CSCI 1101: Computer Science I

Jean-Baptiste Tristan

Spring 2022

Welcome to CSCI 1101: Computer Science I. Here's some important information:

- The course webpage is:
  https://bostoncollege.instructure.com/courses/1627647
- Social office hours take place in CS Lab 122 on Thursdays from 5:30-9:30pm and Fridays from 3-6pm. Feel free to drop by and ask questions or simply work on the homework!
- Everyone's contact information is below. Please remember to contact **your own** discussion section leader for technical questions (who may escalate to Julian if needed) and to contact Professor Tristan for personal questions.

Jean-Baptiste Tristan	Instructor	tristanj@bc.edu
Julian Asilis	Head TA	asilisj@bc.edu
Ananya Barthakur	TA	barthaka@bc.edu
Joseph D'Alonzo	TA	dalonzoj@bc.edu
Jakob Weiss	TA	weissjy@bc.edu
Joanne (Jo) Lee	TA	leebpo@bc.edu
Thomas Flatley	TA	flatleyt@bc.edu
Chris Conyers	TA	conyerch@bc.edu
Brielle Donowho	TA	donowhob@bc.edu

- These notes were taken by Julian and have **not been carefully proofread** – they're sure to contain some typos and omissions, due to Julian.

Contents

§1 Wednesday, January 19

Welcome to the Introduction to Computer Science. The plan today is mostly to talk about the structure of the course – rather than diving headfirst into the course material – and to talk about the spirit of computer science at a high level. In particular, we'll try to convince you that it's useful to learn about computer science even if you don't intend to become a programmer.

§1.1 What is computer science?

First things first: what is the definition of **computer science**? Here's what the dictionary says:

> *The branch of knowledge concerned with the construction, programming, operation, and use of computers.*

Well, what's a **computer**? Let's use the dictionary again:

> *A device or machine for performing or facilitating calculation.*

It's important to note that there's no mention of electronics here! So even something like an abacus is a computer under this definition. There's another dictionary definition of a computer though:

> *An electronic device [...] capable of [...] processing [...] in accordance with variable procedural instructions.*

The latter end of that definition seems to be referring to an **algorithm**, perhaps the single most important object in computer science. Let's see a definition:

> *A procedure or set of rules used in calculation and problem-solving; (in later use spec.) a precisely defined set of mathematical or logical operations for the performance of a particular task.*

The important (and difficult!) part is that algorithms are a very precise set of instructions. Defining exactly what it means to be 'precise' or 'mathematical' is no small feat, though. Formalizing this entire setup (computer, algorithm, etc.) is actually one of the most important feats of the legendary Alan Turing.

§1.2 History of computer science

One of the earliest sets of instructions for performing a computation comes from the Babylonian Empire, circa 1600 B.C. The algorithm (in more modern language) served to calculate certain dimension of a cistern. It was really a flushed out example – rather than an abstract algorithm in the modern sense – but it's thought of as one of the earliest examples of computational thinking.

In ancient Greece (circa 300 BC), Euclid developed an algorithm to compute the greatest common divisor of two numbers. This was a fairly flushed out example, and an important point is that it didn't run in a fixed amount of steps. The number of steps required in the algorithm instead depended upon the size of the inputs fed to it. We'll touch on this idea later in the course.

In the 3rd century, Chinese mathematician Liu Hui developed what is currently known as Gaussian elimination (long before Gauss!). Furthermore, he even proved *correctness*

of the algorithm (i.e., that the algorithm's instructions conclude with the answer that you would like it to, when used on any input).

In the 9th century, Al-Khwarizmi was part of the Islamic Golden Age, which united ideas from Chinese and Indian number theory in order to develop the number system we currently use. Notably, the word algorithm comes from Al-Khwarizmi himself.

Remark 1.1. The way data is represented is *really* important when performing computation. For instance, you learned how to perform addition when you were 5 or 6 years old using the Arabic numerals, and it wasn't too hard. What if you'd had to learn addition using the Roman numeral system instead? What's MCCXXXIV + MMMMCCCXXI? In Arabic numerals, that's just 1234 + 4321 = 5555!

In the 19th century, Ada Lovelace produced perhaps the first program, for computing Bernoulli numbers. She is one of the great pioneers of computer science, and the programming language Ada bears her name.

In the 20th century, Alan Turing started contemporary computer science by formally defining computers and algorithms. He also led the team that decoded the Enigma code in order to locate Axis U-boats in the second world war. In 1946, the first programmable electronic computer was created. One last historical note: the first computer bug was a literal bug that got in the hardware of these early computers (hence the name).

What's really the point here?

- Computer science is about much more than programming electronic devices.
- It will improve your **problem-solving skills**.
  - Design, analysis, and implementation of algorithms to solve problems
- It will introduce you to **computational thinking**.
  - Decompose, generalize, abstract, organize
- It will make you a more rigorous and logical thinker.

One last example to really underscore that computers are not (just) electronic devices. One way to solve a famous problem knows as the *Traveling Salesman problem* is by using slime mold! You can literally place food in a petri dish and the slime mold will connect in the shortest path possible.

§1.3 Course information

Here are some of the things you'll learn in the course.

- Problem solving by designing and analyzing algorithms
- Representing and manipulating data
- Programming an electronic computer
  - Using the Python programming language
- Operating an electronic computer
  - Using a terminal on a Linux instance in the cloud

How will you learn all of this?

- Lectures
  - Usually, no slides, live programming and explanations

  - Not mandatory but highly recommended!
  - Lecture notes posted (and lecture hopefully recorded)
- Free textbook: details on Canvas
- Discussions
  - ~10% of final grade, for participation (both mandatory attendance and effort)
  - Make sure you know who your discussion leader is!
  - No swapping discussion sections, sorry
- 9 homework assignments
  - ~35% of final grade
  - Released on Fridays and due the following Friday at 7pm (via Canvas)
  - No homework on midterm weeks
  - -20% for late homework up to 24 hrs past the deadline
- 2 midterms
  - ~30% of final grade, requires a computer
  - Midterm 1: March 4, Midterm 2: April 20
- 1 final project
  - ~25% of final grade; structure subject to change
  - Programming assignment with a partner, project assigned to you
  - 1-2 weeks to complete

There are about 140 students taking this course, so we need to be careful about how you should get help and interact with course staff. Please follow the protocol below.

- **Step 1**: Social office hours:
  - Thursday from 5:30 pm to 9:30 pm, CS Lab 122
  - Friday from 3:00 pm to 6:00 pm, CS Lab 122
- **Step 2**: Email **your own** discussion leader. The email may be forwarded to the head TA if they can't help you.
- **Step 3**: Ask for one on one help with your discussion leader. Again, the email may be forwarded to the head TA.
- Personal matter? Email Professor Jean-Baptiste Tristan.

One last note: if you're going to miss lecture, no need to email anyone. We encourage you to come, but we certainly understand that issues may come up – if you can't come, no need to notify anyone.

Final thoughts: **please read the syllabus**. There's lots of important information, and we were able to cover most, but not all, of it today. Also, there are no discussion sections or office hours this week. Once again, welcome to the course and we'll see you on Friday!

§2 Friday, January 21

§2.1 JupyterHub and primitive types

The very first homework assignment that you'll be completing will be hosted on Jupyter notebooks, which is a flexible platform for writing code, running code, and writing text/math. In particular, a Jupyter notebook is built of different *cells*, that can contain Python code or markdown for writing text.

Within a notebook, you can create new cells, delete existing cells, run cells filled with code and see the output, and write text between cells of code to describe your thought process.

So, for instance, you can have a cell in a notebook that looks like this:

```
2 + 3
```

This is an example of an **expression** in Python, and if you run that cell, it'll output `5`. Awesome! You can also have a cell like this:

```
2 + 2 * 3
```

And this evaluates to 8, as you'd expect. But it's important to note that there was a choice made here – that expression could have instead been evaluated as $(2 + 2) \times 3 = 12$. The fact that it didn't comes down to **precedence** rules; Python has decided that multiplication should be evaluated before addition, which agrees with the way we usually evaluate expressions as humans.

An important fact to note is that every expression in Python has a **type**, which is roughly the 'species' of the expression, or the kind of thing that it is. Important types to start off with are the **primitive types**, which are some of the most basic, built-in types in Python that underlie more sophisticated ideas. For instance, `int` is the type of all the integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and `float` is the type that (roughly speaking) contains the continuous real numbers, like 0.26, 1/3, π, etc.[1]

Another import type is `str`, which contains the *strings* in pythons, i.e., collections of characters like `'Hello!'` or `'This is a string :)'`. There are some nice built-in operations on strings, like addition between strings or multiplication of a string by an integer. Let's see that in action:

```
'hello' + ' bob'
```

will evaluate to the string `'hello bob'`, and

```
'hello' * 3
```

will evaluate to the string `'hellohellohello'`. There are tons of built-in operations on these primitive types, and we're simply not going to be able to cover all of them. One important skill that we want to instill in you over the course of the class is the ability

[1] Strictly speaking, computers can't work with all the real numbers precisely, because computers are fundamentally discrete. So in practice it works with mere approximations of these numbers, which can sometimes produce strange behavior. The takeaway is just to be a bit careful anytime you're working with floats, keeping in mind that they're just approximations.

and eagerness to Google your programming questions. It's something that even the pros rely on, and it can be one of the fastest ways to answer your questions.

§2.2 Debugging

Let's talk a bit about **debugging**, which is one of the most important skills in programming. Let's work with an example.

```
2 +
```

If we run this in a cell, we'll get an error that reads `SyntaxError: invalid syntax`. Here's the first, golden rule of debugging: **read the error messages**. They're not *always* useful, but they'll often give you most (or all) of the information that you need to fix the problem.

In this case, we got a `SyntaxError`, which is roughly the programming analogue of making a grammatical mistake. The error will even point you to the line where the mistake was made, and we'd be able to see that we forgot to provide one of the arguments to `+`. Let's look at another example.

```
'hello' ** 5
```

In this case, we would get an error message of a `TypeError`, which tells us that one (or more) of our inputs in an expression has the wrong type. In this case, we would realize that we're not allowed to exponentiate a string and a number (what would it even mean to multiply `'hello'` by itself 5 times?). Let's keep going.

```
4 / 0
```

This gives us a `ZeroDivisionError`, which tells us exactly what we need to know: we tried to divide by 0, which isn't even legal mathematically (much less computationally).

§2.3 Variables

The **variable** is the bread and butter of programmers, and serves as shorthand for the expressions. Let's look at how you **bind** a variable, i.e., assign an expression to it.

```
x = 42
```

This is a line of code that assigns the value 42 to the variable `x`. Unlike in mathematics, it is not declaring that `x` equals 42. After all, `x` doesn't even exist before the line of code is run! But from here on out, we can write code with the variable `x` in place of `42`. To drive the point home, let's look at another (perhaps somewhat surprising) example.

```
x = x + 25
```

This code will run happily! `x` will have the value 67 after the line of code is run, and this underscores that `=` plays the role of an action in Python, not a passive test of equality.

```
b = a + 23
```

What if we run the code above? Well, we've never defined `a`, so Python will yell about a `NameError`, which lets us know that it doesn't know what `a` is. (Good thing we read the error message!)

What if we want Python to display information to us? This is achieved using `print` statements, equipped with an argument of what we want to print. For instance,

```
print(x)
```

will display the value `67` on our screen, as that's the value bound to `x`. Similarly, we can write

```
print('the value of x is:', x)
```

which displays `the value of x is: 67`. (Notice that it added a blank space between the colon and 67.) An important note here is that print statements can be extremely useful for debugging! Riddling your code with print statements lets you know exactly what all the variables are bound to when the code fails, at which step the code fails, etc.

Now here's a bit of a puzzle: what if we wanted our code to print `"hello"`, rather than just `hello`? Running `print("hello")` will achieve the latter, so it's not what we want. It turns out that this can be achieved by combining single quotes and double quotes in Python. If we run

```
print('"hello"')
```

then we'll indeed get `"hello"`, as `print` only strips the outer layer of single quotes.

That should be everything you need for the first homework assignment. In discussion section next week, your TAs will help you familiarize yourself with Jupyter notebooks and the process of transferring homework between Canvas and Jupyter Hub. Good luck on the homework!

§3 Monday, January 24

The goal for today is to learn how to use the terminal; it may not be the most exciting topic we'll cover in the course, but it's important for developing skill in using your own computer in advanced ways and for using computers remotely.

§3.1 Terminal basics

To ease the into idea of using a terminal, recall that last time we talked about using JupyterHub for the first homework. JupyterHub is actually just an interface for accessing a virtual machine in a far-off place, like a warehouse with lots of powerful computers.[2] Furthermore, JupyterHub has its own terminal, accessible from the home page. At a high

[2] Informally, a virtual machine is like a sliver of one of those powerful computers, that you share with many other users.

level, the **terminal** is just a powerful interface for interacting with a computer. The bread and butter of terminal usage lies in its basic commands, some of which we'll cover now (and which we'll expect you to know!).

```
$ echo 'hello'
```

This will have the effect of simply printing `hello` back to us. Nothing too fancy yet. How would we learn more about a terminal command (e.g., about its optional arguments)? Using `man`.

```
$ man echo
```

This will display the manual for echo (hence the name `man`), including lots of information about arguments for echo, etc. Now we have lots of junk on our screen, and we might want to clean things up using the `clear` command.

```
$ clear
```

Now our terminal is clean – nice. In order to get information about the machine that we're using, we can use the `uname` command.

```
$ uname
```

In this case, the terminal will tell us that our machine is using Linux, which can be useful to know.

Now let's talk about commands for organizing data stored in the computer. In order to know where the terminal is currently set up within the file system (it's always somewhere!), you can use the `pwd` command.

```
$ pwd
```

Short for 'print working directory', `pwd` will tell us the **directory** (or folder) where the terminal is currently working. In order to move the working directory, you can use the `cd` command, short for 'change directory.'

```
$ cd ~
```

This will take the terminal to your home directory, since you fed it the `~` argument. You could have written `cd /` to navigate to the root directory instead. (How can you learn more about `cd`? Using `man`!). In order to see the files contained in your current directory, use the `ls` command, i.e.,

```
$ ls
```

If you feed the optional argument `-l` (i.e., write `$ ls -l`), then you'll get even more information about the contents of your current working directory (cwd). You can even

make a new directory within your cwd using the `mkdir` command and a name argument, i.e.,

```
$ mkdir my_folder
```

will have the effect of creating a new folder (or directory) in your cwd named `my_folder`. To remove that directory, you would use `rmdir`.

```
$ rmdir my_folder
```

In order to create a file that doesn't exist, say a new text file, you would use `touch`. So

```
$ touch hello.txt
```

will create the file `hello.txt` within your cwd.[3] You can open a file using `open` along with the name of the file, and you can see just the first few or last few lines of the file using `head` or `tail`, respectively. Two last tips for efficiency on the terminal:

1. You can cycle through your previous commands on the terminal using the up arrow; this can save you lots of typing when used correctly!
2. The terminal will auto-complete file and directory names as much as it can when you press tab.

§3.2 Running Python from the terminal

Now let's move on to something a little bit fancier – let's say we've written a Python program in a file called `first.py`. Maybe it looks like this:

```
x = 42
x + 6
```

We can run this from the terminal using the command `$ python first.py`. But nothing happens – why? It's because Python did exactly what we asked; it completed its instructions silently! We can fix this, and ask Python to show us some output, by updating `first.py` as so:

```
x = 42
print(x + 6)
```

Now when we run `$ python first.py`, we indeed see the output of 48. That's an improvement, but it'd be nice to have something more dynamic, perhaps where we can feed the program a number of our choosing at runtime. So we'll update `first.py` again, using the `input` function to request arguments from the user.

[3] If `hello.txt` already exists, then it will just change the 'date last modified' of the file to the current time. That helps explain why it's called `touch` (in fact, using `touch` on a file that doesn't already exist is kind of a degenerate case, even though it may be the most common use).

```
x = input('Give me a number:')
print('Your new number is:', x + 1)
```

Now when we run this, Python actually asks us for input. We can feed it an integer (say 42 again) and look forward to seeing it be incremented by one. But this now gives us a `TypeError` message! Looking more closely, we can see that Python can't compute `x + 1` because `x` is a string.

When Python reads user input via `input()`, it automatically casts it as a `str` type; as humans, we know that we're going to feed the program in integer, but the program itself doesn't know that. So we need to turn `x` into an integer before incrementing it. This leaves us with

```
x = input('Give me a number:')
x = int(x)
print('Your new number is:', x + 1)
```

which will indeed work!

§4 Wednesday, January 26

§4.1 Functions

The goal today is to learn about functions in Python and about approaching algorithmic problems more generally. We'll be running with an example today and for the next couple lectures, concerning solutions of quadratic equations. In particular, recall that a **quadratic equation** is an equation of the form

$$ax^2 + bx + c = 0,$$

where a, b, c are some fixed numbers (with $a \neq 0$) and x is an unknown variable. So a particular quadratic equation might look something like $3x^2 + 2x + 1 = 0$. The name of the game is to find the value(s) of x that make this equation hold true, known as **roots**.

From a mathematical perspective, this problem has been resolved using the quadratic formula, which we'll discuss in more detail shortly. From a computer science perspective, however, there's a bit more going on. The precise problem setup is that there are 3 inputs – the numbers a, b, c (with $a \neq 0$) – and the desired output is a pair of numbers x_1, x_2 such that

$$ax_1^2 + bx_1 + c = 0,$$
$$ax_2^2 + bx_2 + c = 0.$$

Now the goal is to find a generic procedure that works to send *any* valid input values a, b, c to (correct) output values x_1, x_2. As we've discussed previously, this kind of generic procedure for computing output from input is known as an **algorithm**.

Now back to the quadratic formula; here's the precise mathematical statement.

Lemma 4.1 (Quadratic formula)
Let $f(x) = ax^2 + bx + c$ be a quadratic equation (i.e., $a \neq 0$). Then the (complex) roots of f are exactly

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}, \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

A crucial skill for any computer scientist, which is a bit hard to teach, is how to turn a mathematical result (like the quadratic formula) into computer code. So let's try to start by at least computing the roots of a particular quadratic equation, say $x^2 + 4x + 2$. We might start writing code as follows.

```
a = 1
b = 4
c = 2
```

We have an idea of how to perform the addition, multiplication, and division necessary to compute the result of the quadratic formula, but how are we going to calculate the square root? After all, there's no built-in square root character in Python like `+` for addition or `*` for multiplication.

This is where we'll turn to Python's **libraries**, which are collections of functions that other people have written for us. In this case, we'll use the `math` library, which has the function `math.sqrt()` that we're looking for.[4] Now we can write our code as follows.

```
x1 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
x2 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
```

Awesome! Now what if we want the roots of another quadratic equation? We'll need to update the values of `a`, `b`, and `c` and run all these lines of code again. That's not terribly inconvenient in this case, but it would be really inconvenient for larger suites of code, and we can do better. The technique for packaging many lines of code into a single name in Python is the **function**. Before trying to write a function that solves quadratic equations, let's get our feet wet with simple functions.

```
1 def my_func(x):
2     x = x + 1
3     print('Value of x:', x)
4
5 my func(5)
```

In the first three lines, we're defining our own function `my_func` with the single **parameter** `x` that increments `x` by 1 and prints it. In the fifth line, we're actually using `my_func` with the **argument** 5, which will result in Python printing `Value of x: 6`. Simple enough. Now here's a bit of a puzzle – what if we ran the following code?

[4] If you were to forget the name of a library or a function within a library (and everybody does), remember to use Google! Even the pros do it.

```
x = 42
def my_func(x):
    x = x + 1
    print('Value of x:', x)

my func(5)
```

We'll actually still get Python printing `Value of x: 6`. So the first line, in which we wrote `x = 42`, didn't affect the computation of `my_func(5)` at all! The fact that `my_func` doesn't care about the value of `x` outside of its own argument/definition has to do with the idea of **scope**. In particular, the value of `x` on line 3 only comes from the argument to `my_func`, not from anywhere else.

Now what if we were to run `my_func(4+5)`? This would be evaluate as `my_func(9)`, due to the **call by value** nature of Python. In particular, Python first evaluates `4+5` and then sends the result to `my_func`. That's how most programming languages do things, but not all of them![5]

Okay, back to the quadratic equation. Let's try to write a solver for it now that we're warmed up.

```
def solver(a, b, c):
    x1 = (-b + math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    x2 = (-b - math.sqrt(b**2 - 4 * a * c)) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

That works! If we run

```
solver(1, 4, 2)
```

then we'll get the roots of x^2+4x+2 that we saw earlier. If we want the roots of x^2+5x+2, then we only need to change a single character.

```
solver(1, 5, 2)
```

We've made a great leap forward from just writing code in cells (i.e., without functions), but there are still improvements to be made. The code is currently a little bit hard to read and has redundant computation (e.g., we're computing the square root of `b**2 - 4*a*c` two times). Let's try to clean this up a bit, with the goal of making it faster, easier to read, and less prone to errors.

```
def solver(a, b, c):
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
```

[5] If this is confusing, don't worry. The idea is really just that Python does what you'd expect it to do in this situation, and some other languages do stranger, fancier things.

```
5     print('First solution:', x1)
6     print('Second solution:', x2)
```

This code still isn't perfect (we'll see why over the next several lectures), but it's faster and cleaner than the previous version of `solver()`. Now let's conclude with a bit of debugging. What if we were to run:

```
solver(1, 2)
```

Then Python will give us an error message:

```
TypeError: solver() missing 1 required position argument: 'c'.
```

That's pretty descriptive; we now know that we need to feed `solver` its third argument. What if we were instead to go overboard with arguments?

```
solver(1, 2, 3, 4)
```

We get another error message:

```
TypeError: solver() takes 3 positional arguments but 4 were given.
```

Again, that's pretty informative and helps us figure out what went wrong. Now let's stop messing around and really use `solver()` as intended.

```
solver(1, 4, 2)
```

This time we've made sure to give `solver()` the correct number of arguments of the correct type. But we still get an error! Python will complain of:

```
ValueError: math domain error.
```

In this case, what's happened is that `math.sqrt()` is trying to take the square root of a negative number! We should have been more clever in writing `solver()` and made sure that it didn't accept arguments causing this kind of problem. We'll see how to approach these kinds of issues in the coming lectures.

§5 Friday, January 28

§5.1 Announcements

We'll keep talking about functions today, and you should be ready to start tackling Homework 2 by the end of the lecture. Speaking of the homework, we gave a quick look at some of the Homework 1 submissions yesterday, and it looks like the following thing is happening: many of you have already done some programming before, yet you answer a homework problem incorrectly and (in addition) use unnecessarily advanced techniques.

So, please, read the questions carefully before writing your solutions and be cautious about using sophisticated techniques when you really don't need to.

One more thing to mention: there's a collegiate programming contest called the ICPC, which BC used to perform very well in but which we don't seem to participate in anymore.

Professor Tristan is going to try to restart BC's participation in this competition, so please let him know if you're interested. To sweeten the deal, keep in mind that the kind of algorithmic thinking needed for these competitions is great preparation for interviews at places like Google, quantitative finance companies, etc. (As we mentioned earlier, algorithmic thinking is a very difficult and valuable skill to develop!)

§5.2 Problem: sum of roots

Last time we wrote functions for finding roots of quadratic equations. Today, we'll change things up a bit and think about writing a function that returns the sum of a quadratic equation's roots. That is,

$$a, b, c \longmapsto x \text{ such that } \begin{cases} x = x_1 + x_2; \\ ax_1^2 + bx_1 + c = 0; \\ ax_2^2 + bx_2 + c = 0. \end{cases}$$

Now, there are a couple of ways to approach this problem. Perhaps the most obvious is to use our function `solver()` from last time in order to compute x_1 and x_2, and then return their sum. Breaking this down further, there are two ways to implement this idea:

1. Copy and paste the body of our `solver()` function into a new function that prints $x_1 + x_2$ instead of x_1 and x_2 separately.
2. Write a function that uses the output of `solver()` in order to compute $x = x_1 + x_2$.

Path (2) is far, far better than path (1). **If there is a golden rule of programming, it is to not copy and paste code**. When you copy and paste code 5 times, then any bug you find needs to be fixed 5 times, any style change/clean-up you make needs to be implemented 5 times, any efficiency speed-up needs to be implemented 5 times, etc. Also, copy-pasted code is much harder to read than modular code.

Now, in order to implement idea (2), we need to change `solver()` so that it actually returns its output, rather than just printing it. This is achieved using the `return` keyword. Our new `solver()` will be as follows.

```
1 def solver(a, b, c):
2     sr = math.sqrt(b**2 - 4 * a * c)
3     x1 = (-b + sr) / (2 * a)
4     x2 = (-b - sr) / (2 * a)
5     return x1, x2
```

§5.2.1 Tuples

Something a bit subtle is happening in line 5; `solver()` is returning two values at once by using the `tuple` type. In contrast to the **primitive types** that we have seen previously, the `tuple` is a **composite type**, meaning it's a bit more complicated and built from the simpler primitive types. At a high level, the `tuple` is simply a type for placing several items *in order*. For instance,

```
1 x = (2, 3)
```

binds `x` to the tuple with the `int` 2 in the first position and 3 in the second position. If you've seen vectors before, you can think of tuples like that.

In fact, tuples are one kind of a **data structure**, which is a format for storing and manipulating data. If this is a bit confusing, don't worry – we'll encounter several more data structures over the course of the class. In fact, data structures are probably the second most important idea in computer science, after algorithms!

Now we need to familiarize ourselves with tuples a bit. Let's say we have `x = (2, 3)` as before – how can we access the entries 2 and 3 from the variable x? This is achieved by **indexing** into x, via the following syntax.

```
two = x[0]
three = x[1]
```

By running this code, the variable `two` will indeed take the value 2 (corresponding the leftmost entry of `x`), while `three` will take the value 3 (corresponding to the rightmost entry of `x`). An important observation here is that **0-indexing** is used when accessing the entries of `x`. That is, the leftmost entry of `x` has the index 0, while the next one has the index 1, and so on. Simply put, we start off counting from 0 rather than 1.

This is just a convention in computer science. Python could have chosen to start indexing tuples by starting with the number 43, and that would be perfectly legal (though very confusing for humans). For whatever reason, computer scientists often like to start counting at 0. (Kind of like how in Europe, the ground floor of a building is the 0th floor, rather than the 1st.)

Another way to grab the entries of a tuple is via **pattern matching**, as follows.

```
two, three = x
```

This will again have the effect of binding `two` to 2 and `three` to 3. So now that we've learned about tuples, we can write the outer function that uses `solver()` and adds its output (in order to solve our original problem about sums of roots). We can write:

```
1 def solver_sum(a, b, c):
2     x1, x2 = solver(a, b, c)
3     print(x1 + x2)
```

Isn't that nice? We solved our new problem with two lines of code! This is just an example of the power of writing modular code, i.e., code that is split up into several functions that can be reused, rather than huge blocks of copy-pasted code.

§5.3 More operations

Now we're going to learn about more of Python's powers. We just learned about indexing into tuples, which are entries of data in order. That doesn't sound so different from strings (which are just characters in order), so maybe we can index into them as well. Let's try.

```
s = 'Hello, I am Sam!'
s[4]
```

This indeed returns `'o'` – nice! What if we want all the values between two indices of the string? We can do so via **slicing**, i.e.,

```
s[3:12]
```

This returns `'o, I am '`, the values between the 3rd and 12th indices. To get the string's entries from the 3rd index onward, you can write `s[3:]`, and to get the string's entries up until the 7th entry, you can write `s[:7]`.

Now let's move from strings to floats; what if we want to round a `float` to an `int`? We can use the built-in `round()` function, that rounds a `float` to the *nearest* `int`. For instance,

```
round(4.8)
```

returns 5, while `round(4.2)` returns 4. What if we want to round to the next-lowest integer or next-highest integer, rather than the closest? Then we'll need to use the `math` package, with the functions `math.floor()` or `math.ceil()` respectively. For instance,

```
import math
math.ceil(4.1)
```

comes out to 5.

§6 Monday, January 31

§6.1 HW1 postmortem

Quick comment on the homework due last Friday: the goal was to compute the following value, as a function of T and v:

$$T_{wc} = 35.74 + 0.625 \cdot T - 35.75 \cdot v^{0.16} + 0.4275 \cdot T \cdot v^{0.16}.$$

Most of you wrote something like this:

```
1 T = 20
2 v = 15
3 35.74 + 0.625 * T - 35.75 * v ** 0.16 + 0.4275 * T * v ** 0.16
```

That will produce the correct value, but it actually has an imperfection. Namely, the value `v**0.16` will be computed *twice* by Python in the third line. Rather than having Python repeat identical computation, and waste time & energy, we can do better by introducing a variable that stores the value of `v**0.16`.

The intended solution was as follows.

```
1 T = 20
2 v = 15
3 tmp = v**0.16
4 35.74 + 0.625 * T - 35.75 * tmp + 0.4275 * T * tmp
```

In particular, the variable `tmp` (for temporary) introduced in the third line saves Python from repeating computation in line 4.

One more note: Python evaluates code in a line-by-line manner, and it doesn't evaluate the body of a function until the function is actually called with arguments. For instance, consider the following code.

```
1 x = 10
2 y = x + 42
3
4 def test(x):
5     print('hello')
6     return x + 1
7
8 test(y)
```

What will this output? Well, lines 1 and 2 are run in that order, so `x = 10` and `y = 52`. Then `test(x)` is defined but its body is not run (since we haven't called it with any input yet!). In line 8, finally, we call `test(y)`. Since `y` is bound to 52, that comes out to `test(52)`.

So this code will print `'hello'` and line 8 will return the value 53. In particular, that's exactly the same as if we had replaced lines 1 and 2 with the single line `y = 52`. The body of `test()` only cares about the `x` that it is given as an argument.

§6.2 Booleans

The primary goal for today is to introduce a new type along with its primary functionalities. In particular, we will introduce the type **bool** of Boolean values (named after mathematician George Boole). The type `bool` has only 2 values: `True` and `False`. They're really meant to express the usual notions of truthhood and falsehood within Python, and to allow us to branch our computation based on whether something is true (i.e., compute `f(x)` if `P(x)` is true and `g(x)` otherwise).

Let's jump into some examples, demonstrating how we can get `bool`s from types we already know. For instance,

```
2 < 3
```

will evaluate to `True`, just as

```
3 <= 3
```

will evaluate to `True`. On the other hand,

```
2 > 3
```

and

```
3 <= 4
```

will both evaluate to `False`. So `<` and `<=` correspond to testing strict and weak inequalities of numbers.

To test for equality, among numbers and various other types, you can use `==`. (Recall that `=` is already taken for variable assignment, rather than testing equality!). So,

```
3 == 4
```

and

```
'hello' == 'goodbye'
```

will evaluate to `False`, whereas

```
5 == 5
```

evaluates to `True`.

So that's how we can get `bool`s from familiar types. But we can also manipulate `bool`s themselves to get other `bool`s. For instance, `not` applied to a single Boolean `x` will evaluate to `False` if `x` is `True` and to `True` if `x` is `False`.[6]

So, combining what we've learned,

```
not True == False
```

will itself evaluate to `True`! Two keywords for combining two Boolean expressions (rather than a single one) are `and` and `or`. Once again, they're built so as to agree with their plain English names. So,

```
True and True
```

comes out to `True`, while

```
True and False
```

comes out to `False`. On the other hand, `or` of several expressions will evaluate to `True` as long as even a single one of the argument expressions is `True`. For instance,

```
True or False
```

is `True`, and

```
True or True
```

is `True` as well.

[6] Nothing too fancy here; these Python keywords are designed so as to agree with plain English.

§6.3 Conditional statements

Now let's get to the most important use of `bool`s: **branching** computation. In particular, you often want your program to compute `A(x)` if `P(X)` is true and `B(x)` if `P(x)` is false.

The syntax for this lies in the keywords `if` and `else`. Let's learn through example.

```
1 x = 3
2
3 if x == 3:
4     print('x is 3 and I executed the top branch!')
5 else:
6     print('x is not 3 and I executed the bottom branch!')
```

The rule here is the following: if `x == 3` evaluates to `True`, then running this code will run the indented code immediately after `if` and skip the code after `else`. If `x == 3` evaluates to `False`, then running the code will skip the indented code after `if` and run the code after `else`. So, in this case, we'll see this message printed: `'x is 3 and I executed the top branch!'`. If we had set `x = 4` on line 1, then running this code would result in (only) the other message being printed.

You can also branch on more than one condition by using the `elif` keyword, which is short for `else if`. So,

```
1 if x > 1:
2     print('Took first branch')
3 elif x > 2:
4     print('Took second branch')
5 else:
6     print('Took third branch')
```

will send 1.5 to the first branch (and nowhere else), 2.5 also to the first branch (and nowhere else), and 0.5 to the third branch (and nowhere else). In fact, no numerical value of `x` will reach the second branch – can you see why?

Now let's rewind to when we were writing our `solver()` function for finding quadratic roots. Recall that it looked like this.

```
1 def solver(a, b, c):
2     sr = math.sqrt(b**2 - 4 * a * c)
3     x1 = (-b + sr) / (2 * a)
4     x2 = (-b - sr) / (2 * a)
5     print('First solution:', x1)
6     print('Second solution:', x2)
```

One problem we ran into earlier is that `math.sqrt()` doesn't accept negative input, which can sometimes happen in line 2 if we allow for any inputs `a, b, c`. We also want to make sure, as a basic first step, that $a \neq 0$ (otherwise, lines 3 and 4 really don't make sense ...). We can improve this function a bit using our new knowledge of conditional statements.

```
def solver(a, b, c):
    if a == 0:
        print('a should be 0!')
        return
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        print('No real solutions')
        return
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

Now `solver` will return nothing and print a disclaimer when it gets problematic input. Nice. Another problem we had was that `solver()` accepts input from types other than `float` and `int`.

We can get the type an expression using `type()` and test that it equals `int` or `float` using `==`. For instance,

```
type(3) == int
```

comes out to `True`, while

```
type('hello') == float
```

comes out to `False`. Next time, we'll see how to use this kind of technique to improve `solver()` even further by having it only accept numbers.

§7 Wednesday, February 2

A quick errata from last time: we were talking about the implementation of ordering on strings, e.g., how

```
'be' < 'be curious'
```

evaluates to `True`. We thought that this tested whether the left hand side is a substring of the right hand side, but that's actually not true. `<` instead compares strings under the dictionary ordering (i.e., the ordering used to list words in the dictionary, or to list your last names on Canvas).

§7.1 Type checking

Last time we started talking about the need to check the types of arguments given to the functions that we write. For instance, to make sure that `solver()` only takes `int`s and `float`s as input, we can write a helper function `ct()` (for check type).

```
def ct(x):
    return type(x) == int or type(x) == float
```

So `ct()` returns `True` if `x` has the right type for `solver()` and `False` otherwise. Let's remind ourselves of the version of `solver()` that we were looking at last time, and think about how to use our new helper function `ct()`.

```
1  def solver(a, b, c):
2      if a == 0:
3          print('a should be 0!')
4          return
5      tmp = b**2 - 4 * a * c
6      if tmp < 0:
7          print('No real solutions')
8          return
9      sr = math.sqrt(b**2 - 4 * a * c)
10     x1 = (-b + sr) / (2 * a)
11     x2 = (-b - sr) / (2 * a)
12     print('First solution:', x1)
13     print('Second solution:', x2)
```

We're currently checking to make sure `a` does not equal zero, and that the polynomial indeed has real roots (rather than complex roots), which is great. It would be even better to kick things off by checking the types of `a`, `b`, and `c`. Making use of `ct()`, our new `solver()` will look as so:

```
1  def solver(a, b, c):
2      if not (ct(a) and ct(b) and ct(c)):
3          print('One argument has the wrong type!')
4          return
5      if a == 0:
6          print('a should be 0!')
7          return
8      tmp = b**2 - 4 * a * c
9      if tmp < 0:
10         print('No real solutions')
11         return
12     sr = math.sqrt(b**2 - 4 * a * c)
13     x1 = (-b + sr) / (2 * a)
14     x2 = (-b - sr) / (2 * a)
15     print('First solution:', x1)
16     print('Second solution:', x2)
```

And indeed we can check that `solver(1, 2, 'hello')` returns nothing and prints our new error message.

Remark 7.1. We're not doing anything terribly fancy, but note that this version of `solver()` uses all the techniques we've learned about: functions, conditional statements, and type checking!

Now we've made sure that `solver()` doesn't return anything for illegal inputs, but it still runs perfectly well, which isn't ideal. If we were in a larger project with thousands of lines of code, and `solver()` were being used somewhere deep in a complicated process, then our current setup would have some serious drawbacks:

- `solver()` can feed incorrect output (i.e., nothing) to later functions that happily make use of it.
- We won't know the line number or even the function where we used arguments of the wrong type.
- Our entire program (in which `solver()` is just one tiny piece) will still run, even when *we know* there's a mistake.

The way we can fix all these issues is by **raising an error**, which is Python's way of halting a program, spitting out an error message, and pointing the user to the function and line number where the error occurred. (All of these functionalities are extremely useful!). We can rewrite `solver()` to raise errors like so.

```
def solver(a, b, c):
    if not (ct(a) and ct(b) and ct(c)):
        raise ValueError('One argument has the wrong type!')
    if a == 0:
        raise ValueError('a should be 0!')
    tmp = b**2 - 4 * a * c
    if tmp < 0:
        raise ValueError('No real solutions')
    sr = math.sqrt(b**2 - 4 * a * c)
    x1 = (-b + sr) / (2 * a)
    x2 = (-b - sr) / (2 * a)
    print('First solution:', x1)
    print('Second solution:', x2)
```

§7.2 JupyterHub

This Friday (next class!) we'll have a brief programming quiz to help prepare you for the structure of the midterm. The quiz this Friday won't be graded, but it's still a good exercise to make sure you're on track, and you should make sure that you're comfortable working with JupyterHub beforehand (e.g., creating Python files, writing basic Python files, running Python files from the terminal).

So, let's say we're at your terminal in JupyterHub. If our goal is to run a script that prints `'Hello'`, then we would proceed as so.

```
$ touch hello.py
```

Now we've created the new Python file `hello.py` (note that its name needed to end in `.py`!). Then, in `hello.py`, we can write:

```
print('Hello')
print('Goodbye')
```

Now, *after saving the code we wrote in* `hello.py`, we can run the program in the terminal.

```
$ python hello.py
```

And this will indeed print `'Hello'` and `'Goodbye'`. Let's update our code to take in the user's name.

```
print('Hello, my name is HAL')
s = input("What's your name?")
print('Nice to meet you', s)
```

Upon saving `hello.py` and running it from the terminal, we get the desired behavior.

```
$ python hello.py
Hello, my name is HAL
What's your name? John
Nice to meet you, John
```

Cool. It's a little bit annoying to have Python ask the user for values one-by-one, so let's see how can do things a bit more efficiently.

Let's first make a new file.

```
$ touch test.txt
```

And fill it like so.

```
John
```

Now we can use the lines of `test.txt` as input to `hello.py`! The syntax is as so:

```
$ python hello.py < test.txt
Hello, my name is HAL
What's your name?
Nice to meet you, John
```

With this terminal command, `hello.py` will be run, and it will take successive lines of `text.txt` as input any time it requests input. So the second line of `hello.py`, that requests the user's name, went ahead and grabbed the first line from `text.txt`.Make sure you understand this example, because something similar will show up on Friday's quiz!

§7.3 Problem: maximum of 3 integers

Now we're going to discuss a new problem: given three integers a, b, and c (appearing on their own lines in a .txt file), write a program that prints the largest of the 3 integers. Here's a high-level tip on how to approach these kinds of problems: **Start by thinking about the problem with pen and paper, and try to sketch a solution. *Then* try to code up the idea you developed on paper.**

So let's start by thinking about our problem in English (and *not* jump into writing code!). So, one way to cast this problem mathematically looks like this:

$$(a,b,c) \mapsto \begin{cases} a & \text{if } a > b \text{ and } a > c, \\ b & \text{if } b > a \text{ and } b > c, \\ c & \text{if } c > a \text{ and } c > b. \end{cases}$$

We can code that up as follows, in `solution.py`.

```
1  a = input()
2  b = input()
3  c = input()
4
5  if a > b and a > c:
6      print(a)
7  elif b > a and b > c:
8      print(a)
9  elif c > a and c > b:
10     print(c)
```

Now if we run this on inputs `23, 265, 45`, we get the output 45. What? What went wrong here? In lines 1, 2, and 3, Python takes in its input as `str` types, not `int`s. So it is comparing `a`, `b`, and `c` as strings under the dictionary ordering we mentioned earlier! Let's modify `solution.py` and fix this.

```
1  a = int(input())
2  b = int(input())
3  c = int(input())
4
5  if a > b and a > c:
6      print(a)
7  elif b > a and b > c:
8      print(a)
9  elif c > a and c > b:
10     print(c)
```

One important skill for you to learn is to be adversarial when thinking about the code you write. Imagine that someone were to pay you 100$ if you could break your code – what would you do? Always try to think about edge cases that can break your code, for instance. In this case, what if we were to run `solution.py` with inputs `100`, `100`, and `100`?

Uh oh, running this script with `100`, `100`, `100` gives us no output at all. What went wrong? In this case, it turns out our math was wrong! The work we did with 'pen and paper' (or on the blackboard, in this case) was totally bogus. The mathematical formulation we wrote above is incorrect, and fails when there is a tie for the largest number. So the code we wrote based on our math was busted as well.

We can fix `solution.py` by using weak inequalities, rather than strict ones.

```
a = int(input())
b = int(input())
c = int(input())

if a >= b and a >= c:
    print(a)
elif b >= a and b >= c:
    print(a)
elif c >= a and c >= b:
    print(c)
```

This code is now correct, but it's also fairly inefficient. It makes 6 comparisons in order to find the maximum of 3 numbers, which is quite high. (Finding such a maximum can be done with only 2 comparisons!). Next time we'll see how to improve upon this, and we'll be talking much more about the efficiency of our programs later on in the course.

Index