Warmup (L1)

- Log in to Jupyter: https://jupyter.math.uwaterloo.ca/
- Create a new Notebook for “Python 3 (ipykernel)”.
- Write this into Jupyter and press the run button:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/Assignment-00.ipynb
```

- Open “Assignment-00.ipynb” in the left panel
- Do what the first box says.

CS114

Module 1: Introduction

Administrata

CS114 M1

Administrata

- The course web site is the central “hub” for course information:
  - https://student.cs.uwaterloo.ca/~cs114/
- The course outline and contact info for course staff is there
  - Instructor: Gregor Richards
  - Coordinator: Scott Freeman King
  - TAs: cs114@uwaterloo.ca

Why this course?

- All science is done with the assistance of computers, because computers are better at computing than humans are
  - (That's why we call them compute-ers)
- Because of the ubiquity of computers, most people don't know them as compute-ers, but you need to
- You will learn to think about computers and how they fit into science (and life?) differently

Tools and software

- In this course you will be learning to use one popular programming language: Python
- We will mostly be running Python in a web environment called “Jupyter”, by creating so-called “Jupyter Notebooks”.
- Jupyter is a popular environment to use Python on the web, but you will also learn to run it on your own computers.
- All software in this course is free

Resources

- Slides: Available on course web site. Slides may be posted after the lecture day.
- Jupyter notebook: Available from Math department, https://jupyter.math.uwaterloo.ca/
- Assignments: Linked from course web site
- Textbook: Optional, linked via course web page
- Alternate textbook: Also optional, available via library access online
- Later in the term you will need a computer to run Python locally

Lectures

- This course has no textbook
- Lectures are where you learn the material
- The statistics are very clear: students who don’t show up to lectures fail this course
- This course has a deserved reputation as being very difficult; don’t make it harder by not showing up

In-lecture quizzes

- Starting next week, there are in-lecture quizzes
- These are for participation, not correctness
- How participation affects your grade is a bit mathematically complicated (it's not just a % of your course grade), so please read the outline

Course goals

- This course must achieve three goals:
  - Basic introduction to programming
  - Introduction to scientific/numeric programming in particular
  - Sufficient reasoning about program behavior to serve as the intro programming course for a computing minor
- (I didn't choose these goals; don't shoot the messenger!)
- This course is arguably bigger than CS students' first programming course…
- In lectures I try to do as many examples as possible, but there's a very real time crunch

Tutorials

- Tutorials are hands-on experience plus office hours
- No tutorials this week, they start next week (Friday the 18th)
- Even if you're learning the material well, you should attend tutorials; lectures are not very interactive!

Office hours

- Instructor and ISAs have office hours
- Some are in the Physics Tutorial Center
  - Other students in the PTC can help, but only our ISAs know this course. Make sure you check who you're talking to!
- The schedule will be posted on the course web page by next week
- Also start next week

Web resources

- Piazza: Course discussion should go here. Make sure you're signed up. Also a sort of "always-there" office hours.
- Web site: https://student.cs.uwaterloo.ca/~cs114/ . This is the hub for all online resources (everything else is linked from here).
  - Some parts of the course web site may only appear later this week or next week.

Learn

This course does not use learn.uwaterloo.ca

Course components

- Assignments (see outline)
- Tutorial problems (go to Tutorials!)
- Exams
  - You must pass the exam weighted average to pass the course!
- In-lecture quizzes/participation
- Marking scheme described in outline (soon)
- Review academic policies in outline as well

Academic integrity

Cheating on assignments is a disservice to yourself: you won't be prepared for exams, and **you must pass the exams to pass the course**

Academic integrity

- The elephant in the room: AI
  - After this course, many of you will use AI for most or even all programming. That's fine, but
  - AI is stupid and bad at everything. It feels like the opposite when you ask it about something you're not an expert at. It will feel much smarter than you at Python until you learn how mediocre its solutions are.
  - Learning to use AI properly is valuable, but *you* must be the expert, so you must learn the material (programming) first. Use of AI is disallowed on most assignments.

When is it cheating?

- You are allowed and encouraged to talk to other students about *concepts*
- You cross the line when you talk about *code* (that's relevant to this course) or *solutions*
- When in doubt, refrain or ask a member of course staff
  - Feel free to ask on Piazza in a private post

Assignments

- All assignments are programming: writing Python code
- Programming is 10% writing code, 90% figuring out why your code doesn't work
  - (That's just as true of somebody with decades of experience as somebody with days of experience!)
- Start assignments early! Don't judge time by how long you spend programming. Debugging is most of the time!

Submissions

- A uwaterloo CS system called Markus.
  - https://markus.student.cs.uwaterloo.ca/
- Assignments are graded in three stages:
  - Correctness (your code does the right thing)
    - Stage 1: automatic in Markus when you submit
    - Stage 2: by TAs after the deadline
  - Stage 3: Style (your code is understandable)

A subset of Python

- Python is way bigger than we're going to learn in this course
- You are not required to stick to the subset we'll see in this course
- Unfortunately, code that uses advanced features looks like AI ☹
- So, if you do, tell us (in the code) where you learned the features! Cite your sources!

Computation

CS114 M1

What is computation?

- I’m from the last generation that was told this by our teachers:

  “You won’t always have a calculator with you!”

- This advice means well but totally misses the point: knowing how to multiply in your head doesn’t help you know when multiplication is needed
- *Applying* math is a creative endeavor

Calculation

- Calculation is the mechanics of math
  - (Addition, subtraction, multiplication, division, etc.)
- Example: two-body problem

Two-body problem

Given two astronomical bodies' positions and velocities, we can predict their locations and velocities with a simple mathematical expression: just plug in the time

Computation

- Calculation with *repetition* and *decision making*
- Must take a step, then use its results to decide what to do next
- Example: *n*-body problem

n-body problem

With the addition of even a third astronomical body, the problem becomes *chaotic*. Every movement affects the expression. Can no longer predict indefinitely in one calculation.

n-body problem

Instead, we must calculate one small step (a second, a day, a week…), then adjust based on the new positions. Repeat, over and over, until we reach the time we want.

Computation

- Computation is more powerful than calculation
- A computer is to computation as a calculator is to calculation: it is just a dumb computing machine
- Correctly *applying* computation is a creative endeavor

A creative endeavor…

The bad news:

I can’t teach you to creatively apply computation to a problem. No one can.

All I can do is teach you the tools, and give you problems to (hopefully) build your intuition.

Computation and Python

- The only language computers directly understand is electrical impulses
- That's… not very human
- Software exists to translate abstract expressions of computation into those electrical impulses
- *Python* is one such piece of software, and the name for the language to express computation

Why Python

- There are thousands (probably hundreds of thousands) of programming languages
- Why this one?
- Reasons for language choice are *extrinsic*
- Scientists use Python
- It's not that Python is magically good at science, but scientists use Python, and momentum begets momentum

```
def advance(dt, n, bodies, pairs):
    for i in range(n):
        for ([x1, y1, z1], v1, m1,
             [x2, y2, z2], v2, m2) in pairs:
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            dist = sqrt(dx * dx + dy * dy + dz * dz)
            mag = dt / (dist*dist*dist)
            b1m = m1 * mag
            b2m = m2 * mag
            v1[0] -= dx * b2m
            v1[1] -= dy * b2m
            v1[2] -= dz * b2m
            v2[2] += dz * b1m
            v2[1] += dy * b1m
            v2[0] += dx * b1m
        for (r, [vx, vy, vz], m) in bodies:
            r[0] += dt * vx
            r[1] += dt * vy
            r[2] += dt * vz
```

```
def advance(dt, n, bodies, pairs):
    for i in range(n):
        for ([x1, y1, z1], v1, m1,
             [x2, y2, z2], v2, m2) in pairs:
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            dist = sqrt(dx * dx + dy * dy + dz * dz)
            mag = dt / (dist*dist*dist)
            b1m = m1 * mag
            b2m = m2 * mag
            v1[0] -= dx * b2m
            v1[1] -= dy * b2m
            v1[2] -= dz * b2m
            v2[2] += dz * b1m
            v2[1] += dy * b1m
            v2[0] += dx * b1m
        for (r, [vx, vy, vz], m) in bodies:
            r[0] += dt * vx
            r[1] += dt * vy
            r[2] += dt * vz
```

Calculation

Repetition

```
def advance(dt, n, bodies, pairs):
    for i in range(n):
        for ([x1, y1, z1], v1, m1,
             [x2, y2, z2], v2, m2) in pairs:
            dx = x1 - x2
            dy = y1 - y2
            dz = z1 - z2
            dist = sqrt(dx * dx + dy * dy + dz * dz)
            mag = dt / (dist*dist*dist)
            b1m = m1 * mag
            b2m = m2 * mag
            v1[0] -= dx * b2m
            v1[1] -= dy * b2m
            v1[2] -= dz * b2m
            v2[2] += dz * b1m
            v2[1] += dy * b1m
            v2[0] += dx * b1m
        for (r, [vx, vy, vz], m) in bodies:
            r[0] += dt * vx
            r[1] += dt * vy
            r[2] += dt * vz
```

But I thought computers did *x*

- Everything else is just set dressing on computation
- When a computer is displaying this slide, the shape of each letter is computed through a piece of software called a *font rasterizer*
  - (etc., etc., etc.)

Be the emperor of your computer

CS114 M1

Jupyter

- Let’s open a Jupyter notebook and see some Python
- https://jupyter.math.uwaterloo.ca/
  - Other Jupyter servers are available, and you can even run your own, but we’ll be using Math’s here
  - If you ever need an alternate Jupyter server, Google Colab is free

Jupyter as a calculator

- Let's calculate $\frac{4(7+2)^2}{6+5}$
- Note that the result is not an integer, and it's shown rounded (not as an exact fraction)
- Be careful of this rounding; it's real!
- Python has BEDMAS/PEMDAS

Jupyter as a computer

- Python is an *imperative programming language*
- What this means is that you're giving the computer a sequence of *commands* (synonym: imperatives) to run in order
- You're in command. *It only does what you ask.* Imperative comes from the same root as emperor (in Latin, imperator)

Syntax of a program

- Syntax: The grammar of a (programming) language
- Our commands are *statements*
- Mathematical calculations are *expressions*
- Statements contain expressions

Jupyter as a computer

- Calculations are usually incomplete commands
  - For convenience, Jupyter will show the result of the last calculation, but not others
- If you want it to output the result, command it to do so!
  - … by asking it to "`print`", for silly historical reasons

`print(expression)` or
`print(expression, expression…)`

New syntax!

- What's going on with "`print(...)`"?
- `print` is a *procedure*, and this is a *procedure call*
  - (Actually, we far more often use the word "function")
- A procedure is a way of boxing up and parameterizing computational steps

Procedures vs. functions

Brief pedantry aside:

Technically, *functions* are mathematically pure (they aren't composed of imperative steps) and *procedures* are not. In practice, programmers usually don't make this distinction, so we'll call `print` a function.

Strings

- You may want to print text, not just numbers
- In Python, a piece of text is called a *string*
- Just surround it in quotes:

```
print("Hello, world!")
```

(Single quotes also work, and are more common than double quotes. I use double quotes because I'm old.)

The power of change

- Make your steps do something by storing values in *variables*
- Store to a variable with =

```
x = 42
```

- Retrieve the value by using the name

```
x / 12
```

The power of change

- Variables are *variable*. You can not only set them, but *change* them.

```
x = 7
print("x is", x)
x = x * 6 # Huh???
print("x is", x)
```

- Mathematicians hate it!
  (This isn't what "variable" or "=" mean in math)

One step at a time

- Remember, we're running commands *in order*
- If you change a variable after it's been used, that use is in the past

```
x = 7
y = x * 2
x = x * 3
print("x is", x)
print("y is", y)
```

Variable names

- Variable name must
  - start with a letter or underscore, and
  - contain only letters, underscores, and numerical digits. In particular, no spaces.
- Nothing stops you from overwriting `print` or other functions (other than common sense), so try not to step on your own toes!

Variable names

- Multiple words in a variable name: `snake_case` and `camelCase` are both common.
- Style conventions:
  - Use either `snake_case` or `camelCase`, but be consistent (except when an assignment demands)
  - Use descriptive names (not `x`) wherever possible

Variable names

Names are case sensitive
(`a_b_c` is different from `a_B_c` or `A_b_c`)

Jupyter is weird

- Enter this into a cell and run it:

```
z = 42
```

- Then clear the cell, and replace it with this:

```
z / 7
```

- You'll notice that it shows a result. It remembered `z`.

Jupyter is weird

- This “hidden state” is there so that cells can work together (set `z` in one cell, use it in another)
- But, can allow cells to work together through time (set `z` in one cell, delete that assignment, then use it anyway)
- That can be confusing

Jupyter is weird

To make sure you're not relying on this, occasionally use the "run all" button

- This resets all the variables (called "restarting the kernel")

Comments

- In an earlier example, I snuck in the question “Huh?”:
  - `x = x * 6 # Huh???`
- This is called a *comment*. You can make your code easier to read by describing what’s happening using comments.
- Starts with a **#**, goes to the end of the line
- Can be a line of its own

More power

- Right now we have a five-function calculator
- The functions in a scientific calculator are available in a *module* called `math`
- *Modules* are bundles of functions, values, variables, or anything else, grouped together for organization

Pythogoras

- Let’s calculate the length of the hypotenuse of a right triangle given the length of its two sides

Pythogoras

- Let's calculate the length of the hypotenuse of a right triangle given the length of its two sides

```
import math
math.sqrt(a**2 + b**2)
```

Smarter imports

- You can also import a specific name

```
from math import sqrt, log
sqrt(a**2 + b**2)
```

- There's no consistent style for whether to import specific names or whole modules. Use whatever is readable. You *do not* have to be consistent if importing multiple modules.

Too many things!

- The `math` module provides a *lot* of functions… and there are a *lot* of modules!
- Find what's in it with the *tab* key
- The `help` function gives you help on anything you've imported

```
help(sqrt)
```

- You can also use the `dir` function (instead of tab) to find what's in a module

```
dir(math)
```

Reusing (functions)

CS114 M1

Back to Pythagoras

- Note how I wrote the code:

```
sqrt(a**2 + b**2)
```

- What's this *a* and *b*?
- We want to use this code for any value of *a* and *b*
- Right now, if we need it twice, we have to rewrite this code!

Functions

- We already saw functions such as `print` and `sqrt`
- Now let’s write one of our own!

Functions

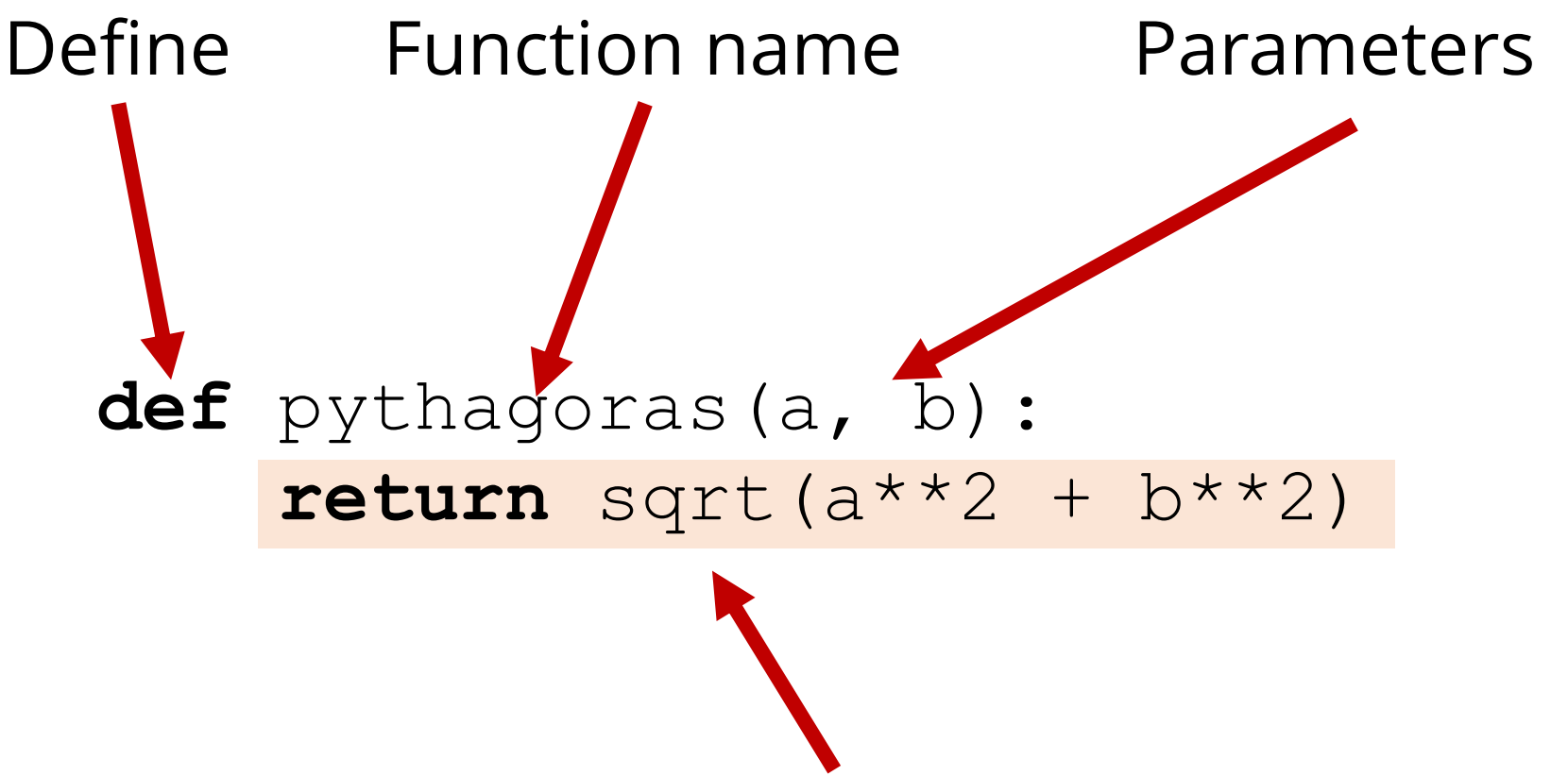
- We already saw functions such as `print` and `sqrt`
- Now let's write one of our own!

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
```

Functions

- We **def**ine a function with the keyword **`def`**
  - (A keyword is a word that has a special meaning in the programming language, other than just a name)
- We want this function to be *parameterized*
  - Similar to `sqrt`. We need to be able to tell `sqrt` what we want to square-root, and we need to be able to tell `pythagoras` the sides of our triangle

Functions


Define
Function name
Parameters
def pythagoras(a, b):
return sqrt(a**2 + b**2)
Body of the function (what it actually does)

Functions

Header

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
```

Body is *indented* past the header
(that's how Python knows it's the body!)

Functions

- Defining the function doesn't make anything happen
- We have to *call* the function for it to run

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
[…]
pythagoras(3, 4)
```

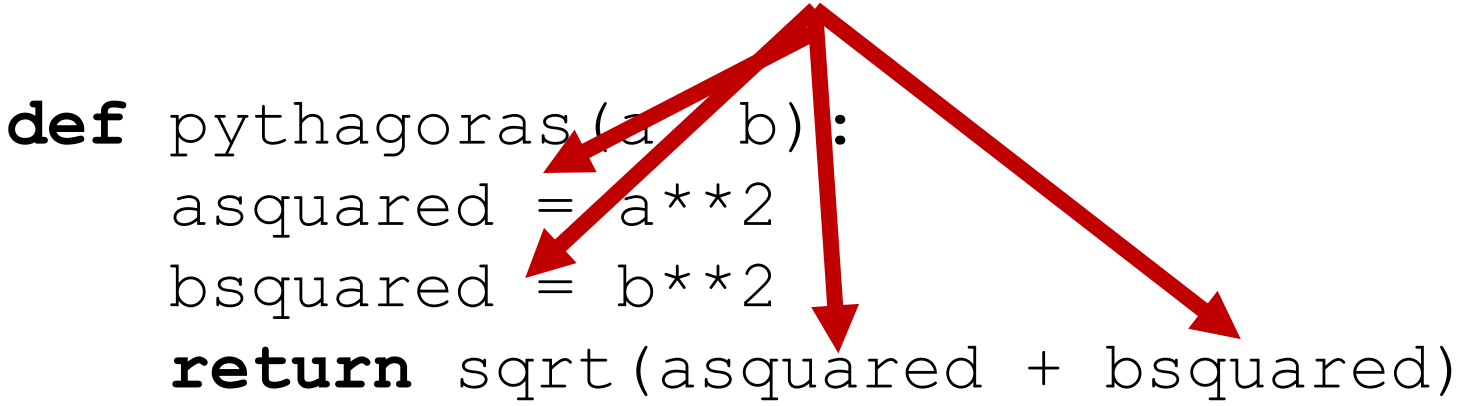
Functions

- A function can be any number of steps

```
def pythagoras(a, b):
    asquared = a**2
    bsquared = b**2
    return sqrt(asquared + bsquared)
```

Functions

Functions can define and use their own variables



```
def pythagoras(a, b):
    asquared = a**2
    bsquared = b**2
    return sqrt(asquared + bsquared)
```

Functions and variables

- They are the function's *own variables*
- They don't exist outside of the function:

```
def pythagoras(a, b):
    asquared = a ** 2
    bsquared = b ** 2
    return sqrt(asquared + bsquared)
pythagoras(3, 4)
print(asquared) # ERROR!
```

Functions and variables

- Parameters are also variables
- You can overwrite them

```
def pythagoras(a, b):
    a = a ** 2
    b = b ** 2
    return sqrt(a + b)
```

- This is usually confusing and should usually be avoided, but use common sense

Local vs. global

- Variables within a function are *local variables* of that function
- Variables outside the function (defined for the whole kernel) are *global variables*

Pedantry aside

- I've used the terms "parameter" and "argument"
  - In `pythagoras`, `a` and `b` are parameters
  - The value you actually pass in is the argument: the argument fills the parameter
- The confusion: if we talk about what `pythagoras` does, we'll talk about `a` and `b` as their future values, the arguments
- Often used interchangeably, but technically not the same

Functions calling functions

- Our `pythagoras` function calls `sqrt`
- We can also call our own functions in functions

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
[…]
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
```

Returning

- Here, we use a call to `pythagoras` as a value in a calculation. What actual number is used in the calculation?

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
[…]
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
```

Returning

- The **`return`** statement of the `pythagoras` function gives a value to whoever called `pythagoras`

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
[…]
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
```

Returning

- ***WARNING!*** Returning *ends* the function, even if there are more statements left!

```
def pythagoras(a, b):
    return sqrt(a**2 + b**2)
    print("This will never be printed!")
```

- Why would you want this? We'll see when we get to decision-making.

Functions calling functions

- `print` is also a function
- Very useful for understanding and debugging code!

```
def pythagoras3(a, b, c):
    h = pythagoras(a, b)
    print("Hypotenuse of one face:", h)
    return sqrt(h**2 + c**2)
```

Understanding

- I cannot overstate enough the value of using `print` to understand and debug your code!
- `print`ing what's happening while your code runs is your first line of defense! Use it! A LOT!

To return or not?

- A function doesn't have to explicitly return

```
def pythagoras(a, b):
    sqrt(a**2 + b**2)
```

- But, if it doesn't, it still returns a value: "`None`", a special value that means "there is no value". … but it is still a value…

None is Hell

- Why have functions return `None`?
- We want functions that are just there to *do* something, rather than *returning* something…
- but Python has no way of distinguishing these two kinds of functions.
- So, Python needed *something* for `x = print("Whoops!")` to do.

None is Hell

- You can store `None` in a variable and Python won't report anything wrong
- You can pass `None` as an argument to a function, and it'll work until (and unless!) it's used
- Forgetting to **`return`** causes a problem to arise *distant from the actual error in the code*

Bugs, bugs, bugs!

CS114 M1

Debugging

- Time for debugging!

```
from math import sqrt
def pythagoras(a, b):
    sqrt(a**2 + b**2)
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
print(pythagoras3(4, 5, 6))
```

Debugging

One thing to be careful of when debugging: errors happen when code *runs*, so if you don't run a buggy function, you won't see the problem!

```
from math import sqrt
def pythagoras(a, b):
    sqrt(a**2 + b**2)
def pythagoras3(a, b, c):
    return sqrt(pythagoras(a, b)**2 + c**2)
print(pythagoras(4, 5))
```

Avoiding bugs

- Develop incrementally (one small step at a time)
- Test every small part by making sure it behaves as you expect. This can be **`return`**ing what you expect or just doing what you expect.
- Use `print` to spot-check

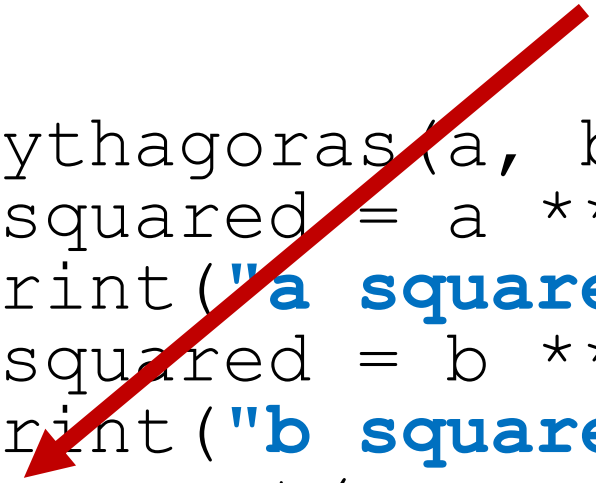
Avoiding bugs: printing

- Make sure whatever you print is descriptive/unique enough that you know which is which

```
def pythagoras(a, b):
    asquared = a ** 2
    print("a squared:", asquared)
    bsquared = b ** 2
    print("b squared:", bsquared)
    r = sqrt(asquared + bsquared)
    print("result:", r)
    return r
```

Avoiding bugs: printing

Don’t be afraid to introduce variables just so that you can print something from the middle of a calculation!



```
def pythagoras(a, b):
    asquared = a ** 2
    print("a squared:", asquared)
    bsquared = b ** 2
    print("b squared:", bsquared)
    r = sqrt(asquared + bsquared)
    print("result:", r)
    return r
```

Avoiding bugs: printing

- When you're done debugging, make sure you remove the prints. They'll confuse our tests!
- You can also *comment out* prints, so you can remove them without forgetting them

```
def hypotenuse(a, b):
    # print("a was", a)
    return sqrt(a**2 + b**2)
```

Avoiding bugs: documentation

- Part of avoiding bugs is *good documentation*
- You shouldn't name a function "`pythagoras`"
  - The Pythagorean theorem is an implementation detail
  - When you're calling the function, you don't care how it's implemented
  - It should've been named "`hypotenuse`"

Avoiding bugs: documentation

- Remember the `help` function?
- We can (and should!) document our functions in the same way, with *docstrings*
- Let's add a docstring to `pythagoras`

Avoiding bugs: documentation

```
def pythagoras(a, b):
    """
    Return the length of the
    hypotenuse of a right
    triangle with side lengths a
    and b.
    """
    return sqrt(a**2 + b**2)
```

Docstrings

- The docstring is the first statement in a function
- Triple-quote can be used to make a multi-line string anywhere; it's only a docstring when it's the first statement

```
def sillystring():
    return """
    Hello world!
    """ # This is not a docstring
```

Docstrings

- In this course, you must write a docstring for every function you write
- Your code is graded not just for correctness (doing what it's supposed to) but for documentation and readability!

Docstrings

- Conventions for good docstrings:
  - Refer to parameters by name

```
"""… with side lengths a and b."""
```

not

```
"""… with the two sides."""
```

Docstrings

- Conventions for good docstrings:
  - Describe what it does, not how it does it

```
"""Return the length…"""
```

not

```
"""Return sqrt(a**2+b**2)"""
```

Docstrings

- Conventions for good docstrings:
  - The docstring should be a command for the function to obey

```
"""Return the length…"""
```

not

```
"""Computes the length…"""
```

Comments

- Comments (with #) should be used to clarify
- It's assumed that whoever's reading the code knows English and Python, so
  - If your variable names are good and the steps are obvious, you may not need comments to make the code understandable
  - but, err on the side of caution: use comments where it *might* be confusing
  - Assume the reader is as stupid as possible while still being able to read English and Python

Avoiding bugs: types

- We've seen numbers and we've seen strings
- Look at the result of `hypotenuse`: note how `hypotenuse(3, 4)` is written as 5.0, not just 5
- Internally, Python stores integers and rationals differently
- It is often useful to distinguish between them

Numeric types

- In Python, we can store a number as an `int` or a `float`. `int` corresponds to integers. `float` corresponds to rationals (and is used to approximate reals).
- Since all integers are rationals, we can store an integer as a `float`.
  - If a number *could have been* a non-integer, it's usually a `float`, even if it *is* an integer in practice!

Numeric types

- In Python, we can store a number as an `int` or a `float`. `int` corresponds to integers. `float` corresponds to rationals (and is used to approximate reals).
- Note "*corresponds to*" here. An `int` can store any integer (as long as your computer has enough memory!), but `float`s have limited capacity. It's hard to intuit about, so just remember: it's rounded!

Float?

- "`float`" stands for "floating point"
- It means the point (the dot separating the integer part from the fractional part) can float (be anywhere within the number)
- Don't overthink the name. It's just a name.

Numeric types

- It mostly won't matter how a number is stored (`float` or `int`)
- … but it can. We'll see situations later where it matters.
- You can convert in a few ways:

```
float(42) # 42.0
int(41.999) # 41
round(41.999) # 42
```

Documenting types

- It's often useful to document what type you expect something to be
  - If it's the wrong type, your code will do something unexpected!
- When writing a function, you can (and should!) document the types of its parameters and the type it returns

Documenting types

```
def hypotenuse(a: float, b: float) -> float:
    return sqrt(a**2 + b**2)
```

- These *type annotations* are documentation. Even if they're wrong, your code will run.
- If you set up Jupyter with `Assignment-00.ipynb`, it will warn you when they're wrong

Documenting types

- Although documentation, annotations are so-called *checked documentation*
- That is, they don't change your code (just document it), and yet we can check that they're correct
- If you don't use types as you describe them, you'll get typing errors, but your code will still run

int vs float

- Every integer is a rational, so it's tempting to write `float` everywhere
- This is poor documentation! If you expect an integer, write `int`

```
def stableNeutrons(protons: int) -> float:
    """
    Return approximately how many neutrons
    are needed to stabilize a nucleus with
    this many protons.
    """
    return protons * 1.5
```

Type errors

- Using an unexpected type won't always stop your code from running
- But it can. E.g., trying to treat `None` like a number will cause an error

```
print(42)/(7) # Note wrong parentheses
```

- Let's explore the errors reported by the above code (it's a lot!)

Typing errors vs type errors

- Typing errors (or type-checking errors) are about the *documented types*. Code doesn't have to run to check, and errors don't prevent code from running.
- Type errors happen while code is running, and stop it from running further
- Both are about types, and the names are confusing, but they're distinct

Learning to read errors

- Making errors readable and understandable is an area of active research (no, really!)
- For typ*ing* errors:
  - Look for “expected” and “got”.
  - Think about which way data is moving (into parameters, out of returns)
- For *type* errors and other errors:
  - Start from the *end* and work your way backwards to understand where it’s happening

Testing

CS114 M1

Testing

- Bad development: keep poking at it until it looks right
- Good development: write examples of how it should behave, then write it until it does behave
- Running these examples is *testing*

Assertions

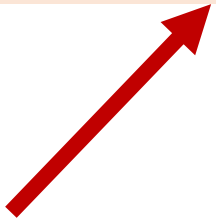
- Python has a built-in technique for testing: **`assert`**

  as·sert (asserted; asserting; asserts)
  transitive verb
  1a: to state or declare positively and often forcefully or aggressively
  (— Merriam Webster dictionary)

- You assert (declare) something to be true, then fix your code ’til it is ☺

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```



The test
(what you want to be true)

Name for the test
(documentation)

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

Since = is variable assignment, == is equality

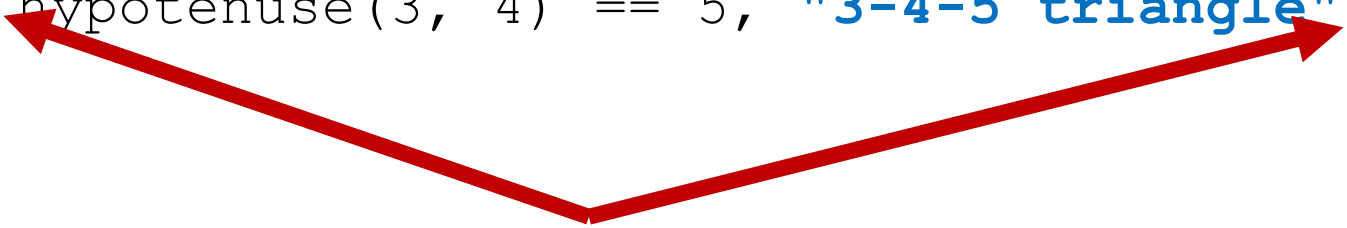
Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```

But *do not use* == *with floats!!!*
(Remember, they're rounded!)

Assertions

```
assert hypotenuse(3, 4) == 5, "3-4-5 triangle"
```



Note no parentheses. **`assert`** is a keyword, not a function! If you add parentheses with the test name, it won't do what you think!

Let's look at

```
assert(1 == 0, "Math is broken")
```

Testing with floats

- `floats` have limited precision (they're rounded)
  - Internally, they're scientific notation in base-2, so have limited base-2 significant figures, but don't try to intuit about base-2 scientific notation...
- The precision can go wrong in very surprising ways:

```
assert 0.3-0.2 == 0.1, "Math too simple to fail"
```

Imprecise numbers, imprecise tests

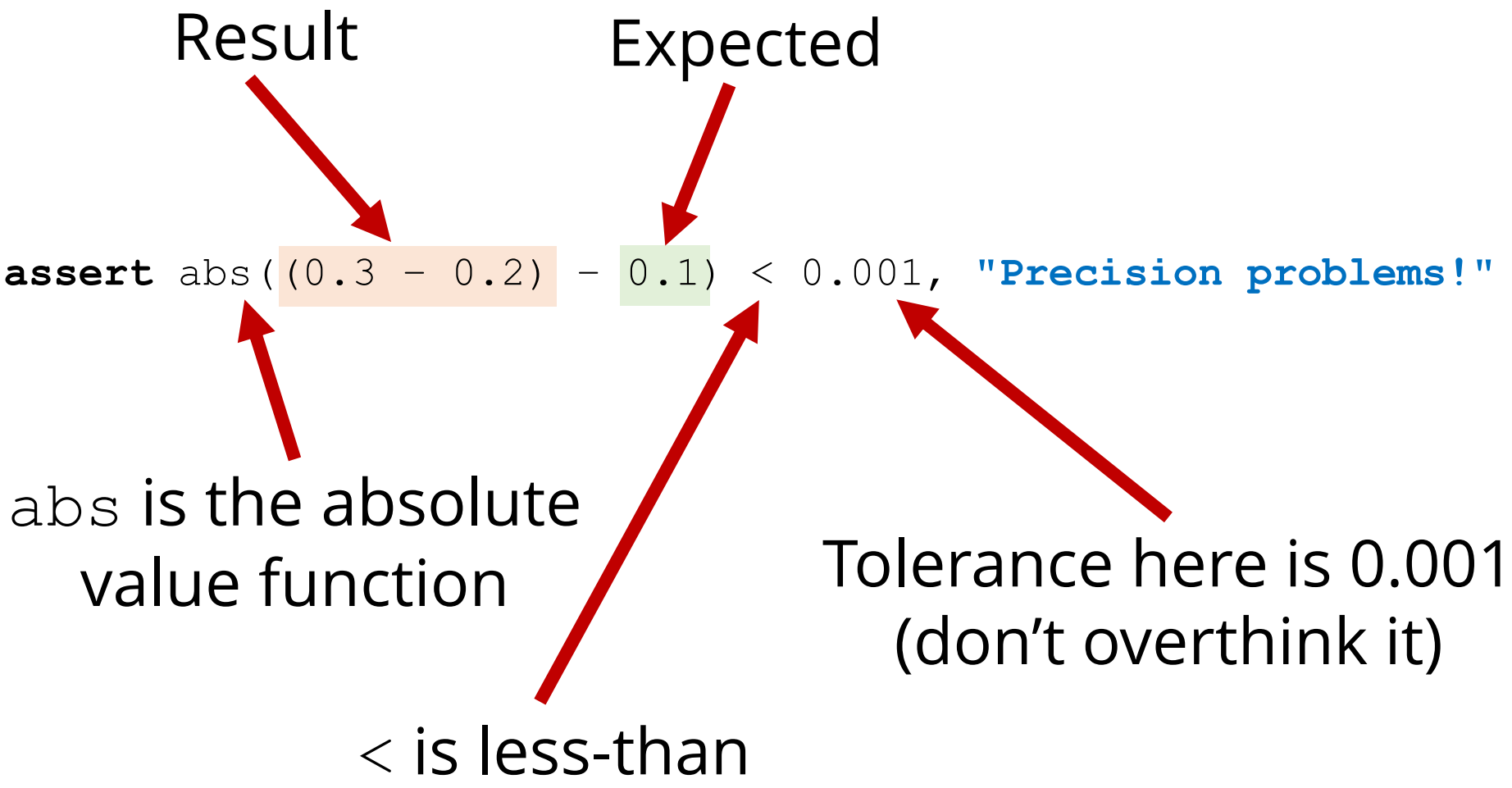
- Always test floating points with a range, called the *tolerance*, to avoid precision problems
- Simplest way to test with tolerance is

$$|result - expected| < tolerance$$

- In Python, that looks like this:

```
assert abs((0.3 - 0.2) - 0.1) < 0.001, "Precision problems!"
```

Imprecise numbers, imprecise tests



Imprecise numbers, imprecise tests

- Use similar code any time your numbers won’t be integers

```
assert abs(hypotenuse(4, 5) - 6.4031242) < 0.001, "…"
```

Code style

- In this course, every function must have
  - A docstring,
  - parameter and return types, and
  - at least two test assertions (other than the ones we provide you)

Code style

- We set two tests as a *minimum*
- For most functions, it won't be enough
- Think about corner cases
- … but don't overthink it. Just make up some tests.

Code style

- In practice, **`assert`** isn't always a practical way of testing a function
- When you're programming outside this course, you may find it more useful to just run your code in a few situations, rather than writing **`assert`** tests
- We just want to make sure that you've learned *to* test

Other assertions

- **`assert`** is often used for tests, but can also be used for other checks
- For instance, if we want to make sure our arguments are positive:

```
def hypotenuse(a: float, b: float) -> float:
    assert a > 0, "a must be positive"
    assert b > 0, "b must be positive"
    return sqrt(a**2 + b**2)
```

Module summary

CS114 M1

Module summary

- We'll be writing Python in Jupyter
- Imperative programming language: give your computer a sequence of commands
- Calculation in Python
- Computation =
  calculation + repetition + decision-making
- Functions box up behavior
- Programming is mostly fighting bugs