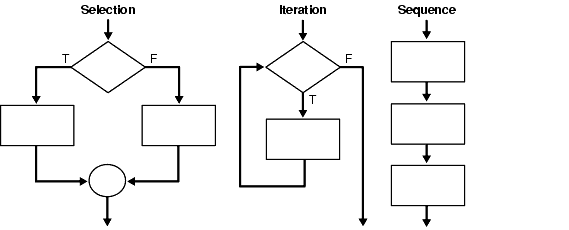
**MIT – 6.00.1x: Introduction to Computer Science and Programming**

**WEEK 2**

**Lecture 3: Simple Algorithms**

Part 1: Iteration

* Iteration
  + Need one more concept to be able to write programs of arbitrary complexity.
    - Starts with a test.
    - If evaluates to True, then execute loop body once, and go back to re-evaluate the test.
    - Repeat until test evaluates to False, after which code following iteration statement is executed.
  + Some Properties of Iteration Loops
    - Need to set an iteration variable outside the loop.
    - Need to set a test variable to determine when done.
    - Need to change that variable within the loop, in addition to other work.

Part 2: Guess and Check Algorithms

* Classes of Algorithms
  + Iterative algorithms allow us to do more complex things than simple arithmetic.
  + We can repeat a sequence of steps multiple times based on some decision; leads to new classes of algorithms.
  + One useful example is “guess and check” methods.
* Guess and Check
  + Remember our “declarative” definition square root of x.
  + If we could guess possible values for the square root (call it g), then we can use definition to check if g \* g = x.
  + We need a good way to generate guesses.
* Finding a Cube Root of an Integer
  + One way to use this idea of generating guesses in order to find a cube root of x is to first try 0 \*\* 3, then 1 \*\* 3, then 2 \*\* 3, and so on.
  + Can stop when reach k such that k \*\* 3 > x.
  + Only a finite number of cases to try.
* Extending Scope
  + Only works for positive integers.
  + Easy to fix by keeping track of sign, looking for solution to positive cases.
* Loop Characteristics
  + Need a loop variable.
    - Initialized outside the loop.
    - Changes within the loop.
    - Test determination depends on that variable.
  + Often, it is useful to think about a **decrementing function**.
    - Maps set of program variables into an integer.
    - When the loop is entered, the value is non-negative.
    - When value is <= 0, loop terminates.
    - Value is decreased every time through loop.
* What Happens if We Miss a Condition?
  + Suppose we don’t initialize the variable?
    - Remove ans = 0
  + Suppose we don’t change the variable inside the loop?
    - Remove ans += 1
* Exhaustive Enumeration
  + Guess and check methods can work on problems with a finite number of possibilities.
  + Exhaustive enumeration is a good way to generate guesses in an organized manner.

Part 3: Loop Mechanisms

* For Loops
  + While loops generally iterate over a sequence of choices (ints in cases we have seen).
  + Python has a specialized mechanism for this case, called a for loop.

for <identifier> in <sequence>:

<code block>

* + Identifier is initially bound to the first value in the sequence.
  + Code block is executed.
  + Identifier bound to the next level.
  + Code block is executed.
  + Continues until sequence is exhausted or a **break** statement is executed.
  + To generate a sequence of integers, use:
    - range(n) = [0, 1, 2, 3, …, n – 1]
    - range(m, n) = [m, m + 1, m + 2, …, n – 1]

Part 4: Floating Point Accuracy

* Dealing with Floating Point Numbers
  + Floats approximate real numbers, but useful to understand how.
  + Decimal number:
    - 302 = 3\*10\*\*2 + 0\*10\*\*1 + 2\*10\*\*0 (Remember \*\* is the Python exponentiation operator)
  + Binary number:
    - 10011 = 1\*2\*\*4 + 0\*2\*\*3 + 0\*2\*\*2 + 1\*2\*\*1 + 1\*2\*\*0
    - 10011 = 16 + 0 + 0 + 2 + 1
    - 10011 = 19
  + Internally, the computer represents numbers in binary form.
* Converting Decimal Integer to Binary
  + Consider the example of:
    - x = 1\*2\*\*4 + 0\*2\*\*3 + 0\*2\*\*2 + 1\*2\*\*1 + 1\*2\*\*0 = 10011
  + If we take the remainder relative to 2 (x % 2) of this number, that gives us the last binary digit.
  + If we then divide x by 2 (x/2), all the bits get shifted left.
    - x/2 = 1\*2\*\*4/2 + 0\*2\*\*3/2 + 0\*2\*\*2/2 + 1\*2\*\*1/2 + 1\*2\*\*0/2 = 1001

Keep doing successive divisions; now remainder gets next bit, and so on.

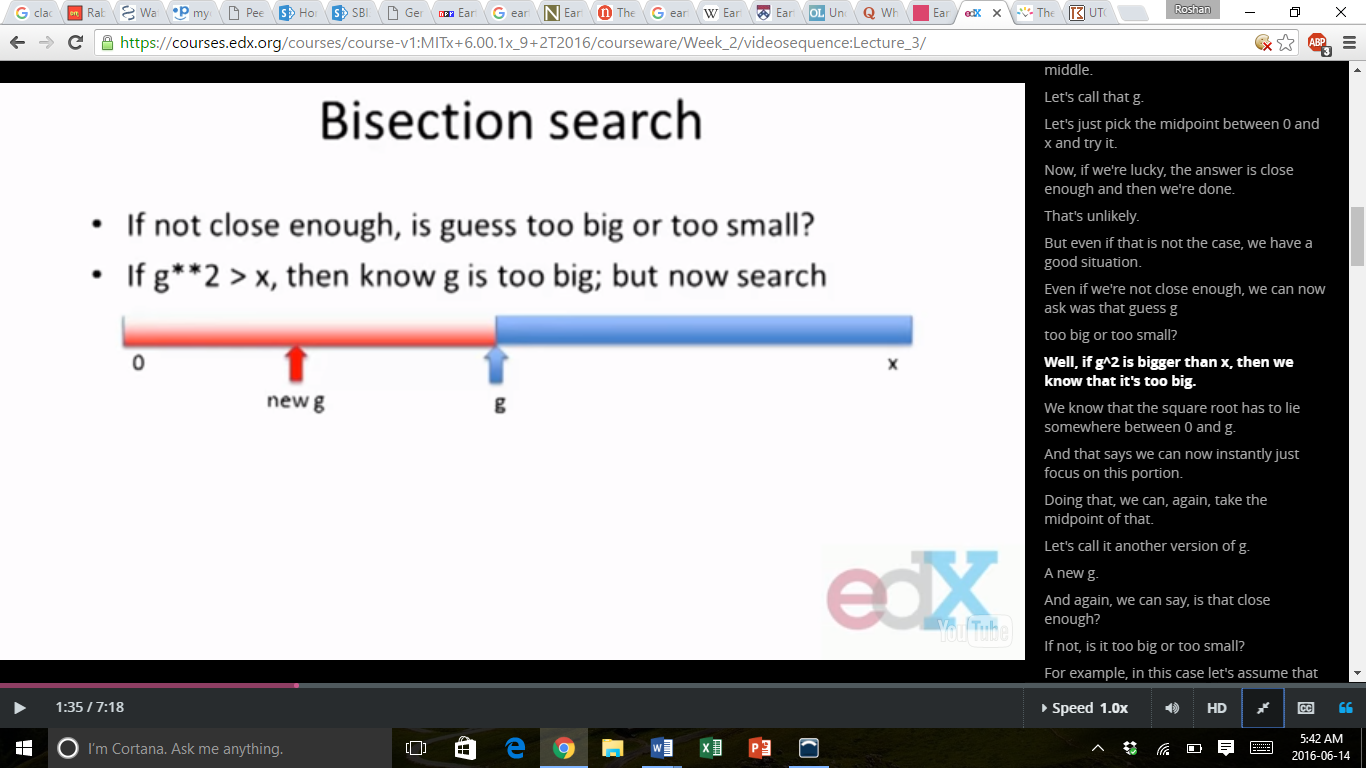
* + - x/2/2 = 1\*2\*\*3/2 + 0\*2\*\*2/2 + 0\*2\*\*1/2 + 1\*2\*\*0/2 = 100
    - x/2/2/2 = 1\*2\*\*2/2 + 0\*2\*\*1/2 + 0\*2\*\*0/2 = 10
    - x/2/2/2/2 = 1\*2\*\*1/2 + 0\*2\*\*0/2 = 1
* So What about Fractions?
  + 3/8 = 0.375 = 3\*10\*\*(-1) + 7\*10\*\*(-2) + 5\*10\*\*(-3)
  + So if we could multiply by a power of 2 big enough to convert into a whole number, which can convert to binary, then we can also divide it by the same power of 2.
  + 0.375 \* (2\*\*3) = 3 (decimal)
  + Convert 3 to binary form (now 11)
  + Divide by 2\*\*3 (shift left) to get 0.011 (binary)
* Some Implications of Converting Decimals to Binary
  + If there is no integer such that is a whole number, then the internal representation of the binary number is always an approximation.
  + Suggests that the testing quality of floats is not exact.
    - Use , rather than
  + Then why does print (0.1) return 0.1, if not exact?
    - Because Python designers set it up this way to round automatically.

Part 5: Approximation Methods

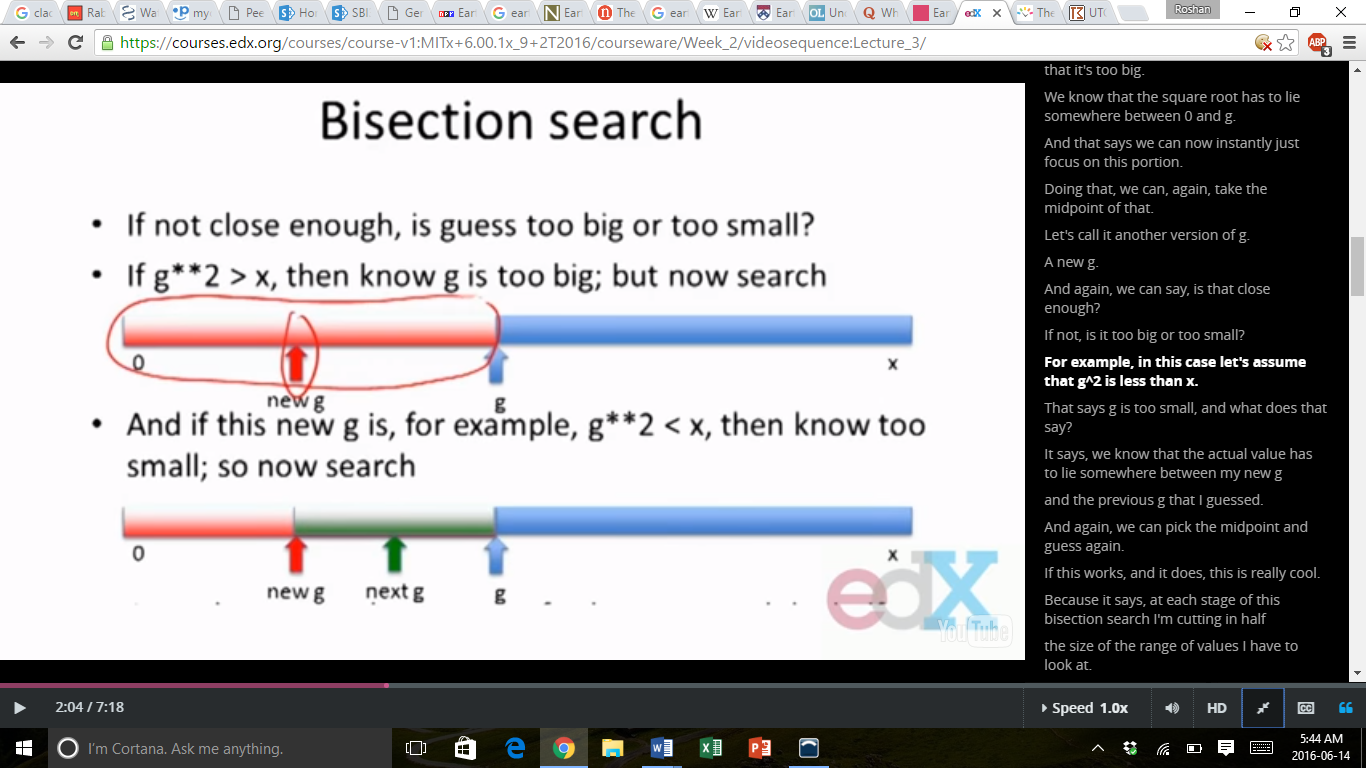
* Approximate Solutions
  + Suppose we want to find the square root of *any* non-negative number?
  + Can’t guarantee exact answer, but just look for something close enough.
  + Start with exhaustive enumeration.
    - Take small steps to generate guesses in order.
    - Check to see if we’re close enough.
* Some Observations of Square Root
  + Step could be any number.
    - If too small, the program will take too long to find the square root.
    - If too large, the program might skip over the answer without getting close enough.
  + In general, the solution to find square root will take x/step times through code to find the solution.

Part 6: Bisection Search

* Bisection Search
  + We know that the square root of x lies between 0 and x, from mathematics.
  + Rather than exhaustively trying things starting at 0, suppose we instead pick a number in the middle of this range.
  + If we are lucky, the answer g, which is at the midpoint of 0, is close enough to x when squared.
  + But even if that is not the case, if it is not close enough, we can ask if guess is too big or too small?
  + If g\*\*2 > x, then we know g is too big; refine search to:



* + And if this new g, is for example, g\*\*2 < x, then we know that the search is too small; refine search to:



* + At each stage of this bisection search, reduce the range of values possible by half.
* Some Observations
  + Bisection search radically reduces computation time – being smart about generating guesses is important.
  + Should work well on problems with “ordering” property – this means that the value of function solved varies monotonically with input value.
    - Here function is g\*\*2, which grows as g grows.

Part 7: Newton-Raphson Root Finding

* Newton-Raphson
  + General approximation algorithm to find the roots of a polynomial in one variable.
  + Want to find a value such that.
  + For example, to find the square root of 24, we need to find the root of the equation.
  + Newton showed that if g is an approximation to the root, then:
  + The above formula gives a better approximation of the root; where is the derivative of.
  + Simple case:
  + First derivative:
  + So, if the polynomial is, then the derivative is
  + Newton-Raphson says that given a guess for the root, a better guess is:
  + This gives us another way of generating guesses for the square root of a number; which we can check – it is very efficient.
* Iterative Algorithms
  + Guess and check methods build on reusing the same code.
    - Use a looping construct to generate guesses, then check and continue.
  + Generating guesses:
    - Exhaustive enumeration
    - Bisection search
    - Newton-Raphson (for root finding)

**Lecture 4: Functions**

Part 1: Creating Functions

* Functions
  + So far, we have seen numbers, assignments, input/output, comparisons, and looping constructs.
    - Sufficient to be **Turing Complete**
    - Technically, we can now compute anything that is computable, according to the definition of Turing Complete
  + But code lacks abstraction
    - I have to reload the file every time I want to use it.
    - I can’t use the same variable names in other pieces of code.
    - It can quickly become cumbersome to read and maintain.
  + Functions give us abstraction – allow us to capture computation and treat as if primitive.
* A Simple Example
  + Suppose we want z to be the maximum of two numbers, x and y.
  + A simple script would be:

if (x > y):

z = x

else:

z = y

* Capturing Computation as a Function
  + Idea is to encapsulate this computation within a scope such that we can treat it as a primitive variable.
    - Use by simply calling name, and providing input.
    - Internal details hidden from users.
  + Syntax

def <function name> (<formal parameters>):

<function body>

* + def is a keyword
  + Name is any legal Python name
    - Within parenthesis are zero or more formal parameters – each variable name is used inside function body.
* A Simple Example
  + When we call or invoke max (3, 4), x is bound to 3, y is bound to 4, and then body expression(s) are evaluated.
* Function Returns
  + Body can consist of any number of legal Python expressions.
  + Expressions are evaluated until.
    - Run out of expressions, in which case the special value None.
    - Or when special keyword return is reached, in which case subsequent expression is evaluated and that value is returned as a value of the function call.
* Summary of Function Call

1. Expressions for each parameter are evaluated, bound to formal parameter names of function.
2. Control transfers to first expression in the body of the function.
3. Body expressions executed until return keyword reached (returning value of next expression) or run out of expressions (returning None).
4. Invocation is bound to the returned value.
5. Control transfers to the next piece of code.

Part 2: Environments

* Environments to Understand Bindings
  + Environments are formalism for tracking bindings of variables and values.
  + Assignments pair names and values in a table, in some sense, or what is formally known as an environment.
  + Asking for value of name just looks up the value in the current environment.
  + Python shell is default (or global) environment. It contains the bindings of all the expressions that we put in as we deal with.
  + Definitions pair function name with details of the function in the environment. It creates a pairing of the name to what we call a procedure object.
* Simple Example

x = 5

p = 3

result = 1

for turn in range(p):

print (‘iteration: ’ + str(turn) + ‘; current result: ’ + str(result))

result \*= x

* + Global Environment (Python shell)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable | Value #1 | Value #2 | Value #3 | Value #4 |
| x | 5 | 5 | 5 | 5 |
| p | 3 | 3 | 3 | 3 |
| result | 1 | 5 | 25 | 125 |

* Back to Functions

x = 5

y = 3

def max (x, y):

if (x > y):

return x

else:

return y

Procedure1

(x, y)

if (x > y):

return x

else:

return y

* + Global Environment (Python Shell)

*pointers*

|  |  |
| --- | --- |
| Variable | Value #1 |
| x | 5 |
| p | 3 |
| max |  |

* When We Call a Function
  + We want to evaluate <expr0>(<expr1>, … , <exprn>)
  + First evaluate <expr0>, which looks up procedure object in environment.
  + Then evaluate each of the other <expri> to get values of parameters in the environment (usually the Python shell).
  + Bind parameter names in procedure object to values of arguments in a new frame which has as a parent the environment in which the procedure was defined.
  + Evaluate body of procedure relative to this new frame.

Part 3: Computing Powers as an Example

* Another Simple Example
  + Suppose we want to compute powers of a number by successive multiplication.
  + Stop when we have multiplied number *a* by itself *p* times, and return the final product.
* Computing Powers
  + Evaluating the body of the procedure causes a new environment to be created, in which the variables x, p, and result are defined separately.
  + Thus, the computation is held in environment 2 (E2) and does not interfere with the global environment (E1).

Part 4: Understanding Variable Binding

* Another Example

def f(x):

y = 1

x += y

print ('x = ' + str(x))

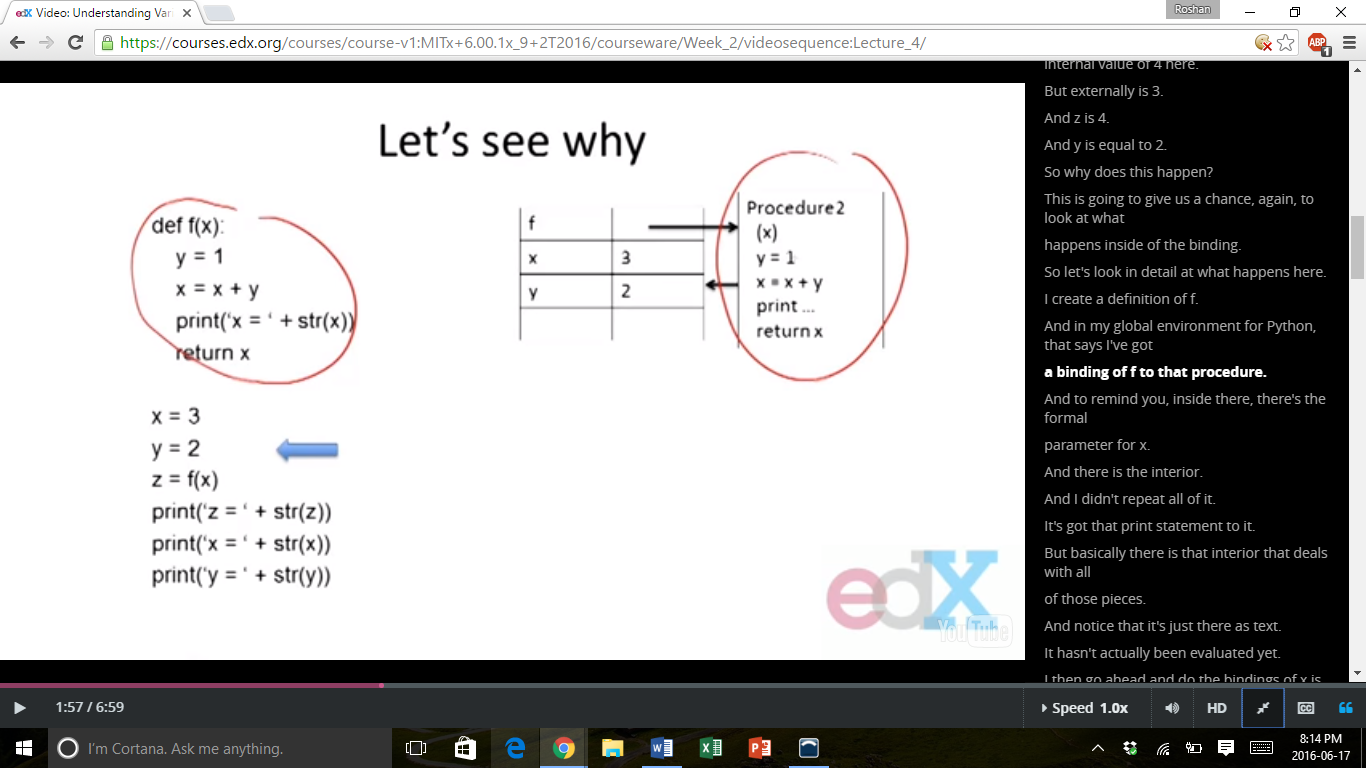
return x

x = 3

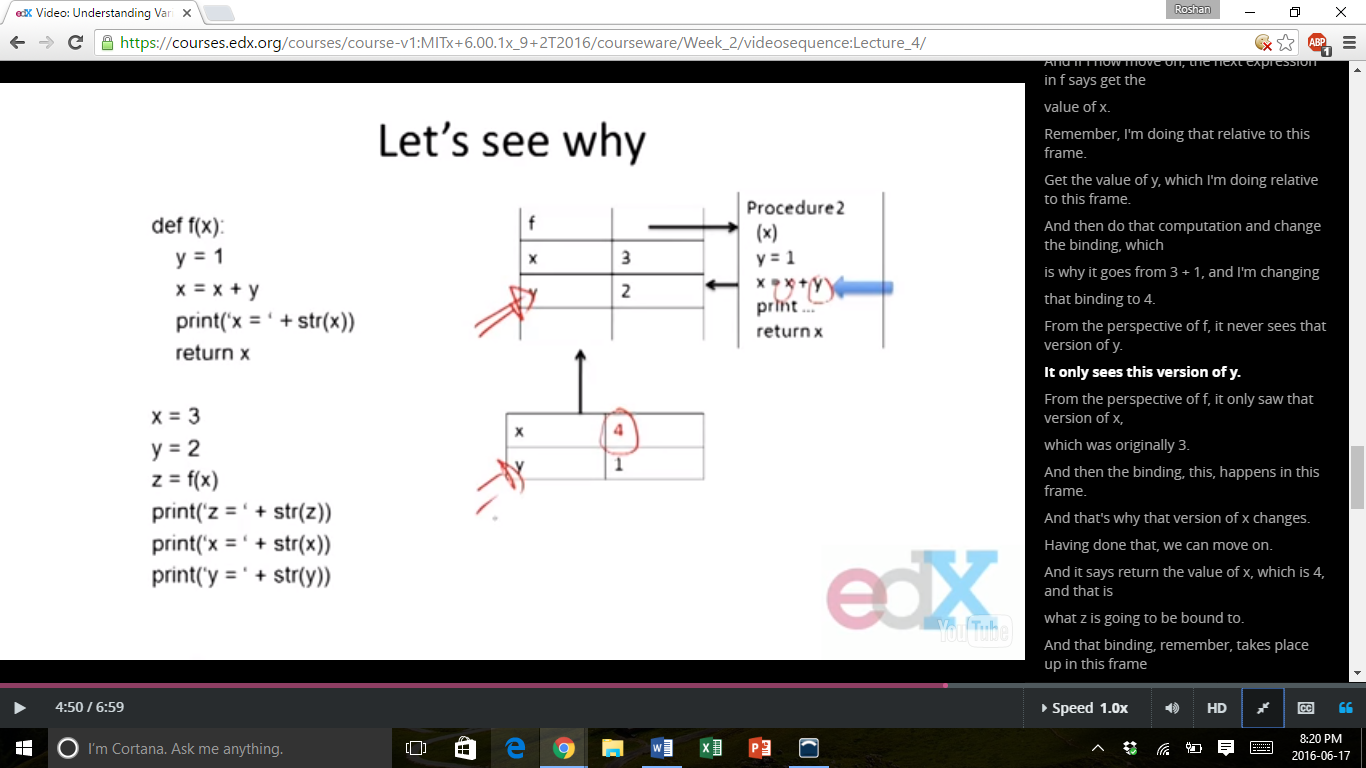
y = 2

z = f(x) = 4

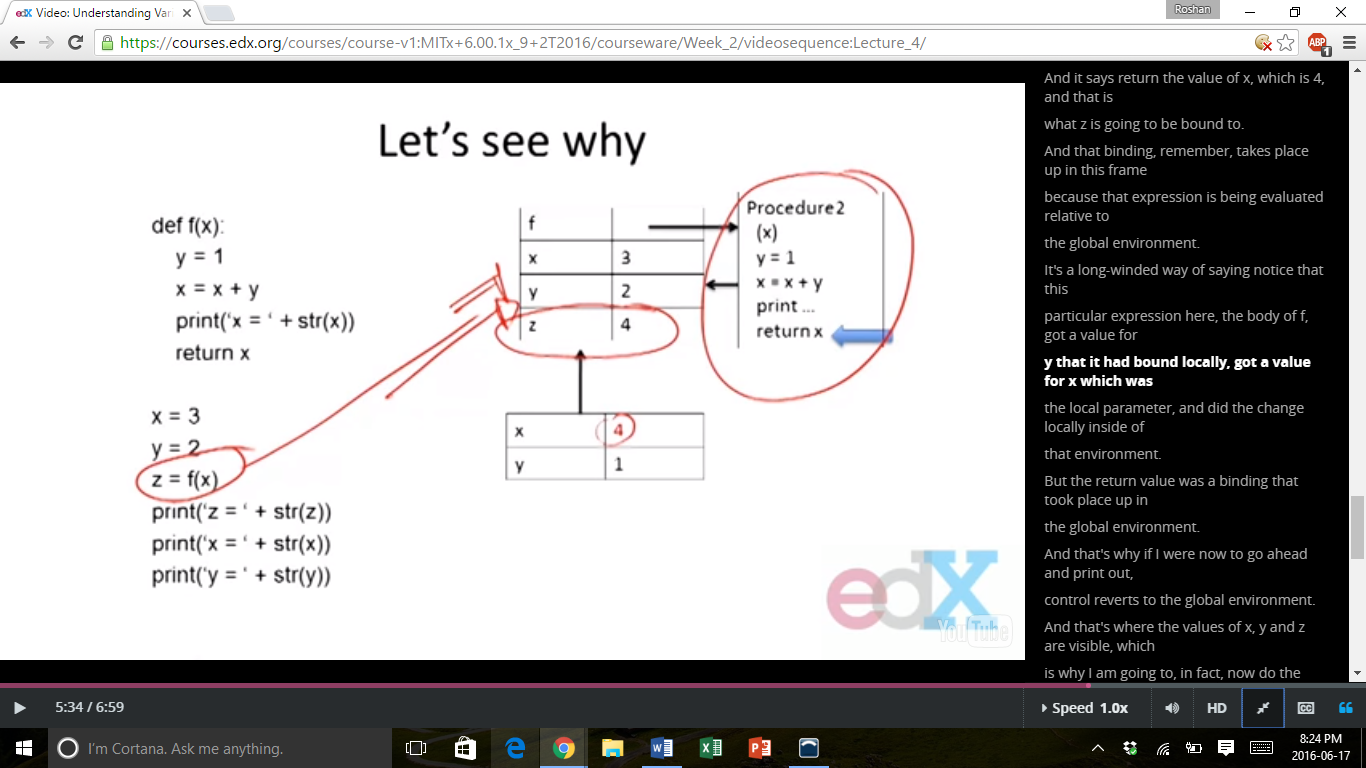
* + This causes the following to happen in Python
    - x = 4
    - z = 4
    - x = 3
    - y = 2
* Let’s See Why



* + Here, it is seen that in the global environment (E1), the values of x and y are bound to 3 and 2, respectively. The letter f is bound to a procedure object whose body has references to x and y, but nothing more.
  + The fact that the procedure object contains x and y is not of concern to the global environment, because these variables are simply used in the computation, but are not bound to the x and y values in the global environment.



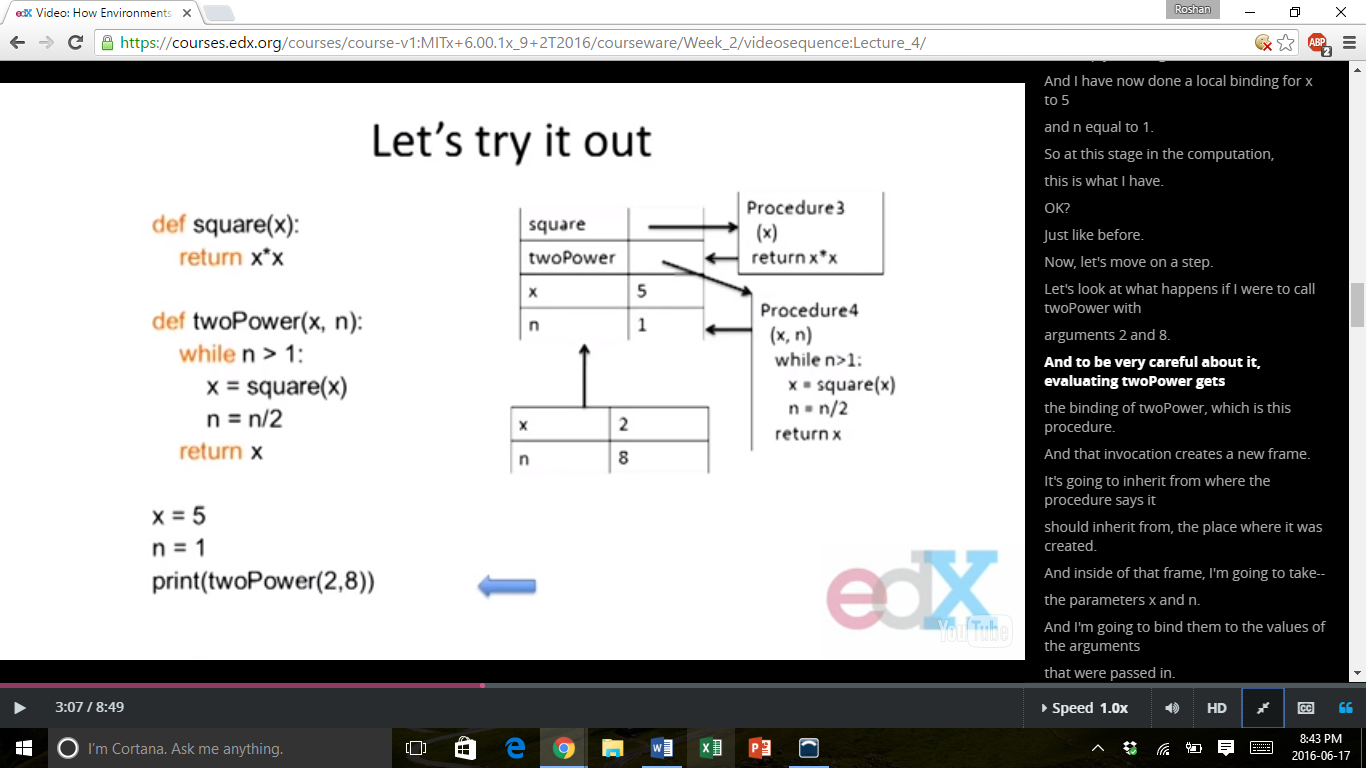
* + Here, it can be seen that y has two bindings – one in the global environment (E1), and one in the local environment of function f (E2). In E2, when it is performing the computation, the function f NEVER sees the variable y in the global environment (E1), and so those values cannot be interchanged.



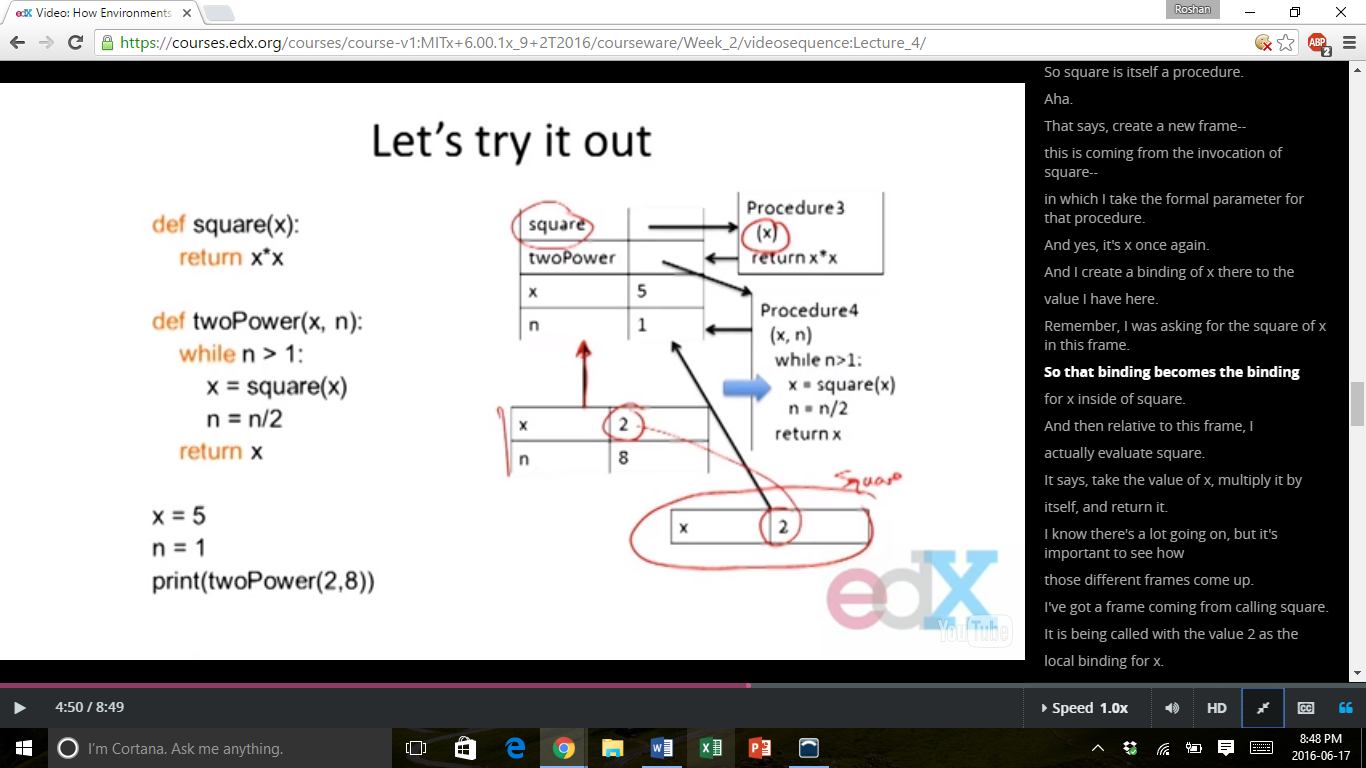
* + Now, it is seen that z is evaluated with respect to the global environment (E1).
  + It references to values in the local environment E2 (of the function f), but, when those values are returned, it does not retain ties to the values in the local environment E2.
* Some Observations
  + Each function call creates a new environment, which scopes bindings of formal parameters and values, and of local variables (those are created with assignments within the body).
  + Scoping is often called static or lexical because the scope within which the variable has a value is defined by the extent of the code boundaries (boundaries are defined by the environment).

Part 5: How Environments Separate Variable Bindings

* Another Example:
  + Let’s look at what happens when multiple functions are called within the global environment, and the local environments have to return respective values.



* + When twoPower (x, n) is called upon, the procedure inherits the values of the formal parameters x and n from the global environment (E1).
  + Inside of the new frame (E2), the values x and n are bound to the arguments of 2 and 8, respectively, because that was how the function was called.



* + Here, when the function square (x) is called within the procedure twoPower (x, n), the local variable defined within the procedure 2 environment (E2) can be used to call procedure 3 with an argument of x that is defined in E2, or x = 2.
  + When the frame E2 changes its local variables to x = 4 and n = 4 due to the instructions in the while loop, then it these variables are put under the same process again, in the while loop, to change the variables as follows:
    - x = 16 ; n = 2
    - x = 256 ; n = 1
  + Here, the computation stops, because n is no longer larger than 1, and the while loop is exited. Then, the value of x is finally returned.
* Some Observations
  + Notice how each call to square created a new frame, with a local binding for x.
  + The value of x in the global environment was never confused with values from frames within function calls.
  + The value of x used by the call to square is different from the binding for x in the call to twoPower (x, n).
  + The rules we described can be followed mechanically to determine scoping of variables.
  + Thus, the same variable names CAN be used, but they just need to be bound in separate environments so that the variables don’t overlap.

Part 6: Understanding Root Finding

* Capturing a More Interesting Example

def findRoot1 (x, power, epsilon):

if (x < 0 and power % 2 == 0):

return None

low = min(0, x)

high = max(0, x)

ans = (high + low) / 2.0

while abs(ans \*\* power - x) > epsilon:

if (ans \*\* power < x):

low = ans

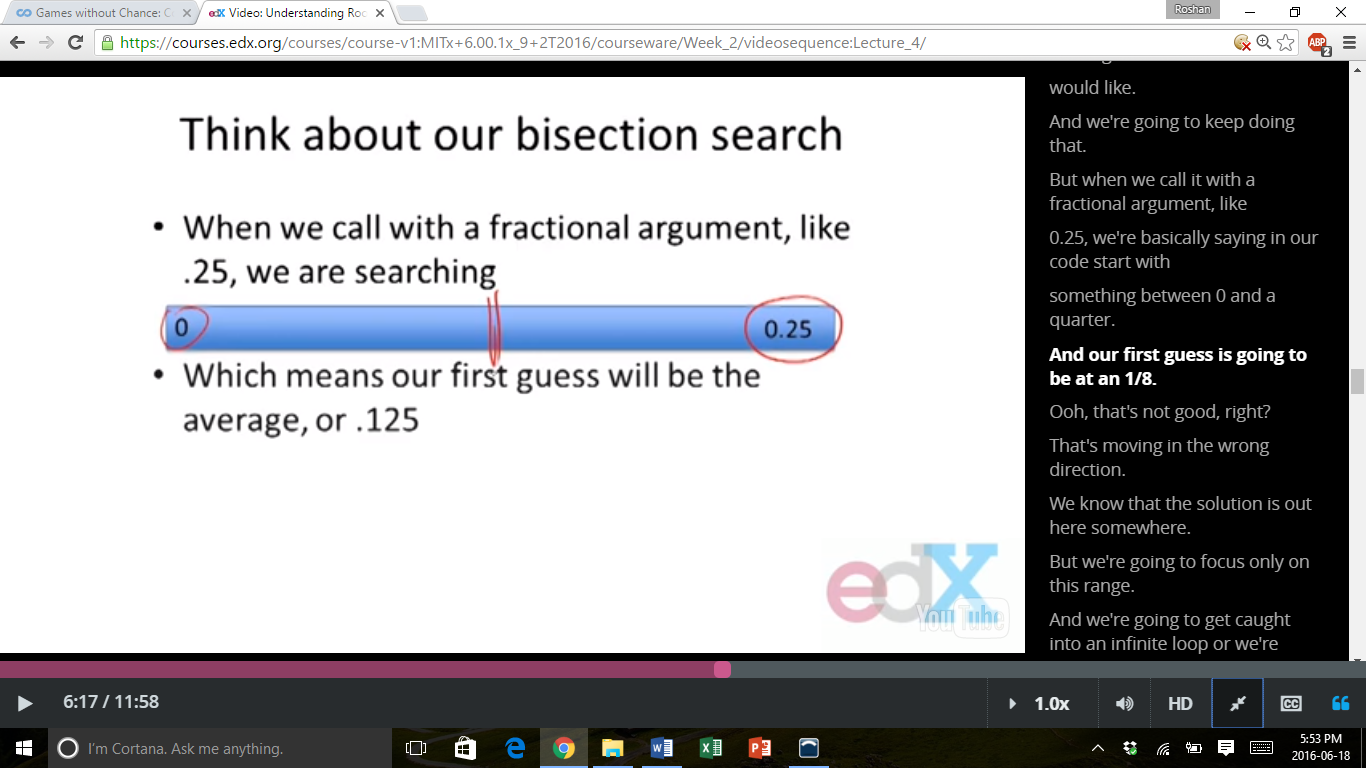
else:

high = ans

ans = (high + low) / 2.0

return ans

* + Here, this function takes in a number x and finds the powerth root of that number within a certain range, epsilon.
* Think About Bisection Search
  + When we call the function with a fractional argument, like 0.25, we are searching in the following range.



* + Which means that our first guess will be the average, or 0.125.
    - Our original idea used the fact that the root of x was between 0 and x, but when x is fractional, the root is between x and 1.
* Specifications
  + Create a contract between implementer of function and user.
    - Assumptions: Conditions that must be met by users of function. Typically constraints on parameters, such as type, and sometimes acceptable range of values.
    - Guarantees: Conditions that must be met by the function, provided that it has been called in a way that satisfies assumptions.
* Functions Close the Loop
  + Can now create new procedures and treat them as if they were Python primitive variables.
  + Properties:
    - Decomposition: Break problems into modules that are self-contained, and can be reused in other settings.
    - Abstraction: Hide details. User does not need to know interior details, can just use as if a black box.

Part 7: Modules

* Using Functions in Modules
  + Modularity is the degree to which a system’s components may be separated and recombined.
  + Thus, the goal of modularity is to group functions together that share a common theme.
  + Place in a single .py file.
  + We can then use the import command to access those files.
* Example

pi = 3.14159

def circleArea (radius):

return pi \* (radius \*\* 2)

def circleCircumference (radius):

return 2 \* pi \* radius

def sphereSurfaceArea (radius):

return 4.0 \* pi \* (radius \*\* 2)

def sphereVolume (radius):

return (4.0 / 3.0) \* pi \* (radius \*\* 3)

* How to Use the Above Functions?

import circle

pi = 3.0 # value in local environment

print (pi) # will print value in local environment, 3.0

print (circle.pi) # will print 3.14159, value from file circle

print (circle.area (3)) # will print 28.27431, uses values from file circle

print (circle.circumference (3)) # will print 18.84954, uses values from file circle

* + The dot notation specifies the context in which the values are read from.
* Another Way to Import Values

from circle import \*

pi = 0.0

print (pi) # will print 0.0, value from local scope

print area (3) # will print 28.27431, uses values from file because area is not locally bound

print circumference (3) # will print 18.84954, uses values from file