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!

Index