ProgrammingParadigms-Lecture24

Instructor (Jerry Cain):Everyone, welcome. I have two handouts for you today. You all know that Assignment 7 is due this evening, 11:59. It is the last assignment that we'll issue a letter grade on. Like Assignment 1, Assignment 8 is intended to be this kind of like transition type of thing. We're trying this out of the course. I'm not planning on putting Python or anything related to Python on the final exam. I'm just trying to fill out the course with a new interesting language that's topical. It certainly represents a paradigm, but I'm just trying to get in the habit or not forcing lesson material on the last few weeks or the last few lectures to come up with a clean endpoint, which is probably I'll call last Friday as far as testable material, but still fill out the quarter with an interesting assignment. The assignment that's due a week from today – I'm sorry, a week from tomorrow. I'm giving you until Thursday evening of next week. It's this very short Python assignment that's just intended to get you used to the framework. It's not gonna teach you everything about the language. It's really just taking a program that works and making it work a little bit differently just so you're forced to deal with Python and its integers and its Booleans and its tests and to fully define functions and classes and things like that, okay.

So as far as how the rest of the quarter will work out, I will definitely lecture today and Friday. I probably will lecture Monday because I don't think I'm gonna get through all the Python material. As to whether or not I lecture next Wednesday, I think will just depend on how quickly I get through the Python material that I want to introduce to you, but I'm not trying to race as much material as possible into the course because I know you all have a lot of other things going on. Fortunately, 107 does not have a lot of stuff going on in the next week. After you get the scheme assignment done, then you're really just in a situation where you can start studying for the final, okay. I'll have a practice mid-term – I'm sorry, a practice final probably out next Monday, a practice solution as well. The mid-term is scheduled for June 9. You can take it either at 8:30 or 3:30 on June 9. I have rooms set up. I forget where they are. I'll put that on the practice exams. Okay, so this thing called Python. Oops, let's spell it right. There are two overarching features that I want to emphasize. It is as opposed to anything you've seen before, Scheme you could argue is a little bit like this, but C and C++ and Java are certainly not. It's an example of what's called a scripting language.

Okay, and then as far as that, we can call that a paradigm. It is also imperative. It is also object oriented. It is also functional or it has functional components in the language just like Scheme does, and I'll show you an example hopefully today, either today or the beginning of Friday where you actually see the equivalent of map, okay, in Python, and you're all intimately familiar with the notion of map right now. You've seen vector map and you've seen iterators and things like that in C++ and you certainly know map from Scheme. There really are lambdas. In Python, there's also the map function. There is the notion of an evaluate statement just like there is in Scheme. So I put these up here kind of to remind you that imperative is the C language, it's the assignment oriented, function oriented type of language, I'm sorry, procedure-oriented type of language. [Inaudible], you're very familiar with that. Everything's centered around the data and we call objects for classes and functional is an example of a paradigm which Scheme adheres to. I would say that Python is object oriented like C++ is. A lot of people program in C++, not so much as C++ programmers, but as C programmers, but like to use objects. Does that make sense?

And they still had this very sequential way of thinking and they organize data in terms of classes as opposed to structs. I actually – I get the impression based on what little Python programming I've done and what good amount of Python code I've seen is that it really is the case that people program imperatively, okay. They rarely subscribe to the functional or object of paradigms. They usually go with this imperative approach where they have a series of tasks that need to be done one after the other, and they just get them done in bullet point format, okay, and subscribe more to the C like or Pascal or just the imperative approach of actually accumulating data piecemeal until you actually have a result at the end. They may incidentally use objects, certainly those that are part of library set that come with Python. They may like to use lambdas and they see an opportunity to use some functional paradigm approach to solve some subset of the problem. They might do that. I really think of it as this where it just happens to have these features inside of it. That's consistent with the fact that it's a scripting language.

The best analogy that I can think of outside of programming for scripting analogy is if you actually ever go see a play and for whatever reason there's like a casting emergency and somebody who normally plays the part and has the lines memorized isn't available, so somebody has to actually fill in the part and they have to hold the script in their hand, okay. I have seen that a couple of times. They have to go through the motions. They have to do everything perfectly, but they actually are reading their instruction sets while they do it, okay. Do you understand what I mean when I say that? Okay. Just think about a set and like a dress rehearsal for a play where you still have a script in hand. I think of a Python script or any kind of shell script for that matter as being very much like that. It's this very what you see is what you get. It digests function definitions and it digests individual statements as they read them and as they read them the side effects of their execution are actually realized while the script is running, okay.

So it's like Scheme in that regard. We normally prepare all our functions and put them in a dot SCM file ahead of time and then load them and then run the result. That still has a scripting feel to it, but the emphasis is on the functional paradigm. With Python, and a lot of things, Pearl, a lot of things that are designed to be run a script, to batch process these together to get some large set of tasks done. I actually see that – I see Python doing that more than anything else. I don't see many games written in Python. I don't see many compilers written in Python, although I do see some servers written in Python. A couple of – I don't want to say the names because I'm not actually sure of them, but there is this set of companies right now that are setting up services, not these companies specifically, but companies like Twitter and Friend Feed that are all trying to basically broadcast information. Some of them, I'm not sure which ones. I think of one, but I don't want to say it and be wrong, are actually writing all of their server and services in Python, okay. Normally those things have been written in Java or in C++. Python has a very rich set of networking libraries and natural support for SMTP and ACTP and ACTPS and all of these networking protocols that are very common today with all of these distributed systems that are set up to provide these types of services, and I know Python is contributing to that.

So there really is some large-scale software that's being written in Python beyond the scripting set that I'm gonna focus on today, okay. As far as other buzzwords that describe Python, it is dynamic. Scheme is also dynamic. C and C++ are not at all, okay. They have a huge compile time element and there's very little type information at run time. Scheme and Python both maintain type information about every single piece of data that exists in the run time, and you can actually query the data type if you want to, okay. Does that make sense? Okay. It is really interesting to code in Python because it feels very much like C and C++ and Java in the way that you just do block structures and if statement and Y loops and things like that. But there's a down play on the emphasis of curly braces and parenthesis, okay. There's a – the block structure is actually decided, not by curly braces or by parenthesis, but by white space and tabs.

And it's the most frustrating thing to get used to the first day you program in Python because you just naturally put in curly braces, and then you realize you do that, and you're like, "It's not gonna work." So when I show you our first – show you the first Python function I want to write, you'll see just that it's just about taxed with very little delimiters. There's double quotes and things like that, but there's no curly braces. There's very few parenthesis. It's really just very strange to look at the very first time you see it, but then you just get used to it after you program in it for a while. So what I thought I'd do is I'd take the Scheme approach to introducing Python and just show you the Python environment, how you can actually talk to Python and get it to announce – like tell you what the sum of the first four integers are and what happens when you type in a string constant and how Booleans work and things like that, just to you get used to the syntax, and so you have the understanding as to what things look like. And I will do that first. Going to the screen, I will try and get as much of this to fit on the screen as possible. Okay, that's not bad at all. Okay, so I launched Python much like we would launch Kawaa, okay. I just type in the word Python. As opposed to Kawaa, Python basically comes on any system that has any Unix backing whatsoever. So the mess, the pods, all of them have Python installed already, so does Mac OS10.

So you just go into a terminal and you type Python and it's probably gonna do what it just did right here. This part should feel very familiar to you. I type in a four and it comes back with a four. I type in hello there, and it comes back with that. It happens to delimit it in single quotes. You can actually use them interchangeable. It allows you to use both double quotes and single quotes because you might want to delimit the entire thing in single quotes if there are a lot of double quotes in your string constant and vice versa. That doesn't mean that a lot of modern languages are paying a lot of effort to because there's so many quotes that appear in HTML documents and things like that they don't want you to have to backslash and escape everything because that just makes it that much error point of an investment to try and code something like that up. It actually has [inaudible] numbers not surprisingly, and that it is also nice that we're back to normal addition, okay. I don't want to say that this is conceptually confusing, but whatever.

We’re used to end fixed notations, so Python went with the in face approach. As far as the Boolean constants are concerned, true is true, false is false. True and true is true. You actually spell out the keywords, okay, kind of like you do in Scheme, but they’re in face again. If you want to – I’m trying to think of what else.

The one thing about strings that are interesting is that there really is no direct support for the notion of a character. It’s pretty clear that this is a string the way I type this in, whoops, quote A, B, C, D, E, like that, okay. This happens to be the best you can do for isolating an individual character. You just think of characters not so much as characters as you think of them as strings of length 1, okay. Does that make sense? Okay. Numbers themselves are just plain old numbers, but strings, let me do this. Hello starts with HB. Whoops, didn’t like that. Oh, that’s my fault. Hold on a second. Oh, I’m sorry. I’m just doing something. I’m sorry, starts, doesn’t like that, sorry. I’m just messing up here. There it is, okay. Capitalized in Scheme, okay. The one thing that’s unique, and this isn’t unique to Python. I’ve seen this in C sharp and managed C++ as well is that you don’t necessarily have to create an object to surround an object constant in order to send messages to it. Does that make sense?

I certainly could have done this. Greeting is equal to hello, and then done something like this, greeting dot starts with worth, that right there. And that’s the way you would have taken – would have done it in C++ or Scheme or something like that, but Python is just smart enough to know that if you’re sending a message to something and it’s actually a constant, it can tell by looking at the constant what data type it is, so it can just build the synonymous object around it so that it really can receive these messages. Does that make sense to people? Okay, so as far as all the primitive types, I’ve more or less touched on everything I wanted to. With regard to Booleans, you have and, or, and not spelled out. That shouldn’t surprise you. True and false are the actual Boolean constants. I will do this. You may look at that and you may say, “Okay, that doesn’t make sense.” But do you understand that as opposed to pure Scheme where everything evaluates to something, that thing right there didn’t evaluate to anything. It had the side effect of actually associating the number five with X, but X equals five didn’t print anything out. Does that make sense? That’s because it evaluates to this constant called none, which is basically the equivalent of void from C and C++, which means there’s really no side effect or nothing of interest in the evaluation of it, okay. Does that make sense?

So you can still do this, okay, but then it adopts a whole different set of functionality in order to propagate the assignment. But the overall assignment and actually evaluates to this constant called none. And none really doesn’t mean very much except that it doesn’t have anything printable about it, okay. So there’s that the – as far as strings are concerned, there are a whole set of methods on the string class. It’s certainly a superset of the set of string methods you see in traditional languages like C++ and Java. Python recognizing that when it was written, it was intended to be something of a web savvy language. There are some very like very strange methods that exist for the string class.

Let me see if I can do some of them. Hello, there. Capitalize right there. That’s a strange method, but it just realizes that a lot of things get printed sentence wise as part of HTML documents, so rather than actually having you go through and uppercase everything that comes at the front of a string, it just has built in code that does this, okay. It has some really weird stuff. Let me see if I can do this. Is title, it comes back with false. So a really weird predicate method to be built into the language, okay. It really just analyzes a string and it goes through the entire thing and sees whether or not every single word that's delimited by white space or by the boundaries of the entire string happens to begin with a capital letter. So something like this as if it's a movie title – like clearly Iron Man is a title. And I'll go back, don't worry. Just like that comes back with a true because it's clearly the name of the movie, like it just knows, okay. Does that make sense to people?

Now I'm not clear what the motivation is to why these things were specifically included in there except for the fact that they just wanted to have – maybe it must have been the case that they just saw these types of things being written several times in other languages and said, "You know what? Let's just take care of it. Why make them rewrite it again? We'll just put it into the core language, okay." Does that make sense? Now as far as writing functions is concerned, it's a little difficult to do at the command prompt for the same reasons it's difficult to write a Scheme function at the command prompt. I do want to introduce one other data structure before I write a function, okay. Let me show you what a list looks like. This is a list constant just like that, okay. The square brackets actually mean something. It means not only is it a list, but it's gonna be a mutable list. So I can do something like this, small nums is equal to one, two, three, four, five, six, just like that. And then I can ask what small nums of zero is, okay. And that doesn't surprise you. I can also do that this is the fun part. What do you think that means?

You're right because it actually understands that it is just – there's no reason to buy a search from the front in terms of the syntax that's available to you, so it recognizes that you might be interested in the last character, but you don't want to do it like this where you do lin of small nums minus one. I'm not even sure this works. Yeah, it does, okay. Lin I've used on strings. I haven't used on list before. You could do it that way, but then it actually has to manually count the entire thing, whereas internally they could just optimize operator square bracket, the C++ terminology to just have access to both sides, and when there's a negative number to just start searching from the back instead. Okay, this is – I mean the neat part. What I can do is I can do that right there and I can get sublists. Does that make sense to people? Okay, so that wants everything from index zero up through, but not including index three. If I do this, that's basically the equivalent of that right there. If for whatever reason this programmatically comes up, I can get the empty sequence, everything from index zero up through but not including index zero is just basically the empty list.

If I'm interested in the last – the end of the list, I can just punt on the thing that follows the colon. It basically says negative two up through the end of the string. Does that make sense to people? If I want everything but the last two characters, I can do something like this, okay. Does that make sense? So it's the syntactic sugar that was introduced to the language because this language is much younger than any of the other languages we've studied. Scheme is really the oldest of all the languages we've looked at, at least in this class. Scheme, we're talking like late 1940s, I'm sorry, late 1950s, early 1960s when there was a lot of research circulated around it. It was really introduced to kind of emulate the lambda calculus and come up with an implementation of that, and it was used for symbolic programming and larger programming and things like that.

C was invented for the purpose of implementing Unix. It's like they wanted access to the hardware. They wanted some form of abstraction. They thought it was brilliant level of abstractions at the time, I'm sure. We know better now, but it actually provided a much cleaner syntax than what was otherwise available to those implementing operating systems, which was assembly code, okay. You had to get down there in the nitty gritty of the hardware. It's hard to do that in Fortran or Scheme, so they actually – they. Those who worked on Unix invented the C language with the intent of actually making it easier to build the OS. C++ was invented late '70s, early '80s. It became fashionable around 1990. It was initially used as C with objects, and then a lot of effort was put into the implementation to introduce templates and a string class and all of those things that you're now familiar with. But we're talking with a language that at this point is 27, 28 years old in terms of specification, okay.

Python is probably 13, 15 years old, okay. It didn't come into any level of popularity until year 2000, 2001 is when I started to hear about it. That doesn't mean that it's all of the sudden popular only because I've heard about it, but when I was doing a lot more consulting work at the turn of the century during dot com era, I heard of a lot of big name companies starting to use Python. Google is the one specifically that I remember hearing about so much that people that went there to work said, "You know what? You should really teach Python in 107 instead of Scheme." I'm like, "No, I like Scheme better. I think it's cooler as far as illustrating a paradigm." But I'm getting to the point where Python is something we have to pay attention to because it has that much more of a presence in industry, okay. If you want this, just another neat little feature of list, you can do this with strings as well. Hello, there. See if this works. Oops, sorry about that, totally can. Okay, it emphasizes the fact that it really is a sequence of characters, something of a list. It turns out as an immutable list of characters. Strings are immutable like they are in Java. The one thing that's really fun, let me just do this. List – I shouldn't do that. Let's do SCQ is equal to – let's do zero, one, two, three, oops, can't count. Do that right there, okay, and I want to go back and correct the problem, okay.

Do you understand that the part where there should be a four is actually the sublist of SCQ and I want to say it's this right here. I may be messing this up, but I don't think I am. I've just identified the splice within the list that's missing the four. Does that make sense? Okay. So what this is gonna do is it basically comes up with an L value and identifies the sequence in the list that is synonymous with four, up to, but not including four, which is this empty region between the three and the five, and you can actually assign to it. Does that make sense? I hope I'm not messing this up, but we'll see in a second. There you have that, okay. Now it turns out this is just cool. It's not anything you can't do in Java or C++ or Scheme already. It's just that it's more gracefully handled by the syntax that you're seeing right here, okay. That's just the product of it being a more modern language. It can learn from all the other language's mistakes and know where a program was cumbersome before and just try to come up with some better solution now, okay.

There is a variation on the list. When I do this, SCQ is equal to five gate – whoops, commas are important now. When I do that, I actually have the option of taking sequence and saying, “You know what? I have fives. I’m gonna replace it with a 14.” And so we’re dealing with a clear script of what’s called a mutable list. I just call it that. There is a variation of the list which was immutable. I do this. It’s also considered to be a list, but the parenthesis, for whatever reason, I don’t know why they’ve dumped the parenthesis, but that marks – I mean is considered to be an immutable list, a collection of data that is read-only. So if I do this, that’s all fine and dandy, but if I try to do this, it freaks, okay. It’s actually not called a list. It’s called a tuple. I’m reading that right now. And but it’s clearly marked behind the scenes as read-only. It just wanted to be able to make the distinction between something that can be updated while the program is running and something that is supposed to be just this bundling of data in some predefined sequence that can’t be updated at run time. So it’s just basically this one little gesture toward const or final in the language, okay. Does that make sense? Okay.

So as far as lists are concerned, they are in fact objects. You haven’t seen any object oriented flavor to this yet, but if I do SCQ is equal to – let’s just do some strings, which is what these really are. SCQ append D works, okay. Does that make sense to people? Okay. So there’s that – there’s a whole list of list methods. There’s actually lists, there’s sets, there’s dictionaries. They’re all these things they’ll talk about in a few minutes. But let me just go ahead and just write some code that does something algorithmic. It’s not gonna be very sophisticated, but I just want to show you what the things look like. I do have to kill the computer for a second. I will come back and show you how it works afterwards. As far as languages go, this language is more like the make file than anything I’ve ever seen before, and it’ll be clear in a second when I actually write some code. I want to write a – I think a fairly juvenile function in terms of algorithm, but I want to illustrate the syntax. I just want to write a function that’s in the handouts called gathered divisors.

So I want it to be able to work like this. I want to be able to pass, let’s say, gather divisors, and I want to be able to pass in something like 24. And I want it to be able to spit back 1, 2, 3, 4, 6, 8, 12, and 24 is fine with me, okay. I want it to accumulate all the things that evenly divide into the number that’s specified, I’m sorry, all the positive integers that divide evenly into it. Keyword is def, okay. Just know INE was too much work to include that, so we just go with the EF, gathered divisors. I’ll just call it number, and there’s a colon right now, which means it’s very clear about that. Everything that follows, any line that follows this one that is indented is under the jurisdiction of the entire definition, okay. Does that make sense to people? So what I want to do is I want you to just assume that the number is a positive number. There’s some error checking in the handout. I’m not gonna worry about that right now. What I’m gonna do is I’m going to set divisors equal to this list constant. There are no semi-colons. I almost put a semi-colon there. It’s very hard not to put a semi-colon, okay, because you put a semi-colon in every other imperative language other than Scheme. It just knows that the lines end and now backslash N means semi-colon, okay.

More so than any other language this space right there, this tab right there, okay, is absolutely required because if it started right there, it would have not only marked a statement that said divisors is equal to the empty list. It would have implicitly marked the end of anything associated with this definition right here, okay. So all the white space that you see in the handout in the definition of this function, every single one of those, it’s not four spaces, it’s actually a tab. It may be stores as four spaces by the editor, but it’s rehydrated into a tab or the equivalent of a tab when you actually load the file again, okay. Not surprisingly, what I want to do is I want to look over every single thing between 1 and 24, okay, and inclusive 24. And I want to decide whether or not the number defies it evenly. I’m speaking like you’re all 106A programmers. I know you know how to do this part. You just may not know the syntax within Python to do it.

This is the way you do it. I’ll say dif in range. I will say zero. I will say number plus one plus one. It looks very for loopish. It certainly is. It has a technically different approach to the way that it generates for loops. You’re used to four I is equal zero, I less than ten, I++, and there’s a test that is checked with every single iteration to see whether or not you should continue. For loops are almost always expressed – I’m sorry, they are always expressed in terms of what are called iterables. That means that the object of the N key right here has to be something that evaluates to a list of things, okay. Now this range function is a built-in. I’ll explain – I’ll type it on the console in a second. But this actually evaluates to the list. I don’t mean zero here. I mean one. It evaluates by default to every single number between and including this one up to, but not including that one right there, which is why I have the plus one there. So this actually evaluates to the list, one, two, three, four, all the way up through number, which in the context of this thing would be 24, and we want 24 in there, okay, because I want it to appear in the result.

There’s an optional third argument right here that can be the step amount. By default, it’s one, which means just include every number in counting up sequence. If I put a ten there, it just would have like done 1 and 11 and 21, etcetera. You can even make it negative if you want to, if you want to generate a list where it’s counting down from some high number to a low number, okay. This idiom right here, it’s technically different than the for loop you’re used to. It just iterates and it associates div with every single number in this iterable, okay. So in the first iteration it’s associated with a one and a two and a three, but for different reasons, okay. If there’s some implicit – there’s no implicit plus equals one or plus plus coding behind the scenes. It’s incidental that the numbers are sequential because that’s the way we constructed the iterable. Does that make sense? Okay, so there’s that.

So with each iteration, this is the easy part. If it’s the case, I need a tab there. I don’t need parenthesis. It doesn’t cause problems, but you don’t need them. If it’s the case that number mod div equals equals zero, then I want to take divisors and I want to append whatever div is on this particular iteration, okay. And I have no semi-colons. When I come right here I’m gonna just return whatever’s accumulated over the course of that iteration divisors. There is no ambiguity as to where the if test ends. I'm sorry. There's no ambiguity as to what's under the jurisdiction of that if test, and there's no ambiguity as to what's under the jurisdiction of that for loop, okay.

The fact that this came all the way back over here means that it's a peer in the sequence of statements right here and this marks the end, not only of this if jurisdiction, but also of the for jurisdiction, okay. If I had moved this over one tab, it would not have been under the jurisdiction of the F, but it would have been under the jurisdiction of the for, which means that it would have returned after the first iteration was over, okay. Make sense? Okay, so there's that. I'm going to store this in a file called divisors dot PY. The dot PY is more required than you'd think. It's actually not required, but I think every single Python file I've ever seen ends in dot PY, okay. So what I will do, so I'll bring the computer back. I'm doing okay on time. This is good. And I will control the exit because I want to see where I am. That's not where I am right now, CD devel, CS107, Python examples, okay. Let me just – I have a better idea. This more divisors right there, you see the gathered divisors right there, so there's clearly some codes in this file that looks very much like the code I just put on the board. Let me open it in a slightly prettier environment.

Text Mate is the coolest little program ever. Does anybody use Text Mate? It's great. Okay, so here's this. Let's just focus on what is visible, nice soft autumn colors, okay. I actually did not include the 24 in my implementation here, but I included it on the one on the board, but conceptually it's exactly the same. The weirdest thing about this is you see this triple double quote at the front. If the very first statement in a function definition like I have here is a string, it's understood to be a documentation string. I'll show you how you can actually find that in the run time interpreter that I was showing before. The triple quote means that the string I'm about to present is actually expected to appear over multiple lines, so don't give me a problem because I don't put a double quote at the end, okay. Does that make sense? Okay, so there's that.

As far as this is concerned, that's the entire function. I have some other code in here. I didn't put this in the handout, but there is prime. There's scattered primes, whatever. It's the same exact idea. So that's the entire thing. There's no real analog to this in C. There's a little bit of an analog to it in C++ where we have name spaces. You're much more familiar with this from Java where you import packages, okay. Do you understand what I'm talking about? Okay, there's a little bit of that in Python as well. The fact that I called this gathered divisors, of course that's relevant if I want to actually call this function. The fact that it's inside this thing called divisors dot PY means that it's inside a module called divisors, okay. So that when I come back and I quit this and I go to a directory like that and I relaunch Python.

Actually, I'm sorry. I didn't mean to do that. CD devel, it says 107 Python examples. Okay, I'm inside this directory that has all these modules of code. I'm gonna go ahead and invoke Python. And if I try to call gathered divisors 24, it's like, "What are you asking me? I've never heard of gathered divisors. It's not in the language. Yeah, it may be in any one of the 15 trillion files in the world, but you have to tell me which one it's in." There's a couple of ways to do it. Import divisors does that. That is relative to the current execution path of the current path where you – I'm sorry, the current directory backing the Python run time. It basically goes in an digests everything inside the Python file, so it's the equivalent of the load statement from Scheme, okay. And you say, "Okay, that's great. I can just do gathered divisors, and now it will work." And the answer is no, it will not. The reason is just because you've digested all the material that was inside the divisors module, yeah, the divisors module, doesn't mean that all of the sudden all of the things that were defined in there are the master copies of code attached to all those symbols. So if you really want to invoke the gather divisors, there's two ways to do it. The more clumsy way is this. Divisors dot gather divisors, oops. Did I actually get that right? Yes, 24. Pray with me, yes.

Okay, so basically the module is the name space, okay, and you have to frame by default all functions you invoke in terms of that name face so it knows specifically which gathered divisors you're talking about. You may think that's ludicrous, but in our real system, you're gonna have presumable, let's say, between 1 and let's say 4,000 Python files that are all contributing to a system. You have to make sure that no two functions are named the same thing because if they are you want – but if there are, you have to qualify which one you're referring to by actually including the packet name inside. Now if there's no danger of ambiguity, you can do this. Let me control D and start over. You can say from divisors, import, gather divisors, and then all of a sudden gather divisors is a top-level function name. So it's just syntax. It's all in the handout. It's not really the emphasis, but I'm just leading you through the full example and hitting on everything that I think is important, okay. Does that make sense to people? Okay, now you know how in Scheme that you modeled everything in terms of lists, and if you had a struct, you'd say, "Oh, we'll just use lists." And the CAR of the list is the name and the CAR of the CDUR is the GPA. You just had to remember the actual structure that was imposed in your lists. Python you don't need to do that.

You can elect to do that if you want to, but most people use classes, which I'll talk about a little bit more on Friday how they work. They're not complicated. It's just that we really do have better modeling schemes available to us in Python because it's just a more modern language and they included those things inside of it. What I thought I would do here is I would pretend that we haven't done that stuff, and I will introduce you to, I think an even more central data structure to Python than anything else involving lists or sequence or tuples for that matter, okay. There is a – the notion of what is called a dictionary. I'm very careful to use the word dictionary in Python because that is the replacement word for map. You can use map and if we're talking about Python, people would know what you're talking about, but dictionary is the operative word for the primary data structure in Python. If I do this, let's say I'm student, and right now it's not a very interesting student. I use square brackets and parenthesis for lists. I use curly braces to delineate all of the content of a dictionary constant. And when I do that, all it means is that the student is an idea at the moment, okay.

I can do this though. Student name is equal to let's say Linda, okay, and there's that. And now all of the sudden student has grown quite a bit, okay. Make sense? I can do this.

GPA is equal to 3.98. Go, Linda. And then we have that, okay. Does that make sense? If I want to say I forgot her GPA I can do this and I'm reminded, okay. Interesting to know is that the dictionary is completely backed by a hash table, okay. And it's a very small hash table, and when I say small, I meant the number of buckets is actually very, very small initially because most of these dictionaries imitate structs and classes and objects in Python. In fact, objects are really dictionaries with just a little bit more embedded inside of them. If I want to update for GPA because it's just not high enough, that's Harvard. That right there obviously updates the dictionary in place, okay.

The reason I'm talking like this is because the dictionary is easily the most malleable, easily manipulated data structure in Python as opposed to lists in Scheme where you pass around lists and you functionally manipulate them to create new lists. Python, at least to the extent that I'm gonna have you exercise it in Assignment 8, really just deals with strings, which you already have enough information about, and dictionaries, okay. There's one thing I forgot to mention last autumn when I taught Python for the very first time, and it impacted people on Assignment 8, so I just want to say it right now before y'all disappear for he day is that all of these objects, and I say objects loosely. Anything, any aggregate data structure like a list or a tuple or a dictionary, they're all passed around by reference. So if you write a function that takes a dictionary, it just makes a shallow copy of it. Any changes that you make within the function are reflected in the original. Does that make sense to people?

That's a huge point for the purposes of getting through Assignment 8 because you're gonna use one of these dictionaries to keep track of a master set of information that accumulates through some intense recursive algorithm, okay. And it just involves some caching of previously computer information so you don't make the same recursive calls several times, okay. So that's why I wanted to say that right there. I think it's pretty clear that the dictionary right here is heterogeneous in the fact that it really doesn't impose any requirements at all on what the values need to be, okay. So I have a string attached to the field called name. I have a double or floating point attached to the thing called GPA. If I wanted to do this, share, and there's that. Whoops, what did I do wrong? Oh, I'm sorry. I meant an array. There's clearly order, student, and then all of a sudden you have this third thing inside here. So you can attach anything you want to these things. The fact that it was printed first just means it happened to – that friends happen to hash to a lower value. That's really what it is, okay.

If I want to bundle all of my ideas, but I don't have any at the moment, I can do that. So there's this one thing where one of the keys happens to be associated with an interdictionary, okay. Does that make sense to people? Okay, so it is really free form. In fact, let me just do something like this, playground is equal to initially it's empty. I don't recommend this. I'll show you the problem with this in a second. It actually doesn't have a problem, okay. So you can have things that are non-strings. It just – anything that is hashable can actually appear as the key inside a dictionary. I don't recommend actually mixing them. I would stick with strings, okay, for the keys, but it's a freefall as to what value you want to attach to those things, okay. Does that make sense? Now come Friday I want to do a lot of things. I'm going to jump ahead to the lecture next week. I forgot that it's – I thought it was gonna cover like this entire handout today and I'm just not – I just didn't at all.

I want to talk about a lot of things about Python specifically that are interesting from a language standpoint. The libraries are certainly interesting, more so than even Java, which you have some familiarity with. It has incredible support for XML processing, for HTTP processing, for building a web server, things like that, just amazing that you can really – of course it's rudimentary, but you can build a web server in about 15 lines of code, and I'm not overstating it. And it's not very sophisticated. It just does the special files and it doesn't do anything clever about like compiling pages or building pages dynamically, but if you just want to fetch pages behind the scene, you can use Python and about 10 lines of code, 15 lines of code, do exactly that, okay. That's usually a statement that the libraries are really taking care of something and they're taking care of it very, very well, okay. On Friday I want to show you a little bit more about dictionaries. I have this very dense example, but nonetheless, I think representative of how Python would solve the random sentence generator problem that you all solved for Assignment 1, okay. The difference is that you actually prepare the grammar and you frame it in terms of a dictionary, okay.

You may think that it's cheating that you're getting the data structure in memory, so you don't actually have to worry about reading it in from a file and parsing it and looking for the angle brackets and things like that. We didn't have the option in C to do anything else, okay. We don't have data structure constants that you can actually very easily assign to variables in C. You have to build them up out of raw deserialized data, okay. In Scheme and also in Python, functions and data – I'm sorry. Functions like the ability to express a constant carried beyond just the set of integers and the set of strings. You can define list constants. You can't do that in C or C++, okay. You can define tuple constants and you can define dictionary constants. So you can just elect to format the RSG grammars in something that kind of looks like a Python dictionary so that when you set it equal to a variable called grammar, it's loaded, okay. Does that make sense? And then you can just write the recursive code and frame it in terms of this variable that you know happens to be a Python dictionary. So you can use operative square bracket and you can assign things to it and you can make random selections from the list of options that are available and associated with each non-terminal, okay.

And I'll go over that. It's in the handout, but I think it's very nice to look at. It also introduces map, this Scheme functional language thing as to how that can work. Okay, so with just a little bit of work you are more than outfitted to tackle this Assignment 8. Even if you struggle on the first like 30 minutes to an hour just to get the syntax down, I think you will recognize that it's not intended to be a lot of work. It's just about getting used to the Python idea, living without a compiler or an interpreter that does very much checking for you. Okay, I will see you all on Friday.

[End of Audio]

Duration: 48 minutes