**MIT – 6.00.1x: Introduction to Computer Science and Programming**

**WEEK 1**

**Lecture 1: Introduction to Computation**

Part 1: Basics of Computation

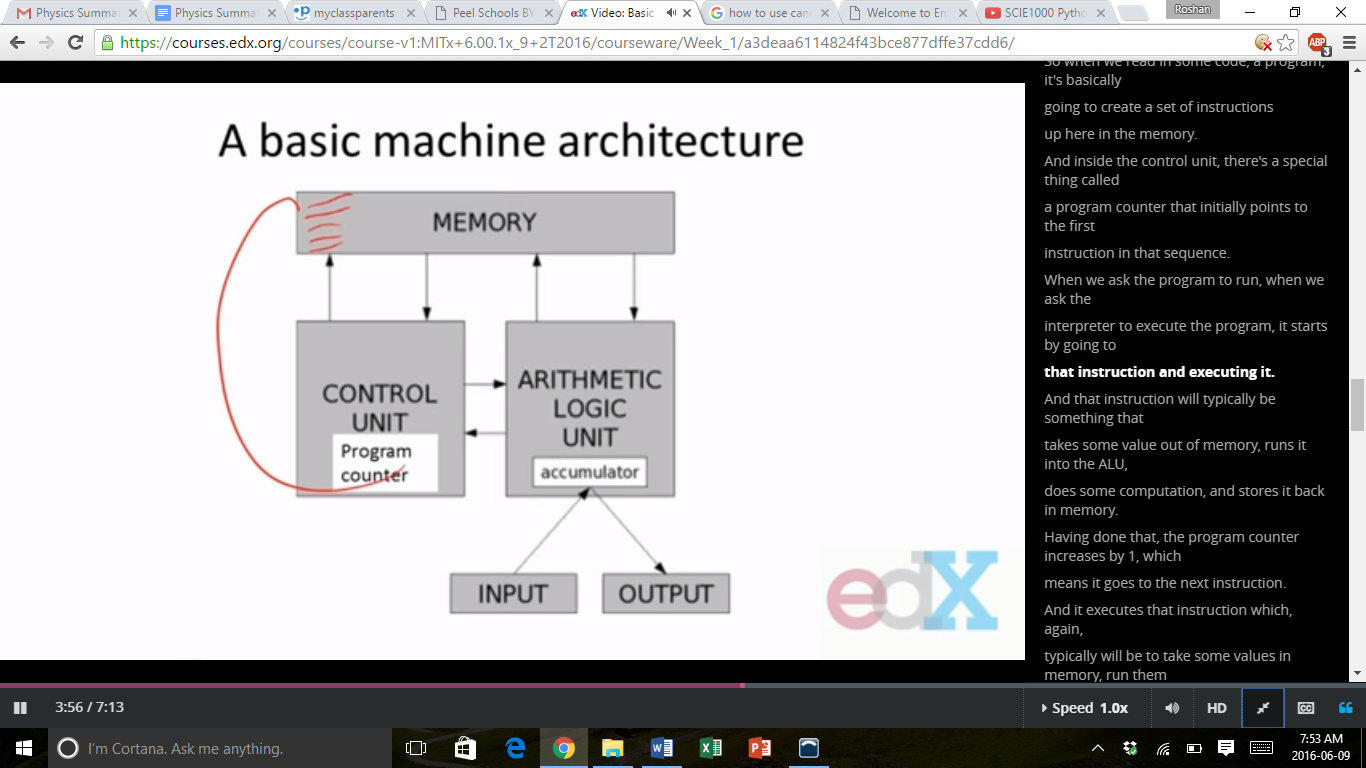
* 6.00 Introduction to Computer Science and Programming
  + Goal:
    - Become skillful at making a computer do what you want it to do.
    - Learn computational modes of thinking.
* What Does a Computer Do?
  + Performs calculations
  + Remembers the result
  + Built in primitives
  + Creating our own methods of calculating

Part 2: Types of Knowledge

* Computational Problem Solving
  + What is computation?
    - What is knowledge?
    - Declarative knowledge
* Statements of fact
  + - Imperative knowledge
* “How to” recipes
* Declarative Knowledge
  + The square root of a number *x* is a number *y* such that *y \* y = x*.
  + You can’t use this to find a particular instance of *x*, but you know that there is such a thing.
* Imperative Knowledge
  + Here is a recipe for deducing the square root of a number.
    - Start with *g*.
    - If *g \* g* is close enough to *x*, then stop and say *g* is the answer.
    - Otherwise, make a new guess, by averaging *g* and *x/g*.
    - Use this new guess, repeat the process until we get close enough.
* Algorithms are Recipes
  + Recipes for cooking food are much like algorithms.

Part 3: Basic Machine Architecture

* How Do We Capture a Recipe in a Mechanical Process?
  + Build a machine to compute square roots.
    - Fixed program computers (e.g. calculator, Alan Turing’s bombe, Atanasoff and Berry’s computer for solving systems of linear equations)
  + Use a machine that stores and manipulates instructions.
    - Stored program computers
* Stored Program Computer
  + Sequence of instructions (program) stored inside the computer.
    - Built from a predefined set of instructions.
* Arithmetic and logic
* Simple tests
* Moving data
  + Special program (interpreter) executes each instruction in order.
* A Basic Machine Architecture



* What are the Basic Primitives?
  + Turing showed that using 6 primitives, can compute anything. But if everything had to be coded down to those 6 primitives, it would be a great pain.
  + Fortunately, modern programming language have a more convenient set of primitives.
  + Also have ways to abstract methods to create new “primitives”.

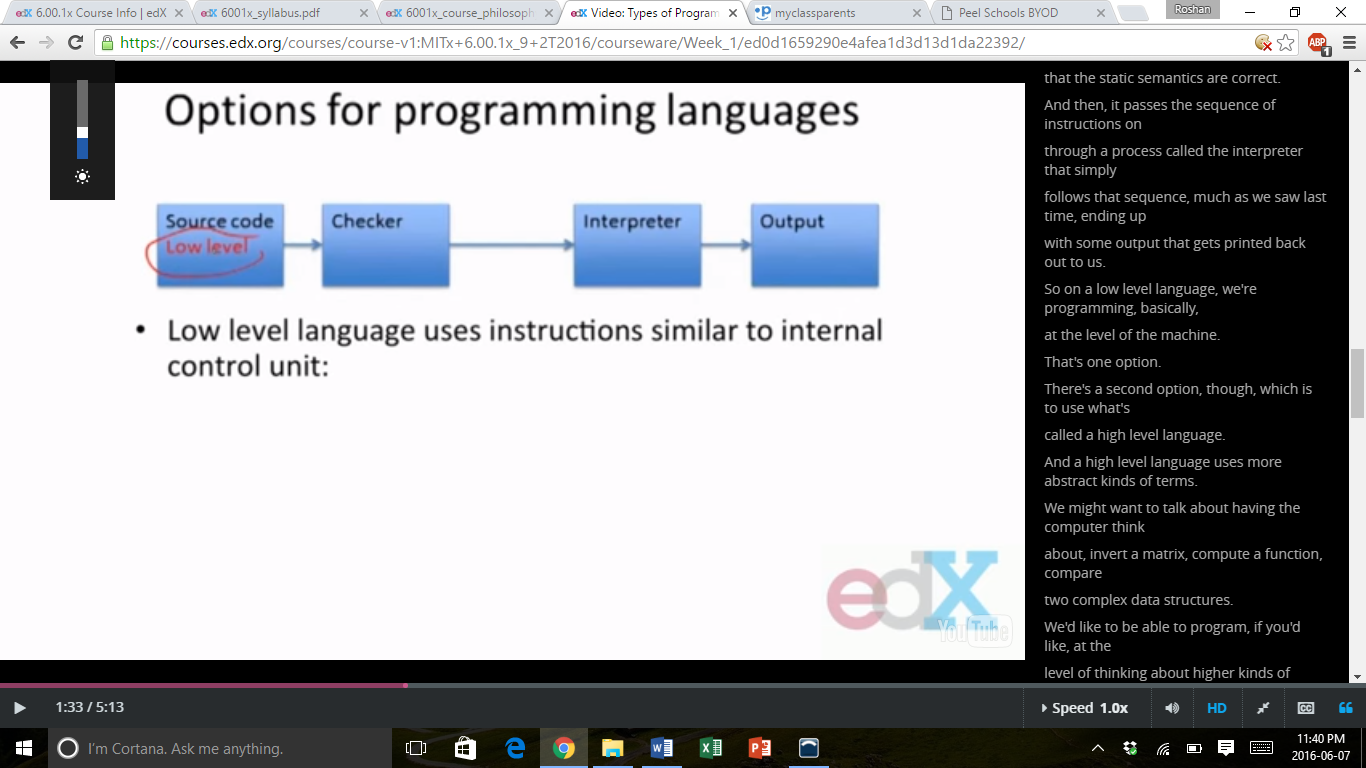
Part 4: Programming Language Characteristics

* Creating “Recipes”
  + Each programming language provides a set of primitive operations.
  + Each programming language provides mechanisms for combining primitives to form more complex, but legal, expressions.
  + Each programming language provides mechanisms for deducing meanings or values associated with computations and expressions.
* Aspects of Languages
  + Primitive constructs
    - Programming language – numbers, strings, simple operators (123, abc, +–=)
    - English – words
  + Syntax – which strings of characters and symbols are well-formed
    - Programming language – we’ll get to specifics shortly, but for example 3.2 + 3.2 is a valid Python expression
    - English – “cat dog boy” is not syntactically valid, as not in form of acceptable sentence
  + Static Semantics – which syntactically valid strings have a meaning
    - English – “I are big” has <noun> <intransitive verb> <noun>, so it is syntactically valid, but is not valid English because “I” is singular, “are” is plural
    - Programming Languages – for example, <literal> <operator> <literal> is a valid syntactic form, but 2.3/’abc’ is a static semantic error
  + Semantics – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
    - English – can be ambiguous
    - Programming Languages – always has exactly one meaning
* But meaning may not be what the programmer intended
* Where Can Things Go Wrong?
  + Syntactic errors
    - Common but easily caught by computer
  + Static semantic errors
    - Some languages check carefully before running, others check while interpreting the program
    - If not caught, behaviour of program is unpredictable
  + Programs don’t have semantic errors, but meaning may not be what was intended
* Crashes (stops running)
* Runs forever
* Produces an answer, but not programmer’s intent
* Our goal
  + Learn the syntax and semantics of a programming language
  + Learn how to use those elements to translate “recipes” for solving a problem into a form that the computer can use to do the work for us

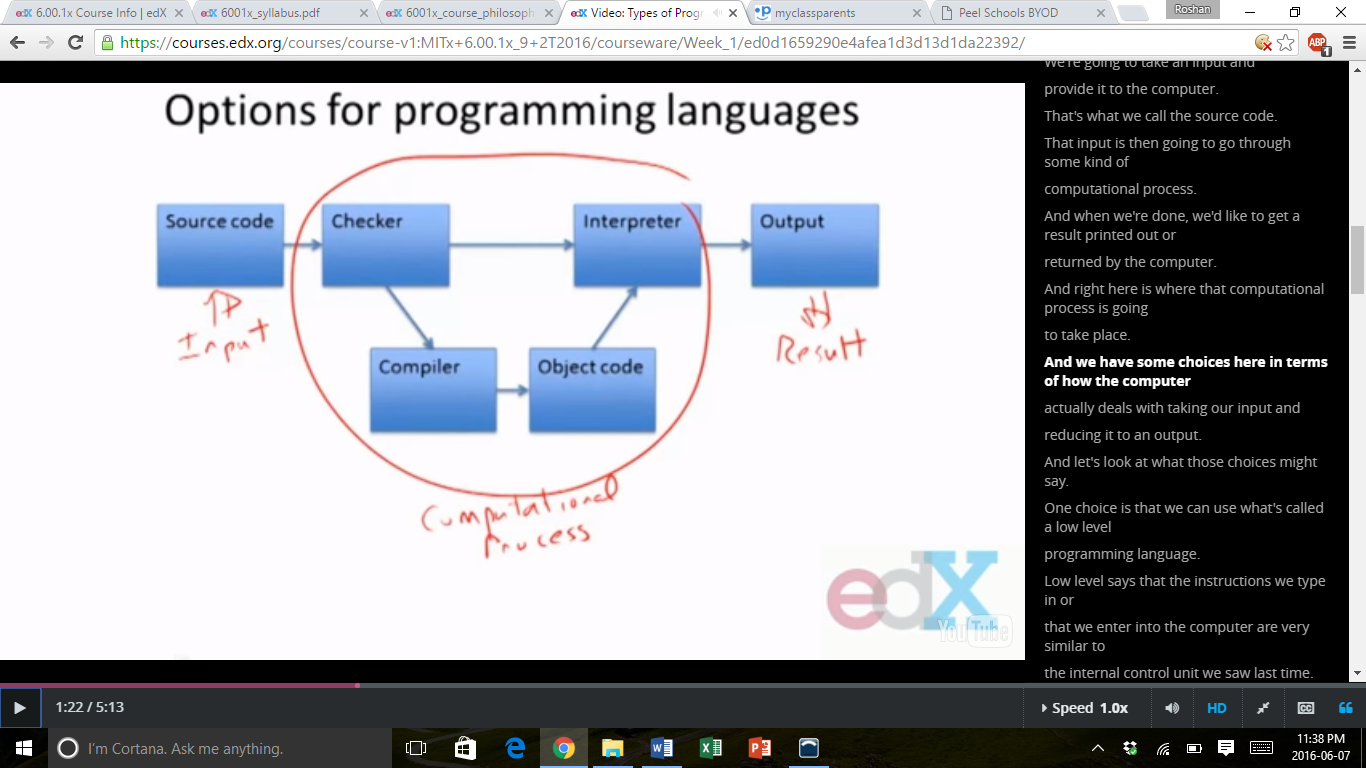
**Lecture 2: Core Elements of Programs**

Part 1: Types of Programming Languages

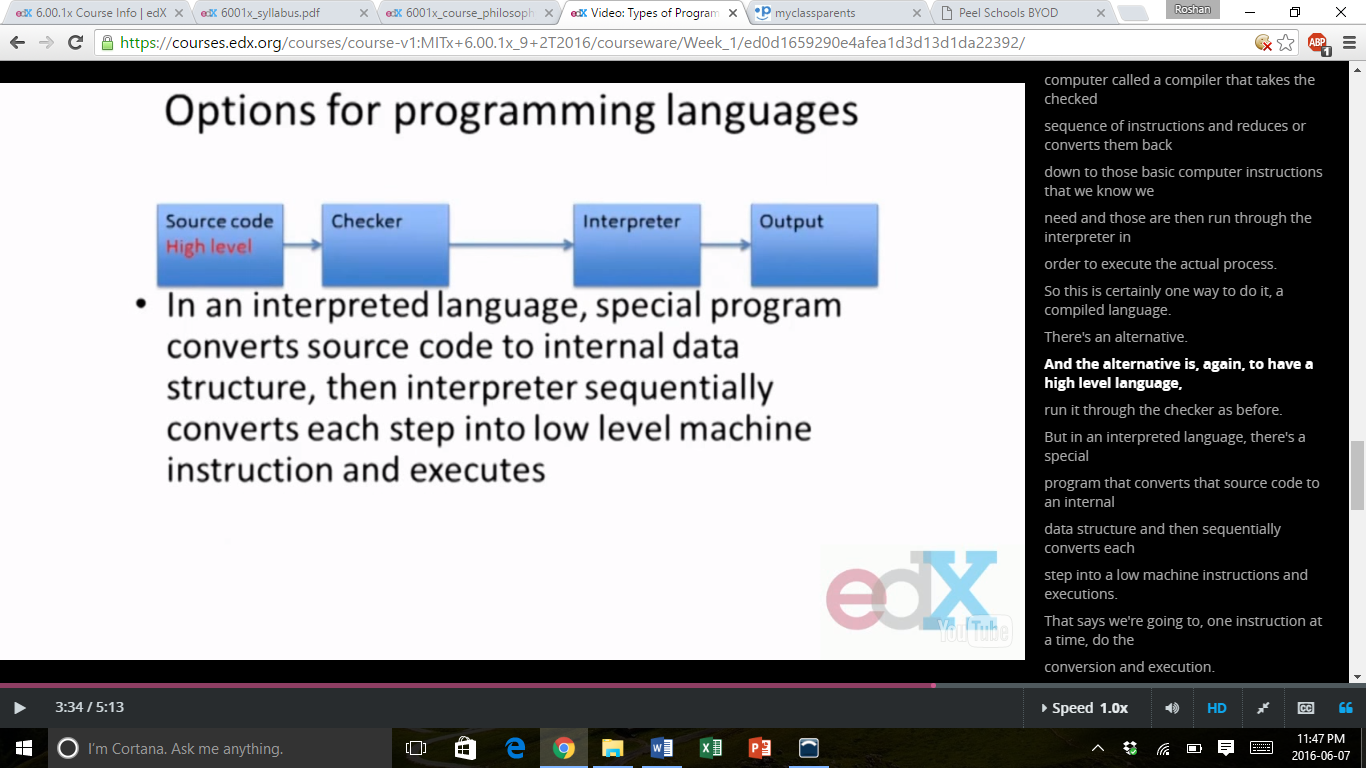
* Programming Languages
  + Goal:
    - Need a way to describe algorithmic steps such that a computer can use them to execute processes.
    - Programming language defines syntax and semantics needed to translate our computational ideas into mechanical steps.
    - Syntax is the study of how to put legal expressions together.
    - Semantics is the study of how to deduce the meanings associated with those expressions.
* Options for Programming Languages



* + Low level language uses instructions similar to internal control unit:
    - Move data from one location to another.
    - Execute a simple ALU (arithmetic logic unit) operation.
    - Jump to a new point in the sequence of instructions based on test.
  + Checker confirms syntax, static semantics correct.
  + Interpreter just follows sequence of simple instructions.



* + A high level language uses more abstract terms – invert a matrix, compute a function, compare two complex data structures, etc.
  + In a compiled language, those abstractions are converted back into low level instructions, then executed.
  + In a compiled language, the code is typically faster since all the work is done ahead of time through the compiler, which converts instructions into low level machine code. So we can make them very efficient, which means that they will run faster.
  + But one of the challenges is if we have a bug/error in the code, it can often be difficult to find out what caused it.



* + In an interpreted language, a special program converts source code to internal data structure, then interpreter sequentially converts each step into low level machine instructions and executes.
  + We are going to use Python, which belongs to this class of programming languages.
  + In an interpreted language, it can be a little slower, not greatly, but sufficiently. This is because we are converting all high level code one instruction at a time. On the other hand, it can be easier to pinpoint what caused an error in our code because we know exactly at what point the error occurred.

Part 2: Python Objects

* Python Programs
  + Program (or script) is a sequence of **definitions** and **commands**.
    - Definitions evaluated and stored away, and commands executed by Python interpreter in a **shell**.
    - Can be typed directly into a shell, or stored in a file that is read into the shell and evaluated. It’s called a shell because it covers up the details of the OS and just lets us interact with things.
  + **Command** (or **statement**) instructs interpreter to do something.
* Objects
  + The first thing we have to do is describe data objects inside a computer. These are the things that capture information and that can be manipulated in some way by the computer to determine more information.
  + At heart, programs will manipulate data objects.
  + Each data object has a **type** that defines kinds of things programs can do with it.
  + Objects are typically:
    - **Scalar** (i.e. cannot be subdivided) or
    - **Non-scalar** (i.e. have internal structure that can be accessed)
* Scalar Objects
  + 3 different kinds of scalar objects:
    - int – used to represent integers, e.g., 5 or 10082
    - float – used to represent real numbers, e.g., 3.14 or 27.0
    - bool – used to represent Boolean values true and false

>>> type (3)

<type ‘int’>

>>> type (3.0)

<type ‘float’>

* Expressions
* Objects and operators can be combined to form **expressions**, each of which denotes an object of some type.
* The syntax for most simple expressions is <object> <operator> <object>.
* Operators on Integers and Floats
  + i + j – sum – if both are ints, result is int, if either is float, result is float
  + i – j – difference
  + i \* j – product
  + i / j – division – if both are ints, result is int, representing quotient without remainder, it rounds down to the nearest integer
  + i % j – remainder
  + i \*\* j – i raised to the power of j

>>> 3.0 + 4

7.0

>>> 3 / 2

1

>>> 3.0 / 2

1.5

>>> 3 \*\* 2

9

>>> (2 + 3) \* 4

20

* Performing Simple Operations
  + Parentheses define sub-computations – complete these to get values before evaluating larger expression.
  + Operator precedence:
    - In the absence of parentheses (within which expressions are first reduced), operators are executed left to right in the following order:
* \*\*
* \* and /
* + and –
* Comparison Operators on Integers and Floats
  + i > j – returns True if strictly greater than
  + i >= j – returns True if greater than or equal to
  + i < j – returns True if strictly less than
  + i <= j – returns True if less than or equal to
  + i == j – returns True if equal
  + i != j – returns True if not equal

>>> 3 >= 4

False

>>> 3 != 4

True

* Operators on Booleans
  + a and b – is True if both are True
  + a or b – is True if either is True
  + not a – True if a is False, and False if a is True
* Type Conversions (Type Casting)
  + We can often convert an object of one type to another, by using the name of the type as a function.

>>> float (3)

3.0

>>> int (4.3)

4

Part 3: Variables and Naming

* Simple Means of Abstraction
  + While we can write arbitrary expressions, it is useful to give names to values of expressions, and to be able to reuse those names in place of values.
  + The following are examples of assignment.
  + pi = 3.14159
  + radius = 11.2
  + area = pi \* (radius \*\* 2)
* Binding Variables and Values
  + The assignment pi = 3.1415 assigns the name pi to the value of the expression to the right of the equals sign.
  + Think of each assignment as creating a binding between a name and a value stored somewhere in the computer.
  + We can retrieve the value associated with a name or variable by simply invoking that name, e.g. pi
* Changing Bindings
  + Variable names can be rebound, by invoking new assignment statements.
  + For example, this is what is changed:
    - radius = 14.3
    - area = 394.08104
  + Note that this doesn’t change the value associated with area.

Part 4: Strings

* Non-Scalar Objects
  + We will see many different kinds of compound objects.
  + The simplest of these are strings, objects of type str.
  + ‘abc’
    - “123” – think of this as a string of characters, not numbers

>>> foo = “abc”

>>> foo

“abc”

>>> type (foo)

str

>>> 3 \* ‘ab’

‘ababab’

>>> ‘a’ + ‘b’

‘ab’

>>> ‘a’ + str (123)

‘a123’

>>> len (‘abc’)

3

* Extracting Parts of Strings
  + Indexing
    - ‘abc’[0] returns the string ‘a’
    - ‘abc’[2] returns the string ‘c’
    - ‘abc’[3] is an error
    - ‘abc’[-1] returns the string ‘c’ (essentially counting backwards from the start of the string)
  + Slicing
    - If s is a string, the expression s[start:end] denotes the substring that starts at start, and ends at end at –1.
* ‘abc’[1:3] has the value ‘bc’

Part 5: Simple Scripts

* Programs (or Scripts)
  + While we can type expressions directly to a Python interpreter (for example, using an interface such as an IDLE shell), in general we will want to include statements in a program file.
  + Executing an expression from a script will not produce any output; for that we need statements (not expressions), such as:
    - print (‘ab’)
    - print (3 \* ‘3’)
* Some Simple Code
  + You can use variable names anywhere you might use the expression whose value it holds.

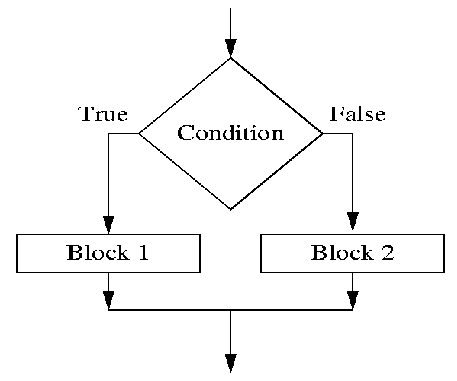
>>> myString = ‘Too much’

>>> weather = ‘snow’

>>> print (myString + ‘ ’ + weather)

Too much snow

* Some Observations
  + Comments appear after a #.
    - These are very valuable, as they help a user understand decisions the programmer has made creating the program.
    - Well commented code should be very readable by a user.
  + A straight line program simply executes each statement in order, with no variation in order.
  + Most programs require more sophisticated flow control.



Part 6: Branching Programs

* Branching Programs
  + The simplest branching statement is a **conditional**.
    - A test (expression that evaluates to True or False)
    - A block of code to execute if the test is True
    - An optional block of code to execute if test is False
* Some Observations
  + The expression x % 2 == 0 evaluates to True when the remainder of x divided by 2 is 0.
  + Note that == is used for comparison, since = is reserved for assignment.
  + The indentation is important – each indented set of expressions denotes a block of instructions.
    - For example, if the last statement were indented, it would be executed as part of the else block of code.
  + Note how this indentation provides a visual structure that reflects the semantic structure of the program.
  + We can use nested conditionals or compound Booleans within our conditionals to make them more sophisticated.
* What Have We Added?
  + Branching programs allow us to make choices and do different things.
  + But it’s still the case that at most, each statement gets executed once.
  + So the maximum time to run the program depends only on the length of the program.
  + These programs run in **constant time**.