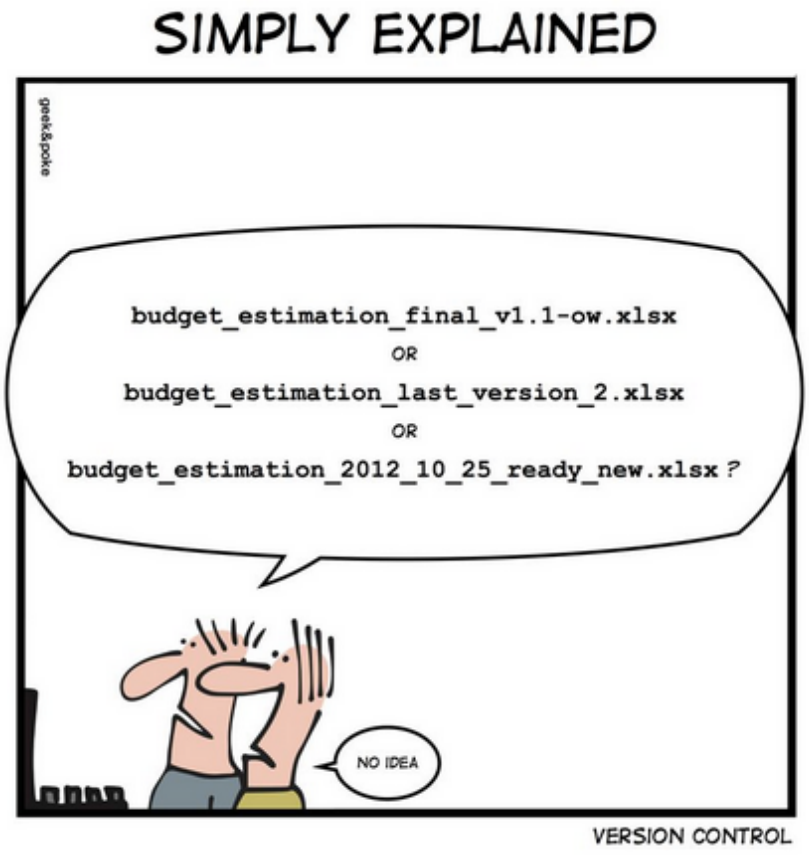
You can ask Colab to save a copy in GitHub for you (not required, of course, you can edit things locally). And then you can open from GitHub back into Colab. There's a nice discussion here that you can read at your own pace: https://colab.research.google.com/github/googlecolab/colabtools/blob/master/notebooks/colab-github-demo.ipynb

What is this git ? Why are we using it?



https://www.groovecommerce.com/ecommerce-blog/guide-to-version-control-for-magento-using-git-and-beanstalk/

Version control!

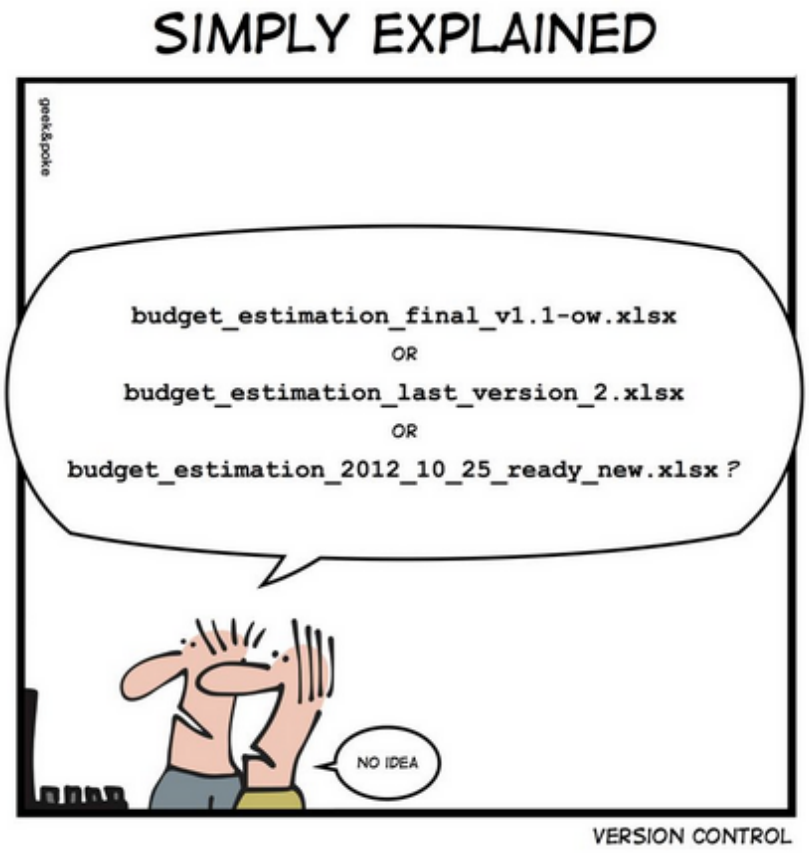


Version control

Think of it as a way to **back up your work** while having **different versions** to track what you did and when (and what others did and when)

I started out using CVS (concurrent versions system), then moved to SVN (subversion) and now git

We will use only the simplest features of git, which can be quite complex



Re: git

en.wikipedia.org/wiki/Git

Naming [edit]

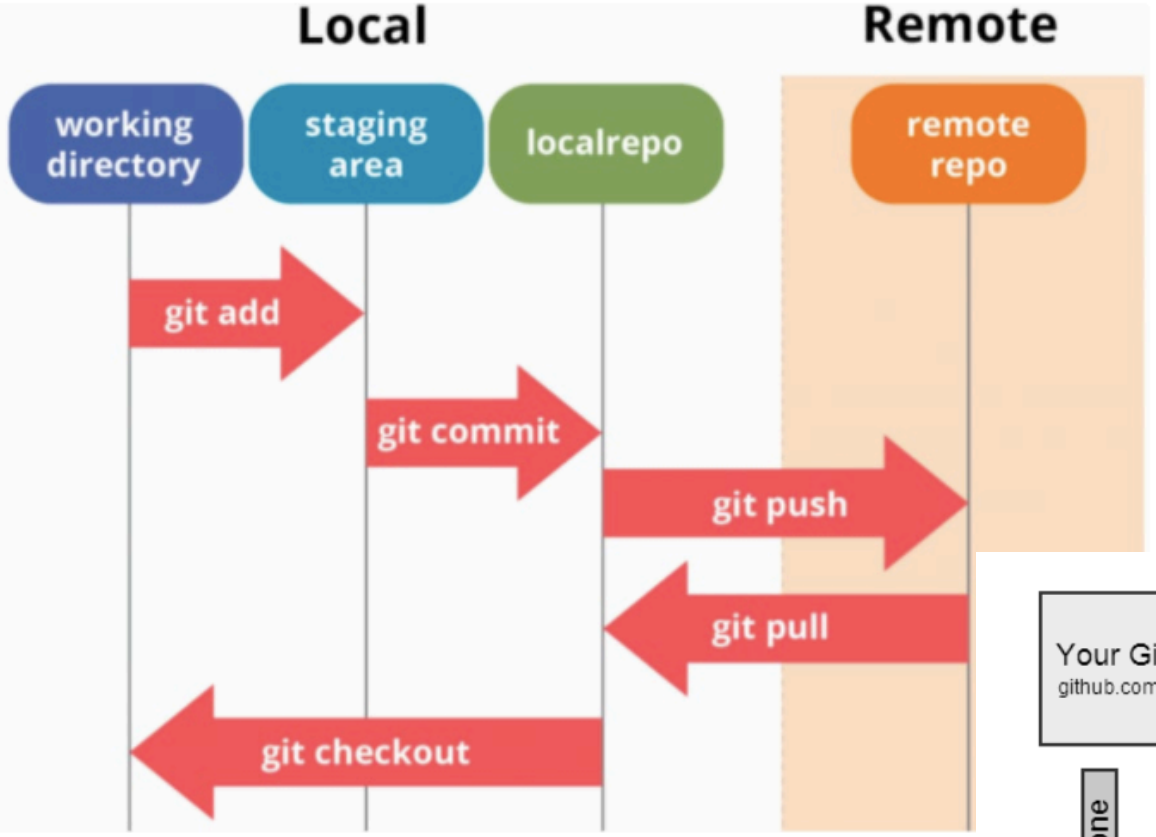
Torvalds sarcastically quipped about the name *git* (which means "unpleasant person" in British English slang): "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."[24][25] The man page describes Git as "the stupid content tracker".[26] The read-me file of the source code elaborates further:[27]

> "git" can mean anything, depending on your mood.
>
> - random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
> - stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
> - "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
> - "goddamn idiotic truckload of sh*t": when it breaks

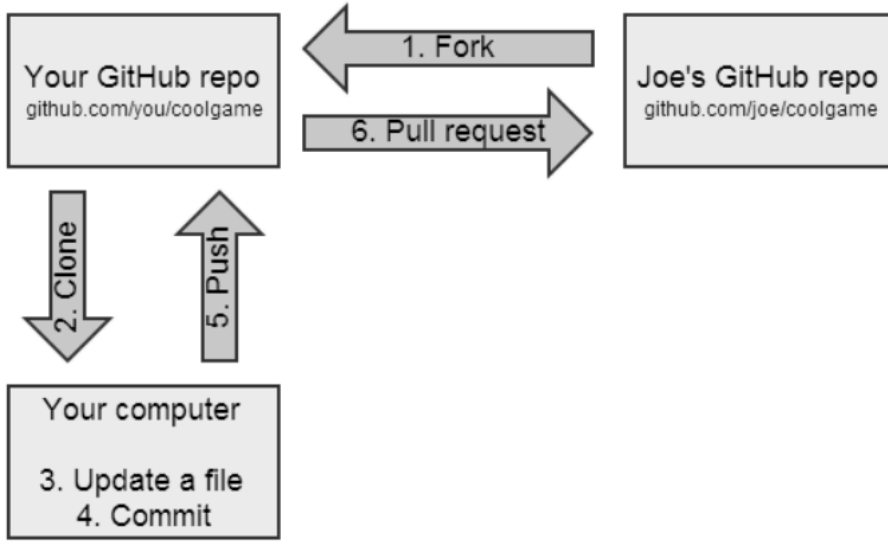
No comment

Workflow on git

https://dev.to/mollynem/git-github--workflow-fundamentals-5496



Not worrying about branches (different versions of the same code for different development and use) so things should be straightforward



Why the complication?

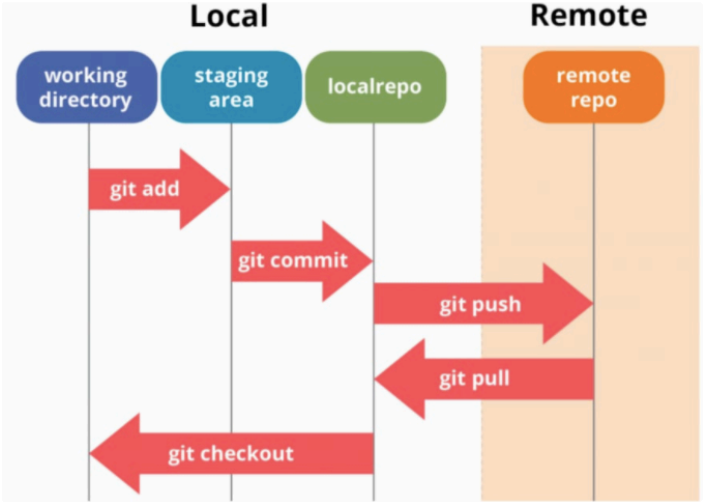
Your local areas is YOURS. You can do what you want with it, make your own branches, make mistakes, fix things and only when things are really working do you "push" to the central, remote repository of code

Can access other people's code and suggest updates too!

Reminder: We will use only the simplest features of git, but it's very useful to know

Useful to understand workflow:

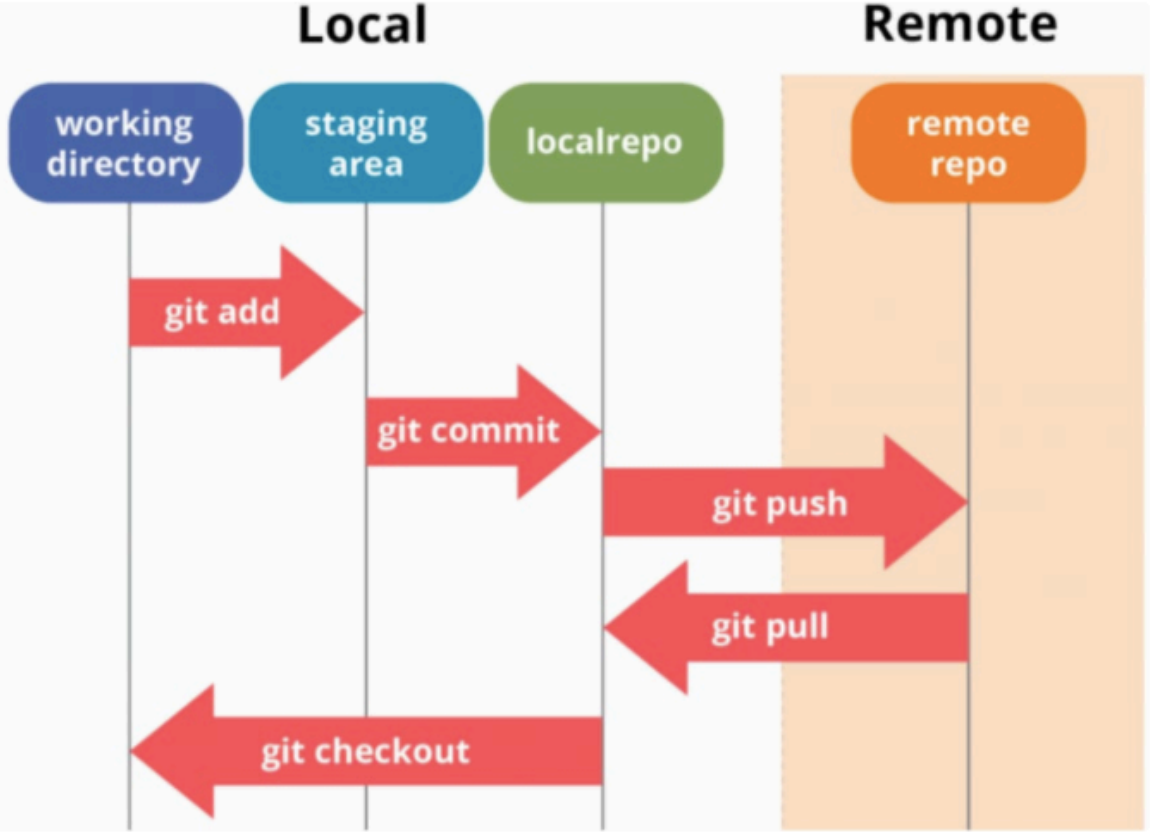
https://guides.github.com/introduction/flow/



Useful to understand how to use it:

https://guides.github.com/activities/hello-world/

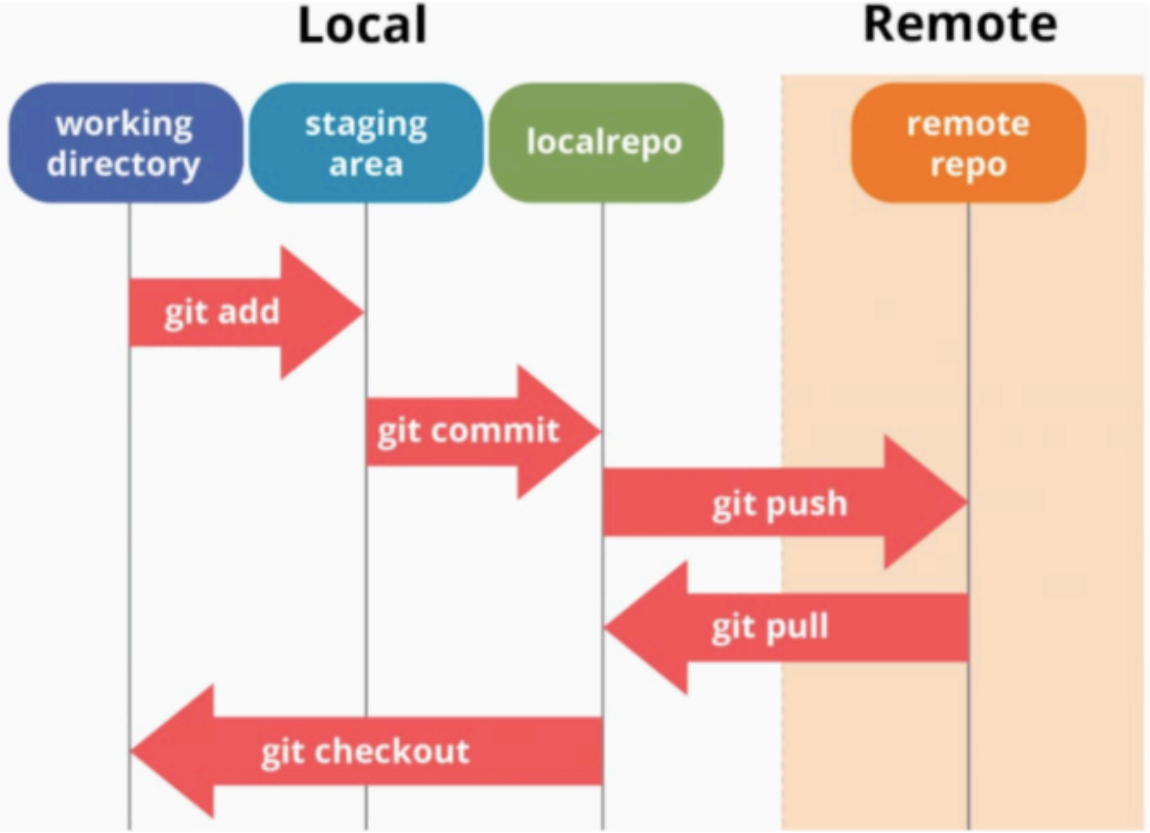
More on git and assignments



I will have access to anything you have pushed to your repo for any assignment. I will not look at it until the due date, but if you need help in advance, you can push updates for me to look at

Create your accounts and submit “test” assignment (hello world) right away so you don’t get stuck when we have real assignments to hand in

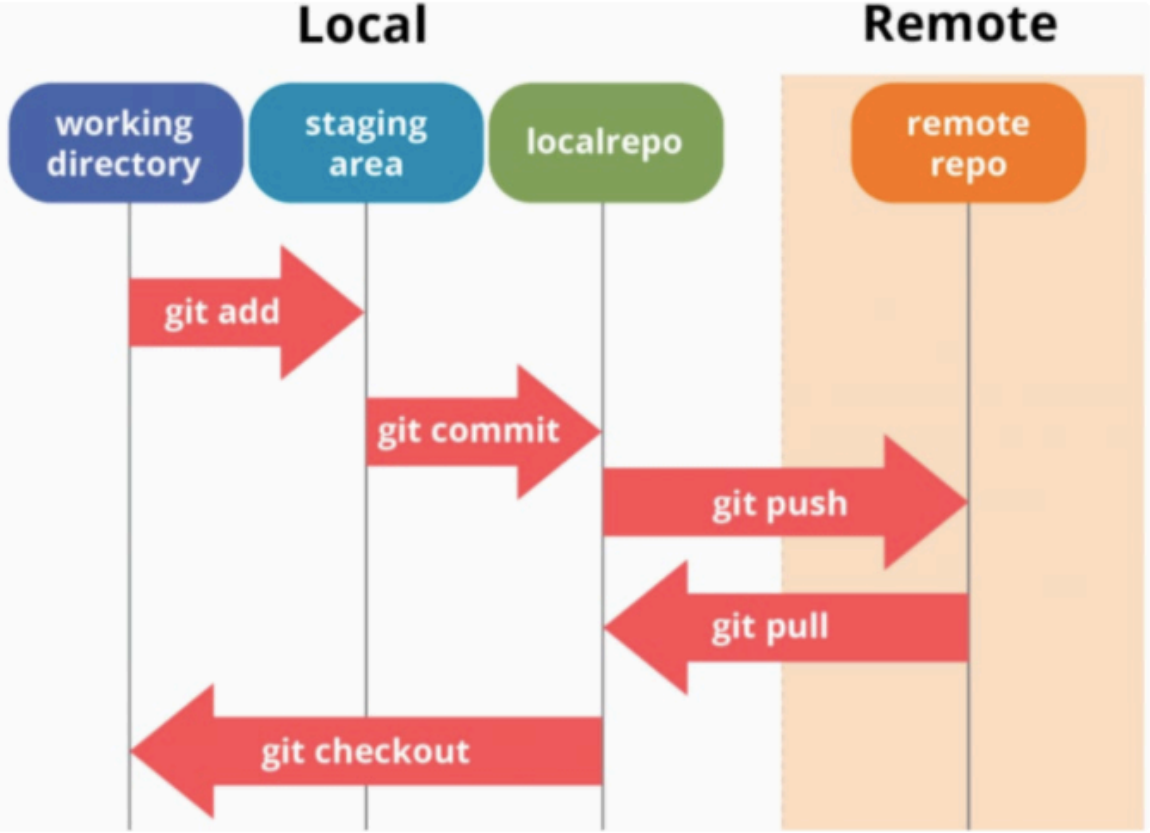
More on git and assignments



I will have access to anything you have pushed to your repo for any assignment. I will not look at it until the due date, but if you need help in advance, you can push updates for me to look at

You do NOT have to do python development on colab, though we will use it a bit and it is quite intuitive and easy. But you can do python development in a different environment and just upload the python file or notebook, if you prefer (though I will test in colab, so you should as well!)

More on git and assignments



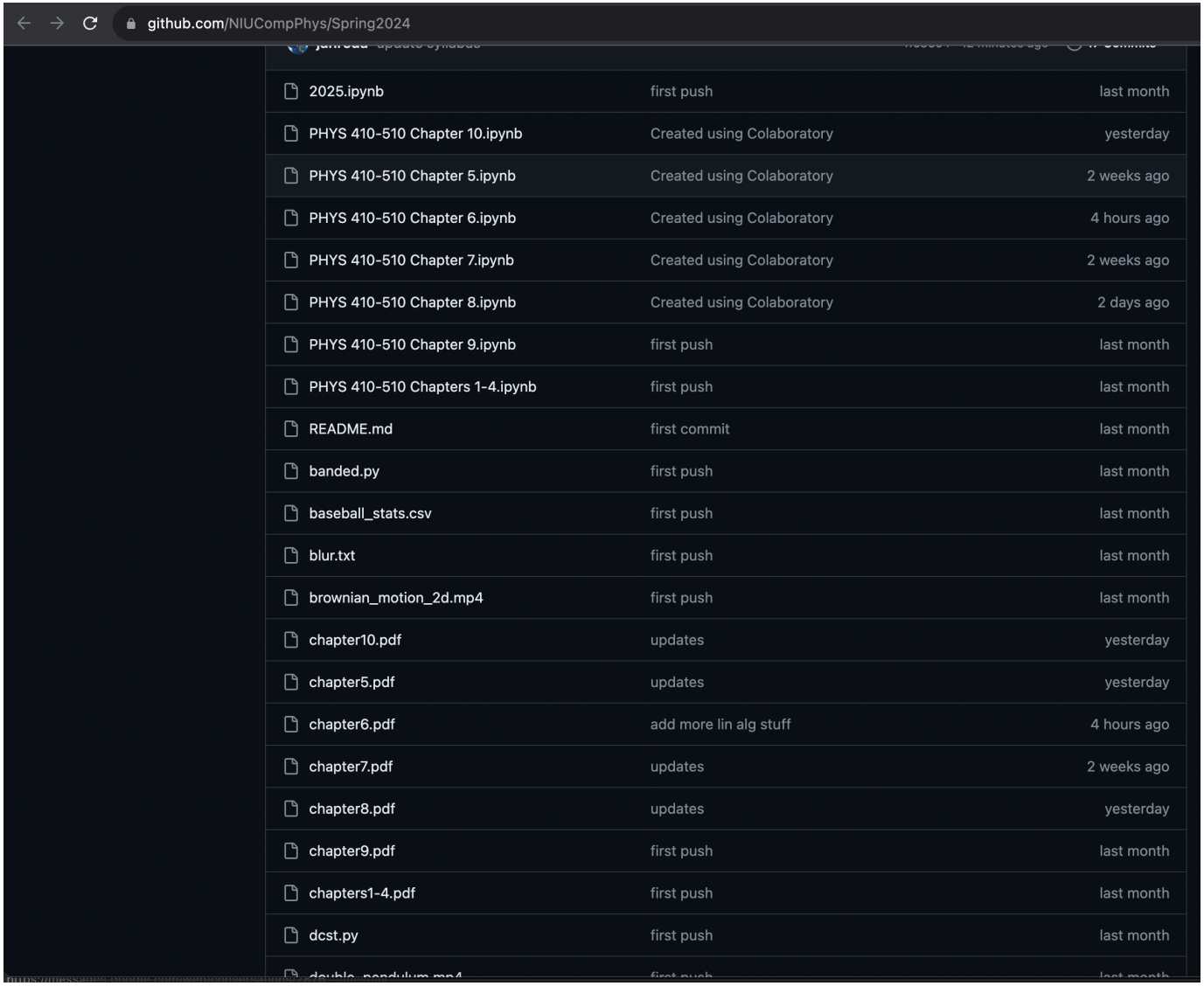
I will have access to anything you have pushed to your repo for any assignment. I will not look at it until the due date, but if you need help in advance, you can push updates for me to look at

You will have homework that isn't strictly coding. Just upload a PDF of your work! For some assignments I am also happy for you to submit things as Colab text

What else is in our git repo?

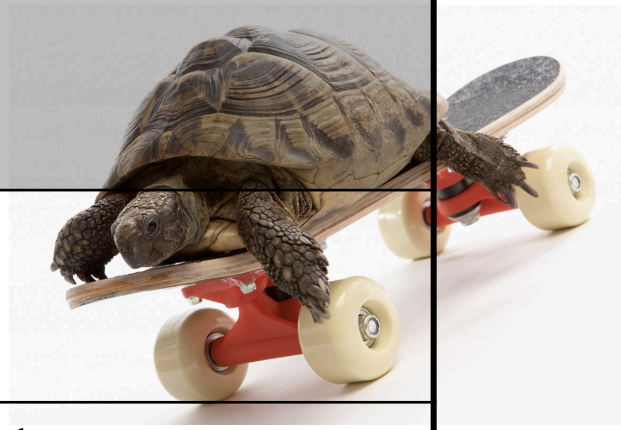
https://github.com/NIUCompPhys/Spring2024

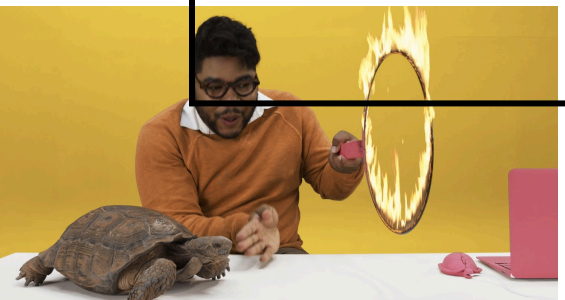
All the notebooks (hint: lots of useful things for your assignments), all the material we go over in each chapter… and all the slides!



More on our choice of programming language

C++	Python
Compiled	Interpreted
Difficult to develop	Fast to develop (interpreted code helps with this!)
Often not so intuitive, difficult to debug	More intuitive, easier debugging
Fast execution	Slow to execute, but we'll see ways around this





More on our choice of programming language

https://en.wikipedia.org/wiki/Python_(programming_language)

Naming [edit]

Python's name is derived from the British comedy group Monty Python, whom Python creator Guido van Rossum enjoyed while developing the language. Monty Python references appear frequently in Python code and culture;[174] for example, the metasyntactic variables often used in Python literature are *spam* and *eggs* instead of the traditional *foo* and *bar*.[174][175] The official Python documentation also contains various references to Monty Python routines.[176][177]

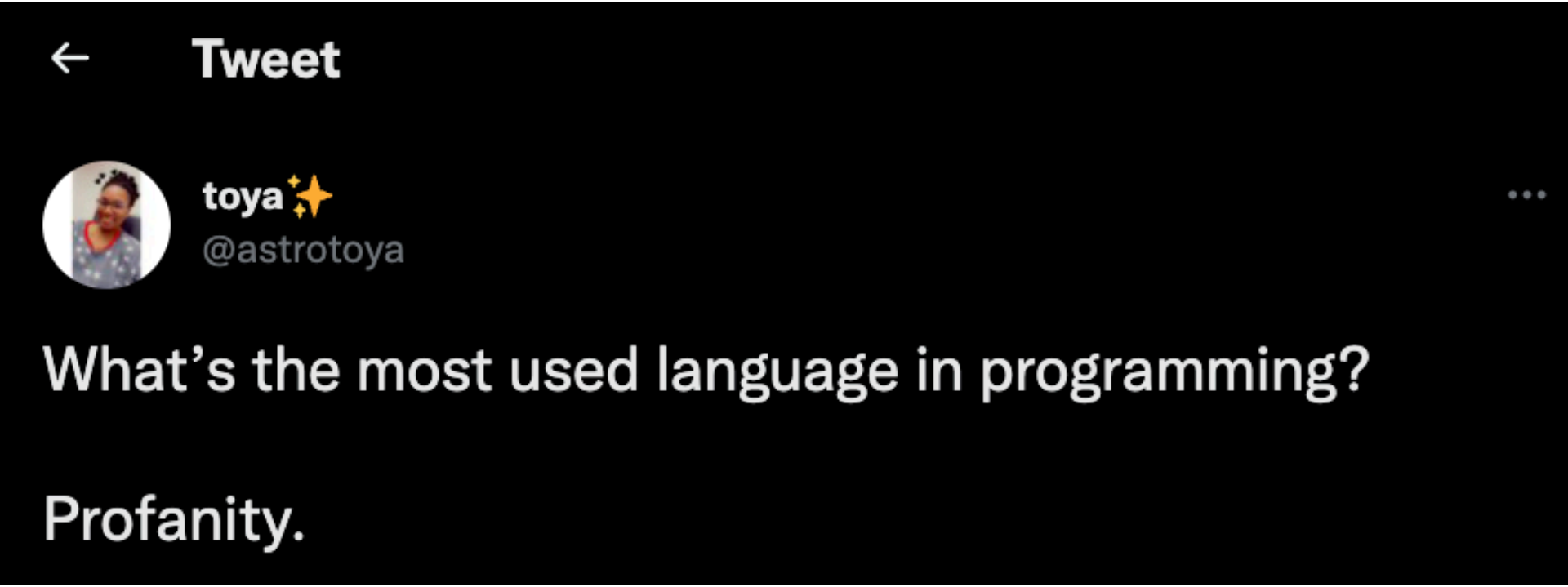
The prefix *Py-* is used to show that something is related to Python. Examples of the use of this prefix in names of Python applications or libraries include Pygame, a binding of SDL to Python (commonly used to create games); PyQt and PyGTK, which bind Qt and GTK to Python respectively; and PyPy, a Python implementation originally written in Python.

https://www.youtube.com/watch?v=ZmInkxbvlCs

Some … HW (ungraded) is to watch this video and make sure to laugh

Yet more on our choice of programming language

https://twitter.com/astrotoya/status/1358164363497078784



All good physicists and computer scientists need a sense of humor

Ideally how we use both Python and C++

- Write all the algorithms in Python
- Ensure that Python is using libraries pre-compiled in C++
- Best of both worlds! This is what numpy and scipy (which we will see) do
- If you don't know Python, please read the first few chapters of the textbook, which are easy to read and very instructive

Other programming language choices

- matlab is not free, see no reason to use it (though we'll use plotting code inspired by it!)
- Fortran (in various iterations) is very old, not used by many people anymore outside of obscure areas of academia
- There are many other new, interesting programming languages. But I won't help you with them, so you will be on your own if you decide to use them

Pseudocode

We will often discuss algorithms and tools and techniques using **pseudocode**, which is not for any code that is specific to any one programming language, but is instead designed to be easy to read and follow by humans (as opposed to be computer interpreters or computer compilers)

```
def studentGrade:
  hw = average(HW grades)
  quiz = average(quizzes)
  exam = average(exam scores)
  finalgrade = 25%*hw +
      10%*quiz + 20%*exams +
      45%*final
  if (perfectAttendance)
    finalGrade = finalGrade+Bonus
  if (cheated) letterGrade = F
  else if (finalGrade > 90%)
    letterGrade = A
  else if (finalGrade > 80%)
    letterGrade = B
  else if (finalGrade > 70%)
    letterGrade = C
  else if (finalGrade > 60%)
    letterGrade = D
  else
    letterGrade = F
```

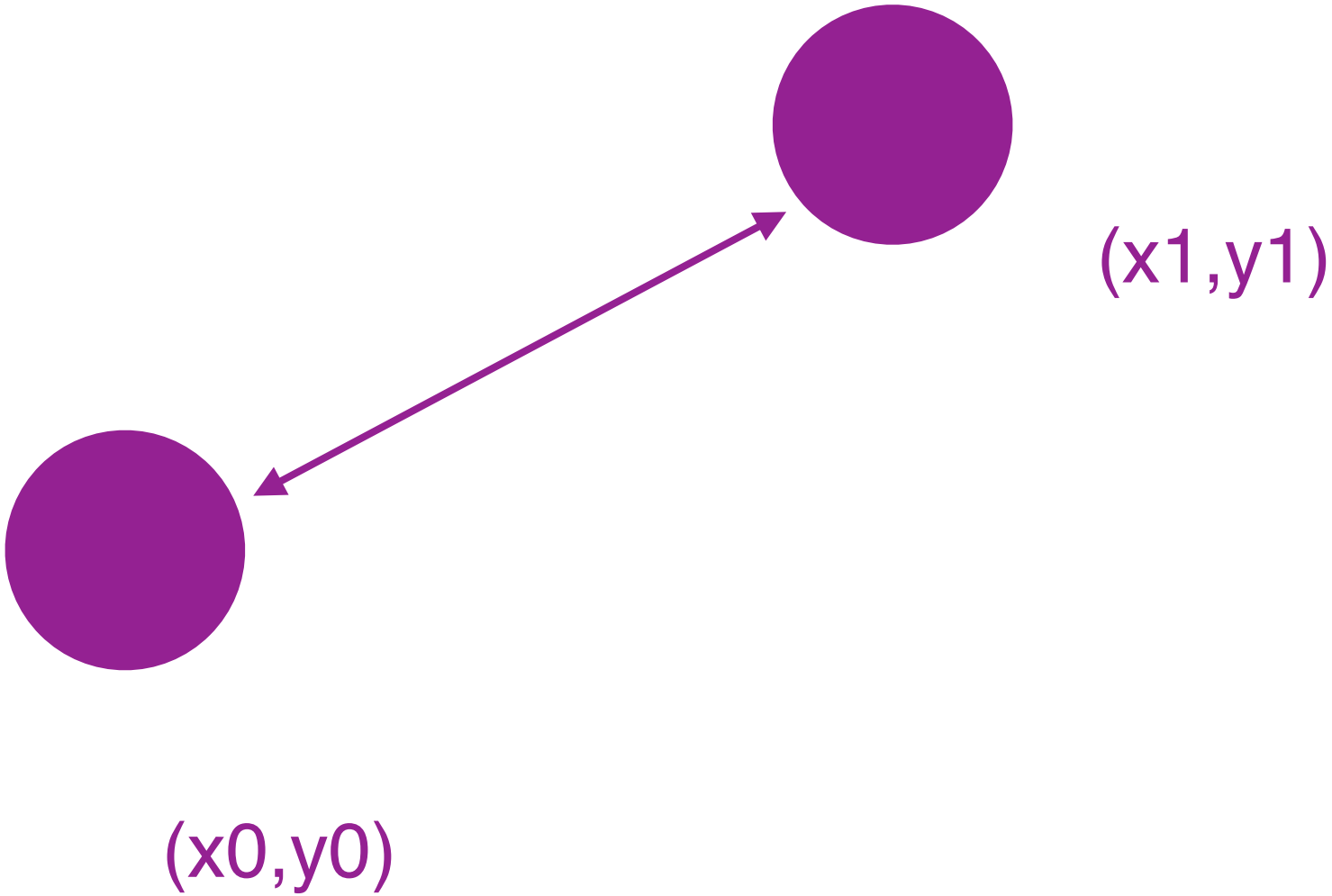
Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

"Weeks of coding can save you hours of planning." - Unknown

What we'll try to do now

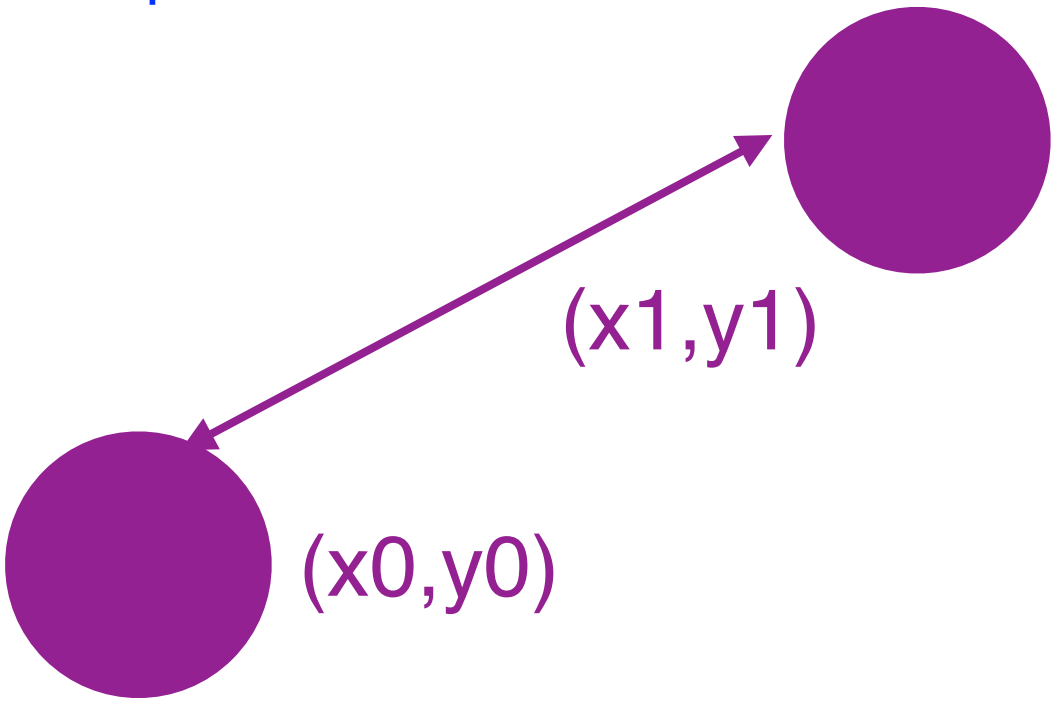
What is the slope between these two points?



Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

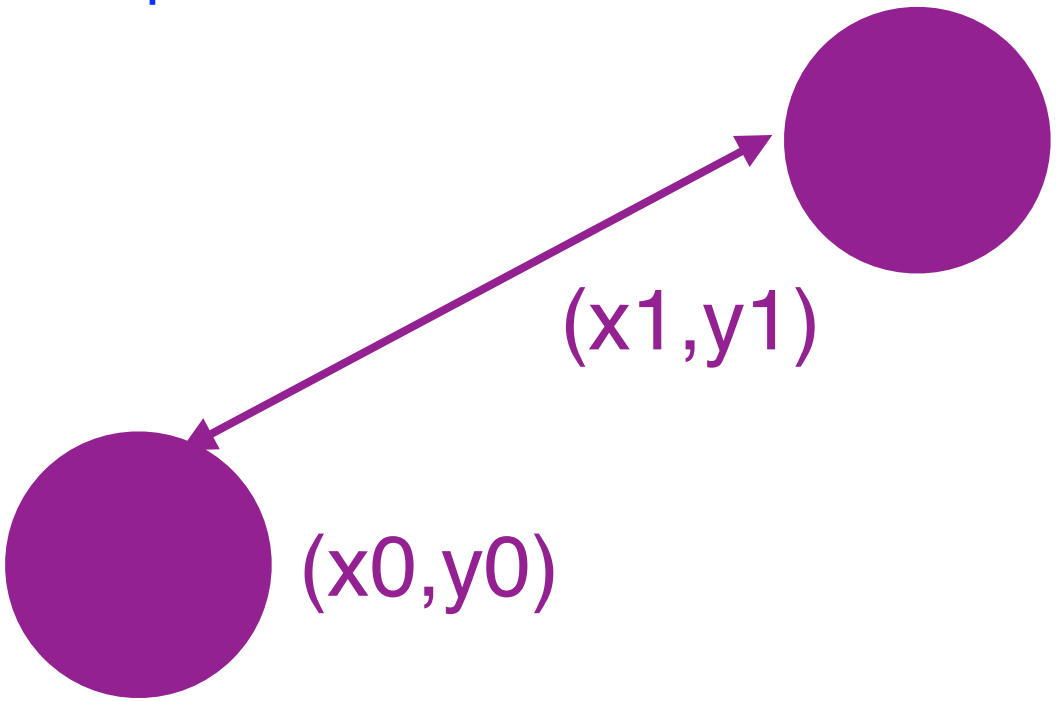
Slope = delta(y) / delta(x) = (y1-y0)/(x1-x0)



Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

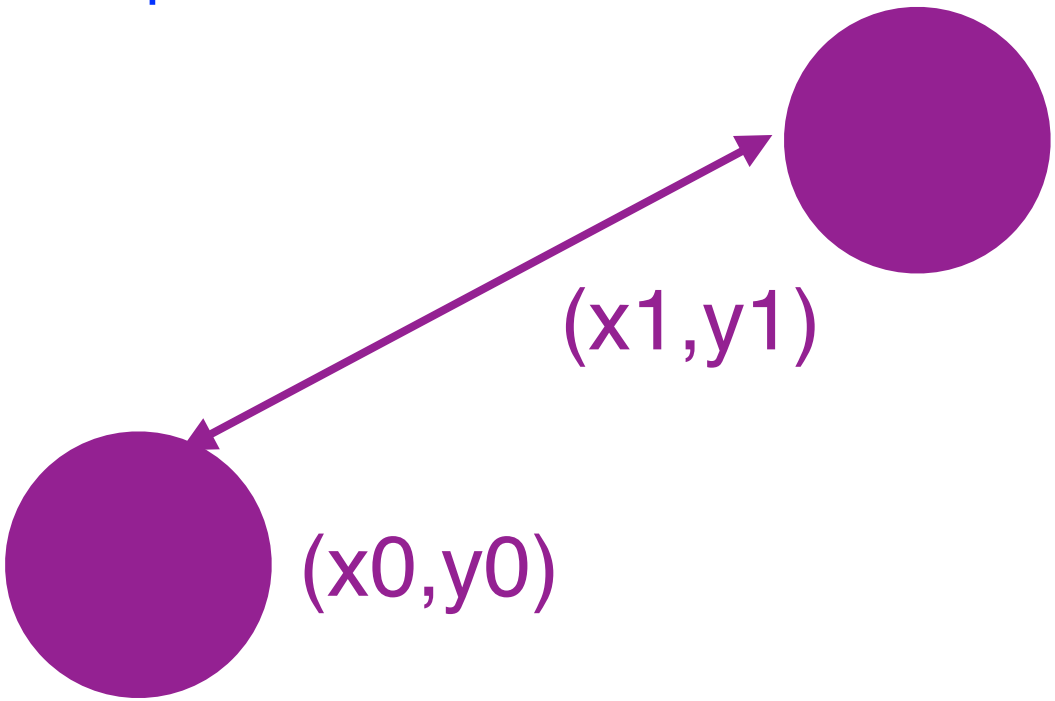
Slope = delta(y) / delta(x) = (y1-y0)/(x1-x0)



Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

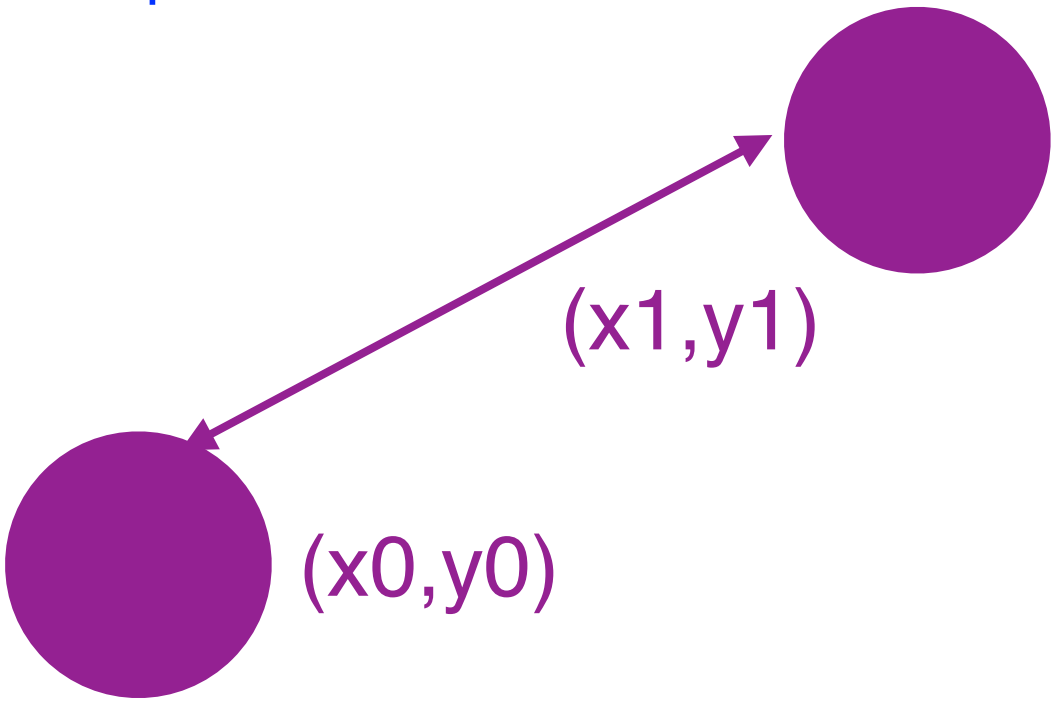
```
slope = (y1-y0)/(x1-x0)
return slope
```



Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

```
def getSlope(x0,y0,x1,y1):
  slope = (y1-y0)/(x1-x0)
  return slope
```



Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

```
>>> def getSlope(x0,y0,x1,y1):
...     slope = (y1-y0)/(x1-x0)
...     return slope
...
>>> slope = getSlope(0.0,0.0,2.0,4.0)
>>> print(slope)
2.0
```

Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

```
>>> slope = getSlope(2.0,3.0,2.0,4.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in getSlope
ZeroDivisionError: float division by zero
```

Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

```
>>> from math import *
>>> epsilon = 1e-6
>>> def getSlope(x0,y0,x1,y1):
...   dy = y1-y0
...   dx = x1-x0
...   if abs(dx) > epsilon: ## only calculate if den != 0
...     return dy/dx
...   else:
...     print("Can't get the slope, dx is too close to zero")
...     return None
```

Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

```
>>> from math import *
>>> epsilon = 1e-6
>>> def getSlope(x0,y0,x1,y1):
...   dy = y1-y0
...   dx = x1-x0
...   if abs(dx) > epsilon: ## only calculate if den != 0
...     return dy/dx
...   else:
...     print("Can't get the slope, dx is too close to zero")
...     return None
...
>>> getSlope(0.0,0.0,2.0,4.0)
2.0
>>> getSlope(2.0,3.0,2.0,4.0)
Can't get the slope, dx is too close to zero
```

Stages of development (this and below from UB course)

- Step 1: Write things down on paper
- Step 2: If you don't understand everything goto step 1
- Step 3: Write Pseudocode
- Step 4: Write code
- Step 5: Check code with unit tests
  - Check all criteria
  - If unit tests fail, goto step 4
- Step 6: Publish

Use github to submit assignment, go celebrate

Some thoughts on our simple example

- Computers do not understand infinite precision
  - See: upcoming chapters
- Computers follow instructions, no more and no less
  - They are not smarter than us!
- Stepping back and thinking a bit before writing code will ultimately save you a lot of time in the long run

C++ version


```
/Users/jahreda/Desktop/temp cat slope.C
#include <iostream>
#include <math.h>
using std::cout;
using std::endl;

// Better slope calculator : Error handling when
// denominator is zero
float getSlope( float x0, float y0, float x1, float y1, float epsilon = 1e-6) {
   cout << "Getting slope for x0 = " << x0 << " and x1 = " << x1 << " and y0 = " << y0 << " and y1 = " << y1
<< endl;
   float dy = y1 - y0;
   float dx = x1 - x0;
   if ( fabs(dx) > epsilon )
   {
      return dy / dx;
   }
   else {
      cout << "----> invalid input, setting to -99999.99" << endl;
      return -99999.99;
   }
}

int main(int argc, char *argv[])
{
   float slope = getSlope( 0.0,0.0,2.0,4.0);
   cout << slope << endl;
   slope = getSlope(2.0,3.0,2.0,4.0);
   cout << slope << endl;
   return 0;
}
/Users/jahreda/Desktop/temp g++ -o slope slope.C
/Users/jahreda/Desktop/temp ./slope
Getting slope for x0 = 0 and x1 = 2 and y0 = 0 and y1 = 4
2
Getting slope for x0 = 2 and x1 = 2 and y0 = 3 and y1 = 4
----> invalid input, setting to -99999.99
-100000
```

Already see that Python is easier to read and follow than C :)

Back to C++ vs Python

A lot of the examples that we'll use this semester will be shown in Python, but you don't have to submit any Python code at all if you don't want to. C/C++ is fine, too, though you need clear instructions for how I am to run the code, and in many cases you will need visualization tools. ROOT is the tool used in HEP, but there are plenty others. Up to you.

Why Python? Looks a lot like pseudocode! So you should be able to follow this even if you don't know Python

```
>>> from math import *
>>> epsilon = 1e-6
>>> def getSlope(x0,y0,x1,y1):
...   dy = y1-y0
...   dx = x1-x0
...   if abs(dx) > epsilon:
...     return dy/dx
...   else:
...     print("Can't get the slope, dx is too close to zero")
...     return None
```

What are these languages doing?

```
/Users/jahreda/Desktop/temp cat slope.C
#include <iostream>
#include <math.h>
using std::cout;
using std::endl;

// Better slope calculator : Error handling when
// denominator is zero
float getSlope( float x0, float y0, float x1, float y1, float epsilon = 1e-6) {
   cout << "Getting slope for x0 = " << x0 << " and x1 = " << x1 << " and y0 = "
<< y0 << " and y1 = " << y1 << endl;
   float dy = y1 - y0;
   float dx = x1 - x0;
   if ( fabs(dx) > epsilon )
   {
      return dy / dx;
   }
   else {
      cout << "----> invalid input, setting to -99999.99" << endl;
      return -99999.99;
   }
}

int main(int argc, char *argv[])
{
   float slope = getSlope( 0.0,0.0,2.0,4.0);
   cout << slope << endl;
   slope = getSlope(2.0,3.0,2.0,4.0);
   cout << slope << endl;
   return 0;
}
/Users/jahreda/Desktop/temp g++ -o slope slope.C
```

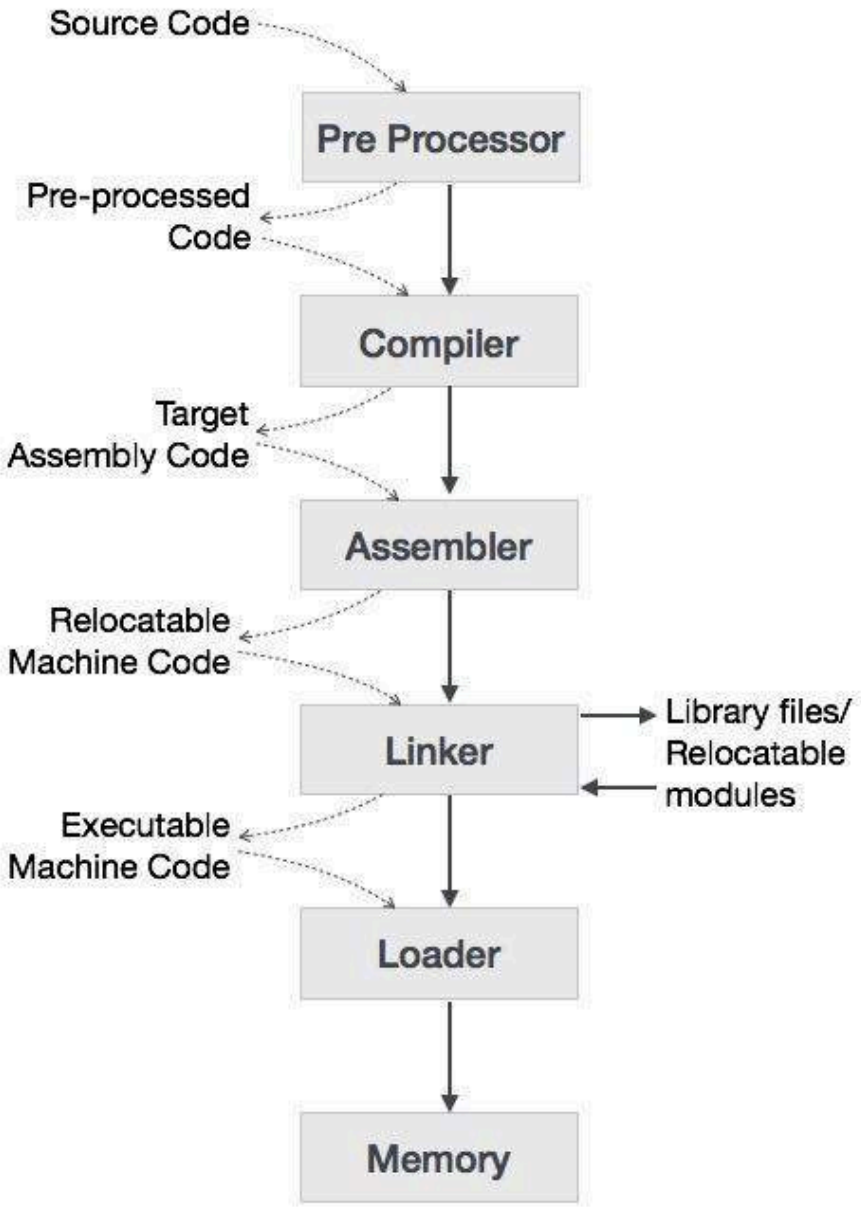
```
>>> from math import *
>>> epsilon = 1e-6
>>> def getSlope(x0,y0,x1,y1):
...   dy = y1-y0
...   dx = x1-x0
...   if abs(dx) > epsilon:
...     return dy/dx
...   else:
...     print("Can't get the slope, dx is too close to zero")
...     return None
```

No matter what they are doing, ultimately your program will be executed on your phone/laptop/ computer, likely on a CPU but maybe on some other processing unit that **only knows 0's and 1's**

What is compiled code (like C++)?

https://sites.google.com/site/mscsystemsoftware/compiler-design

- The compiler converts source code to assembly language (low-level language)
- This is then translated into machine code
- A linker links/ combines the pieces of the program for execution
- The program is loaded into memory, and then executed



Assembly language?

https://commons.wikimedia.org/wiki/File:Motorola_6800_Assembly_Language.png

- No one wants to write assembly code. For very good reason. But it is a set of low-level basic instructions: Move Register 1 into Register 2, Add Register 1 contents to Register 2 contents, store these in Register 3
- Assembly language is specific to a hardware architecture, so the assembler needs to know what hardware you are using
- Assembly language is then closely connected to the exact hardware instructions run on your machine. But it's … ugly :)

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

C000                    ORG     ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS     #STACK

                        ****************************************
                        * FUNCTION: INITA - Initialize ACIA
                        * INPUT: none
                        * OUTPUT: none
                        * CALLS: none
                        * DESTROYS: acc A

0013                    RESETA  EQU     %00010011
0011                    CTLREG  EQU     %00010001

C003 86 13      INITA   LDA A   #RESETA   RESET ACIA
C005 B7 80 04           STA A   ACIA
C008 86 11              LDA A   #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04           STA A   ACIA

C00D 7E C0 F1           JMP     SIGNON    GO TO START OF MONITOR

                        ****************************************
                        * FUNCTION: INCH - Input character
                        * INPUT: none
                        * OUTPUT: char in acc A
                        * DESTROYS: acc A
                        * CALLS: none
                        * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04   INCH    LDA A   ACIA      GET STATUS
C013 47                 ASR A             SHIFT RDRF FLAG INTO CARRY
C014 24 FA              BCC     INCH      RECIEVE NOT READY
C016 B6 80 05           LDA A   ACIA+1    GET CHAR
C019 84 7F              AND A   #$7F      MASK PARITY
C01B 7E C0 79           JMP     OUTCH     ECHO & RTS

                        ****************************************
                        * FUNCTION: INHEX - INPUT HEX DIGIT
                        * INPUT: none
                        * OUTPUT: Digit in acc A
                        * CALLS: INCH
                        * DESTROYS: acc A
                        * Returns to monitor if not HEX input

C01E 8D F0      INHEX   BSR     INCH      GET A CHAR
C020 81 30              CMP A   #'0       ZERO
C022 2B 11              BMI     HEXERR    NOT HEX
C024 81 39              CMP A   #'9       NINE
C026 2F 0A              BLE     HEXRTS    GOOD HEX
C028 81 41              CMP A   #'A
C02A 2B 09              BMI     HEXERR    NOT HEX
C02C 81 46              CMP A   #'F
C02E 2E 05              BGT     HEXERR
C030 80 07              SUB A   #7        FIX A-F
C032 84 0F      HEXRTS  AND A   #$0F      CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF   HEXERR  JMP     CTRL      RETURN TO CONTROL LOOP
```

An assembly language listing for a Motorola 6800 8-bit microprocessor. This is a page from a "Monitor" program that communicates to a serial terminal connected to a MC6850 Asynchronous Communications Interface Adapter (ACIA). The routines here initialize the ACIA (INITA), read an ASCII character (INCH) and read a hexadecimal digit (INHEX).

What about Python?

https://commons.wikimedia.org/wiki/File:Motorola_6800_Assembly_Language.png

- Even as an interpreted language, it still needs to execute those same basic instructions on a CPU, but it does so by executing (ie interpreting) your code line-by-line
- The python interpreter is a Virtual Machine that interprets your python scripts and executables (though it does some small conversion to python bytecode first)
  - Naively may be slower, though there are ways around this

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2

C000                    ORG     ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS     #STACK

                        ****************************************
                        * FUNCTION: INITA - Initialize ACIA
                        * INPUT: none
                        * OUTPUT: none
                        * CALLS: none
                        * DESTROYS: acc A

0013                    RESETA  EQU     %00010011
0011                    CTLREG  EQU     %00010001

C003 86 13     INITA    LDA A   #RESETA    RESET ACIA
C005 B7 80 04           STA A   ACIA
C008 86 11              LDA A   #CTLREG    SET 8 BITS AND 2 STOP
C00A B7 80 04           STA A   ACIA

C00D 7E C0 F1           JMP     SIGNON     GO TO START OF MONITOR

                        ****************************************
                        * FUNCTION: INCH - Input character
                        * INPUT: none
                        * OUTPUT: char in acc A
                        * DESTROYS: acc A
                        * CALLS: none
                        * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04  INCH     LDA A   ACIA       GET STATUS
C013 47                 ASR A              SHIFT RDRF FLAG INTO CARRY
C014 24 FA              BCC     INCH       RECIEVE NOT READY
C016 B6 80 05           LDA A   ACIA+1     GET CHAR
C019 84 7F              AND A   #$7F       MASK PARITY
C01B 7E C0 79           JMP     OUTCH      ECHO & RTS

                        ****************************************
                        * FUNCTION: INHEX - INPUT HEX DIGIT
                        * INPUT: none
                        * OUTPUT: Digit in acc A
                        * CALLS: INCH
                        * DESTROYS: acc A
                        * Returns to monitor if not HEX input

C01E 8D F0     INHEX    BSR     INCH       GET A CHAR
C020 81 30              CMP A   #'0        ZERO
C022 2B 11              BMI     HEXERR     NOT HEX
C024 81 39              CMP A   #'9        NINE
C026 2F 0A              BLE     HEXRTS     GOOD HEX
C028 81 41              CMP A   #'A
C02A 2B 09              BMI     HEXERR     NOT HEX
C02C 81 46              CMP A   #'F
C02E 2E 05              BGT     HEXERR
C030 80 07              SUB A   #7         FIX A-F
C032 84 0F     HEXRTS   AND A   #$0F       CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF  HEXERR   JMP     CTRL       RETURN TO CONTROL LOOP
```

An assembly language listing for a Motorola 6800 8-bit microprocessor. This is a page from a "Monitor" program that communicates to a serial terminal connected to a MC6850 Asynchronous Communications Interface Adapter (ACIA). The routines here initialize the ACIA (INITA), read an ASCII character (INCH) and read a hexadecimal digit (INHEX).

What is being executed, again?

Everything comes down to specific memory locations in your computer/ phone/Playstation, which are transistors (physics!) that store things as on/off, 1/0, yes/no, one/zero. We typically use 1/0 when we talk about **binary (base 2)**. Each of these numbers is stored in a specific **memory address**

Address	Value
0	10000001
1	10101010
2	11111111
3	10000000

Bits = **Bi**nary Digi**ts**

Sidebar on pointers (for the C++ crowd)

Great slide for the C++ folks from Sal Rappoccio…. please keep in mind!

C++: ptrs/refs in functions

- Pass by value:
  - COPIES value into a temporary variable called "x"

```
void func( int x);
```

- Use when you don't want to modify value, and cheap to copy

- Pass by reference:
  - Pass REFERENCE to variable, temporarily called "x"

```
void func( int &x);
```

- Use when you want to modify value, and expensive to copy, ptr=0 disallowed

- Pass by pointer
  - Pass POINTER to variable, pointer is called "x"

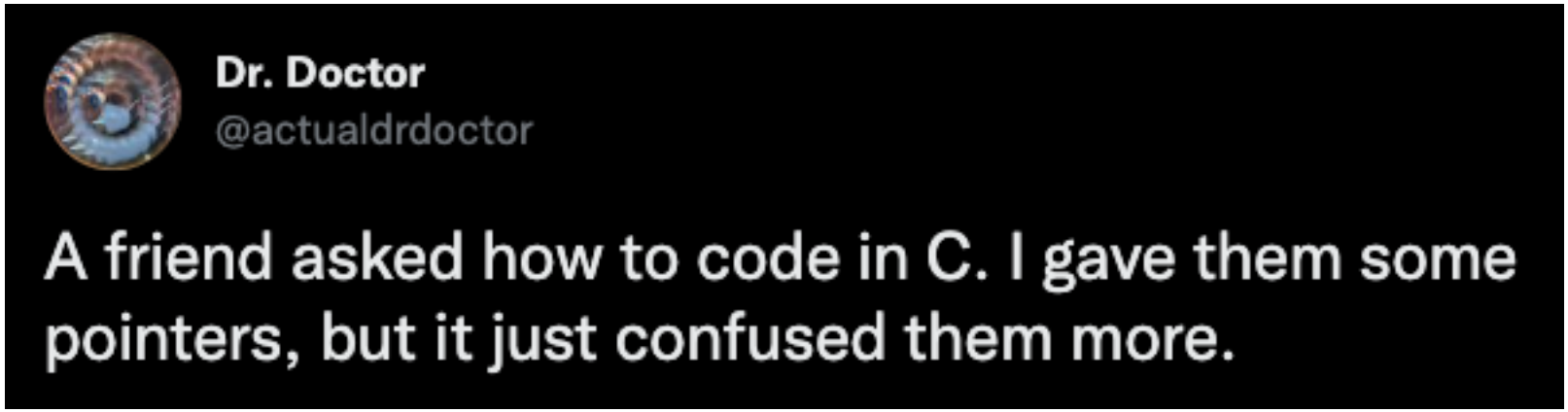
```
void func( int *x);
```

- Use when you want to modify value, and expensive to copy, ptr=0 allowed

Also usable with CONST

Sidebar on pointers (for the C++ crowd)

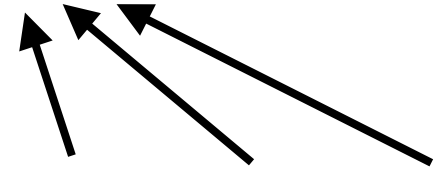
https://twitter.com/actualdrdoctor/status/1527708842422837256?t=lBeSI6V6SkiDIkl9fCehVQ&s=03



Not focusing on pointers (especially for the small bits of code we will be developing) should be very useful and helpful

Binary

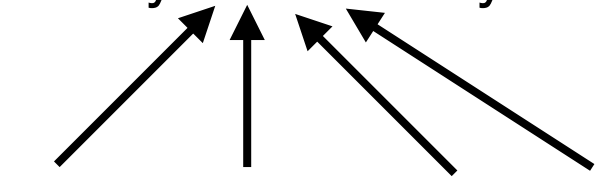
How do we store actual numbers in binary representation? A reminder that 127 in decimal really means:



$1*10^2 + 2*10^1 + 7*10^0$

In other words, when using decimals, each added digit (to the left, ie of more significance) is worth an extra power of ten

How to use binary instead? In binary 1010 really means:



$1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8+0+2+0 =$ 10 (in decimal)

Of course, 1010 in decimal means…. 1010 (in decimal). So we need to know which base we're using!

How to know what base we're using?

Without any indication otherwise, the assumption is we're using base 10 (of course, there is a good reason that this is how we count!)

A 0b prefix implies binary (so 0b1010 = 10)

But writing out numbers in binary and reading them is cumbersome and a pain to us 10-digit machines

```
>>> bin(1234567890)
'0b1001001100101100000001011010010'
```

Other useful bases

Machines use base 2, which don't easily match up to base 10 since neither is a simple exponent of the other (0b1010 = 10). So we often use two bases that are more compatible with binary and are closer to base 10, octal and hexadecimal

A 0 prefix implies octal (we otherwise don't write the leading zero!) and a 0x prefix implies hexadecimal
0b1000 = 8 = 010 (simpler!)
0b10000 = 16 = 0x10 (simpler!)

Digits in octal go from 0-7. Digits in hexadecimal go from 0-f !
(0xa = 10, 0xb = 11, 0xc = 12, 0xd = 13, 0xe = 14, 0xf = 15)

Comparison of bases

Decimal	Binary	Octal	Hex
1	0b1	01	0x1
7	0b111	07	0x7
8	0b1000	010	0x8
15	0b1111	017	0xf
22	0b10110	026	0x16
39	0b100111	047	0x27

Let’s check all these together carefully!

An oft-seen phrase

There are 10 types
of people in the
world.
Those who understand
binary, and those who
don't.

On egg salad

The novel further describes an intra-Lilliputian quarrel over the practice of breaking eggs. Traditionally, Lilliputians broke boiled eggs on the larger end; a few generations ago, an Emperor of Lilliput, the Present Emperor's great-grandfather, had decreed that all eggs be broken on the smaller end after his son cut himself breaking the egg on the larger end. The differences between Big-Endians (those who broke their eggs at the larger end) and Little-Endians had given rise to "six rebellions ... wherein one Emperor lost his life, and another his crown". The Lilliputian religion says an egg should be broken on the convenient end, which is now interpreted by the Lilliputians as the smaller end. The Big-Endians gained favour in Blefuscu.

Gulliver and the Emperor of Lilliput, from a French edition of *Gulliver's Travels* (1850s)

The Big-Endian/Little-Endian controversy reflects, in a much simplified form, British quarrels over religion. Less than 200 years previously, England had been a Catholic (Big-Endian) country; but a series of reforms beginning in the 1530s under King Henry VIII (reigned 1509–1547), Edward VI (1547–1553), and Queen Elizabeth I (1558–1603) had converted most of the country to Protestantism (Little-Endianism), in the episcopalian form of the Church of England. At the same time, revolution and reform in Scotland (1560) had also converted that country to Presbyterian Protestantism, which led to fresh difficulties when England and Scotland were united under one ruler, James I (1603–1625).[13]:31

Wikipedia: https://en.wikipedia.org/wiki/Lilliput_and_Blefuscu
(Gulliver's travels)

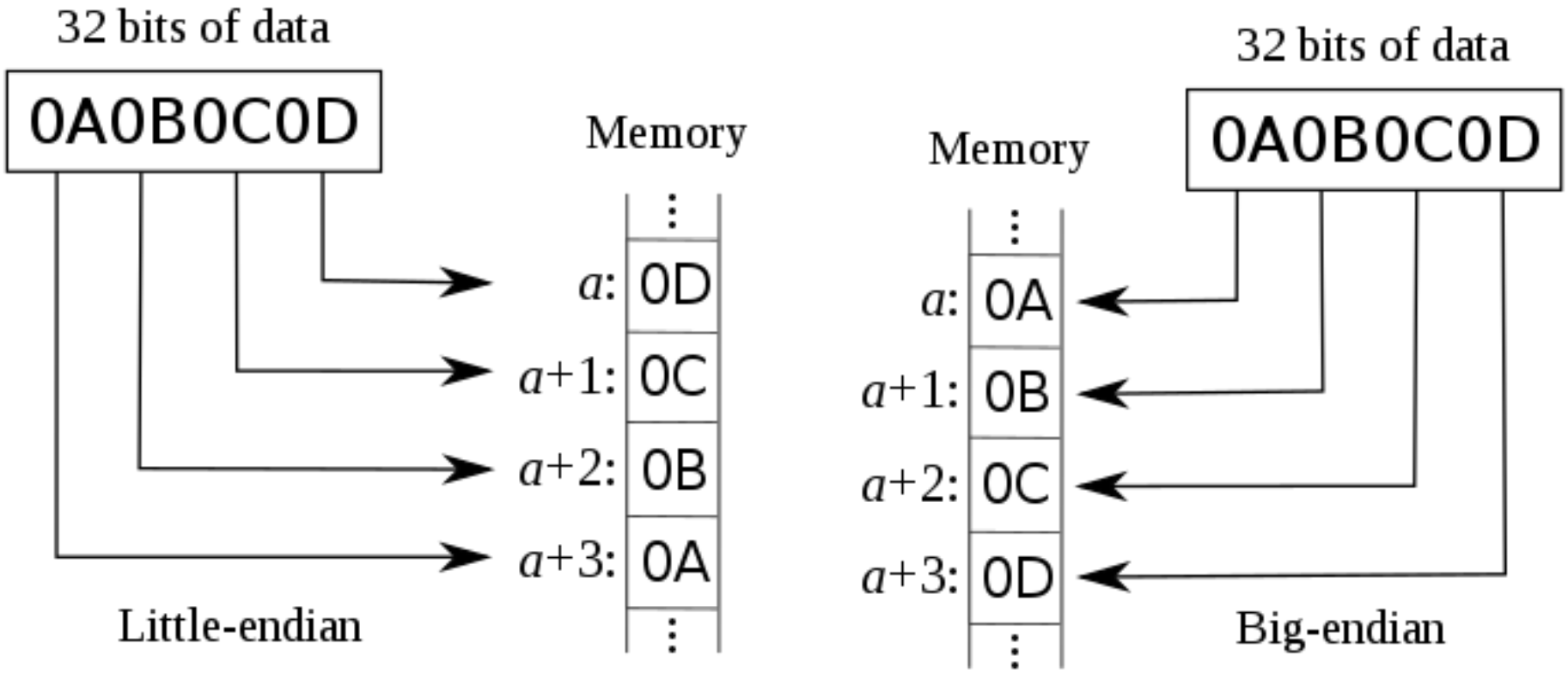
What do deviled eggs have to do with us?

How do we decide how to store our bits in the transistors of the computer (or in any other binary system)?

Computer memory has no concept of “leftmost digits” or “rightmost digit” but rather “address a, address a+1, address a+2, etc”

Big endian: Most significant digit is first
Little endian: Least significant digit is first

Endianness, pictorially



https://en.wikipedia.org/wiki/Endianness

Just conventions, neither one is better or more right than the other, but important in many places to know which one your machinery is using!

And how do we deal with negative numbers?

Decimal	Binary
1	0b1
7	0b111
8	0b1000
15	0b1111
22	0b10110
39	0b100111
-39	**0b-100111 = 0b1100111**

Great, but how do we store a negative sign? Well, we can just use another bit (choose one, for example the most significant bit, 1 = “negative”, 0 = “positive”, though this is just one choice)

Let's try some elementary school math

Decimal: 6+2 = 8, Binary: 0b0110 + 0b0010 = 0b1000

Let's check that we see this together

Decimal: 6 - 2 = 4, Binary: 0b0110 - 0b0010 = 0b0100

Let's check that we see this together, this works

And something else you learn in elementary school:

Decimal: 6 + (-2) = 4

Binary (adding in signs in **bold**): 0b**0**0110 + 0b**1**0010 = 0b**1**1000

Let's check this together… but that clearly isn't right!!!!!

We want: 0b**0**0110 + 0b**1**0010 = 0b**0**0100

A better way to store negative numbers

Two's complement to store negative numbers: If we have N bits available to represent a number in binary, subtract the negative of the number we want the two's complement of from 2^N

Example: 12 = 0b**0**1100 (5 bits)

Two's complement of -12 = 2^5 - 12 = 32-12 = 20 = 0b**0**10100

Let's check that we see this together

Naive storage of -12: 0b**1**1100

Two's complement version: 0b10100 (drop extra bit)

A (much) easier way to get the two's complement: flip the bits of the negative of the number and add 1:

12 = 0b**0**1100 (start with negative of the number, ie +12)

0b**1**0011 (bits flipped)

0b10100 (add one)

Why is two's complement useful?

Let's check what 12-5 is… using 5 bits.
I hope you can do this in decimal :)

In two's complement, positive numbers remain as expected, so
12 = 0b**0**1100

What is 12-5? It is equal to 12+(-5)

What is -5 in two's complement?

-(-5) = 5 = 0b**0**0101
Flip the digits -> 0b**1**1010
Add 1: 0b**1**1011

So 12-5 = 0b**0**1100 + 0b**1**1011 = 0b100111, and we truncate to get 0b**0**0111 = 7! It works!!!! It's nice when simple arithmetic gives the expected results

What does two's complement work?

First, a reminder that to get the right number using two's complement, you must know the number of bits available (since inverting a 0 means you get a 1 in that place!)

Next, remember that the negative of a number is equivalent to 0 minus that number. Now let's go back to our example from the last slide and work out together what happens if we do that.

In order to correctly deal with negative numbers, we also need to deal with overflow and truncation. So instead of subtracting from zero, we can subtract from 10000000000…… (where we have N zeros, so the 1 is an overflow extra bit). And how do we write that number we subtract from?

It is (1 + 11111111….) Subtracting from 11111111… is the equivalent of inverting the bits of the digits. And then we add back in that last one!

Two's complement, some examples

Assuming 8 bits! Let's check a few of these together

Decimal	Binary
1	0b01111111
7	0b01111001
33	0b01011111
-1	0b11111111
-7	0b11111001
-33	0b11011111
0	0b00000000

0b**1**0000001 -> 0b**0**1111110 -> 0b**0**1111111

0b**1**0000111 -> 0b**0**1111000 -> 0b**0**1111001

0b**1**0100001 -> 0b**0**1011110 -> 0b**0**1011111

0b**0**0000001 -> 0b11111110 -> 0b11111111

0b**0**0000111 -> 0b**1**1111000 -> 0b**1**1111001

0b**0**0100001 -> 0b**1**1011110 -> 0b**1**1011111

Other useful bases

One bit is the smallest unit on a computer, but it's very small and not typically useful. We typically refer to a byte as 8 bits (though this can vary on architectures)

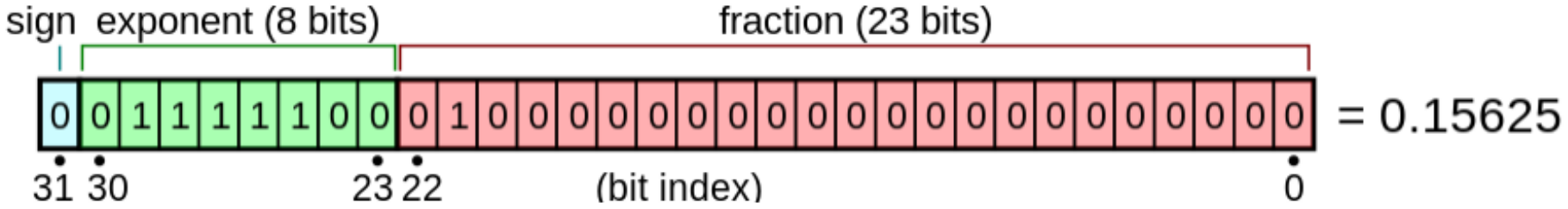
So on the previous slide, each collection of 8 bits was one byte. A byte can store $2^8 = 256$ different numbers. If unsigned, 0-255. If signed, -128 to +127!

We can cleanly represent a byte as 0x0-0xff, and we can put bytes together to make bigger quantities. If we use 2 bytes (16 bits, 0x0-0xffff) we have access to 2^{16} numbers, so 0-65535 or -32768 to +32767

What about non-integer numbers?

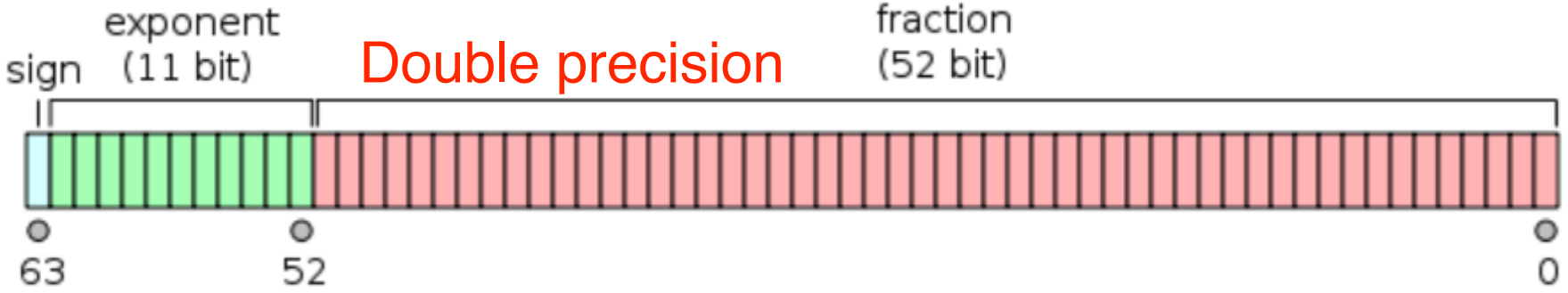
How to store 3.14159 in discrete bits? Use scientific notation! Typically use 32 bits/4 bytes or 64 bits / 8 bytes. One of the bits is often used for the sign, so:

Single precision



(Wikipedia)

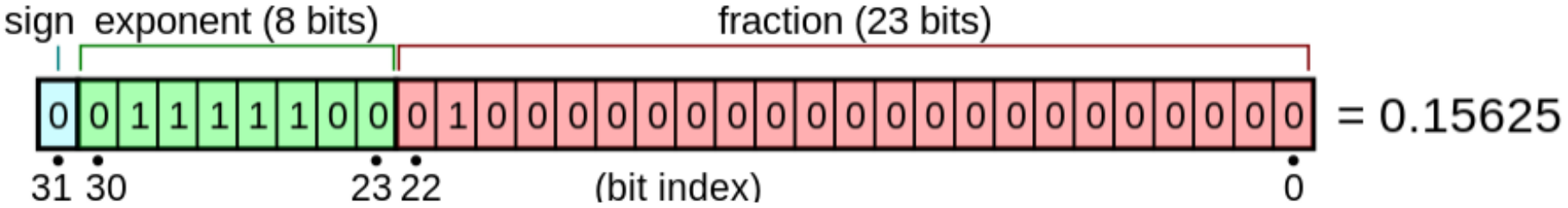
Double precision



How does that IEEE single precision work?

1.f = mantissa. Note that we can only compare floating points within the precision of the mantissa! Better to assign a tolerance level you are willing to live with: if(abs(x-pi) < epsilon): ….

Here's a simple example from Wikipedia



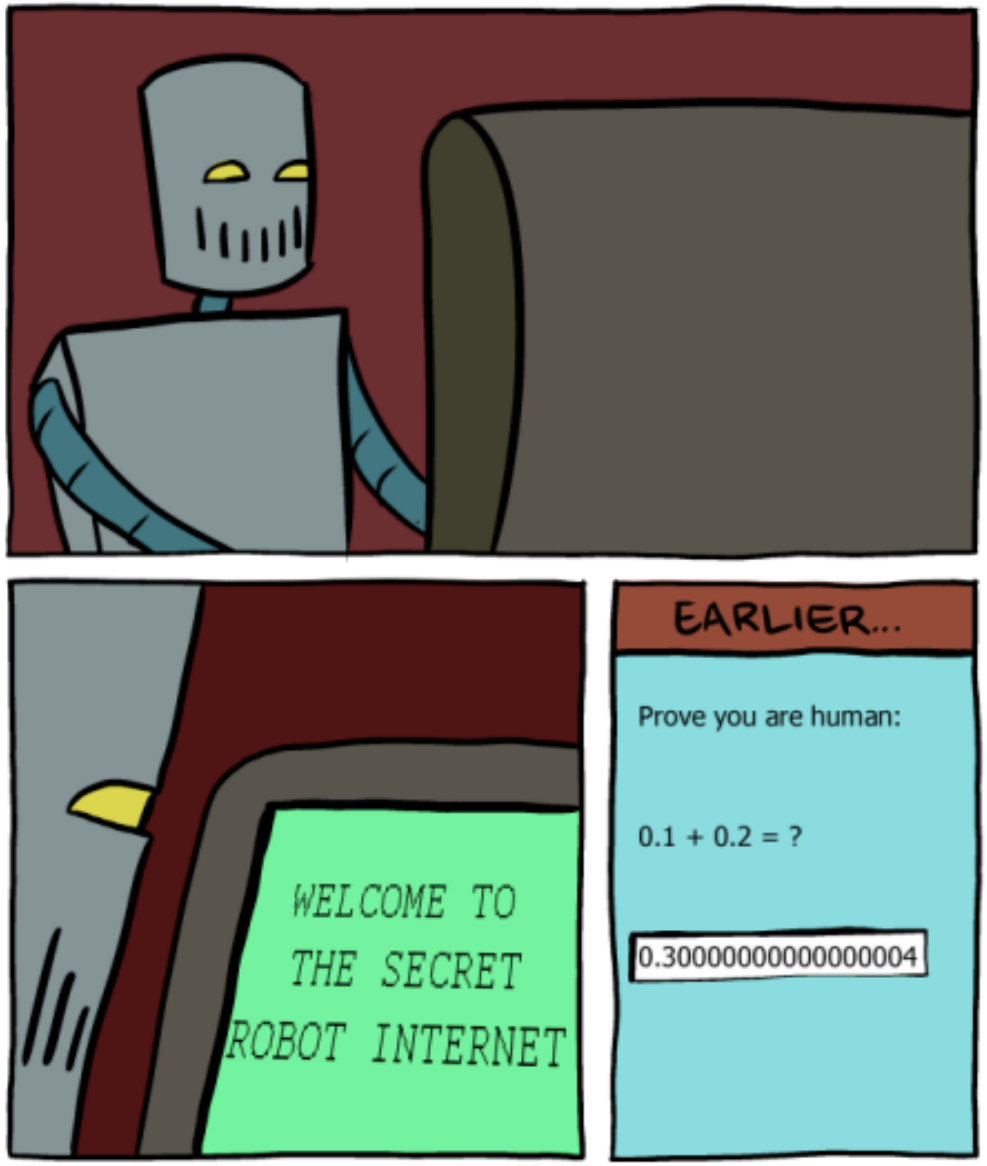
(Wikipedia)

$$\text{value} = (-1)^{\text{sign}} \times 2^{(e-127)} \times \left(1 + \sum_{i=1}^{23} b_{23-i} 2^{-i}\right)$$

Sign is $(-1)^0 = 1$ Exponent is 0b01111100 = 124

Fraction is $1+1*2^{-2} = 1+0.25=1.25$ Value is $1*2^{-3}*1.25 = 0.15625$

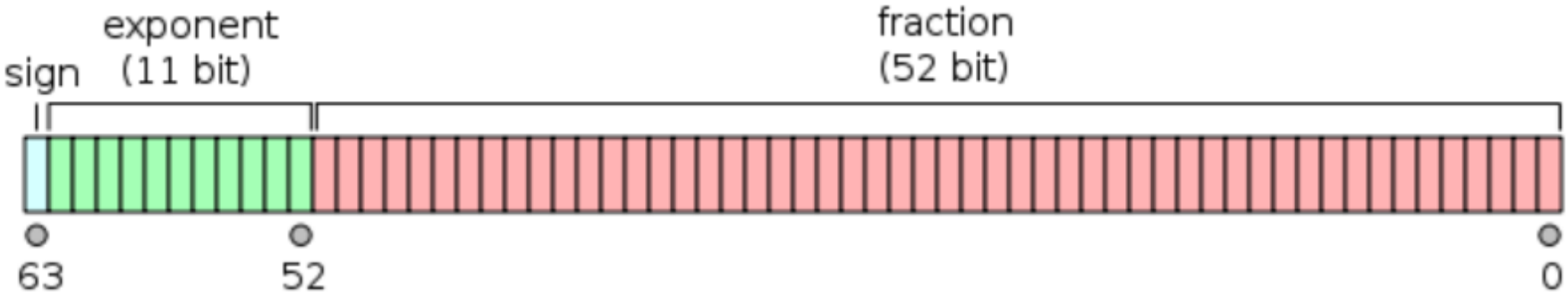
Comparisons of floating points



http://www.smbc-comics.com/?id=2999

How does that IEEE double precision work?

Here's a simple example from Wikipedia



$$(-1)^{\text{sign}} \left(1 + \sum_{i=1}^{52} b_{52-i} 2^{-i}\right) \times 2^{e-1023}$$

(Wikipedia)

Note that in either format, integers can be exactly represented (if small enough). But otherwise, there is an implicit precision involved for floating point numbers! Keep in mind as you use them

How do we add numbers in these formats?

Can't just take the bits and add them as expected! Our IEEE formats are really (binary) scientific notation. How do we add numbers in scientific notation? We need to make sure we add the numbers with the same exponents

Take the number with the smaller exponent and shift the mantissa so that it has the same exponent as the number with the larger exponent. Then we can add things as expected

And subtraction is then using two's complement and doing the same thing. Multiplication and division? Add/subtract exponents and then work on the mantissas

C++ vs Python

C++ is a strongly typed language. You have to declare in advance the sort of object and number you are representing - Is it signed? Unsigned? Is it an integer? Or a floating point? How many bits do you want to use to represent it?

This is often a source of confusion (and occasional misery!), though many of the types can easily be converted from one to another. C++ prefers that you declare this in advance so that you know what the computer is doing and why.

For example, if you are using two bytes to store a number, the difference in the range of possible values depending on whether it is signed or not (-32768 to +32767 vs 0 to 65535) is important!

Types in Python are not static and can change!

```
>>> x=4/3.
>>> type(x)
<type 'float'>
>>> x
1.3333333333333333
>>> x=4/3
>>> type(x)
<type 'int'>
>>> x
1
```

TRUNCATION!
Let's discuss

Python2 vs Python3


```
/Users/jahreda python3
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 26 2016, 10:47:25)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2/7
0.2857142857142857
>>> 2//7
0
>>> ^D
/Users/jahreda python2
Python 2.7.14 |Anaconda, Inc.| (default, Dec  7 2017, 11:07:58)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 2/7
0
>>> 2//7
0
```

C++ types

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
`short`	`short int`	at least **16**	**16**	**16**	**16**	**16**
`short int`						
`signed short`						
`signed short int`						
`unsigned short`	`unsigned short int`					
`unsigned short int`						
`int`	`int`	at least **16**	**16**	**32**	**32**	**32**
`signed`						
`signed int`						
`unsigned`	`unsigned int`					
`unsigned int`						
`long`	`long int`	at least **32**	**32**	**32**	**32**	**64**
`long int`						
`signed long`						
`signed long int`						
`unsigned long`	`unsigned long int`					
`unsigned long int`						
`long long`	`long long int` (C++11)	at least **64**	**64**	**64**	**64**	**64**
`long long int`						
`signed long long`						
`signed long long int`						
`unsigned long long`	`unsigned long long int` (C++11)					
`unsigned long long int`						

https://en.cppreference.com/w/cpp/language/types

C++ types

Type	Size in bits	Format	Value range	
			Approximate	Exact
character	8	signed		**-128** to **127**
		unsigned		**0** to **255**
	16	unsigned		**0** to **65535**
	32	unsigned		**0** to **1114111** (**0x10ffff**)
integer	16	signed	**± 3.27 · 10^4**	**-32768** to **32767**
		unsigned	**0** to **6.55 · 10^4**	**0** to **65535**
	32	signed	**± 2.14 · 10^9**	**-2,147,483,648** to **2,147,483,647**
		unsigned	**0** to **4.29 · 10^9**	**0** to **4,294,967,295**
	64	signed	**± 9.22 · 10^{18}**	**-9,223,372,036,854,775,808** to **9,223,372,036,854,775,807**
		unsigned	**0** to **1.84 · 10^{19}**	**0** to **18,446,744,073,709,551,615**
floating point	32	IEEE-754	• min subnormal: **± 1.401,298,4 · 10^{-45}** • min normal: **± 1.175,494,3 · 10^{-38}** • max: **± 3.402,823,4 · 10^{38}**	• min subnormal: **±0x1p-149** • min normal: **±0x1p-126** • max: **±0x1.fffffep+127**
	64	IEEE-754	• min subnormal: **± 4.940,656,458,412 · 10^{-324}** • min normal: **± 2.225,073,858,507,201,4 · 10^{-308}** • max: **± 1.797,693,134,862,315,7 · 10^{308}**	• min subnormal: **±0x1p-1074** • min normal: **±0x1p-1022** • max: **±0x1.fffffffffffffp+1023**

https://en.cppreference.com/w/cpp/language/types

Why do we care about all this stuff?

If you want to know how long it will take for my car to go from point A to point B, it probably doesn't matter that your floating point answer has finite precision.

But finite precision is a source of **error**, and in many cases, these sources of error can build up and accumulate and become quite critical or even dominant for many calculations

Can consider the absolute error on a number in absolute terms. If x_0 is the true value and x is the stored/estimated value in the computer, then the **absolute error is $|x_0-x|$**. This is useful in many cases, but an error of 1 earth-year on the age of the Earth is probably not a big deal for the age of the Universe, even if it is absolutely a long time. So we can also consider the relative or fractional error, **$|x_o-x|/x_0$**

Quadratic equations example

Exercise 4.2 in the book is an interesting one. Let's look at the solutions to quadratic equations

x = [-b +/- sqrt(b^2 - 4ac)]/(2a).

Let sqrt(b^2-4ac) = s

Then x = [-b +/- s][2a]

Multiply x by 1 = [-b -/+ s]/[-b -/+ s] to get

x = [b^2 +/- bs -/+ bs - s^2][-2ab -/+ 2as]

x = [b^2 - s^2][-2ab -/+ 2as]

But s^2 = b^2 - 4ac so

x = [b^2 - b^2 + 4ac]/[-2ab -/+ 2as]

x = [4ac]/[2ab -/+ 2as] = [2c]/[-b -/+ s]

Two "equivalent" ways to write the solutions to a quadratic equation. But what do they give?

```
[8]  from math import sqrt
     def quad(a,b,c):
       print("")
       print(a,b,c)
       squareroot = sqrt(b*b-4*a*c)
       sol1 = (-b + squareroot)/(2*a)
       sol2 = (-b - squareroot)/(2*a)
       sol1_prime = 2*c/(-b-squareroot)
       sol2_prime = 2*c/(-b+squareroot)
       print("Standard approach:",sol1,sol2)
       print("And standard approach we get back:",a*sol1*sol1+b*sol1+c,a*sol2*sol2+b*sol2+c)
       print("Alternate approach:",sol1_prime,sol2_prime)
       print("And alternate approach we get back:",a*sol1_prime*sol1_prime+b*sol1_prime+c,a*sol2_prime*sol2_prime+b*sol2_prime+c)

     quad(2,10,5)
     quad(0.001,1000,0.001)
```

Quadratic equations example

```
quad(2,10,5)
quad(0.001,1000,0.001)
```

```
2 10 5
Standard approach: -0.5635083268962915 -4.436491673103708
And standard approach we get back: 8.881784197001252e-16 -7.105427357601002e-15
Alternate approach: -0.5635083268962916 -4.436491673103709
And alternate approach we get back: 0.0 0.0

0.001 1000 0.001
Standard approach: -9.999894245993346e-07 -999999.999999
And standard approach we get back: 1.0575401665491313e-08 7.247924804689582e-08
Alternate approach: -1.000000000001e-06 -1000010.5755125057
And alternate approach we get back: 0.0 10575.62534720993
```

Looks nice

Eeeeek!

x = [-b +/- sqrt(b^2 - 4ac)]/(2a).

Let sqrt(b^2-4ac) = s

Then x = [-b +/- s][2a]

Multiply x by 1 = [-b -/+ s]/[-b -/+ s] to get

x = [b^2 +/- bs -/+ bs - s^2][-2ab -/+ 2as]

x = [b^2 - s^2][-2ab -/+ 2as]

But s^2 = b^2 - 4ac so

x = [b^2 - b^2 + 4ac]/[-2ab -/+ 2as]

x = [4ac]/[2ab -/+ 2as] = [2c]/[-b -/+ s]

What happens if b is large? Standard solution has a value (-b) being added to another value (the stuff in square root). If b is large then the square root ~ b, and we are computing -b + b! Large errors!

Quadratic equations example

```
quad(2,10,5)
quad(0.001,1000,0.001)
```

```
2 10 5
Standard approach: -0.5635083268962915 -4.436491673103708
And standard approach we get back: 8.881784197001252e-16 -7.105427357601002e-15
Alternate approach: -0.5635083268962916 -4.436491673103709
And alternate approach we get back: 0.0 0.0

0.001 1000 0.001
Standard approach: -9.999894245993346e-07 -999999.999999
And standard approach we get back: 1.0575401665491313e-08 7.247924804689582e-08
Alternate approach: -1.000000000001e-06 -1000010.5755125057
And alternate approach we get back: 0.0 10575.62534720993
```

Looks nice

Eeeeek!

x = [-b +/- sqrt(b^2 - 4ac)]/(2a).

Let sqrt(b^2-4ac) = s

Then x = [-b +/- s][2a]

Multiply x by 1 = [-b -/+ s]/[-b -/+ s] to get

x = [b^2 +/- bs -/+ bs - s^2][-2ab -/+ 2as]

x = [b^2 - s^2][-2ab -/+ 2as]

But s^2 = b^2 - 4ac so

x = [b^2 - b^2 + 4ac]/[-2ab -/+ 2as]

x = [4ac]/[2ab -/+ 2as] = [2c]/[-b -/+ s]

What happens if b is large? Standard solution has a value (-b) being added to another value (the stuff in square root). If b is large then the square root ~ b, and we are computing -b + b! Large errors! Prefer to add two negative values or two positive values

Quadratic equations example

```
from math import sqrt
def quad(a,b,c):
  print("")
  print(a,b,c)
  squareroot = sqrt(b*b-4*a*c)
  if (b > 0.0):
    sol1 = 2*c/(-b-squareroot)
    sol2 = (-b - squareroot)/(2*a)
  else:
    sol1 = 2*c/(-b+squareroot)
    sol2 = (-b + squareroot)/(2*a)
  print("New approach:",sol1,sol2)
  print("And new approach we get back:",a*sol1*sol1+b*sol1+c,a*sol2*sol2+b*sol2+c)

quad(2,10,5)
quad(0.001,1000,0.001)
quad(2,-10,5)
quad(0.001,-1000,0.001)
```

```
2 10 5
New approach: -0.5635083268962916 -4.436491673103708
And new approach we get back: 0.0 -7.105427357601002e-15

0.001 1000 0.001
New approach: -1.000000000001e-06 -999999.999999
And new approach we get back: 0.0 7.247924804689582e-08

2 -10 5
New approach: 0.5635083268962916 4.436491673103708
And new approach we get back: 0.0 -7.105427357601002e-15

0.001 -1000 0.001
New approach: 1.000000000001e-06 999999.999999
And new approach we get back: 0.0 7.247924804689582e-08
```

Much nicer!

Our code is of course missing one thing. What is it?

Error propagation

Let x and y be two real numbers, and x* and y* their floating point approximations.

Now, let the computer calculate x-y:
First, x and y are replaced by x* and y*.
The final result is then (x*-y*)*.

Example: x=301/2000≈.15050000 and y=301/2001≈.150424787

The exact result is x-y=301/4002000≈.00007521239

Imagine we have decimal floating point numbers with 4-digit mantissa, i.e., x*=.1505 and y*=.1504 so that d*=(x*-y*)*=0.0001
Relative error on x = 0, Relative error on y = 10^{-4} … but relative error on d is 25%!

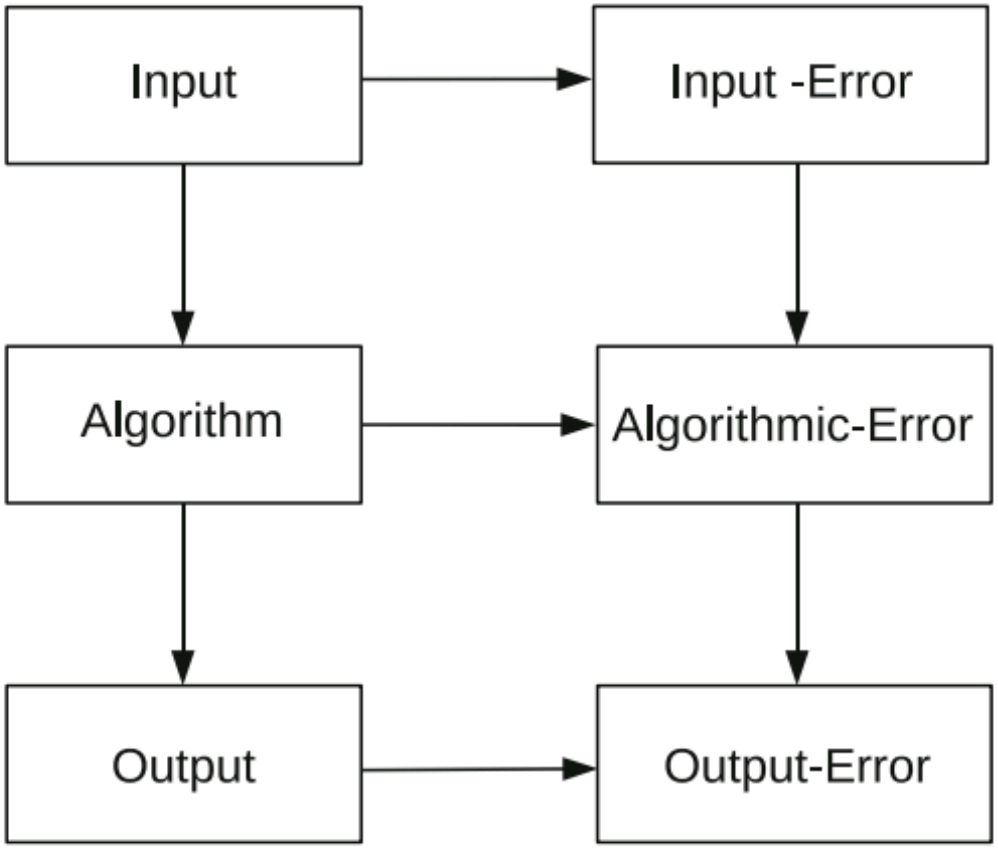
Putting it all together

We will be devising and implementing algorithms this semester. All algorithms are going to be the result of operations on bits, the most basic objects that we have. We have a finite number of these bits, and anytime we move away from integers (which seems pretty likely) we will only have approximations of the solutions.

We want to keep the size of these errors and approximations as small as possible

Errors

Schematic classification of the errors occurring within a numerical procedure



roundoff errors and measurement errors contained in the input data, e.g. input as (round) decimal number → converted to floating point

roundoff errors during evaluation and of **methodological errors** due to mathematical approximations

roundoff errors

Main sources of errors

In any applied numerical computation, there are several key sources of error:

Modeling/Measurement errors

i. Inexactness of the mathematical model for the underlying physical phenomenon.

ii. Errors in measurements of parameters entering the model.

These do not concern us too much. The latter is the domain of the experimentalists. I disagree,

Algorithmic errors but these do not concern us **this semester**

i. **Roundoff errors** in computer arithmetic (finite numerical precision imposed by the computer).

ii. **Discretization (methodological) errors**: continuous processes are replaced by discrete ones (e.g., summation to calculate an integral).

iii. "Termination" errors: infinite algorithms are terminated after a finite number of steps (e.g., iterations like $x_{n+1}=(x_n+a/x_n)/2 \rightarrow \text{sqrt}(a)$, $n \rightarrow \infty$).

See book discussion on program speed!

The latter two are the true domain of numerical analysis and are a consequence of the fact that

- Most systems of equations are too complicated to solve explicitly
- Even with known analytic solution, directly obtaining the precise numerical values may be difficult

Methodological Errors

Result from replacement of mathematical expressions by approximate, simpler ones.

E.g. pendulum period:

$$\tau = 4\sqrt{\frac{\ell}{g}}\int_0^{\frac{\pi}{2}} \frac{\mathrm{d}\alpha}{\sqrt{1-k^2\sin^2(\alpha)}} \qquad k = \sin(\theta_0/2)$$

$$= 4\sqrt{\frac{\ell}{g}}K_1(k)\ .$$

Truncate K_1 series at some N:

$$K_1(k) = \frac{\pi}{2}\sum_{n=0}^{\infty}\left[\frac{(2n)!}{2^{2n}(n!)^2}\right]^2 k^{2n}$$

$$= \frac{\pi}{2}\sum_{n=0}^{N}\left[\frac{(2n)!}{2^{2n}(n!)^2}\right]^2 k^{2n} + R_N(k)$$

Program speed and also ability to implement!

R_N: **truncation error**

Or for derivatives (Chapter 2):

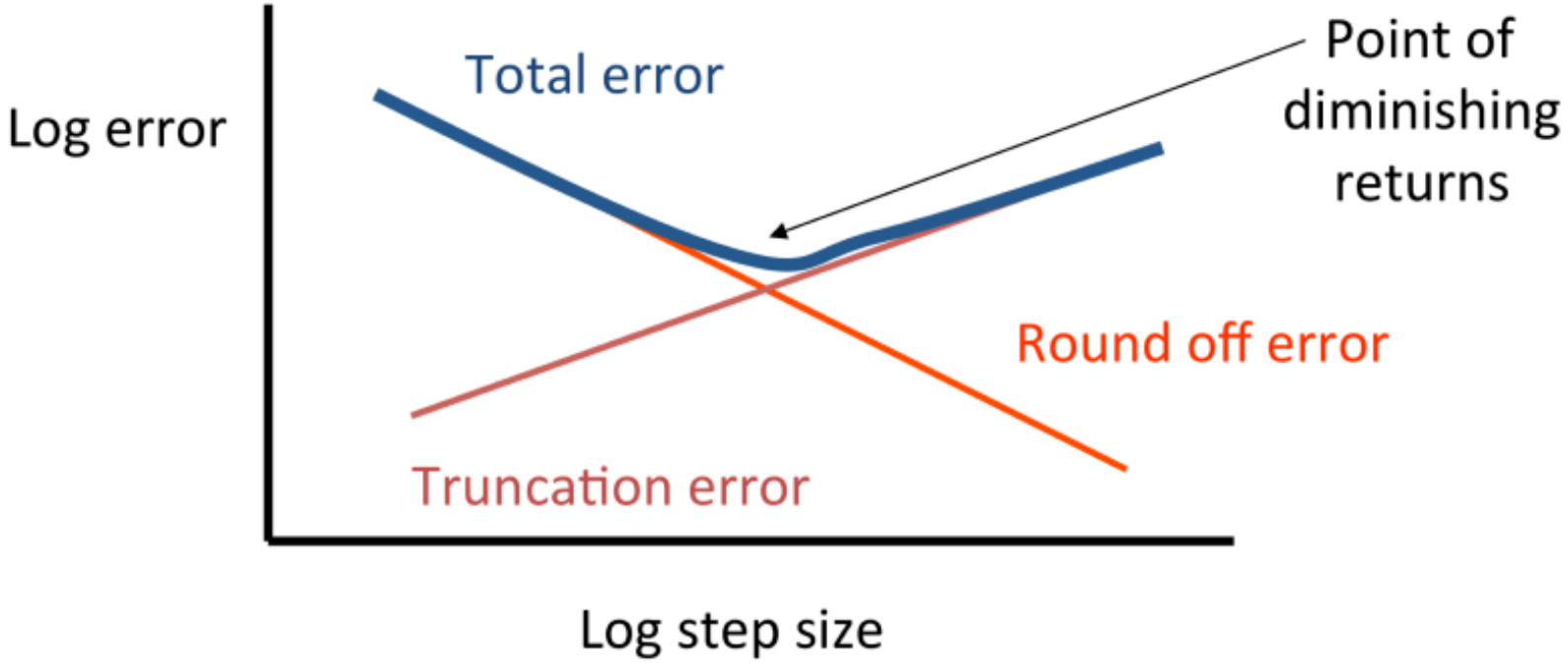
$$\left.\frac{\mathrm{d}}{\mathrm{d}x}f(x)\right|_{x=x_0} = \lim_{h\to 0}\frac{f(x_0+h)-f(x_0)}{h} \approx \frac{f(x_0+h)-f(x_0)}{h}$$

For some finite h → finite difference (relative error can be minimal for some finite! h)

Error trade-off

$$f'(x_0) = \frac{\mathrm{d}}{\mathrm{d}x} f(x)\Big|_{x=x_0} \approx \frac{f(x_0+h) - f(x_0)}{h}$$

- Using a smaller step size reduces truncation error.
- However, it increases the round-off error.
- Trade off/diminishing returns occurs: Always think and test!



Some other thoughts….

From the textbook:

```
def f(x):
  return x*(x-1)

def derivative(x,epsilon):
  return (f(x+epsilon)-f(x))/epsilon

x=1
for epsilon in (1e-4,1e-6,1e-8,1e-10,1e-12,1e-14):
  print (epsilon, derivative(x,epsilon))
```

```
/Users/jahreda/Desktop python3 derivative.py
0.0001 1.0000999999998899   OK
1e-06 1.0000009999177333    Better
1e-08 1.0000000039225287    Even better
1e-10 1.000000082840371     Only better again
1e-12 1.0000889005833413    Getting worse
1e-14 0.9992007221626509      Pretty bad
```

True value of f’(x) = 2*x-1, f’(1) = 1

Stability

An algorithm, equation or, even more general, a problem is referred to as unstable or ill- conditioned if small changes in the input cause a large change in the output.

Example 1:

$$x + y = 2.0,$$
$$x + 1.01y = 2.01$$

Solution: x=1.0, y=1.0

Let us suppose we make a small error in the rhs of the second equation:

$$x + y = 2.0,$$
$$x + 1.01y = 2.02$$

Solution: **x=0.0, y=2.0**

I.e. a 0.05% input error resulted in a 100% wrong result!

Furthermore, if the y-coefficient 1.01 in the original equation would have a 1% error and be 1.0, the equation **would be unsolvable** altogether!

Previous slides from Andreas

Example 2:
$$\begin{cases} \ddot{y} - 10\dot{y} - 11y = 0\,, \\ y(0) = 1, \qquad \dot{y}(0) = -1 \end{cases}$$

General solution: $y = A\exp(-x) + B\exp(11x)$

With initial conditions: $y = \exp(-x)$

Adding a small input error in initial conditions: $y(0) = 1 + \delta$ and $\dot{y}(0) = -1 + \epsilon$

Yields:
$$\bar{y} = \left(1 + \frac{11\delta}{12} - \frac{\epsilon}{12}\right)\exp(-x) + \left(\frac{\delta}{12} + \frac{\epsilon}{12}\right)\exp(11x)$$

i.e. the relative error is
$$\epsilon_r = \left|\frac{y - \bar{y}}{y}\right| = \left(\frac{11\delta}{12} - \frac{\epsilon}{12}\right) + \left(\frac{\delta}{12} + \frac{\epsilon}{12}\right)\exp(12x)$$

→ problem is ill-conditioned: for large values of *x* the second term overrules the first one

See Book for another example for induced instability and the relation to Chaos Theory!

Bringing us to computational physics

Despite what you may learn in all your intro courses, the vast majority of your undergraduate courses and even a good fraction of your graduate courses, it's rare these days that we can completely solve useful physics problems with pen and paper

Most problems don't have analytical solutions, and must be solved with numerical, approximate methods. And even those problems with analytical solutions likely can't be done by you with your pencils. In addition, visualization of solutions is becoming an ever-critical part of physics, and thus of computational physics

What sorts of problems will we need to solve? Integrals, derivatives, eigenvalue/vector problems, Fourier analysis, differential equations, systems of equations, simulation, etc

Another simple example

```
>>> from math import pow
>>> from numpy import arange
>>> exponent = arange(1,20)
>>> exponent
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
>>> for val in exponent:
...    epsilon = pow(10,-val)
...    x = 1 - epsilon
...    y = 1 - x ### should be epsilon
...    print(epsilon,y)
...
0.1 0.09999999999999998
0.01 0.010000000000000009
0.001 0.0010000000000000009
0.0001 9.999999999998899e-05
1e-05 9.999999999954 49e-06
1e-06 1.0000000000287557e-06
1e-07 9.999999994736442e-08
1e-08 1.0000000050247593e-08
1e-09 9.999999717180685e-10
1e-10 1.000000082740371e-10
1e-11 1.000000082740371e-11
1e-12 9.999778782798785e-13
1e-13 1.000310945187266e-13
1e-14 9.992007221626409e-15
1e-15 9.992007221626409e-16
1e-16 1.1102230246251565e-16
1e-17 0.0
1e-18 0.0
1e-19 0.0
>>>
```

Let's take a close look at this

Homework #1

1) Exercise 4.4 in the textbook. You may want to use:

```
import time
tic = time.perf_counter()
…
toc = time.perf_counter()
time_in_seconds = toc - tic
```

2) Give the standard (IEEE) single precision binary representation of the machine approximation for –1/3. Don't find an algorithm to do this, but write it out yourself
3) Find the smallest positive integer that is not exact in single precision. Show your work
4) What are the two's complement representations for these decimals numbers? Use 12 bits: 9, 455, 1021, -12, -1022, -1023.
5) What are the octal, binary and hexadecimal representations of the decimal numbers 13, 133, 1333 and 13333? Show your work