ProgrammingParadigms-Lecture25

Instructor (Jerry Cain):Hey, we're on. Look at that. How'd that happen? I have one handout for you today. Actually, I don't expect to get to the material today, but I don't see the disadvantage in giving it to you just in case you're bored over the weekend and have nothing else to do, you can read about XML and Python and networking.

What I want to do today is I want to focus a little bit more on dictionaries and show you an example where dictionaries actually contribute to a meaningful program. I'm gonna re-write RSG from a sign of one in Python. We're gonna do it in laughably little space. I'm gonna be able to illustrate a very small program that has the imperative, object oriented, and the functional paradigm all in it. Okay.

I'll illustrate lamdas that they actually exist in Python and it's really just a matter of coming up – knowing the different syntax, the Python way of doing what you did in Scheme a week and a half ago or just until I guess last night or two nights ago rather. I also want to talk a little about objects and classes in Python, and talk about the object model.

It's very interesting to look at it because when you learn about objects and object orientation, and in a class like 107 where you look under the hood to figure out how these things are implemented, you can very easily get the impression like I did for years to be honest, that all objects, and everything is implemented the exact same way in all languages, and that's just not the case.

So I will talk a little about that today. With regard to dictionaries, I mentioned on, I guess it was Monday – what's today, Wednesday, sorry, Wednesday – that dictionaries are a central data structure in Python. They are basically a very simple syntax layer over what is very well understood to be a hash table.

They keys can be anything that are hashable. They don't even have to be heterogeneous. You can have some integers and some strings as your keys, and the things that they map to, the values, they can be any type whatsoever, and they don't have to be consistently the same type over the dictionary.

It is very amorphous, very heterogeneous, just like Scheme is with its data structures. You can put anything on a list, you can mix up data types you can do the same thing with Python structures. Okay.

What I want to do is I want to emphasize the fact that as opposed to C and C++ where you don't have a clear way to texturally initialize a data structure – like think about a C++ map or a C vector from Assignment 3 looks like after you've populated it with say 20 values. It actually depends on the binary representation of the data. Okay. And there's no way to say C++ vector of int is equal to 12345. Okay.

You don't have object literals in C or C++ or for that matter Java either. Okay. You can have array literals in Java in a way that you can't in C and C++, but the real sophisticated containers, there's no real way to actually specify container constants. That's not the case with Python.

The first of three or four pieces of the RSG example – I won't write it out in full grammar, but what I'm basically doing here is I'm coming up with a file format for a random sentence generator grammar. You are familiar with this. I know it's been eight weeks ago, but you remember that it involved some angle brackets for non-terminals and things that were terminals were just normal strings. If I do this grammar I'm actually writing Python code.

Now I will make it clearly a dictionary. I'm going to have grammar that is just a gesture to what the full grammar could be like, but my grammar is going to be expressed as a dictionary literal where the keys are structured this way, and they map to arrays which are called lists in Python, or at least the way I use them I call them lists, lists of expansions. And for a start, I'm only gonna have one expansion.

So here's the list. It happens to be a list of a length of one, and that item is itself a list, and I'm gonna do it this way. It's not a very sexy sentence it's just a placeholder. This object is here. If I had a second or third option, they would have been common delimited list. I'm not gonna have that for this one. I'm just gonna have one option just so you see the structure in a small example, but I will let object map to a couple of options.

It can map to the standalone thing like computer. I'll let it map to this par, this assignment. I'm not interested in grammatically meaningful sentences; I'm just interested in getting the grammar on the board. I think this is complicated enough. You look at this and you may think that there's really no other sensible way to do this, but realize that kind of like you have in Scheme that you are allowed to represent the data in serialized object form. Okay.

And that this actually gets grammar to be an in-memory dictionary where it has two keys and each one maps to a list of lists where this one happens to be a list of length one and this happens to be a list of length three. Does that make sense? Okay, I'm being fastidious about the white spacing just because I'm careful about it in the handout as well.

What I want to do is I basically want to conceptually expand the start symbol to generate a random sentence, and knowing that it's gonna select this, I'm gonna want to expand this as a terminal, I'm gonna wanna expand what's in there, I'm gonna wanna expand what's in there. This and this are actually simple, it's just supposed to print. This is supposed to recursively do the same thing as if object is the start symbol, and it should expand to whatever it's supposed to expand to. Okay. Does that make sense?

Forget about the libraries, let me just invent function names. They happen to be the real function names. We're generating random numbers and names and things like that. Okay. If I assume that grammar is global variable. Just assume it's a global variable. We'll

correct that in a little bit. Just because we're dealing with Python doesn't mean we should just be lazy about globals, but just to illustrate all three paradigms at once right now. I wanna define this function def expand and I'm just gonna call it symbol.

Now you know I do one of two different things depending on whether or not the symbol which I'm just gonna assume is being cast in as either a terminal or non-terminal, and if it's a non-terminal then it's definitely in my grammar. If it's the case that symbol starts with this string which is incidentally the best way to represent a character. If that's the case then what I want to do is recognize that I don't want to print to the non-terminal, I want to select one of its definitions from its definition set randomly, and then basically expand, in order, all of the terms that make up the list. Okay. Does that make sense to people?

So all I'm gonna do is this. Definitions is equal to the global grammar of this thing that I'm assuming really in the grammar and no semicolon. Okay. That brings in this or this entire thing. Now there's a built-in which I'm just gonna do this. I'll just say expansion is equal to – there's a built-in function called choice which takes a list – it actually takes either an integer or a list. I'm gonna give it a list. Choice is basically a get random function in Python as a built-in. If it's given an integer it gives you a number between zero and that integer exclusive just with random probability equal distribution.

If it gives you a list it selects anyone of the elements from the list with equal probability. That's kind of what I want. I want it to choose this, this or this with probability one third. I want it to choose that with probability one. That's exactly what this line is gonna do right here. Okay.

Now on Assignment 1 when you did it using C++, you did not necessarily use recursion. You probably used the iterator to just visit everything, but if you're thinking in terms of Scheme and how it dealt with lists, we didn't use iteration there. We didn't know about mapping technically when we did Assignment 1, but on behalf of something like this, if I understand that expanding this space will just print this space, and expanding space is here period we'll just print to that after I get the else clause up here. Okay.

And I can also recursively expand that. I can do this map, the expand function, over what's locally recognized as the expansion so far. Okay. Does that make sense? So if this is capable of being levied against start with an angle bracket, it's certainly capable of being levied against that, and I'm just gonna implement the else case where it isn't a non-terminal to just do sys out write.

That's basically the equivalent of print out for [inaudible] and less than without any new lines, it's a raw character printer, where I will just print out the symbol. Ultimately every single non-terminal becomes a terminal or a series of terminals to be printed, so this is what's going to be doing the printing. Okay. Does that make sense?

Object orientation, clear imperative style, and I am electing to go with this functional Scheme-ish approach of actually looking at everything that's making up the expansion list

as peers and having them all publish themselves whether there's recursion involved or not. Okay. Does that make sense to people?

As far as how to do this from the get-go, as a global function, I could actually call the seed function that just basically is like randomized from the CS106 library set. We're just shaking up the dice a bit so that we really general pseudo random numbers, and don't start from the same number every time we run it. And then I might do something like this expand – now the handout version is a little bit cleaner about how it actually paginates the answers and things like that, but that's basically the gist of it. Okay. Does that make sense to people? Yes, no? Okay.

So what I want to do now is I want to bring the computer up. I'm gonna talk about classes, and how to define classes in a second, but there's gonna be – I wanna emphasize some things that confused people last autumn when they did this DNA assignment, and it was more or less because I just didn't mention these things specifically last autumn.

It was probably from me teaching Python for the first time last autumn when I just didn't know what the problems to expect were going to be. This is in the handout. There's a clear output. You can actually run this. This is up in the CS107 Website underneath WWW you can go and actually find this if you want to and run it. Okay. Any questions about this before I divorce myself from this example? Yep?

Student: So to do recursion in this language then you have to call map?

Instructor (Jerry Cain):Oh absolutely no. I just elected to use map. If I wanted to write Fibonacci I could have a definition of Fibonacci and call it Fibonacci. If I wanted to, I could have done a four loop from 4I in range of zero though length of expansion, and I could have just called expand inside a four loop. I don't have to use, I mean, we can always make a direct recursive call in virtually any language that I know of. Okay.

I just elected to go with the map approach here because I think it is cleaner, and it is kind of the functional way of doing things. Not necessarily that that's the goal, but I'm just illustrating all three paradigms in the same example. Okay. Question in the back? Student:

Could you repeat what C does again.

Instructor (Jerry Cain):Oh, C right here. Computers even though they – when they generate random numbers, they do so deterministically. Okay. So they're not really random, but there as seemingly random as a computer that's completely deterministic in its operation can be. Normally what happens, every time you launch the Python interpreter it uses zero or something related to zero to generate the first random number, and then it algorithmically uses prime number theory to generate the next sequence in the random number sequence.

If you always start from zero then you get the first same random number every time you run something, and you get the same second number every time you run something. You get the same sequence, and it kind of breaks the fallacy – I'm sorry, it's exposes the fallacy that these aren't really random numbers. That's actually not a bad thing. A lot of times I recommend getting rid of that when you're testing so you do get the same output every single time, and so you can really debut a lot more easily.

But C just says, you know what, just populate – just set it so the first random number that's going to be generated is related not to zero, but usually this is in the case of C, and I'm sure it's the same in the case of Python that it usually takes the number of milliseconds or the number of seconds since the computer was turned on, and relates the first random number to that instead. This is basically the equivalent of s rand from C if you're familiar with that function. Okay. Yeah?

Student:[Inaudible]

Instructor (Jerry Cain):Yeah, they're right here. This seed and this choice, they're both from the same module, and technically if I'm complete about everything up top, I should do this. I should do import sys – that's short for system obviously. It's because of that that I'm allowed to go in and publish the console. And I could either import the random library and then do random.choice and random.seed or I could use the sexier way of doing it which is from random specifically import choice and seed.

And that's another way of doing it. Okay. It's just boilerplate, more boilerplate learning the language, things like that. Actually before I leave this example, I wanna do something. I wouldn't write it this way, I'm actually fine with the idea of grammar as a global here because it's just a little script, and it's really executing this as a statement, and then this as a statement, and then this as a statement. I don’t really think of grammar as a global.

I think of the entire file as a function that it gets executed, but if you wanted to be a purist about this, and you didn't want this right here to be a global variable, you could pass in the grammar like that. There's a little bit of a breakdown because now all of the sudden expand is a binary argument function as opposed to a unary argument function. That doesn't mesh too well with the way I've called map. Map and Scheme actually can deal with multiple lists depending on the arity of the function that's being mapped.

That's not the case with Python. This has to be a unary function. It doesn't have to be a named function. Okay. If I wanted frame the implementation and I wanted to frame this function, in terms of expand, but as a unary function I can do that using the Scheme idea lamdas. Lamdas actually exist in Python as well. What I can do, it's just syntax, I can invent a function right here. I'm not just writing the Scheme code here, I'm really writing Python.

Lambda, I'll call it item for lack of a better word, and use a colon for the same reasons you use a colon every place else. Okay. And I just equate it with expand of item

grammar. I'll abbreviate that. That's the first argument. Do you understand how that's a one-argument function? Just believe the syntax, it is right. Okay.

It is scripted as an anonymous one-argument function whose implementation is framed in terms of that one argument, and this thing that it available as a local variable in the outer scope, and I want to map that over expansion. That's another way to do this, and it really has the functional components of Scheme that are interesting to me, and probably to a lot of people as it is mapping, and it has lamdas and the closures that come with lamdas. Okay. Does that make sense? Okay, good. Okay.

So what I want to do is I want to talk a little bit about the object model from a memory standpoint. I'm gonna talk about this to a couple of degrees. I'm just gonna worry about dictionaries at first. The manner in which dictionaries are passed around is precisely, at least for our purposes, precisely the same as the manner in which objects are passed around in Java. Everything is always passed around by alias or by reference. Okay.

So if I do this, let me just deal with lists because they're easier to draw, and I'll just generalize the dictionaries. If I do this, it doesn't print anything out, but if I just print X, naturally it does this. If I do this, and then I print out Y, it prints out 1, 2, 3. Okay. That shouldn't surprise you at all. Okay. There's some languages where what I'm about to do would actually be different, but if I go ahead and call X, which is a list and I do append, and I append a 4, if I go ahead and print out X, you know for a fact that it's that, and that isn't the least bit surprising.

You've logically updated X. However, what you may not realize is that X was evaluated, and the result of the evaluation was assigned a Y. All X did was it evaluated to the pointer, to the lead note on the list or the lead element in array. So when I go ahead and print this out, I'm gonna get that because I changed Y behind its back when I actually updated X. Does that make sense to people? Yep, go ahead.

Student:[Inaudible]

Instructor (Jerry Cain):I'll get to that in a second. It doesn't happen by default. You have to invoke one of two functions depending on whether it was just a one-level copy or a full deep copy is available to you. This is one thing I did not mention last autumn, and it caused problems. My particular solution to Assignment 8 doesn't use any copying whatsoever. It just shared the same master dictionary throughout the entire implementation, but I should not have assumed that everyone would want to do it exactly the same way.

So you do need the ability, at least initially, until you convince otherwise, to be able to clone dictionaries. I'm gonna talk about that in a second. Let me give you some sophisticated examples. If I set Z equal to the list, let's say 10, 12, 14, and I go out and print Z, it's of course going to be this, but then I do this. I'll do W is equal to, and I'll construct a list that way. Okay. Z and Z evaluates to the list 10, 12, 14. Okay. So when I go ahead and I print out W, not surprisingly, I get this. Okay.

I bet everyone believes that. What you might not recognize is that it does not make any deep copies. You do not transfer full ownership of the Z lists into the list that's owned by W. So if I do this, z.append 17 and print out Z, it's now this, but more interestingly, if I print out W, I get this: 12, 14, 17, 10, 12, 14, 17. These are obviously contrived examples that are very, very small, but it's illustrating – for some reason to me at least it was more mysterious because it felt like it was this higher level language, you didn't necessarily know that objects were backing these things.

It was very easy to wonder whether or not it's a deep clone or a shallow clone, but almost always, at least initially, it just makes a shallow copy of it. Okay. It turns out that the shallow copy is preserved for the lifetime of the data in Python, and it happens to be different in a language called PHP which I use a lot at work where when you pass one dictionary around or one list around, it doesn't actually clone it, but it does actually label it [inaudible] it was copied from something else, so as you change it, it actually does a copy on write, and actually branches off the part that changes so that the two logically look different even though they're sharing memory. Okay.

None of that happens in Python though. Okay. If you want to make a copy, there's two ways you can do it. There is a module called copy. Okay. I suspect it's not written in Python. I'm suspecting it's written in C, but I could import something called copy. There's actually another function I'll talk about in a second. If you want to clone an object, X = 14, 15, 21, and you print out X, you will get 14, 15, 21.

If you do this, this is the same as all the prior examples and you print out Y, you get this, but that shouldn't surprise you. This shouldn't surprise you either. You don't know the syntax, but there's a key word in the language called is, X is Y is basically equality at the pointer level, and because X and Y really are aliasing the same exact physical memory, not only are they logic identical, but they're memory-wise identical as well, you expect this to come back with a true. Okay.

If you want to make a clone of something because you want to make some changes to a data structure without it affecting the original, you can do that. Z is equal to a functioned called copy, it's this copy right there that does it and I want to make a copy of X, not surprisingly Z is 14, 15, 21, but I'm out of room, but if I did Z is X, I would actually get a false back because they're memory independent at least to some degree.

Now copy, you would think because it's called copy, that copy really means decline, do this depth first reversal and make sure every piece of memory that's generated on behalf of Z is independent from that that was accessible from X, that's not the case. This particular copy is what's known as a shallow copy, it only goes one level down. Okay.

So if I had a list of atoms, like I do right here, these really are fully memory independent, but if I were to have lists of lists of lists of lists – does that make sense? The top-level list would be replicated from a memory standpoint, but everything inside would just be a shallow copy. So it's almost like it generates new memory for the top-level array, but that

does a mem copy behind the scenes for everything below that. Does that make sense to people when I say that?

If you really do want a deep copy, then you have to use this strangely named function called deep copy, and this is an interesting example. If I do M is equal to the list 1, 2, 3, I do N is equal to the list M, M, and that's good enough actually. Recognize that I have the list 1, 2, 3 in memory. That's my abbreviated version of showing the link list in memory, and this thing is associated with the variable M. Okay.

The link list that's generated on behalf of N is really this. Does that make sense the way I drew that? Okay. It's the same list that appears in Index 0 and Index 1, but the same list is being pulled in two different scenarios. Okay. If I do this, P = deep copy of M, which is interesting to me, not only does it do a recursive descent clone of everything, but if there are any cycles like there kind of our in this, forget about M, that doesn't interest me anymore. Do you understand why those two arrows point to the same 1? Okay.

The deep copy not only figures out how to make a deep logical clone of the entire thing, but it actually figures out how to preserve the graph structure. Okay. Does that make sense to people when I say graph structure? Yes, no? Okay. So this would be associated with something that was completely independent, but would have the same aliasing internally. Okay. That's actually just a wise thing for it to do so that it can re-hydrate objects from their serializations if you ever wanted to go to that part of the Python language. Okay. Making sense?

Now it is – I'm thinking that you do not need to use these functions for Assignment 8, but whatever is said right here on behalf of lists, also applies to dictionaries. Okay. You use dictionaries and strings quite a bit in the Assignment 8 solution. You don't have to use lists all that much at all. You can if you want to.

I use dictionaries to more or less bundle information that would otherwise be aggregated in a struct, like C and C++ would require it, and I use that to kind of aggregate information that's related to one another. Okay. That's I think a common practice in a lot of these dynamic modern languages. Python, certainly Perl to some degree, I don't know Perl as well, but Python I certainly know that's the case. I also know it's the case with PHP. Okay. So there's that.

What I want to do now, is I want to show you a little bit about objects and classes in Python, okay, and show you how they really are very little more than just dictionaries. Okay. This is gonna be interesting because – it's gonna be interesting to me, and hopefully it's interesting to you as well. Remember when we learned about the object representation? I'm sorry, the activation record layout of a C struct, right.

You have a clear order in which fields are declared. You actually declare the fields ahead of time because there's a compile time element to it so you can bother doing that, and the first field goes at the bottom, the second field goes above that, the third field goes above that, etc. Okay. Objects and structs, classes and structs in C and C++, and actually classes

in Java all adopt that model. All classes do is recognize that structs and classes can be the same thing, they both have data fields.

They don't store the method pointers or anything like that in the struct, they just lay things out according to the same exact formula. When you see something like that, you just assume that every language that's ever going to be invented from that point on is going to use exactly the same model. However, in a language like Scheme or Python, which have no compile time element whatsoever, you don't pre-declare the types ahead of time.

You just add stuff to these dictionaries which is basically Python's answer to the struct or the class. Okay. Does that make sense to people? And so you can't specify an order of fields ahead of time because you're not even specifying the fields ahead of time. You just kind of add things to the dictionary as it is suitable for your algorithm, and if you add X, and then Y, and Z, but in some other execution you add Y and then Z, and then X, it should still logically have the same set of keys, they just happen to not be inserted in the same order.

Python and most modern languages, and to some degree Objective C, that's the Cocoa language for Apple, uses this for storing methods. They actually just put all the fields, and store them as strings in a dictionary. Okay. That's exactly what happens backing for dictionaries, and also for classes in Python which are backed by dictionaries, and we'll get there in a second. Okay. So let me go ahead and just show you and example of a class, and I'll do some tinkering at the command line.

Okay, this is good enough, and let's just focus on this right here. I don't think I can – that's not good. Okay. I'll try to make the font bigger in the meantime. It might be too big, but we'll correct it in a second. That's not bad actually. Is it up there? Now it's huge. Actually that's okay. That's a nice word over there. Let's bring it and make it nice good word. Okay.

That's good enough I think. I'm not worried about the comment that's being clipped off, then I can bring this down, and that's all I'm interested in. Okay. That's great. Actually it's like artwork. Okay. You see the class keyword that should just be immediately obvious to you that it's going to be defining some class in whatever sense Python defines classes.

You see this underscore underscore, and init underscore thing, that just means it's a special method. Not surprisingly it's related to construction. I'll show you how to construct a lexicon object in a second. Remember how in C++, I probably said it this way about a millions times, and now it's a million and one, that it always silently passes the address of the receiving object as the negative one parameter, right.

Well, Python doesn't do that. It's very explicit about passing the address of the relevant container or object to all methods including the constructor. By convention it's called self, it doesn't have to be, but self is a keyword that's borrowed from Objective C, and I think

that's right, yeah, actually it is. Objective C's self, it's basically just Python's equivalent of this right here. Okay.

Because objects are initialized dynamically, there's no compile time element whatsoever, it takes this empty object that really is a default object called object, that's part of the language, it's like java.lang.object, but you just add stuff to it, and what's happening here is that I've created two local variables, one's called in file, and one's called words.

From that point on it initializes fields, but these two lines right here, forget about them being inside a constructor, there just methods, f open happens to be the function that opens a file. Read lines is this built-in thing that actually takes an entire text file and builds an array where every single entry is populated with one line from the original file.

I'm not gonna show you the file, you can just imagine that the words file that's opened by default is an alphabetically ordered list of all the English words in the language with no intervening white space except for the backslash N's for new lines. Okay. So I synthesize this right here. The read lines that actually preserves the backslash N which is really annoying, but it just does it, so what I do here is I introduce the first ever fields to the lexicon object by saying, you know what, you didn't have anything before, but now you have this words field.

If it had it before it would just reinitialize it, and rebind it to something, but since I'm taking a raw object, I'm actually inserting one more thing into the dictionary that's backing the object, and it's initially set equal to the empty array. Okay. And then I just do brute force for looping. This is what's called an interable, it's actually an array. That's why I synthesize this right here. Okay.

And it just goes through, and it takes whatever words in that words array, truncates off the backslash N and puts it in the words array that's embedded inside the lexicon. Okay. Does that make sense to people? Okay. To whatever degree it's successful for you, just subscribe to your C++ and Java sensibilities as far as what constructors are for and what they're intended to do.

A lot of this is just different syntax. It has some quirks, its dynamics, but it doesn't have a class definition. There's no dot h file, there's no implied interface by a dot java file. It really is just this kind of deal with it as it runs type functionality, but nonetheless this entire thing is responsible for taking a raw object, and building it up to be a logically sound lexicon. If I just show you the next – oops, didn't mean to do that, but I'll bring it back in a second – the next method, it really is a method. It's a function definition that just happens to be – I'm not sure I can get the entire thing in there – the def is over there on the left, but you can just look at this.

Self dot words, there's this bisect function, it's just like B search it's a little bit different. It's more like the lower bound function from C++ in that it just returns either the index of the matching element or the index where the thing could be inserted in order for it to maintain alphabetical ordering. I initialized the words array so that it was alphabetically

ordered. All I'm doing here is I'm asking whether or not the word that's explicitly supplied when I invoke this contains word function, whether or not when I do a binary search for it, and get the insertion index for it, whether or not that actual slot in the words array matches the word I passed in.

So I'm actually going to slide this over and see what the double equals is, whether or not it's equal to the word local variable. Okay. You get the gist of what I'm attempting there? Okay. I have some other methods, I'll just name them. They're not that algorithmically interesting. Word contains everything. I'll show you how that works in a second. List all words containing. I'll show you those in a second.

What I want to do is I want to bring my terminal back and show you how this works. Let me make sure I'm in the right directory. I am, and I do in list to make sure I have words. I do. Let me invoke Python. This is how you deal with objects in Python. It's a little quirky, but there's a good narrative that can be made as to why things work the way they do.

If I want to use this lexicon class, just like all the functions inside divisors.py, and all the functions inside the copy module and random module and things like that, I have to import – I'm sorry, from LEX, because this is all stored in lex.py, I want to import the class or the symbol, whatever functionality is associated with the symbol lexicon. Okay. And I do that and it works. I can do this, I don't want to use L, yes I will. I'll spell it out, is equal to lexicon, just like that.

Student:[Inaudible]

Instructor (Jerry Cain):Oh, I'm sorry, well, you can tell me. I didn't do that. Okay. Do this. Wait, can you see it now? Barely.

Student: Yes.

Instructor (Jerry Cain):Okay. Now you can't here me. This right here initializes, there's no new keyword in Python. The name of the class in this case is what's called a callable, and if you invoke it as a function it's a request to build an instance of that class. When I do this, it actually builds EL, I can do a couple things here. Lexicon, if you just type it in, it just reminds you that it is in fact a class. Here's one thing. Just to show you how relevant dictionaries are to the lexicon class, there's a special meta variable that's inside all objects, inside a lot of things, but in a particular object at the moment.

You ask for its dictionary – it looks like gibberish, but you see gestures to a lot of things. You see some weirdly named keys like underscore underscore, module, and underscore underscore init. Okay. Does that make sense? You can clearly see that they're keys in some dictionary the way the way it's scripted out. So the in-memory model of an actual class definition is a list of all the symbols that are embedded inside.

The ones that we created are underscore underscore init, and contains words and list all words containing, and contains all characters or whatever I called it or word contains

everything. Okay. Does that make sense? So the actual class object is modeled by a dictionary. That doesn't happen in C or C++ at all. Everything's done in compile time, and it just generates all these 1's and 0's that are consistent with the original definition of the class.

In Java there really is something like this, it's actually not a dictionary, it's a standalone class that gets stored in memory, but it's not that different from this right here, but there really is an in-memory representation of the class idea itself. I can't show you EL; that would be a lot. Well, actually I will show you EL. If I do this, it tells you it's an instance of the lexicon in the LEX module.

When I do this, it's gonna show you the dictionary that's not associated with the lexicon class, but the dictionary that's associated with the instance of that class that I just created. Now this is gonna whiz by, it's gonna take a couple of seconds – actually that wasn't that bad. I can't scroll up because we're dealing with a 150,000 words. Okay. Well, actually I could, but I'm not going to.

What I will do is I will break the – oops, it didn't like that, sorry. Okay. That's a little weird that I did that. Remember that words was a field that I introduced inside, there's no notion of privacy whatsoever in Java. You're only supposed to deal with that by policy, and understand that there probably are fields inside you're not supposed to touch. There is some way to actually mark something as intentionally private using underscores. I'm not saying it's not used; it's kind of a hack.

There really is no enforced encapsulation in Python. It just relies on the programmer to be a good programmer, and to not touch inner fields or functionality that it doesn't think it needs to touch. But now if I do this you won't get the original dictionary, but you'll get that right there. Okay. Before words had this array that went all the way through though zizzaba, and I just happened to empty it out. Okay.

But what I'm illustrating here is that the object itself is backed by a dictionary and all the attributes, in this case there was only the words attribute, okay, but all those attributes are actually keys that are associated with some inner dictionary that's identified by an underscore underscore dict underscore underscore. Okay. Does that make sense? So you see how central the dictionary is to the Python language. Okay. I'm guessing you do. Okay.

So there's that. What else to I want to do. Let's actually just, because it's fun, not because it's really important, let's rebuild the lexicon contains word hello, just to show that it actually works, it does. Okay. And then I wanna list all the words – I love doing this – all the words that have all the vowels. Oops, I didn't do that. That's not – List all words containing – spelling right, yes, okay. Isn't that neat? And it's very brute force. This isn't the magic of Python, that's just they way I implemented it. It's just a regular class.

But the implementation of this is in the last page of the handout I gave out last time, the Python basics handout, and it's just a matter of gleaning syntax. I can tell you right now

from experience that when you go out and get a programming job, it very well may be in C++ or Java, it may be in some language you have very little experience with, but you can't be, um, I don't know how to do it. But binary search is mysterious in all other languages. It really isn't.

It's just a matter of learning the other syntax and the idioms that the language supports for getting things like iteration and recursion and classes and structs and objects and all that kind of stuff down. Let's see if there's this. Just that one, and then obviously there's nothing else. Let's see if there's any words that have A, B, C, D, E, and F in them. That's not bad. Okay. N, G. Nope. Okay. Okay. So that's just playground stuff. That's kindergarten stuff just doing that. What I wanna do now is I wanna show you – let me just draw – I'm trying to think if I wanna do some stuff. I do actually. Let me put this to bed.

What I wanna do is make a few remarks about the illusion of objects just being dictionaries. When you do something like this, let's say at the prompt you do O = object. This is the equivalent of java.lang.object. You do that right there. You get absolutely nothing in response to this. You just get O is an instance of the object, but if you do this you actually get that right there. Does that make sense to people?

I'm asking it to identify the collection of attributes that have been accumulated inside the object I'm calling O or the instance of the object I'm calling O. When I do this, and then I do this, and I do this, they're really instructions to insert a new value in the dictionary that is underscore underscore dict underscore underscore. So if I do this, I actually get three fields. I'll assume they hash in this order. Okay.

When you do this you're not respecting capsulation. You're supposed to let methods do this, but I told you already that it doesn't respect any kind of privacy. There really is no support for privacy that's genuine and real. There's that, but doing these three lines right here is like doing this, and in fact it more or less translates to this. And then O dot underscore underscore dict underscore underscore c = [inaudible].

This right here is syntactic sugar. It doesn't look like syntactic sugar, but it is in the case of Python, because this is just taken to be an instruction to not really try to do this, but to do that right there. It's really physically levying an operation against the dictionary that's inside of it. Okay.

So the takeaway point from this – I'll talk a little bit more about this in the first five or ten minutes on Monday, but the takeaway point here is that the objects are backed by these growable, shrinkable containers, they have to be growable and shrinkable in ways that they don't have to be for C, C++, and Java because C, C++ and Java do all this wedding planning up front where everything has to be set in stone before any code executes whatsoever.

Python is a fully dynamic language. Everything's supposed to be able to grow and shrink. Everything about it is supposed to be dynamic, so you're not supposed to impose limits

on how big something can be unless it's an implementation limit like you just can't store more than like 2 to the 12th keys in a dictionary or something like that. Okay. So activation record as an idea, it works beautifully for C and C++ where it can actually do all the scrubbing up front, but for Python it can't; it has to back all the objects with a different model, and this is exactly how it does it.

Okay. So you definitely have enough information to crank through Assignment 8. Come Monday, I wanna talk a very little about inheritance. I know you know a little bit of inheritance from either AP Java or from 106A. Okay. If you've taken 108, then everything will be easy. But I will just talk a little about inheritance and how it's relevant to some of the, I think, more interesting ways of doing networking operations which is what this handout I gave out today is all about. Okay. So you guys have a good weekend, and I'll see you Monday.

[End of Audio]

Duration: 49 minutes