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

**WEEK 6**

**Lecture 11: Classes**

Part 1: Classes: User-Defined Types

* Objects
  + Python supports many different kinds of data: integers (1234), floats (3.14159), strings (“Hello”), etc.
  + These are the primitive data types, but there are compound data types as well, like lists ([1, 2, 3, 5, 7, 11, 13]) and dictionaries ({“CA”:“California”, “MA”:“Massachusetts”}.
  + Each of the above is an **object**. Objects have specific properties:
    - A type (a particular object is said to be an **instance** of a type, and the type also determines the kinds of operations that can be performed on it, or **behaviour**)
    - An internal data representation (primitive or composite; how do we put the data together in a package, and how do we access that object)
    - A set of procedures for interaction with the object
* Example: [1, 2, 3, 4]
  + Type: list
  + Internal data representation
    - int length L, an object array of size S >= L, or
    - A linked list of individual cells
* < data, pointer to next cell >
  + - Internal representation is private – users of the objects should not rely on particular details of the implementation. Correct behaviour may be compromised if you manipulate the internal representation directly.
    - Procedures for manipulating lists
* L[i], L[i:j], L[i, j, k], +, \*
* len(), min(), max(), del L[i]
* L.append(…), L.extend(…), L.count(…), L.index(…), L.insert(…), L.pop(…), L.remove(…), L.reverse(…), L.sort(…)
* Object-Oriented Programming (OOP)
  + Everything is considered an **object** and has a **type**
  + Those objects are a data abstraction that encapsulate:
    - Internal representation of the data
    - **Interface** for interacting with the object
* Defines the behaviours, hides implementations of object
* Attributes: data, methods (procedures)
  + - Some languages have support for “data hiding”, which prevents access to private attributes. Python does not – one is expected to play by the rules!
  + One can:
    - Create new instances of the objects (explicitly or using literals)
    - Destroy objects
* Explicitly use del or just “forget” about them
* Python system will reclaim destroyed or inaccessible objects – called “garbage collection”
* Advantages of OOP
  + Divide-and-conquer development
    - Implement and test the behaviour of each class separately
    - Increased modularity reduces the complexity
  + Classes make it easy to reuse code
    - Many Python modules define new classes
    - Each class has a separate environment (no collision on function names)
    - Inheritance allows subclasses to redefine or extend a selected subset of superclass’ behaviour

Part 2: A Class Example

* Defining New Types
  + In Python, the class statement is used to define a new type

class Coordinate(object):

# define attribute here

* + As with def, indentation is used to indicate which statements are part of the class definition.
  + Classes can inherit attributes from other classes, in this case, Coordinate is said to be a **subclass** of object, and object is a **superclass** of Coordinate. One can override an inherited attribute with a new definition in the class statement.
* Creating an Instance
  + Usually when creating an instance of a type, we will want to provide some initial values for the internal data. To do this, an \_\_init\_\_ method needs to be defined.

class Coordinate(object):

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

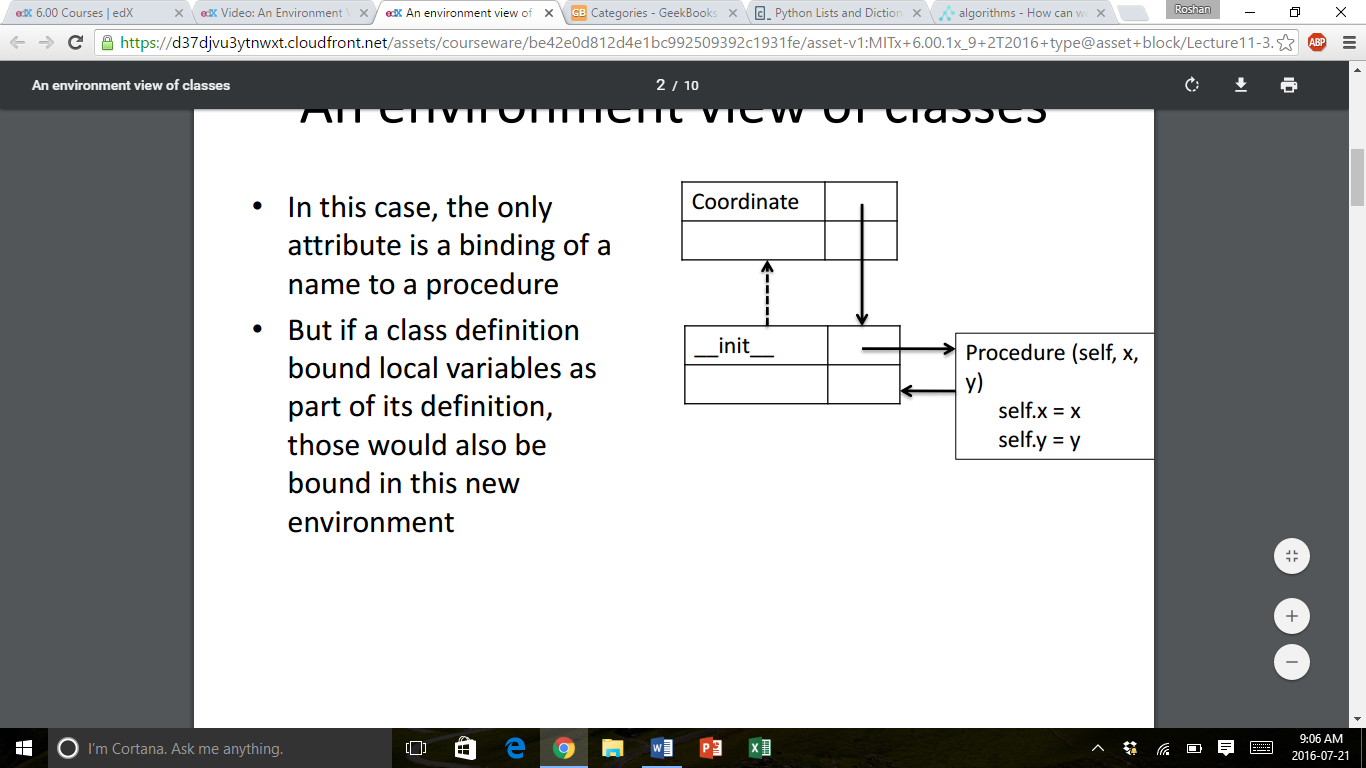
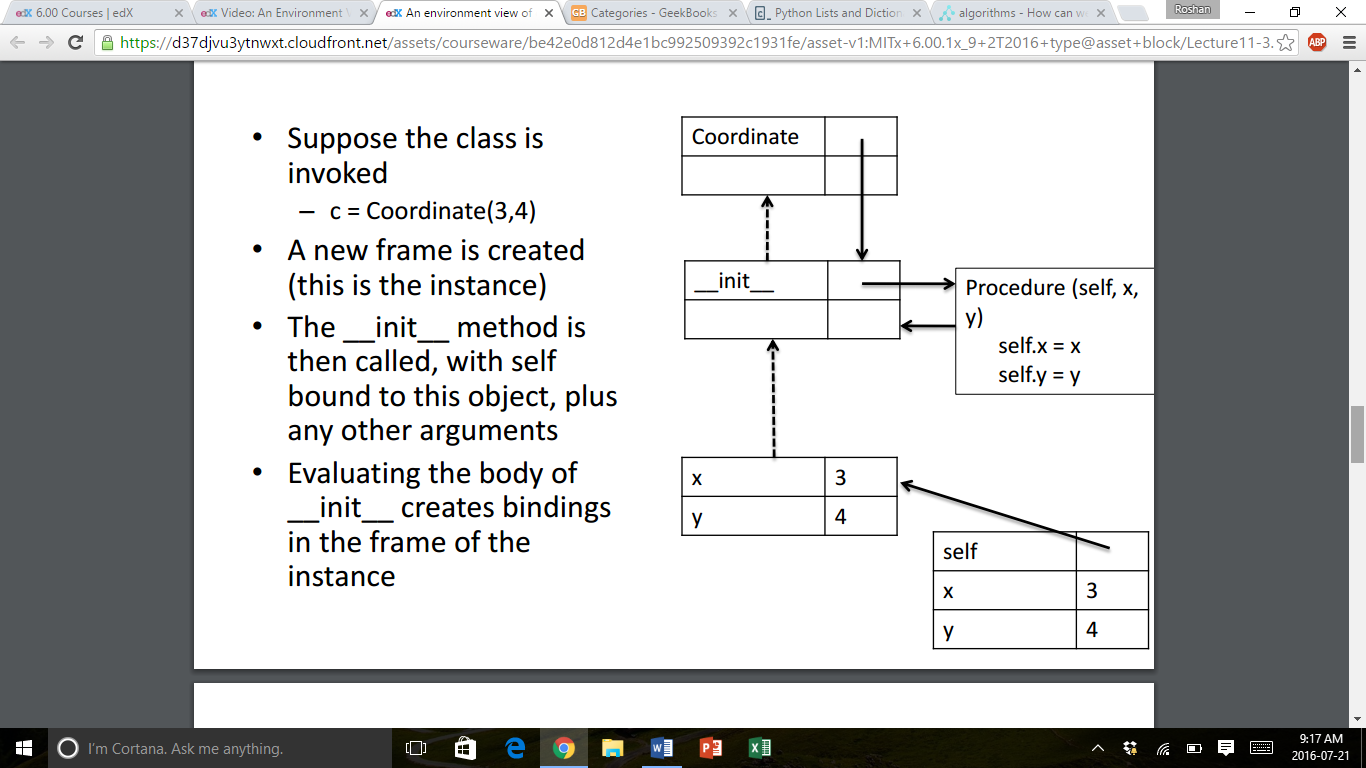
c = Coordinate(3, 4)

origin = Coordinate(0, 0)

print c.x, origin.x

* + A method is another name for a procedure attribute, or a procedure that “belongs” to this class.
  + When calling a method of an object, Python always passes the object as the first argument. By convention, we use self as the name of the first argument of methods.
  + The “.” operator is used to access an attribute of an object. So the \_\_init\_\_ method above is defining two attributes for the Coordinate object: x and y.
  + When accessing an attribute of an instance, start by looking within the class definition, then move up to the definition of a superclass, then move to the global environment.
  + Data attributes of an instance are often call instance variables.
  + The expression classname(values) creates a new object of type classname and then calls its \_\_init\_\_ method with the new object and values as the arguments.
  + When the method is finished executing, Python returns the initialized object as the value.
  + Note that you don’t have to provide the argument for self, because Python does this automatically.

Part 3: An Environment View of Classes

* An Environment View of Classes
  + Class definition creates a binding of class name in global environment to a new frame or environment.
  + That frame contains any attribute bindings, either variables or local procedures.
  + That frame also knows the parent environment from which it can inherit attributes.
  + In this case, the only attribute of the class is a binding of a name to a procedure.
  + But if a class definition bound local variables as part of its definition, those would also be bound as attributes in this new environment.
  + We can access parts of a class using Coordinate.\_\_init\_\_.
  + Python interprets this by finding the binding for the first expression (which is a frame), and then using the standard rules to lookup the value of the next part of the expression in that frame.
  + Now suppose that the class is invoked via an instruction c = Coordinate(3, 4). A new frame is created (this is an instance of a class).
  + The \_\_init\_\_ method is then called, with self bound to this object, plus any other arguments.
  + The instance of the class knows about the frame in which \_\_init\_\_ was created.
  + Evaluating the body of \_\_init\_\_ creates bindings in the frame of the instance.
  + Finally, the frame created by the class call is returned, and bound in the global environment to c.
  + Given such bindings, calls to attributes are easily found. For instance, c.x will return 3 because c points to a frame, and within that frame x is bound locally.
  + Note that c also has access to any binding in the chain of environments. For example, the line c.\_\_init\_\_(5, 6) will change the bindings for x and y within c.

Part 4: Adding Methods to a Class

* Print the Representation of an Object
  + Left to its own devices, Python uses a unique but uninformative print representation for an an object.

>>> print c

< \_\_main\_\_.Coordinate object at 0x7fa918510488 >

* + One can define a \_\_str\_\_ method for a class, which Python will call when it needs a string to print. This method will be called with the object as the first argument and should return a str.

class Coordinate(object):

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "<" + str(self.x) + ", " + str(self.y) + ">"

* Type of an Object
  + We can ask for the type of an object.

>>> print type(c)

<class \_\_main\_\_.Coordinate>

* + This makes sense, because of the following.

>>> print Coordinate, type(Coordinate)

<class \_\_main\_\_.Coordinate> <type ‘type’>

* + Use isinstance() to check if an object is of type Coordinate.

>>> print isinstance(c, Coordinate)

True

* Adding Other Methods
  + We can also add our own methods, in addition to the built in methods.

class Coordinate(object):

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_str\_\_(self):

return "<" + str(self.x) + ", " + str(self.y) + ">"

def distance(self, other):

return math.sqrt((self.x - other.x) \*\* 2 + (self.y - other.y) \*\* 2)

* + Then, we can call the method, and it should return a value.

>>> print Coordinate.distance(c, origin)

>>> print c.distance(origin)

5.0

5.0

Part 5: A Set of Integers

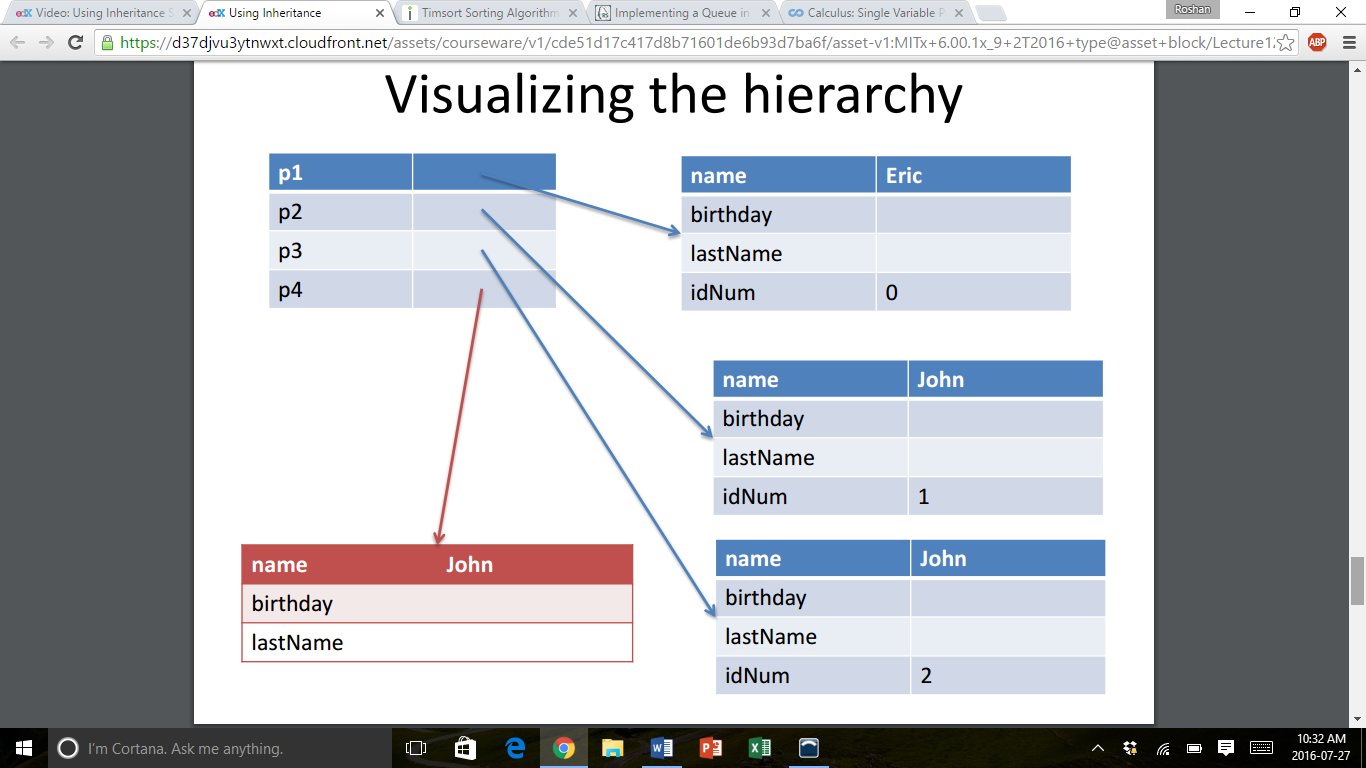
* Example: A Set of Integers
  + Create a new type to represent a set (or collection) of integers.
    - Initially, the set will be empty.
    - A particular integer only appears once in a set.
* This constraint, called a **representational invariant**, is enforced by the code in the methods.
  + Internal Data Representation
    - Use a list to remember the elements of the set.
  + Interface
    - insert(e) – insert integer e into the set if it is not already in the set
    - member(e) – return True if integer e is in set, False otherwise
    - remove(e) – remove integer e from set, error if not present

**Lecture 12: Object-Oriented Programming**

Part 1: Inheritance

* Using Inheritance
  + Let’s build an application that organizes information about people.
    - Person: name, birthday
* Get last name
* Sort by last name
* Get age
* How Does the .sort() Method Work?
  + Python uses the *timsort* algorithm for sorting sequences – a highly optimized combination of merge and insertion sorts that has very good average case performance.
  + The only knowledge needed about the objects being sorted is the result of “less than” comparison between two objects.
  + Python interpreter translates obj1 < obj2 into a method to call on obj1 🡪 obj1.\_\_lt\_\_(obj2).
  + To enable sort operations on instances of a class, implement the \_\_lt\_\_ special method.

Part 2: Using Inheritance Subclasses to Extend Behaviour

* Using Inheritance
  + Let’s build an application that organizes information about people.
    - Person: name, birthday
* Get last name
* Sort by last name
* Get age
  + - MITPerson: Person + ID Number
* Assign ID numbers in sequence
* Get ID number
* Sort by ID number
* Examples of Using this Hierarchy of Classes

p1 = MITPerson(‘Eric’)

p2 = MITPerson(‘John)

p3 = MITPerson(‘John)

p4 = Person(‘John)

* Visualizing the Hierarchy
* Suppose We Want to Compare Objects
  + Note that MITPerson has its own \_\_lt\_\_ method. This method “shadows” the Person \_\_lt\_\_ method, meaning that if we compare an MITPerson object, since its environment inherits from the MITPerson class environment, Python will see this version of the \_\_lt\_\_ method, not the Person version.
  + Thus, p1 < p2 will be converted into p1.\_\_lt\_\_(p2), which applies the method associated with the type of p1 onto p2 using the MITPerson version of the method.
* Who Inherits from Whom?
  + Why does p4 < p1 work, but p1 < p4 doesn’t?
    - The expression p4 < p1 is equivalent to p4.\_\_lt\_\_(p1), which means that we use the \_\_lt\_\_ method associated with type(p4), or the Person class (the one that compares based on names). This is alright because both p1 and p4 have properties of the Person class, which means that they can both use Person class methods.
    - The expression p1 < p4 is equivalent to p1.\_\_lt\_\_(p4), which means that we use the \_\_lt\_\_ method associated with type(p1), or the MITPerson class (the one that compares based on IDNumber). Since p4 is a Person, it does not have an IDNumber, and thus an error is raised.

Part 3: Using Inheritance: Designing a Class Hierarchy

* Using Inheritance
  + Let’s build an application that organizes information about people.
    - Person: name, birthday
* Get last name
* Sort by last name
* Get age
  + - MITPerson: Person + ID Number
* Assign ID numbers in sequence
* Get ID number
* Sort by ID number
  + - Students: several types, all MITPerson
* Undergraduate student: has class year
* Graduate student
* Creating Undergraduates and Graduates
  + Below, we have created new classes for UnderGraduate, Graduate, and isStudent.

class UnderGraduate(MITPerson):

def \_\_init\_\_(self, name, classYear):

MITPerson.\_\_init\_\_(self, name)

self.year = classYear

def getClass(self):

return self.year

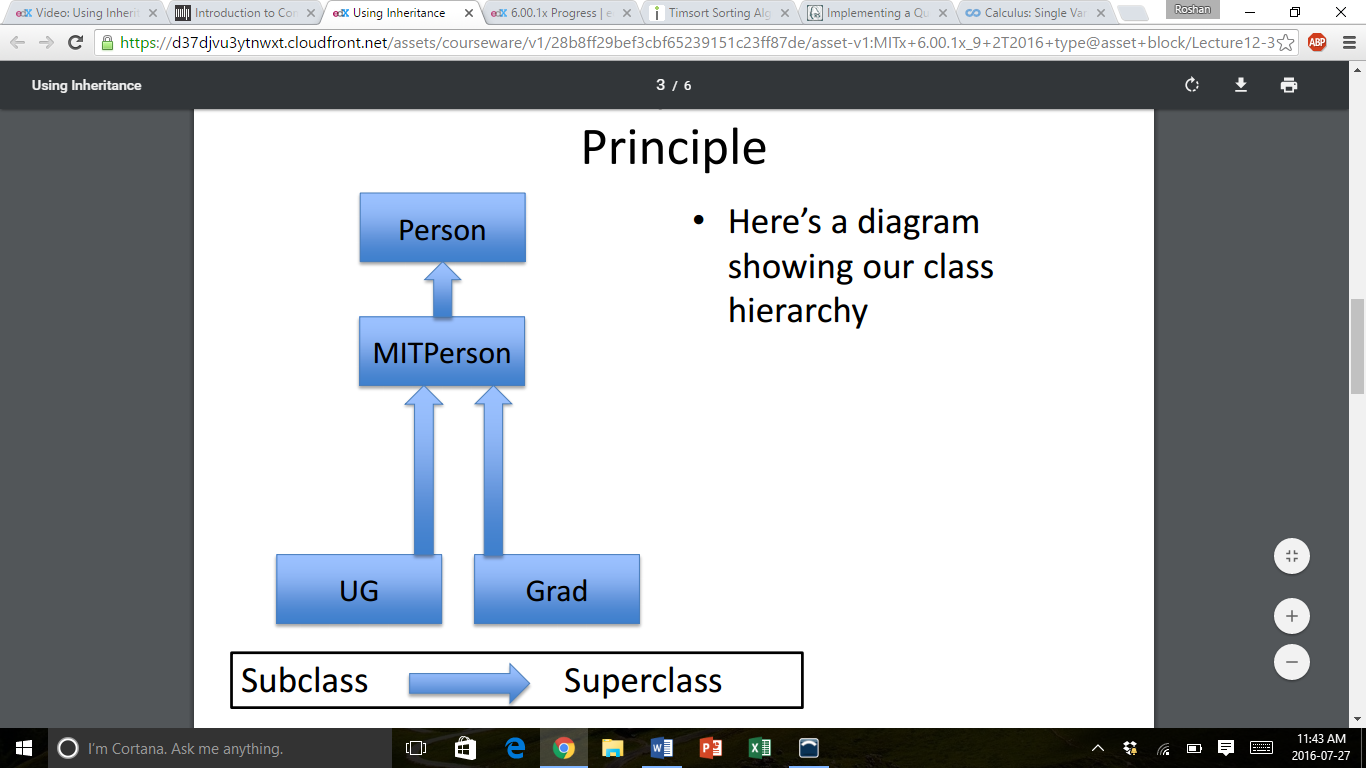
class Graduate(MITPerson):

pass

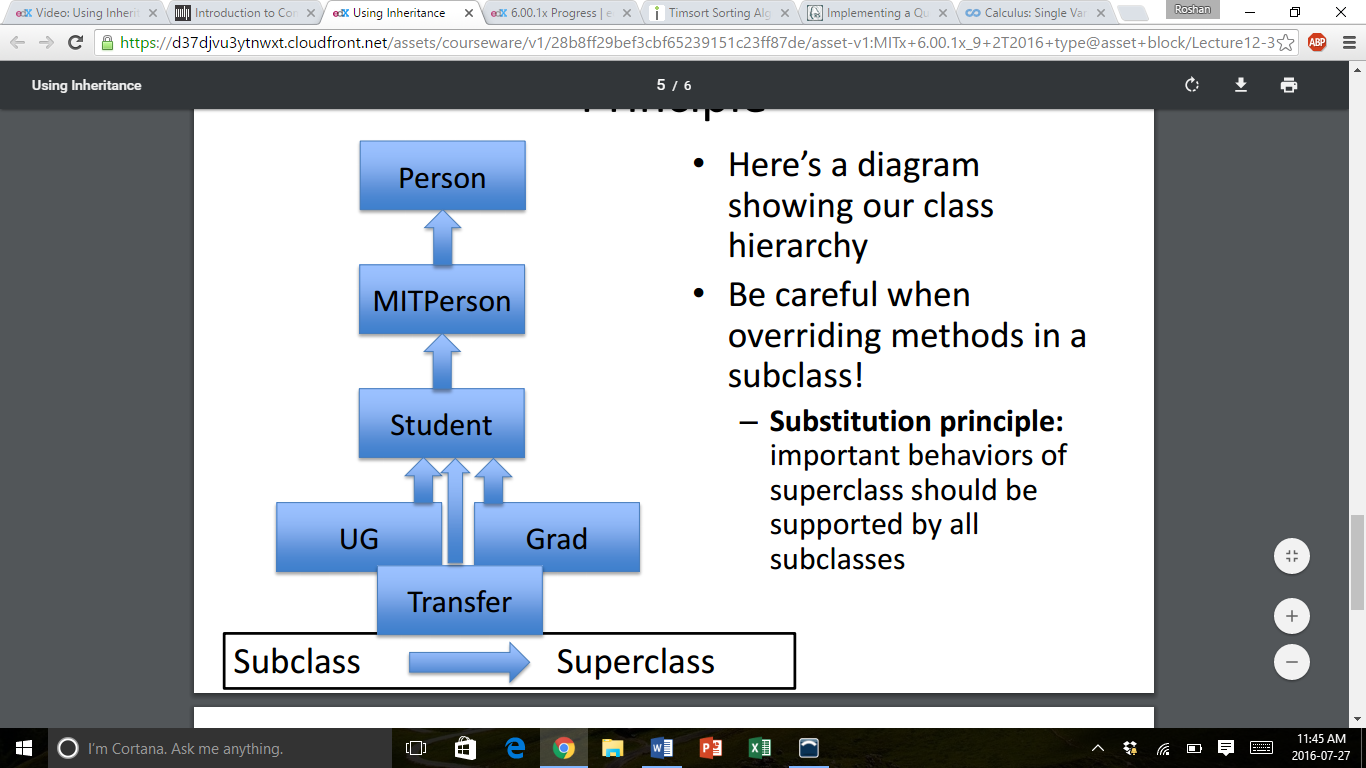
def isStudent(obj):

return isinstance(obj, UnderGraduate) or isinstance(obj, Graduate)

* + You may be wondering, from the code above, why the class Graduate was created and why it has a special word pass listed under it.
  + This simply means that the Graduate class inherits from the MITPerson class, but has no unique methods of its own.
* Class Hierarchy and Substitution Principle
  + Here’s a diagram showing our class hierarchy



* + Now, we are going to rethink our hierarchy because it does not support other classes of students, like transfer students.



* + Be careful when overriding methods in a subclass!
    - **Substitution Principle:** Important behaviours of the superclass should be supported by all the subclasses.
* Cleaning Up the Hierarchy
  + It is better to create a superclass that covers all the types of students, because that then an MIT student is well defined. Similarly, we could create other classes called Faculty or Staff to classify other types of MIT people.
  + In general, creating a class in the hierarchy that captures common behaviours of subclasses allows us to concentrate methods in a single place and lets us think about subclasses as a whole.

Part 4: Example: A Gradebook

* Example Class: A Gradebook
  + Create a class that includes instances of other classes.
  + Concept Outline:
    - Build a data structure that can hold grades for students.
    - Gather together data and procedures for dealing with them in a single structure, so that users can manipulate data without having to know the internal details.
* Using this Example
  + I could list all the students using:

for s in sixHundred.allStudents()

print s

* + This prints out the list of student names sorted by IDNumber. So, why don’t we just do the following?

for s in sixHundred.students:

print s

* + This violates the data hiding aspect of an object, and exposes the internal representation.
    - If I were to change how I want to represent a grade book, I should only need to change the methods within that object, not the external procedures that use it.
* Comments on the Example
  + Nicely separates the collection of data from the use of the data.
  + Access is only through the methods associated with the gradebook object.
  + However, the current version of this gradebook is relatively inefficient – to get a list of all the students, I create a copy of the internal list.
    - This lets me manipulate the list without altering the internal structure.
    - However, it is expensive to do so in a MOOC with ~100,000 students.

Part 5: Generators

* Generators
  + Any procedure or method with a yield statement is called a **generator**.

def genTest():

yield 1

yield 2

* + genTest 🡪 < generator object genTest at 0x201b878 >
  + Generators have a next() method that starts/resumes execution of the procedure. Inside the generator:
    - yield suspends execution and returns a value
    - Returning from a generator raises a StopIteration exception.
* Using a Generator
  + Let’s say we create a generator and implement it to do something.

foo = genTest()

foo.next() # returns 1

* + Execution will proceed in the body of foo, until it reaches the first yield statement, then returns the value associated with that statement.

foo.next() # returns 2

* + As the next yield statement says yield 2, the value 2 is returned at this point. A StopIteration exception will be raised the next time foo.next() is called.
* Using Generators
  + We can use a generator inside a looping structure, as it will continue until it gets to a StopIteration exception.

>>> for n in genTest():

print n

1

2

* A Fancier Example
  + Evaluating fib = genFib() creates a generator object.

def genFib():

fibn\_1 = 0

fibn\_2 = 1

while True:

next = fibn\_1 + fibn\_2

yield next

fibn\_2 = fibn\_1

fibn\_1 = next

* + Calling fib.next() will return the first Fibonacci number, and subsequent calls will generate each number in the sequence.
* Why Generators
  + A generator separates the concept of computing a very long sequence of objects from the actual process of computing them explicitly.
  + This allows one to generate each new object as needed as part of another computation (opposed to computing a very long sequence, only to throw most of it away while you do something on an element, then repeating the process).
* Fix the Grades Class
  + Now, instead of returning a copy of the list of students, we will simply return each element or student in order, as the user needs each student.

def allStudents(self):

if not self.isSorted:

self.students.sort()

self.isSorted = True

for student in self.students:

yield student