Programming Methodology-Lecture13

Instructor (Mehran Sahami): So welcome back to Week 5 of CS 106A. We're getting so close to the middle of the class, so a couple of quick announcements before we start. First of which, there are three handouts. Hopefully, you should pick them up. There's this week's section from some string examples, and kind of some more string examples that we're gonna go over today.

And now, since we're getting to the middle of the quarter, it's that time for the mid-term to be coming up. So, in fact, the mid-term is next week. It's a week from Tuesday. I know. The quarters go by so quickly. If you have a conflict with the mid-term, and by conflict, I mean an unmovable academic conflict.

Like, if you have another class at the same time, not like, oh, yeah, Tuesday night, you know, October 30, yeah, around dinnertime, I was gonna go out with some friends for dinner, but the mid-term is really cramping my style so I can't take it then. That's not an unmovable academic conflict. So if you have an unmovable academic conflict with the mid-term time, check the schedule.

Send me email by 5:00 p.m. this Wednesday, so you have a few days to do it, 5:00 p.m. by this Wednesday. Don't just tell me you have a conflict. Let me know all the times you are free next week to be able to take the alternate exam 'cause what I'm gonna do is a little constraint satisfaction problem. I'm gonna take everyone who's got a conflict with the mid-term, find out the time that everyone can take the exam an alternate time.

Hopefully, there will be one, and we'll schedule an alternate time. So only email me if you have an unmovable academic conflict. Make sure to list all the times you're free, and let me know by 5:00 p.m. Wednesday 'cause after 5:00 p.m. Wednesday, I figure out what the time is, and we ask for room scheduling. And that's it. I can't accommodate any more requests after that. So 5:00 p.m. Wednesday, know it, live it, learn it, love it, mid-term.

A couple of things about the mid-term you should also know, one of which is it is an open-book exam. It will be all written so leave your laptops at home. You won't be using a laptop. But if you ever get a job in computing where someone says, hey, guess what, you have to memorize, like, the Java Manual, don't take that job. So in computer science, we say, hey, you don't need to memorize all this stuff 'cause people, actually, look it up in the real world when they need it.

So it's an open-book, open-note exam. You can bring in printouts of your programs, and I would encourage you to do that. You can bring in your book, only the book for the class, by the way. Don't bring in a whole stack of books. And the reason why we make that restriction is because we – actually, in the past, there has been bad experiences with someone who brings in a whole stack of books. And they spend so much time trying to look things up in a whole stack of books that they just run out of time on the exam. It doesn't make any sense.

All the information you're gonna need for the exam will be in your book, and in the Karel Course Reader. So open book, that’s course text, open Karel Course Reader, open any handouts or notes you’ve taken in the class, and printouts of your own programs. So you can bring all that stuff in. Another thing about the mid-term is you may be wondering, hey, what kind of stuff’s on the mid-term, so on Wednesday I will give you an actual sample mid-term and the solutions for it.

I would encourage you to actually try to take the mid-term, the sample that I give you under sort of time constraint conditions. So you get a sense for 1.) What kind of stuff is on the mid-term? 2.) How much time you're actually spending thinking about the problems versus looking stuff up in books. And that will give you some notion of what you actually need to study to sort of remember quickly versus stuff you might want to look up.

Another good source of mid-term-type problems is the section problems. So like, the section handout this week, actually, has more problems than will probably be covered in section, just to give you some more practice. So section problems are real good. I’ll give you a practice mid-term. That will give you even more sort of examples.

And if you want still more problems to work through, I’d encourage you to work through the problems of the exercise at the back of every chapter of the book. Those are also similar in many cases to the kinds of problems that’ll be on exams. So you'll just have problems up the wazoo if you want to deal with them, okay. So any questions about the mid-term next week? We can all talk about it more as the week goes on, next week, Tuesday.

All right, so with that said, any questions about strings? We're gonna do a bunch more stuff today with strings and characters. But if there’s any questions before we actually dive in to things, let me know now. And if you could use the microphone, that would be great. One more time, take those microphones out, hold them close to your heart. Air and gear, they're lots of fun, they're your friend. Keep the microphone with you.

Student: Actually, sorry, about the mid-term, is it going – what’s the cutoff of the mid-term in terms of, like, caustes.

Instructor (Mehran Sahami): Right, so the mid-term for stuff you need to know, the cutoff will be Wednesday’s class. So, basically, you'll have a whole week of material that you won’t need to be responsible for that will be from this Wednesday up until the mid-term. The other thing, though, is to keep in mind that a few people have asked, well, do I need the book versus your lectures for the mid-term.

You need to know the lectures, and you need to know all the material from the book that is covered with respect to lectures, which is most of the material from the book. But there’s a few cases where we go over something very quickly in class, or I say refer to this page of the book or whatever. That stuff you're responsible for knowing. Stuff that

I’ve, like, explicitly told you you don’t need to know, like, polar coordinates, aren’t gonna be on the exam, okay.

So the exam will be more heavily geared towards stuff from lecture, but you still should know all the stuff from the book that we've kind of referred to in lecture as we've gone along, allrighty. All right, so let’s dive into our next great topic. Actually, it’s a continuation of our last great topic, which is strings. And so, if we think about strings a little bit, one of the things we might want to do with strings, is we want to do some string processing that also involves some characters.

So how are we gonna do that? One thing we might want to do is let’s just do a simple example to begin with, which is going through a string, and counting the number of uppercase characters in the string. And the reason why I’m gonna harp on strings a whole bunch – we talked about it last time – we’re gonna talk about it this time – guess what your next assignments gonna be. It’s gonna be all about string processing. So it’s good stuff to know, okay.

So we might want to have some function. Count uppercase. And that’s a function I’ve actually given to you in one of the handouts, so you don’t need to worry about jotting down all my code real quickly, but you might want to pay close attention. And what this does, is it gets past some string, STR, and it’s gonna count how many uppercase characters are in that string. So it’s gonna return an int.

And let’s just say this is part of some other program, so we'll call this private, although you could make it public if it was in some class that you wanted to make available for other people to use. So if we want to count the number of uppercase characters, what do we want to think about doing? What’s the kind of standard idiom that we use for strings? Anyone remember? What? We want to have a foreloop, [inaudible] somewhere over here.

Yeah, it’s just raining candy on you. We want to have a foreloop that goes through all the characters of the string, sort of counting through the character. So we can do that by just saying for N2i equals zero, “i” less than the length of the string, right. So STR.length is the method we use to get the length of the string, and then i++. And this is gonna loop through all the characters of the string. Okay, where actually it’s gonna loop through some number of times, which is the number of characters in the string.

Now we want to pull out each one of the characters, individually, to check to see if it’s an uppercase character. What method might we use to do that? Get a character out of a string in a particular position. Come on, I’m begging for it. Char at – it’s like where’d it go? It’s just gone, char at, and we'll just for the delayed reaction, we'll do it in slow mo. Anyone remember “The Six Million Dollar Man,” that show? No, all right. Get another man.

I’m just getting so old, I gotta hang it up. And the thing is, I’m not that much older than you. But it’s just amazing what big a difference a few years makes. So char CH is going to be from this string. Were gonna pull out the char at apposition i. So now, we've actually each. We're gonna loop through each character of the string, pulling out that character, and we want to check to see if the character's uppercase.

We could actually have an if statement in here to check to see if that CH is in between uppercase A and uppercase Z, which is kind of how you saw last time we could do some math on characters. We're gonna use the new funky way, which is to actually use one of the methods from the character class, and just say if. And the way we use the methods from the character class, we specify the name of the class here as opposed to the name of an object because the methods from the character class are what we refer to as static method.

There is no object associated with them. There are just methods that you call and pass into character. Is uppercase because this returns a Boolean, and will pass at CH to see if CH is an uppercase character, okay. If it is an uppercase character, okay. If it is an uppercase character, we want to somehow keep track of the number of the uppercase characters we have. So how might we do that? Counter, right.

So have some int count equals zero, up here, that I want initialized. Who said that? It came from somewhere over here. Come on, raise your hand. Don't be shy. It's a candy extravaganza. So if character is uppercase, CH then got count, we're just gonna add 1 to. Otherwise we're not gonna increment the counts. It's not an uppercase character. And then, we end the foreloop. So this is gonna go through all the characters of the string. For every character check seeks the uppercase.

If it is, increment our count, and at the end, what we want to do is, basically, return that count, which tells us how many uppercase characters were actually in the string, okay. Is there any questions about this? This is kind of like an example of the sort of vanilla string processing you might do. You have some string. You go through all the characters of the string. You do some kind of thing for a character of the string. In this case, were not creating a new resulting string. We're just counting up some number of characters that might be in the string.

So we can do something a little bit more funky. This is kind of fun, but it's sort of like, yeah, just basic kind of string and character stuff. Let's see something a little bit more funky, which is actually to do some string manipulation to break the string up into smaller pieces. And so what we want to do is replace some occurrence of a substring in a larger string with some other sting, sort of like when you work on [inaudible], when you do Find/Replace.

You say, hey, find me some little string, or some little work that's actually in my bigger document. I'm gonna replace it with some other word. We're actually gonna implement that as a little function, okay. So what this is gonna do, we'll call this replace occurrence just to keep the name short, but, in fact, all we're gonna do is replace the very first occurrence in a string. So we're gonna get past int some string, STR, and what we want to do is, basically, have some original string, which is the thing that we want to replace, with some replacement string.

So we're gonna get past three parameters here. Will call this RPL for replace, okay. Which is the large string, a piece of text that I want to replace some word in, the original word that I want to replace, and the thing that I want to replace it with, okay. And so what I want to do because strings are immutable, right. I can't change the string in place. I have to actually return a new string, which has this original replaced by this string.

So, this puppy's gonna return the string, and we'll just make this private again, although we could have made it public if we wanted to have it in the library that other people would use, or a class that other people would use, okay. So how might we think about the algorithm for replacing this original string with the replacement. What's the first thing we might want to think about that we want to do with the original string.

Do, do, do, do, a little concentration music. We want to find it, right. We want to see if this original string appears somewhere on that string, right because if it doesn't we're done. Thanks for playing, right, but that's actually the good things for playing. It's sort of like you got no more work to do. And there's, actually, some methods from the string class that we can use to do that.

So there's a string in the string class called "index of." And what index of does is I can pass it some string, like the original string I want to look up, and it will return to me a number. That number is the index of the position of the first character of this string if it appears in the larger string. So the larger string is the one that I'm sending the message to, and I'm asking it do you have this original string somewhere inside you.

If you do, return me the index of its first occurrence. And if you don't, it returns a negative 1. So I'm gonna assign this thing to some variable I'll call index, and first of all, I want to check to see if I have any work to do. If index is not equal to negative 1, then I have some work to do. If it is equal to negative 1, that means hey, you know what, you want it to replace this original string, inside string STR. That original string doesn't exist, so I got no work to do.

You just called, like, find and replace in the word processor, and the thing you wanted to find wasn't there, okay. So in that case, all I would do is I would just return STR, right. Sort of unchanged, if I assume that I'm not doing what's inside the braces. If I do find that string, though, I'm gonna get some index, which is not negative 1, which is the position of this original string.

So let's do a little example just to make this a little bit more clear what's going on. So if we were to call this function – do, do, do, do, do – and pass in the string, STR. So here's STR that we're gonna pass in. We'll just put it in a big box, and we'll say, at this point in life, everyone's just friendly. So we say Stanford loves Cal, right. Sometimes you have to distort reality in order t make an example. All right, so we have Stanford loves Cal.

That’s our original string, STR, and we might want to say, well, you know, this is, really, not always the way life is. Really, the way life is, is we want to replace the occurrence on STR of the word “loves” with kind of a more realistic example, like the word “beats,” right. So what we want to do – and then we're gonna – this is gonna be some string that comes back, will find it back to STR.

And the question is, when we call this, what index are we actually gonna find in here of the original string. So strings we start counting from zero. Zero, 1, 2, 3, 4, 5, 6, 7, 8. The nine is where the L is at. And it keeps going. And 11, 12, 13, just put these all together – 15, 16, 17 is the L and that would be the end of the string. Sorry, the numbers are a little bit small. But the key is this L is at 9, okay.

So when I call string index up original, it says there’s the word, or the string, “loves,” up here somewhere in the larger string. Yeah, it does. It appears at Index 9 so that’s what you get. So if I’ve just gotten Index 9, and what I want to do is construct some new string that, essentially, is going to have this portion removed from it, how do I want to do that.

What I want to think about is the way I construct that string, it’s from three pieces. The first piece is everything up to the word I want to replace. That’s Piece No. 1. The second piece is the thing that I actually want to replace, the string I’m replacing with, right. So this becomes Piece No. 2. And then, everything else after the piece I’ve replaced is Piece No. 3. So if I can concatenate those three pieces together, I’m going to essentially get the new string, which has this part replaced.

And the question is how do I find the appropriate indexes inside my larger string to be able to actually do the replacement, okay. So first thing that I’m gonna do here is say get me the first portion. So what is, essentially, the substring of the original string up to this L position. So the way I can do that is I can say STR, substring, and I’m gonna get the substring starting at zero 'cause I want to start at the beginning of the string, and I want to go all the way up, but not including the L.

That means the last position in substring. Remember, in substring you give it two indexes. You give it the starting point, and the position up to, but not including that last chapter. That’s Position 9. Where am I getting Position 9 from this thing? From index, right. Index says where does love start. It starts at Position 9. I’m, like, hey, that’s fantastic. So zero up to index, or zero up to 9 is Stanford and the states. It does not include the L. So I get that portion.

Then I say well, to that – I’m not done yet, so premature [inaudible] in there. Always gotta watch out for that, bad time. So what we're gonna add to that is we're gonna add the string that we want to replace in here, “beats,” which happens to be the string called the replacement, or RPL. And then to that, we want to add one more string. And that’s, essentially, everything from after “loves” over, to get that third piece, okay.

So what I want to know is what’s the index of the position at which I need to get characters over to the end. That happens to be Position 14. What is 14 equal to, relative to the kinds of things I have over here? It’s index 'cause I have to first get over to the 9, then I need to jump over the length of this thing, which is the length of my original string. So if I add to index, what’s my original dot link, what that gives me is the index from which I want to take a substring over to the end of the string.

So if I want to take a substring, this becomes an index to the substring function, or the substring method. And so from the string, what I do is I take the substring, starting at Position 14. Notice I haven’t given a second index here. In this case I gave two indexes. I gave a start and end position. Here I just gave one index, and what happens if I only give one index? It goes to the end.

So that’s part of the beauty is a lot of times you just say, hey, from this position go to the end. And so that’s what I get when I put all these string things together. And what I need to do is these three things are just pieces. I’m concatenating them together. I assigned them back to STR. And then, when I return STR here, I’ve gotten those three pieces concatenated together. Is there any questions about that? Un huh.

If love appears more than once, index has just returned the index of the very first occurrence. There’s actually a version of index sub that takes two parameters. One is the thing you're looking for, and the second is from which position you should start looking for it at. And so you could actually say look for love starting at Position, you know, 13, and then it wouldn't actually find love in the remainder of the string. So there’s a different version of index of, but index of always returns the index of the very first occurrence of the string you're looking for in that string.

So let’s actually do a little example of this in a running program. Do, do, do, do, do. And we'll do replace occurrence. And one thing that actually goes on at Stanford, which I thought was an interesting thing when I got here professionally, is we don’t like to speak in full terms. So if we want to Stanfordize some strings, we do all these string replacements.

We sort of say, you know what, if you have Florence Moore in your string, that’s really FloMo. And Memorial Church is memchu; AmerSc, [inaudible]; psychology is psyche; economics, econ; your most fun class, CS 106A. So it’s just what Stanford’s all about. And so if we go ahead and run this, right. Here’s the function we just wrote. Here’s our little friend, replace first occurrence. Over here we called it replace occurrence. I’m being explicit and saying it’s only replacing the first occurrence.

You could think of a way to generalize this to replace all occurrences in a string if you wanted to. But I didn’t give you that version 'cause I might give you that version on another problem set at some point. So what we're gonna do is we're gonna ask the user, enter a line to Stanfordize. Notice I want to put Stanfordize inside double quotes. So I put it in these characters, /quote, which just means a single, double-quote character. That’s how I print double quotes.

So it says read line for Stanfordize in quotes. I want to keep reading lines and Stanfordizing them until the user gives me an empty line. How do I do that? I check to see if the line the user gives me is equal to a quote-quote. So if it's equal to a quote-quote, it's equal to the empty string. That means, hey, you entered in – if we ask the user for a string, they just hit enter. They didn't enter any characters. That's the empty string, so we would break out the loop. It's our little loop and a-half concept.

Otherwise, we say at Stanford we say, and we Stanfordize the line. And when someone's finally done, we say thank you for visiting Stanford, ha, ha, ha. That'll be $45,000.00. [Laughter]. All right, so it's money well spent, trust me. Really. Okay, so replace occurrence string we want to run, and we come along, and it's running, it's running, it's running. Sometimes my computer's running a little bit slow.

I notice this weird thing last night. I'm gonna tell you a story while the computer's actually running. I couldn't type N's on my keyboard for some reason. And then I reset my computer, and I could. So at this point, I don't know if I can type N's. So let's just hope we can. So I live in – oh, I got the N – Florence – you should have been here last night. I was like, N, N, and I wasn't getting it – Florence Moore, major in economics – I can't even type today – and spend all my time on my most fun class.

And so, at Stanford we say I live in FloMo, major in Econ, and spend all my time on CS 106A, okay. And now, I hit return, Thank you for visiting Stanford. Go home. All right, so that's kind of a simple version of replace first occurrence. And notice you can actually replace multiple things in the same string, as long as the string that you're doing the replacement on you assign back to itself. And then we kind of do all bunch of these replacements in a row, okay. Is there any questions about that? Are you feeling okay about doing replacement. All right.

So now, it's time for something completely different. Although it's not completely different, it's just kind of different. And the idea is sometimes – and I always say that – sometimes you want to do this. Yeah, 'cause sometimes you want to do it, and other times you don't. Sometimes you feel like a nut. Sometimes you don't. Oh, man, I gotta start watching TV in this decade.

So, tokenizers. What is a tokenizer? A tokenizer is something, as they say it's a computer science term. All a tokenizer is, is we have some string of text. What we want to do is break it up into tokens. That's called tokenization. So you might say, Marilyn, what is a token? Like, last time I remember what a token was, is when I gave a dollar at the arcade and I got back, like, ten tokens instead of quarters. And you're like, yeah, Marilyn, I never did that. I had an XBox.

All right, so a tokenizer – anyone ever go to an arcade? All right, just checking. All right a token, basically, is a piece of string – a piece of string – is a string that has on the two sides of it, white space. So if I say, hello there, Mary, hello there and Mary are tokens. They are something that we refer to as delimited by what space, which means there is either spaces, or tabs, or returns, or whatever, in between the individual tokens.

We like to think of tokens as words, but computer scientists say token. Token is a more general term 'cause if I actually said hello there comma Mary, the “there comma” might actually be considered one token by itself 'cause it’s just delimited by space. Here’s a space here and has a space there, so the comma’s in there. And you would think why comma’s not part of the word. Yeah, that’s why we call them tokens and not words.

So if we want to tokenize, there is a library that we can use in Java that actually has some fun stuff in it for tokenization. And that’s Java util, so we would import Java.util.*, and what we get for doing that, is we get something called the string tokenizer, which is a class that we can use to tokenize text. All right, so we get this thing called the string tokenizer.

How do I create one of these? Well, I paste string tokenizer as the type 'cause that’s the class that I have, and I’ll call it tokenizer equals I want to create a new tokenizer. So I say new string tokenizer, and the question that comes up here is well, what is the string you're gonna tokenize? That is the string that we passed to the string tokenizer’s constructor when we create a new one.

So we might have some line here that we passed in. And now, line is just some string that maybe we got from the user for example by doing a read line. Maybe we were unfriendly and didn’t give the user a prompt. We just like, if a blinking comes up, and there’s like oh, I gotta turn and write something. It’s just like when you're writing a paper, right. The blinking cursor comes up and there’s nothing there. You just gotta fill it in.

So you write some line, and then we can say, hey, string tokenizer, I’m gonna create a new one of you, and the line I want you to tokenize is this line that I’m giving you to begin with. So once you get that line, there’s a couple of things you can ask the string tokenizer. One of them is a method that returns a booleon, which is called has more tokens.

And the way this puppy works is you just ask this string tokenizer, like you would say tokenizer dot has more tokens, like; do you have more tokens? Have you processed the whole string yet? So if you’ve just created the new line, and this line is kind of sitting here like that, and it’s saying do you have any more tokens. Yeah, I got tokens, man. I got tokens up the wazoo. You want tokens, I’ll give you tokens. And so, has more tokens [inaudible] true.

If you process the whole string, when you will see when we get there, it’ll say no, I don’t have any more tokens. How do you get each token? Well, you ask for next token. And what next token does, when you call the tokenizer with next token, is it gives you the next token of the string that it’s processing, as a separate string.

So if I started off the tokenizer with this line, I say hey, do you have more tokens. It says yeah. Well, give me the next token. So what it will return to you is hello. And it will be sort of sitting here waiting to give you the next token. You can ask if you have more tokens. It says yeah, give me the next token.

It will give you “there” and the comma 'cause the default version of the tokenizer, the only think that delimits tokens – delimit is a funky word for splits between tokens – are spaces, or tabs, or return characters. But for a single line, you won’t have returns in that. And then you said you had more tokens. Yeah, give me the next token. It will give you “Mary” as a token that’s sitting here.

And then when you say do you have more tokens, that’s all, okay. And at that point, you shouldn’t call next token. You can if you want. You can experiment with this if you want to experiment with random error messages, but there’s no more tokens to give you. It’s all out of love. It’s so lost without you. It has no more tokens.

Yeah, Air Supply. Not that I would recommend that you have to listen to Air Supply, but sometimes you hear a song and you can’t get it out of your head as much as you wish you could. Sometimes selective brain surgery would not be a bad thing, but that’s important right now. What is important right now is how do we put all this together at the tokenizer line. So let me show you an example of the tokenizer. This one’s very simple.

All we're gonna do here is we're gonna ask the user – I’ll just scroll over a little bit. We're gonna ask the user to enter some lines to tokenize and we're gonna write out the tokens of the string R, and then we're gonna call the message sprint token. What sprint token’s gonna do, it’s gonna take in the string you want to tokenize.

It creates one of these string tokenizers – I’m so lost without you. [Laughter]. Can we make Marilyn snap? No. I know it’s like great fun to listen to when you're, like, 14, and you just broke up with a girlfriend for the first time. And then, after that, you want to kill the next time you hear it. Fine, So, tokenizer. I’m glad we're having fun though.

So what I’m gonna do is I’m gonna count through all the tokens. So I’m gonna a foreloop interestingly enough. Here’s something funky. I’m gonna have a foreloop, but the thing I’m gonna do in my foreloop, my test, is not to check to see if I’ve reached some maximum number. But my test is actually gonna be to see if tokenizer has more tokens.

So I have a foreloop that’s just like a regular foreloop, but I start off with a count that’s equal to zero, and you're like that looks okay. I do a count ++ over here, and you're like that’s okay, what are you counting up to Marilyn. And I say I’m counting up to however many tokens you have. And you go, oh, interesting. So my condition’s to leave, or to continue on with the loop, is tokenizer has more tokens.

If it has more tokens, then I’m gonna do something here to get the next token. I’m gonna keep doing this loop. But what the counter’s gonna give me is a way to count through all my tokens. So I can write out token number count, and then a colon, and then write out the next token that the tokenizer gives me. Is there any questions about there? Let’s actually run this puppy [inaudible]. Do, de, do. You can feel free to keep singing now if you want, if you want.

All right, so we're gonna do our friend. What’s our friend called? The tokenizer example. Do, do, do, do, we're running the tokenizer, interline’s tokenized, so I might say “I, for one, love CS.” We're very formal here. And it says the tokens of the string are on notice. It got the “I” and the comma together as one token because as we talked about, spaces are the delimiter. And so “for” and then “one” with the comma, and “love” and “CS,” and that’s all the tokens we got.

And so at this point you might be thinking, yeah, man, that’s great, but you know what. I really don’t like punctuation. And sometimes I don’t like punctuation, but I can’t stop the user from using punctuation because even though I don’t like to be grammatically correct, they do. So how do I prevent them from being grammatically correct as well, which is kind of a fun thing to do.

What you can say is hey, what I want to do is change my tokenizer, so that it not only stops at spaces, but it’s gonna stop or consider a delimiter, any of this list of characteristics that I give it. So you give it a list of characteristics as a string. So here I’m gonna give it a comma and a space, okay. And this version of the string tokenizer constructor, what it will do is it will actually tokenize the string.

But think of the thing that you're using as your delimiter, or what chops up your individual tokens as either a comma, or a space, or anything you want to put in that string there. Each of the individual characters in that string is treated as a potential delimiter. So if you say “I for one love CS,” ah, no commas. Why? Because commas are considered delimiter.

So it just gives you everything up to a comma or a space, and you could imagine you could put in period, and exclamation point, and all that other stuff, if you just want to get out the non punctuation here. So tokenizing is something that’s oftentimes useful if you get a bigger piece of text, and you want to break it up into any individual words, and then maybe do something on those individual words, okay. Any questions about tokenization? Hopefully, it’s not too painful or scary. All right.

So the next thing I want to do, will just pay for the smorgasbord of string, is I want to teach you about something that’s really gotten to be an important thing about computer science these last few years, which is, basically, this idea known as encryption. And encryption is something that’s been around for thousands of years. All encryption is, is it’s kind of like sending secret messages.

You have some particular message. You want to send it to someone else, but you want to send a secret version of that message. And people have been doing this for thousands of years, actually, interestingly enough. They just didn’t have very good methods of doing it until about the last, oh, 50 years. But you know they did it for a long time, and people broke encryption.

As a matter of fact, there’s this really interesting book by Simon Singh. I’ll bring in a copy, perhaps next class, if you're really interested, about the whole history of encryption.

It goes back thousands of years, and how, like, wars, and queenships, and kingships, and stuff, were, basically, lost in one on the strength of how well someone could break a piece of code.

But the basic idea of encryption, and it probably dates back even further than this, but one of the most well-known ones is something that's known as the Caesar cipher, not to be confused with the salad. But the basic idea with the Caesar cipher – I picked up the wrong newspaper – the Caesar cipher is that what we want to do is, basically, take our alphabet and rotate it by some number of letters to get a replacement.

What does that mean? That's just a whole bunch of words. So let me show you a little slide that just makes that clear. So in Caesar's day – I will now play the role of Caesar. I actually considered wearing a toga to class today. I just thought that was fraught with way too much peril. So I just decided to bring my little Caesar crown. And that's what I'm trying to find my little crown of reason stuff, but I couldn't. So I just got a little hat. [Laughter].

And so the basic idea – say you are Caesar – well, I did crown myself, actually. I knew someone here could actually take the crown from [inaudible]. That was Napoleon, a whole different story. I really like to take history and mix it up. It's just to see if you're actually paying attention. All right, the basic way the Caesar cipher works is we take our original alphabet. Here's all of our letters from A through Z.

We take that whole alphabet, and we shift it over some number of letters. Like let's say we shift it over three letters. So I take this whole thing, I shift it over three letters, so now the D lines up over here where the A should have been so I've shifted over these bottom characters. And the characters that kind of went off the end here like the A, B, and C, were kind of like whoa, we're going off the end. Where do we go? We just kind of shuffle them back around over here.

So the basic idea is we're gonna rotate our alphabet by N letters, and N is 3 in the example here, and N is called the key. So the key of the Caesar cipher is how many letters you're actually shifting. And then we wrap around it again. And now, once we've done this little wraparound, we take our original message that we want to encrypt. That's something that's referred to as the plain text. The plain text is your actual original message.

And we want to encrypt that or change it to our cipher text, which is what the encrypted message is, by using this mapping. So every time an A appears in the original, we replace it by a D. And a D appears in the original, we replace it by G, and a C appears in the original, we replace it by an F, etc., for the whole alphabet. Is there any questions about the Caesar cipher?

This is actually an actual cipher that, evidently, historians tell us that Caesar used in the days of yore. And you know, evidently, he was killed, so it didn't work that well. But you know most people, that's one of the things that when you were a little kid, and you had like the Super Secret Decoder Ring, you were probably getting a Caesar cipher. All right, any questions about the basics of the Caesar cipher.

So what we're gonna do is let’s write a program that actually can be able to encrypt and decrypt text according to a Caesar cipher, and we'll do it doing pop-down design. So we'll actually just do it on the computer together 'cause it’s more fun that way. And because I’m Caesar, I will drive. So we're gonna have my Caesar cipher, all right. And I just gave you a little bit of a run message here. It’s kind of the very beginnings of the program.

But all this does – it’s not a big deal. It says this program uses a Caesar cipher for encryption. It’s going to ask for the encryption key. That means it’s asking for the number by which it’s gonna rotate the alphabet to create your Caesar key, or to create your Caesar cipher, and that’s just our key that’s an integer. So our plain text, that’s the original message that we want to encrypt. We ask the user for the plain text, so we just get a line form the user.

And then what we're gonna do is we're gonna create our cipher text, or the encrypted text by calling a function called encrypt Caesar. We’re sort of giving a directive. It’s kind of like an inquisitive tape. Encrypt Caesar, and we give it the plain text, and we give it the number for the key that we want it to encrypt using. And then, hopefully, that will give us back the encrypted string, and we're just gonna write that out, okay. S

o how do we do this encryption? All right, so at this point, and it should be clear that the thing we want to write is probably encrypt Caesar. So what we're gonna do is we're gonna write a pleasant message, and what is this puppy gonna return to us? String, right 'cause that’s what we're expecting, the encoded version of this particular message as a string. So we'll call this encrypt Caesar.

And what’s it getting past? It’s getting past the string, which we'll just call STR, and it’s getting past an integer, which will we will refer to as the key. So if I want to think about doing the encryption, right, what I’m gonna do is, on a character-by-character basis, I want to do this replacement. I want to say for every character that I see in my original string, there is some shifted version of that character that I want to use in my encrypted string.

So in order to do that, I’m gonna use my standard kind of string building idiom, which says I start off with a string, which I’ll call results, which starts of empty, right. It says, quote, quote, empty string. And I’m gonna do a foreloop through my string that I’m giving to encrypt. So up the string’s length, I’m just gonna count through and get each character. So I’ll so sort of a standard thing.

I’m gonna say CH and I’m gonna essentially get the character that I want to get from the string, so I’ll say STR.char@chat@char@I. So I’ve now gotten my character. I want to figure out how to encrypt that character, okay. So I think to myself, wow, gee, while encrypting the character involves all this stuff, doing the shift and all that, that’s kind of complicated. Maybe I should just create a function to do it. All right, that’s the old notion of pop-down design. Any time you get somewhere, well, you're, like, wow, that’s kind of complicated.

Maybe I don’t want to stick this all in here and figure it out. But it’s the smaller piece, which is just dealing with a single character instead of dealing with the whole string. Let me write a function that will actually do it, or a method that’ll actually do it. So what I’m gonna do, is I’m gonna append to my results what I get by calling encrypt, a single character.

So I’ll just call it encrypt char and what I’m gonna pass to it is the character that I want encrypt, and I need to also pass to it the key so it knows how to do the appropriate shifting to encrypt that character. And after it does this encryption, I’m just gonna say hey, if you’ve successfully encrypted all of your strings, what I want to do is return, RTN, my results, right. That’s your standard string idiom.

I start off with an empty string. I do some kind of loop through every character of the string. I’m gonna do the processing one character at a time, and return my results. Everything in that function that you see, or in that method that you see, except for that one line, should be something you can do in your sleep now. You’ve seen it, like, over and over. We just did it a couple of times today. We did it a couple times last time. It’s the standard kind of thing for going through a string one character at a time.

And now, we reduced the whole problem of encrypting a whole string to the problem of just encrypting a single letter. So what I’m gonna have in here is private, and this is gonna return a single character called – and this puppy’s called encrypt char. And it’s gonna get passed in some character to encrypt as well as the key that it’s gonna use to encrypt it. And now I want to figure out how do I encrypt that single character.

So what’s something I could do to think about how this character actually gets encrypted. How do I want to do the appropriate shifting of the character. So let’s say I’ve gotten an uppercase A. Let’s assume for right now all my characters are uppercase. As a matter of fact, that’s a perfectly fine assumption to make.

The solution you’ve gotten to, it assumes all the characters are uppercase, so assume all the plain text is uppercase, and I want to return to the encrypted cipher text also in uppercase. Let’s say I’ve gotten an uppercase A, okay. And my T is 3. So I want to do, is take that A somehow, and convert it to a D. How do I do that?

Student: [Inaudible].

Instructor (Mehran Sahami): Un huh. I want to add 3 to the character. Now the only problem is I might go off the end of the character. If I just add 3, and I have a Z, I’m gonna – if I just have the A and go to D, that works perfectly fine, but if I have a Z, I’m gonna get something like an exclamation point, or something I don’t know 'cause I go off the end of the character. So I need to do slightly a little bit more math. And what I’m gonna do is say take this character, and subtract from it uppercase A.

That’s gonna tell me which character in the alphabet it is, which number character it is, right. Now, if I add the key, what I get is the number, or the index, of the shifted character. So if I had an uppercase A, and I subtract off uppercase A, I’m gonna get a zero. I now add the key, so I get 3. And you might say, well, if you just convert that to a character, you get a D. That’s perfectly fine. Yeah, but if I had a Z and I subtract off an uppercase A, I get 25.

If I add 3 to 25, I get 28, which is now outside the bounds of the alphabet. How do I wrap around that 28 back to the beginning of the alphabet. Mod it by 26, or we do with the remainder operator by 26, right. So what that does is it says if you’ve gone off the end, basically, when you divide by 26 and take the remainder, if you’ve gone off the end, it kind of gets rid of the first 26, and wraps you back around the beginning.

So if I do that, this will actually work to get me the position of the character wrapped around, and once I’ve gotten the position of the character, here’s the funky thing. I need to add the A back in because if I have, let’s say, an uppercase A to being with, and I subtract out uppercase A, that gives me zero. I add the key. That gives me 3. I do the remainder by 26.

Three divided by 26 as the remainder is still 3. So now I have the number 3. I need to get that 3 converted to the letter D. How do I do that? I add the letter A to that 3, okay. Is there any questions about that?

Now, the final funky thing that I need to do, is if I want to assign this to a character, I can't do this directly. Notice if I try to do this directly, I get this little thingy here. And you might say Marilyn, what’s going on? Like you told me characters were the same as numbers, and everything I’ve done so far has to do with numbers, so why can’t I assign that to a character?

And this little error message comes up. And this has to do with the same thing when we talked about converting from real values to integers. Remember when we went from a real value to an integer. We said you'll lose some information if you try to truncate a real value, like a double to an integer. So you explicitly have to cast it from being a double-splint integer. Same thing with characters and integers. The set of possible integers is huge. It’s like billions and billions. The set of characters is much smaller than that.

So if you want to go from an integer back to a character, you need to explicitly say convert that integer back to a character. So we need to explicitly do a cast here back to a character. And if we do that, then we're happy and friendly. Did I get all my friends right? One, two, three, one two three. All right, why is this still unhappy? Oh, duplicate variable CH, yeah. Let me call this C.

Actually, let me make my life easier. This is a thing I just want to return, so I’m just gonna return it. Do, do, do, do, do. I won’t even assign it to any temporary variable. We'll just return it 'cause now I’m upset. No, I’m really not upset. We're just gonna return it.

So, hopefully, that will give us our little Caesar cipher. So let’s go ahead and run this, and see if, in fact, it’s working. Any questions about this while this is running?

I’ll sort of scroll this down a little bit so you can see what’s going on for that single character. So this was my Caesar cipher. So we say, et tu, brute. Illegal number format. Yeah 'cause that’s not the thing I wanted to encrypt. My encryption here is 3, then I will give it the plain text I was to – everyone’s like what did he do. Sometimes it’s the obvious that’s wrong, and you just need to read. All right, there we actually go.

Now, there’s a little problem here. See, the little problem is the spaces actually got encrypted. We don’t want to encrypt spaces. We only want to encrypt things that are actually valid characters. So we're not quite done yet. What we need to do is come back over here and say, hey, you know what, for my encrypt character, I wasn’t quite as bright as I thought I was. I need to make sure this thing’s actually in uppercase character before I try to encrypt it.

So we can sort of do that if I just call my little friend character. And the thing we want to say is, is uppercase, and I’ll pass at CH. So if it’s already – if it’s an uppercase character, then I’ll return this. Otherwise, what I’ll do – I’ll tab this in – is I will just return CH unchanged. So if I’ve gotten something that isn’t actually a character, then I’ll return – do, do – why is this unhappy again. Oh, semicolon, thank you. All right.

Student: [Inaudible]

Now I got an extra one. Notice it doesn’t give me an error on the extra one 'cause, actually, semicolon without a statement is the emsin statement. It’s perfectly fine, but thank you for catching the straight semicolon. So we'll go ahead and run this, and we'll try our friend, et tu, Brute, again. Sometimes it’s all about texting, and so we have et tu, Brute, and now we're okay 'cause we're not encrypting anything that is not a letter.

So sometimes we think we're okay. We need to go back and just make sure we actually do the texting. Any questions about this? If this all made sense to you, nod your head. If this didn’t make sense to you, shake your head. Feel no qualms about shaking your head. If you're someone in the middle, just stare and stare at me. No, if you're someone in the middle, shake your head. Okay, un huh.

Student: [Inaudible].

Instructor (Mehran Sahami): Why don’t I need an L statement, like, say here? 'Cause if I hit the return, I return from the function immediately and I never, actually, get it down to this return. So if I hit this return statement, I’m done with the method. As soon as I hit that return, it doesn’t matter if there’s any more lines in the method. I’m done. I actually return out.

So the one other thing we might like to do with this that doesn’t quite actually work right now. Let’s actually try running this, then I’ll show you what happens, just to show you that it’s bad time. If I actually encrypt something like et tu, Brute, and I want to decrypt it, I might say hey, try to use minus 3 as your key, and if I try to put in the text – I don’t even remember what the text was that I wanted to encrypt. I guess this funky thing with questions marks, and it’s just not working to move in the negative directly.

So I want to allow for my Caesar cipher to also be able to decrypt information, which means if I got a Caesar cipher by encrypting with a key of 3, if I give it the text that’s been encoded, and I give it the key minus 3, it should shift it back three letters and actually work for me. So how do I do that? Well, it’s something that has to deal with each individual character.

If I want to encrypt each individual character, I need to figure out what’s the right way of using the key, okay. Think about a key of minus 3. What’s a key of minus 3 equivalent to? A key of 23, right. A few people mumbled it, so we'll just throw out some candy. If I want to go 3 in the opposite direction, if I want to go 3 sort of this way, as opposed to this way. It’s the same thing as going 23 characters in the opposite direction.

So if I want to think about doing that, I can say, if my key is a negative number, so if my key is less than zero, there’s some shifting I need to do of the key to actually get this puppy to work. So if my key is less than zero – as a matter of fact, I’m gonna do this once down here. So rather than doing it, and encrypting each character, I’m gonna do it over here by saying you know what, once I shift my key over, I want to use that same key to encrypt all my characters.

So I want to do the shifting just once up here. It makes sense to do it once for the string, and then I’ll use my updated version of key. So here’s what I’m gonna do. I’m gonna say take the key, and the way I’m gonna update key is I’m gonna say it’s 26. And this looks a little bit funky, but I’ll explain it to you in just a second – modded by 26.

And you might say Marilyn, why do you need all this math to actually pull it off 'cause if you do something, you could say why can’t you just take 26 and subtract from it your key. So if you want to say minus – or add toward your key. So if you want to have a key of minus 3, isn’t that just the same as adding minus 3 to 26. You'll get 23, aren’t you fine. Yeah, that’s fine for sufficiently small values of key.

So if this thing actually is minus 3, minus, minus 3 gives me 3, and if I were to – oh, missing a minus in here. Sorry, my bad. I had two minuses. I want to have another minus right there. So if key is minus 3, and I take a negative of minus 3, that gives me 3. Twenty-six minus 3 by itself would give me 23, which is the value I care about and that’s perfectly fine.

But what happens if this key that someone gives me is something, for example, that’s larger than 26. That’s kind of bad time because if I subtract a number that’s larger than 26 from the 26, so if this happens to be minus, let’s say 27, and I say minus minus 27 is positive 27. And I subtract 27 from 26, I get minus 1. That’s bad time.

So the reason why I have this 26 in here, is it says first take the key. They gave me some negative value. Take the negative of that, which gives you some positive value. When you mod it by 26, you will guarantee that that value they've given you is less than 26 'cause if it was 26, 26 mod, 26 is zero. Something larger than 26 gives me a remainder.

So as long as I mod by 26, I will always get back the appropriately mapped value, less than 26, and then I will subtract that [inaudible]. So just to make sure this actually works, what I'm gonna do is in my main program, I'm gonna say encrypt Caesar using this key. And then, do, do, do, so I have some cipher text.

I'm going to now – well, actually, let me write out the cipher text so I'll still use this print link. And then, I'm going to have some other string, new plain. A new plain is just going to be doing encrypt Caesar on my cipher text, so that should be my encrypted text with the negative of the key. So I want to, essentially, switch back to what I've got. And so I'll have print link new plane quote dot whatever the new – man, I cannot type to save my life – L print link. Thank you.

So now I run this puppy in our final moment together, 3 et tu, Brute. Well, at least I got it back even though I misspelled it. I got my mixed-up characters, and then I got my new plain text, which is the same as my original text, which I got just by, essentially, shifting in the negative direction. So any questions about that. Allrighty, then we're done with strings for the time being, and I'll see you on Wednesday.

[End of Audio]

Duration: 50 minutes