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

**WEEK 3**

**Lecture 5: Recursion**

Part 1: Iterative Algorithms

* Iterative Algorithms
  + We have been using looping constructs (e.g. while or for loops) that lead naturally to **iterative** algorithms.
  + Can conceptualize as capturing computation in a set of “state variables” that update on each iteration through the loop.
* Iterative Multiplication by Successive Addition
  + Imagine we want to perform multiplication by successive additions:
    - To multiply a by b, add a to itself b times.
  + State Variables:
    - i – iteration number; starts at i = b
    - result – current value of computation; starts at result = 0
  + Update Rules
    - i 🡨 i – 1; stop when i = 0
    - result 🡨 result + a

Part 2: Recursive Algorithms

* Recursive Version of Multiplication
  + An alternative is to think of this computation as:

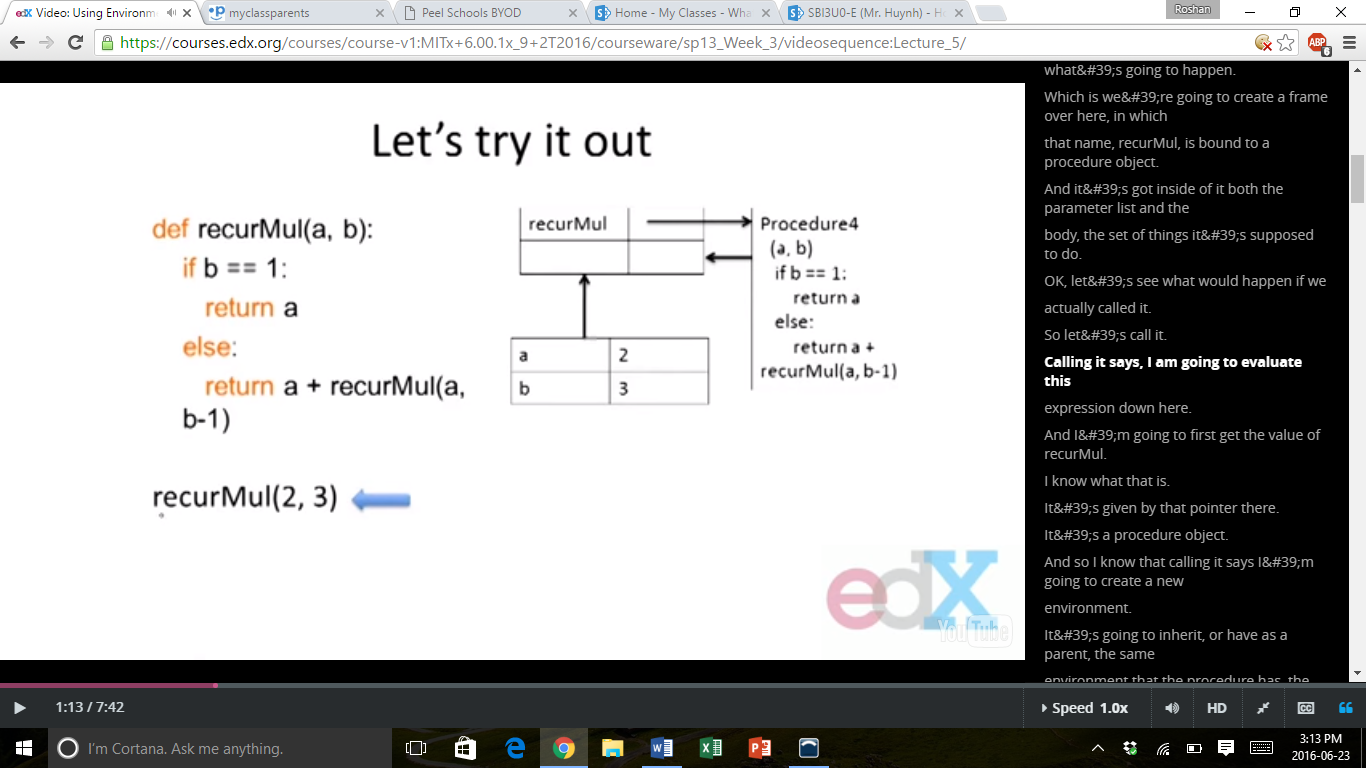
a \* b = a + a + … + a 🡨 b copies

a \* b = a + a \* (b – 1) 🡨 recursive method

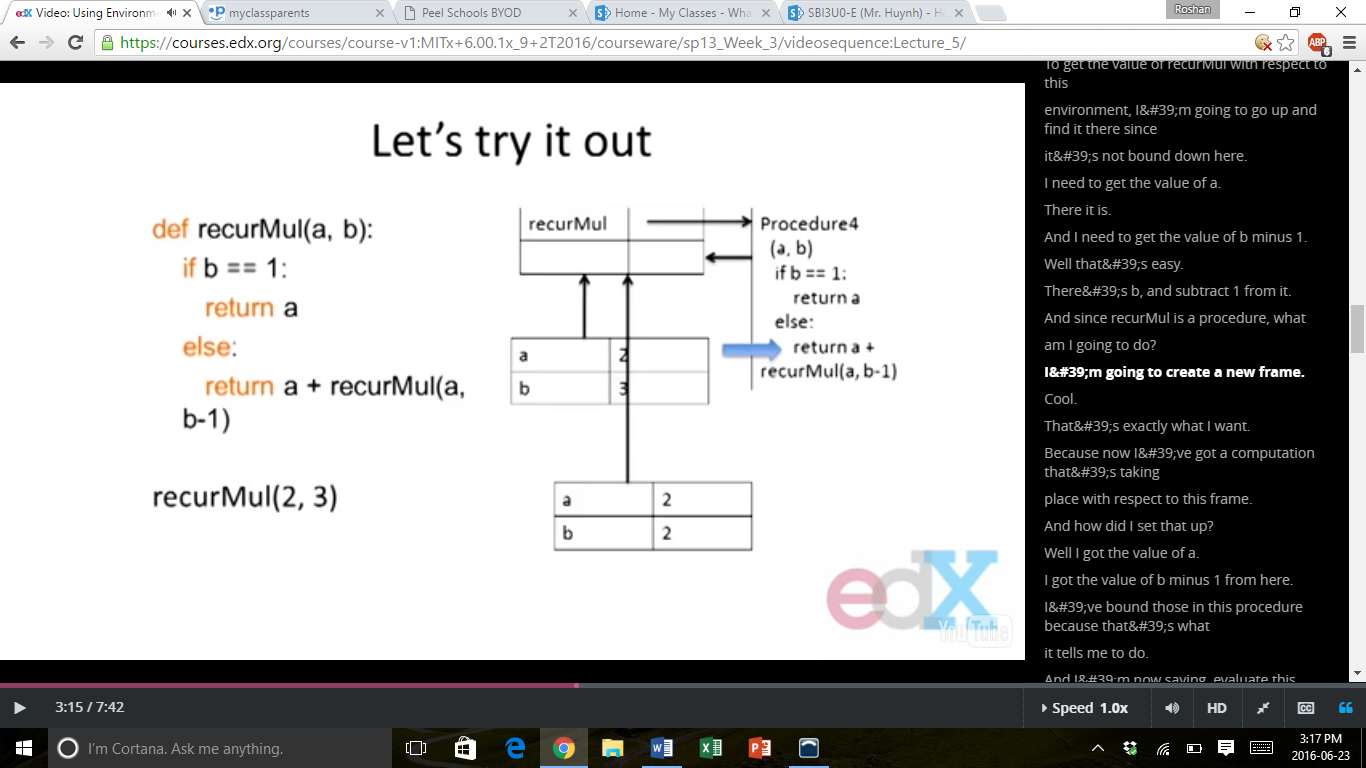
* Recursion
  + This is an instance of a **recursive** algorithm.
    - Reduce a problem to simpler (or smaller) version of the same problem, plus some simple computations.
* **Recursive Step**
  + - Keep reducing until reach a simple case that can be solved directly.
* **Base Case**
  + a \* b = a; if b = 1 🡨 (Base Case)
  + a \* b = a + a \* (b – 1) 🡨 (Recursive Step)

Part 3: Using Environments to Understand Recursion

* Recursion Example Using Environments



* + If recurMul(2, 3) is called, then it loads this environment with a procedure object.
  + The frame below will perform the computation that is described in the procedure recurMul(a, b).
  + recurMul(a, b) says that the program should return a + returnMul(a, b – 1). Thus, a new frame is created, as shown below.



* + These layers of frames keep building up until b reaches 1. When b = 1, then the program finally returns the value of a. Each time the value of a travels back up the ladder of environments it has to pass through, it is added by a, and so a can be added b number of times using this recursive sequence.
* Some Observations
  + Each recursive call to a function creates its own environment, with local scoping of the variables.
  + Bindings for variables in each frame are distinct, and bindings are not changed by the recursive call.
  + Flow of control will pass back to earlier frame once function call returns value.

Part 4: Inductive Reasoning

* Inductive Reasoning
  + How do we know that our recursive code will work?
  + We know that iterMul terminates because b is initially positive, and is decreased by 1 each time when looped; thus b must eventually reach 0.
  + We know that when recurMul is called with b > 1, this makes a function call with a smaller value of b; and this must reach call with b = 1.
  + We know that when recurMul is called when b = 1, this has no recursive call and returns a value.
* Mathematical Induction
  + To prove a statement indexed on integers is true for all values of:
    - Prove it is true when n is the smallest value (e.g. or).
    - Then prove that if it is true for an arbitrary value of, then one can show that it must be true for.
* Example
  + For example, let’s consider the sum:
  + Proof:
  + To assume that this property is true for some value, we need to show that:
  + What does this have to do with Code?
    - Same logic applies

def recurMult (a, b):

if (b == 1):

return a

return a + recurMult (a, b - 1)

* + - For base case, we can show that recurMul must return the correct answer.
    - For recursive case, we can assume that recurMul correctly returns an answer for problems of size smaller than b, then by the addition step, it also must return a correct answer for problem of size b.
    - Thus, by induction, this code correctly returns the answer.

Part 5: Factorial

* The “Classic” Recursive Problem
  + Factorial
  + A way of thinking about this function recursively is:

Part 6: Towers of Hanoi

* Towers of Hanoi
  + The story:
    - 3 tall spikes
    - Stack of 64 different sized discs – all start on one of the spikes.
    - Increasing order of size – smallest at top and largest at bottom.
    - Need to move the entire stack, in the same order, to the second spike.
    - Can only move one disc at a time, and a larger disk can never cover up a smaller disk.
  + Having seen a set of examples of different sized stacks, how would you write a program to print out the right set of moves?
  + Think Recursively!
    - Solve a smaller problem
    - Solve a basic problem
    - Solve a smaller problem

Part 7: Fibonacci Sequence

* Recursion with Multiple Base Cases
  + Fibonacci Numbers
    - Leonardo of Pisa (AKA Fibonacci) modelled the following change.
* Newborn pair of rabbits (one female, one male) are put in a pen.
* Rabbits mat at the age of 1 month.
* Rabbits have a 1 month gestation period.
* Assume rabbits never die, that female produces one new pair (one male, one female) every month from its second month onwards.
* How many rabbits are there at the end of 1 year?

|  |  |
| --- | --- |
| Month | Females |
| 0 | 1 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 5 |
| 5 | 8 |
| 6 | 13 |

* + Let’s Model This
    - After one month (call it 0) – 1 female
    - After second month (call it 1) – still 1 female, now pregnant
    - After third month (call it 2) – two females, one pregnant, one not.
    - In general.
* Every female alive at month will produce one female in month.
* These can be added to those alive in month to get the total alive in month.
  + Fibonacci Model
    - Base Cases
    - Recursive Case

Part 9: Recursion on Strings

* Recursion on Non-Numeric Values
  + How could we check whether a string of characters is a palindrome (i.e. reads the same forwards and backwards)?
    - “Able was I ere I saw Elba” – attributed to Napolean
* How to Solve this Recursively?
  + First, convert the string to just characters, by stripping out punctuation, and converting upper case to lower case.
  + Then
    - Base Case: a string of length 0 or 1 is a palindrome
    - Recursive Case: if the first character matches the last character, then it is a palindrome if the middle section is a palindrome
* Example
  + ‘Able was I ere I saw Elba’ 🡪 ‘ablewasiereisawelba’ 🡪 since first and last characters are same 🡪 ‘blewasiereisawelb’
* Divide and Conquer
  + This is an example of a “divide and conquer” algorithm.
    - Solve a hard problem by breaking it into a set of sub-problems such that:
* Sub-problems are easier to solve than the original.
* Solutions of the sub-problem can be combined to solve the original.

Part 10: Global Variables

* Global Variables
  + Suppose we wanted to count the number of times “fib” calls itself recursively.
  + We can do this using a global variable.
  + So far, all functions have communicated with their environment through their parameters and return values.
  + Though a bit dangerous, we can declare a variable to be global – means name is defined at the outermost scope of the program rather than at the scope of the function within which it appears.
* Global Variables
  + Use with care!
  + Putting a variable as a global variable destroys the locality of the code.
  + Since global variables can be modified or read in a wide range of places, it can easily break locality and introduces bugs.

**Lecture 6: Objects**

Part 1: Tuples

* Compound Data Types
  + Have seen a sampling of different classes of algorithms.
    - Exhaustive enumeration
    - Guess and check
    - Bisection
    - Divide and conquer
  + All have been applied so far to simple data types.
    - Numbers
    - Strings
  + There are several compound data types.
    - Tuples
    - Lists
    - Dictionaries
* Tuples
  + A tuple is an ordered sequence of elements (similar to strings).

t1 = (1, ‘two’, 3)

print(t1)

* + Elements can be more than just characters.

t2 = (t1, ‘four’)

print(t2)

* Operations on Tuples
  + Concatenation – when different tuples are added together, it returns the elements of the tuples in the order they are listed in.

t1 = (1, ‘two’, 3)

t2 = (t1, ‘four’)

print(t1 + t2)

* + Indexing – we can access a tuple and find a certain element within it.

print((t1 + t2)[3])

* + Slicing – we can take portions of tuples or lists of elements, just as string slicing would take a list of characters.

print((t1 + t2)[2:5])

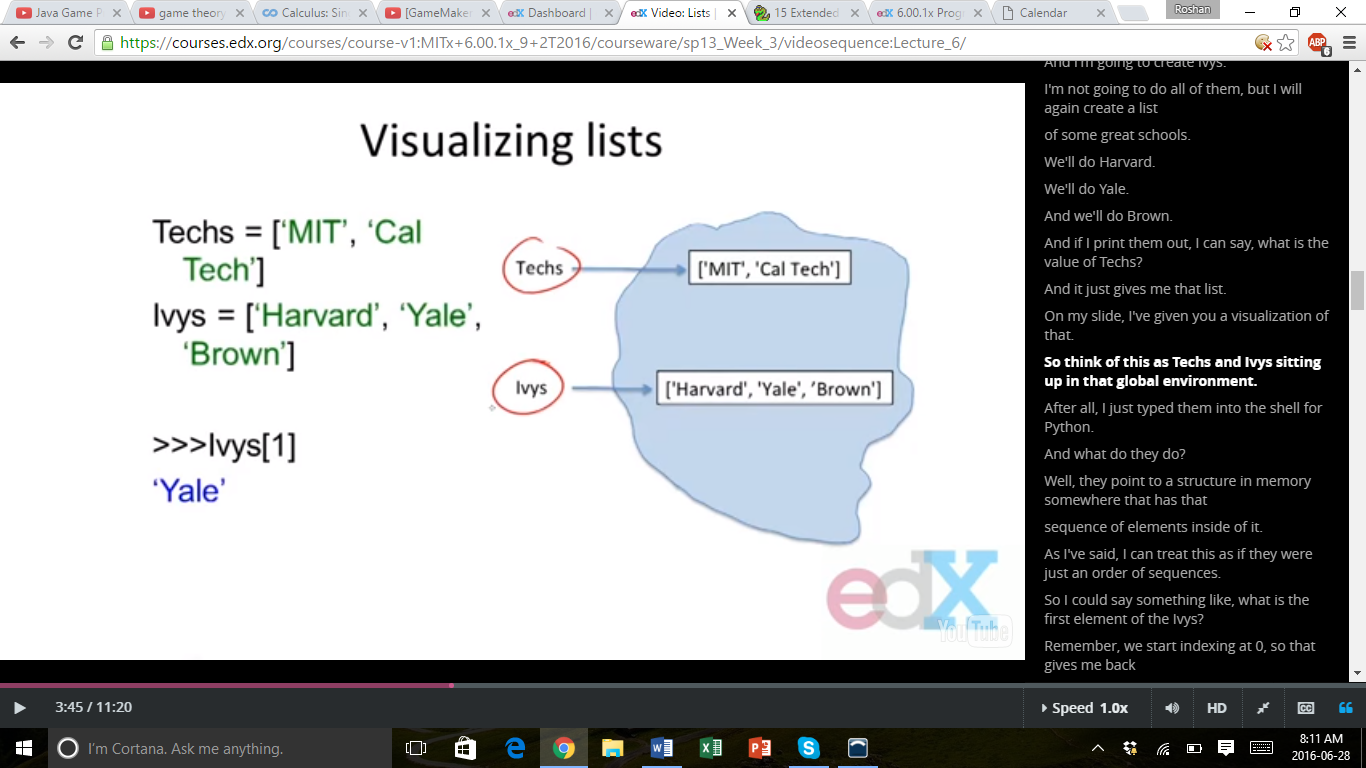
* + Singleton – to only have one value stored in a tuple, there is a special way to do it so that the tuple is not confused with other data types.

t3 = (‘five’,)

print(t1 + t2 + t3)

* Manipulating Tuples
  + Can iterate over tuples just as we can iterate over strings.

Part 2: Lists

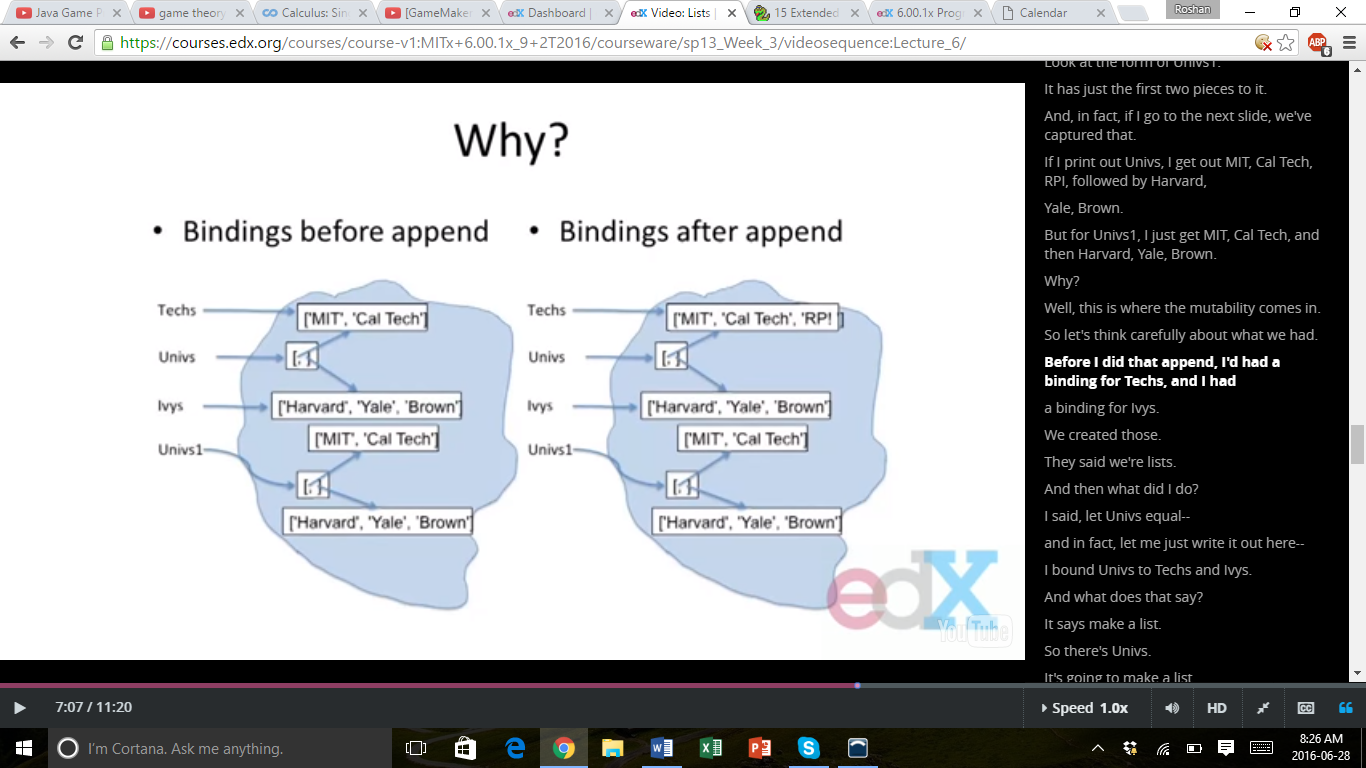
* Lists
  + Look a lot like tuples
    - Ordered sequence of values, each identified by an index.
    - Use [1, 2, 3] instead of (1, 2, 3).
    - Singletons are now just [4] instead of (4,).
  + BIG DIFFERENCE
    - Lists are mutable!
    - While tuple, int, float, str are all immutable, lists are mutable.
    - So lists can be modified after they have been created!
* Why Should This Matter?
  + Some data objects we want to treat as fixed.
    - Can create new versions of them.
    - Can bind variable names to them.
    - But don’t want to change them.
    - Generally valuable when these data objects will be referenced frequently but elements don’t change.
  + Some data objects may want to support modifications to elements, either for efficiency or because these elements are prone to change.
  + Mutable structures are much more prone to bugs in use, but provide great flexibility.
* Visualizing Lists
  + Techs = [‘MIT’, ‘Cal Tech’]
  + Ivys = [‘Harvard’, ‘Yale’, ‘Brown’]
  + >>> Ivys[1]
  + ‘Yale’
* Structures of Lists
  + Consider

Univs = [Techs, Ivys]

Univs1 = [[‘MIT’, ‘Cal Tech’], [‘Harvard’, ‘Yale’, ‘Brown’]]

* + Are These the Same Thing?
    - They print the same thing
    - But let’s try adding something to one of these.
* Mutability of Lists
  + Let’s evaluate

Techs.append(‘RPI’)

* + Append is a method (hence the dot notation) that has a **side effect**.
    - It doesn’t create a new list, it mutates the existing one to add a new element to the end.
  + So if we print Univs and Univs1, we get different results.
  + Univs will contain ‘RPI’, but Univs1 won’t.
* Why?
  + Bindings before append
  + Bindings after append
* Observations
  + Elements of Univs are not copies of the lists to which Techs and Ivys are bound, but are the lists themselves.
  + This effect is called **aliasing**:
    - There are two distinct paths to a data object.
* Through the variable Techs
* Through the first element of the list object to which Univs is bound.
  + - Can mutate object through either path, but effect will be visible through both.
    - Convenient but **treacherous**.

Part 3: Operations on Lists

* Operation on Lists
  + Iteration
  + Combining Lists
    - Using .append() 🡪 mutating the same list
    - Using the + operator 🡪 creates a new list
  + Removing Elements
* Why Removing Elements is Not Working
  + Inside for loop, Python keeps track of where it is in the list using an internal counter.
  + When we mutate a list, we change its length, but Python doesn’t update its internal counter.
  + Thus, it is better to clone over the list rather than iterating over it.

Part 4: Functions as Objects

* Functions as Objects
  + Functions are **first class objects**:
    - They have types
    - They can be elements of data structures like lists
    - They can appear in expressions
* As part of an assignment statement
* As an argument to a function
  + Particularly useful to use functions as arguments when coupled with lists.
    - AKA **higher order programming**
* Example
  + The examples are stored in the Python files.
* Generalizations of Higher Order Functions
  + Python provides some higher order procedures (HOPs), like map()
  + In its simplest form – map() takes a unary function and a collection of suitable arguments for that function
    - map(abs, [1, -2, 3, -4]) 🡪 [1, 2, 3, 4]
  + General form – an n-ary (function that takes n arguments) and n collections of arguments
    - L1 = [1, 28, 36]
    - L2 = [2, 57, 9]
    - map(min, L1, L2) 🡪 [min(L1[0], L2[0]), min(L1[1], L2[1]), min(L1[2], L2[2])]

🡪 [1, 28, 9]

Part 5: Dictionaries

* Dictionaries
  + Dictionary (dict) is a generalization of lists, but now, the indices (ways to get to elements of the dictionary), don’t have to be integers – can be values of any immutable type.
  + Refer to indices as keys, since they are arbitrary in form.
  + A dictionary (dict) is a collection of < key, value > pairs.
  + Syntax
    - monthNumbers = {‘Jan’:1, ‘Feb’:2, ‘Mar’:3, 1:‘Jan’, 2:‘Feb’, 3:‘Mar’}
* We Access Dictionaries Using Keys
  + Entries in a dictionary are unordered, and can only be accessed by a key, not an index.
* Operation on Dictionaries
  + Insertion

monthNumbers[‘Apr’] = 4

monthNumbers[4] = ‘Apr’

* + Iteration

collect = []

for e in monthNumbers:

collect.append(e)

collect = monthNumbers.keys()

* Keys Can Be Complex
  + Here is an example where the keys are actually tuples themselves.

myDict = {(1, 2):'twelve', (1, 3):'thirteen'}

print(myDict[(1, 2)])

* + Note that keys must be immutable, so you have to use a tuple, not a list.
  + We will return to dictionaries and their methods later on.