# Why are you here?

Top Hat: "why are you here"

## *Why study programming?*

**Career, entrepreneurship, changing the world...**

**Responsibility**

**Important way of thinking about problem-solving**

- algorithmic / computational thinking comes up in **every** Engineering discipline!
- interesting problems to solve... can be *fun!*

Even if you don't write much software in your career, we live in a ______________________ ______________________________________________________. Engineers use computers in lots of ways, whether we put computers into the systems we just use computer software to help us design things. Engineers are ___________________________________: you can't blame your tools for faulty design! "My software gave me the wrong answer" is no better an excuse than "my calculator is broken", so we need to _____________________________ ____________________________.

Computer programming gives us ________________________________, not to mention __________________________________________________. Every engineer should be able to express solutions to certain classes of problems using *algorithms* (structured, step-by-step solutions to problems). Many first-year Engineering students think this is the last they'll hear of programming, but Statistics begs to differ, as does Fluid Dynamics, as does seismic analysis... even effective use of ________________ is greatly helped by some fundamental understanding of programming. And, of course, good software designers are ________________________ _________

Finally, whether or not you currently believe it, I hope to show you this term that __________________________

## Solving problems with software

- **specify** engineering processes using *propositional logic*
- **recognize** the pervasiveness of computing in engineered systems and **explain** its *systemic impacts* [...]
- **explain** how computers store values, *compute* on them
- **choose** *data types* to solve engineering problems
- **analyze** software; explain how it works (or why it doesn't)
- **construct** solutions using *imperative programming*
- **design** *algorithms* to solve engineering problems though *top-down procedural decomposition*

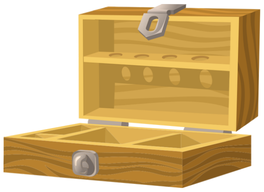
Q: What do you notice about these learning outcomes for ENGI 1020?

A: None of them say anything about ______________!

This course is about how to ____________________________, not how to _____________ __________. The lessons you take away from this course should be portable to all kinds of programming languages, and even to non-programming situations where structured, analytical thinking is called for — which ought to be _________________________!

We use software to address incredibly complex problems, from building self-driving cars (or self-driving submarines, as some folks do here at Memorial) to training ship's crews to securing networks against hackers (as in my own research). In fact, computer programs are among the most complex artifacts ever built.

## Programming toolbox



- logic
- computation
- data types
- imperative programming…
- … in Python

That said, we do need to put this theory into practice, and that requires a programming language. Like a ________________, theory alone is insufficient: in order to *really* get it, you need to ________________________________________ and work on it until it moves from your ____________ to your ___________. Just like you can apply musical theory to a piano or a guitar or any other instrument, so we can apply our programming tookbox to any programming language, but in our exercises, assignments and labs, we will use the Python programming language.

Python is a very useful language because it supports many different ways of thinking about programming, it includes lots of useful software tools for interacting with users and other computers, and it provides a good foundation from which you can learn other programming languages. However, please remember throughout the course that our goal is to learn about _____________________________, not just a particular programming language.

## How to succeed

### Active engagement:

- Try things out
- Practice, practice, practice!
- Ask questions
- Work together **and** independently

A good way to underperform in this course is to come to lectures, passively consume them, only do the things you are explicitly told to do and let you lab partner do things without your understanding. The value you get out of this course will be proportional to the effort you put in to ________________ and ____________________ the material. Specifically, it is helpful to:

When I show you some example code and explain how it works, don't take my word for it: try it yourself! Try running the same code with different inputs to see if you can predict how it will react. Try finding different solutions to the same problem or to slightly different problems to gauge your own understanding of the material.

Software design and development, like other kinds of Engineering design, blends both ______________________. I will teach you things about designing software-based solutions to problems, but you can't really say you've learned design unless you practice designing. This is much like learning to play a musical instrument, you need to learn some theory, but _________________ ______________. The Tools page provides details on how to get set up with software tools for writing, compiling and testing Python code. Please take a look at this page and make sure that you are comfortable with ___________________ of these tools during the ________________ ___________.

In class or labs, in Office Hour or the Success Centre, with friends or with me, via Top Hat or in Brightspace, if you don't understand something, ____________.

Most students will find that, to help them truly succeed, they need to _____________ work with other students on some things ____________ work alone on others. In this course you will encounter lots of exercises and examples that you should work through, and you may find it helpful to do so with your peers so that you can help each other out as your grow in your understanding of the material. You will also have (at least) one partner with whom you will work on prelab questions, experiments and writing. So, there will be plenty of opportunity to work in groups and in pairs to help you form your ideas about programming and how it's done.

It's also a good idea to complete at least some of the exercises ___________________ to check that you're able to think algorithmically yourself, not just when you're working with others. After you've formed your understanding together, checking your understanding individually can help give you confidence that *you* are building mastery of this material.

## Resources

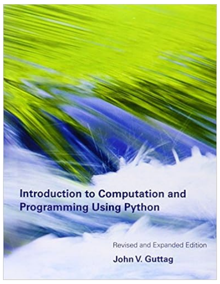
**Course website: https://engi1020.ca**

**Textbook useful but *optional***

**Lab kit *mandatory***

**Web resources: Gradescope, Top Hat…**

https://www.gradescope.ca/courses/14930
https://app-ca.tophat.com/e/610620 (also linked from Brightspace)


Introduction to Computation and Programming Using Python
Revised and Expanded Edition
John V. Guttag

## Human resources

- office hours: in the course outline
- online: email me (address in the course outline)

**Lab instructor: Alice Faisal (CSF-4111)**

**Engineering One Success Center:**

Tutors: consult “Engineering One” Brightspace shell
Supplemental Instruction (SI) leader: Riyana Afroze

## *Actions for today*

**Ensure you can access the following resources:**

**Brightspace** (grades)
https://online.mun.ca/d2l/home/567329

**Gradescope** (assignments, labs)
https://www.gradescope.ca/courses/14930

**Top Hat** (lectures, interactive response)
https://app-ca.tophat.com/e/610620

**If not, please email me *today***

## What computers are

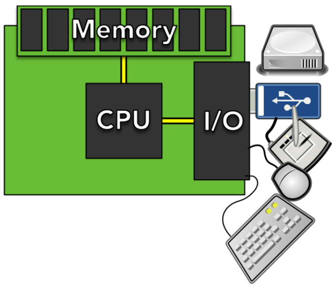
### Memory

- a.k.a., *random access memory*
- stores information in *bits*

### Central processing unit

- performs *computation*

### Input/output devices



These three functional components allow the computer to ________________, to ________________ and to ____________________. There are lots of other things inside of a modern computer, but they all exist to serve and facilitate these main tasks.

This overall structure is the same for the computer you're using to read these lecture notes, the Arduino computers that you'll use in the lab, the computer(s) inside your watch and even the computers that run cars, lighting fixtures, solar panel trackers, etc.! All of these computers have the ____________________. What's different about them is the power of their CPUs, the quantity of their memories and the kinds of I/O devices they have.

## What computers do

### Remember things (memory)

- numbers
- *strings* of characters (words, sentences, etc.)
- program instructions

### Follow instructions (CPU)

## CPUs follow instructions

### One step at at time

- Computers are dumb
- Computers are fast

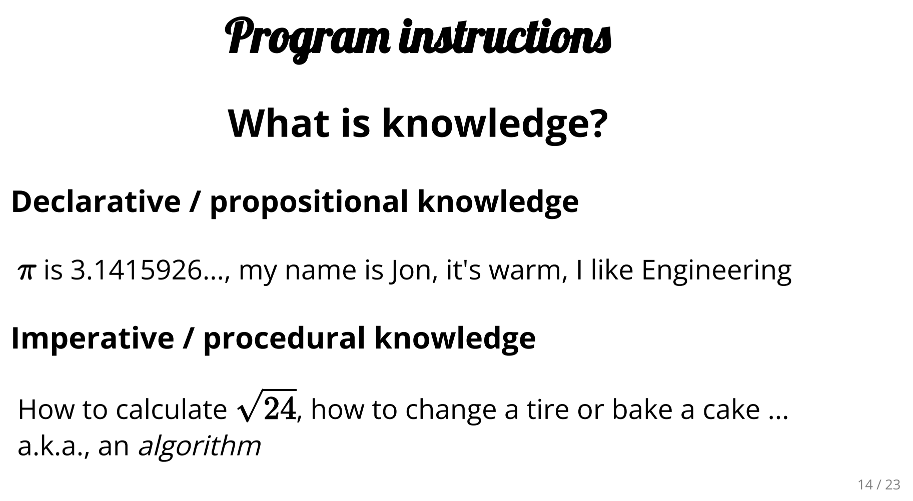
### Where do the instructions come from?

The CPU is the brain of the computer, and we will spend most of our time thinking about how it does its work (later courses in programming, microprocessors and data structures spend more time on these).

### "How do you eat an elephant" joke

The key thing to keep in mind is that a CPU ______________________________, and it does this ____________________________. The CPU follows a ___________________________________.

Computers aren't capable of thinking, but they *can* execute simple instructions very quickly!



## Program instructions

### What is knowledge?

#### Declarative / propositional knowledge

$\pi$ is 3.1415926..., my name is Jon, it's warm, I like Engineering

#### Imperative / procedural knowledge

How to calculate $\sqrt{24}$, how to change a tire or bake a cake ...
a.k.a., an *algorithm*

Logical ________________ are statements that can be either true or false. The propositions on this slide are all true — at least today! We leave as an exercise for the reader determining which of these propositions could be false, or alternatively, determining just how bad a day it would be if each of them *were* false!

________________ or ________________ knowledge concerns things that you ________ ________ to do. The procedural examples on this slide include activities for which you could write instructions (using the ________________ verb tense); someone else could follow these instructions in order to do the task. The instructions might be encoded in a math textbook, a shop manual or a recipe book, but the fundamental construction of a procedure is the same.

Such a set of step-by-step instructions for accomplishing a task is called an ________________.

## Algorithm

*A step-by-step procedure with decisions*

**Example algorithm:**

1. Let $y = \frac{x}{2}$.
2. If $2y = x$, $x$ is even.

**"bake until golden brown", "for each tire"...**

### Why the Great British Bake-off?

An algorithm is like a recipe

Note that this is more than just a mathematical formula or equation: it involves a step-by-step approach that may (for many algorithms) be difficult to represent as an equation.

## Another algorithm

### Sum of integers

e.g., find the sum of the set $S = 4, 7, 2, 11, 5, 8, 1$

**Mathematically:**

$$\sum_{i=1}^{n} S_{1..n}$$

**Algorithmically? (do this as an exercise)**

**Suggested methodology:**

- work out an example or two
- break it into steps
- explain the steps to someone else (pretend they're a computer)

## Software

*Description of instructions for a computer*

**You express meaning using a *programming language***

**Python:**

```python
y = x / 2
if (2 * y == x):
    print(x, 'is even')
```

**C++:**

```cpp
int y = x / 2;
if (2 * y == x)
    cout << x << " is even\n";
```

**Also:**

Assembly, C, C#, Go, Java, Matlab, Perl, R, Rust, Scala, SPIN...

**Your code is translated into *machine instructions***

What is software?

### Softer than hardware?

### There's also "firmware", but...

Software, whether written in C++, Java, Python or another programming language, is a way of ______________________________________. Consider the following implementations of the algorithm from above that checks whether or not a number is even.

You should note that these two code snippets look a bit different, as the details of the two programming languages are different, but the ________________________________ ________.

In both the Python and C++ examples, a program has to translate the programmer-readable *source code* into computer-readable *machine code* for the CPU to execute. This process will happen mostly transparently to us as we work with Python via online Python environments or *integrated development environments* (IDEs). I have provided information about getting started with Python tools on the tools page.

## Programming languages

### vs natural languages

- like natural languages
- unlike natural languages

### Syntax and semantics

- *syntax*: rules of **well-formed** language
- *semantics*: the **meaning** of it all

Like natural languages, a medium for expressing semantics

Unlike natural languages, highly constrained (more like math). Allows succinct yet powerful constructions.

## *Write some software*

### Yes, right now!

1. Think about a problem, e.g., what is $1 + 2 \times 3 - 4$?
2. Compute an answer
3. Check your answer with Python

Type `1 + 2 * 3 - 4` into, e.g., https://www.pythonmorsels.com/repl, then press Enter

## What did you just do?

**Wrote an *expression***

**Expression was *evaluated***

## What is an expression?

**Algebra: values, operators that evaluate to a value**

**Programming: the same! (even operator precedence)**

# Exercise 0

## Submit a Python expression that evaluates to 42

Submit `.py` file to Gradescope

**Not:**

```
Python 3.9.1
>>> x = 21
>>> y = 21
>>> x + y
42
```

**Or:**

```
x = 21
y = 21
print(x + y)
```

**Just:**

```
21 + 21
```

**Or even:**

```
42
```

**See: "Resources" > "Tools"**

## Questions?