Intro to Computer Science

Lesson 01 Notes

How to Get Started

Introduction

Welcome to CS101. I'm Dave Evans. I will be your guide on this journey. This course will introduce you to the fundamental ideas in computing and teach you to read and write your own computer programs. We are going to do that in the context of building a web search engine. I'm guessing everyone here has at least used a search engine before. Like Google, DuckDuckGo or even my personal favorite - DaveDaveFind. You type in what you are looking for, and voila - in literally a blink of an eye, about a tenth of a second, back come the results. This might not be enough to make you wise, but it is pretty amazing. A goal of this class is to turn some of the magic of the search engine into something a bit more understandable. Our biggest goal though is to learn about computer science. Computer science is about how to solve problems, like building a search engine, by breaking them into smaller pieces and then precisely and mechanically describing a sequence of steps that you can use to solve each piece. And those steps can be executed by a computer. For our search engine the three main pieces are: finding data by crawling web pages, building an index to be able to respond quickly to search queries, and ranking pages so that we get the best result for a given query. In this course we will not get into everything that you need to build a search engine as powerful as Google, but we will cover the main ideas and learn a lot about computer science along the way. The first three units will focus on building the web crawler. We will talk more about that soon. Units 4 and 5 will cover how to respond to queries quickly. And unit 6 will get into how to rank results and cover the method Google uses to rank pages, that made it so successful. But first, let's talk about how to build a web crawler that we are going to use to get data for our search engine.

Advice from Sergey Brin



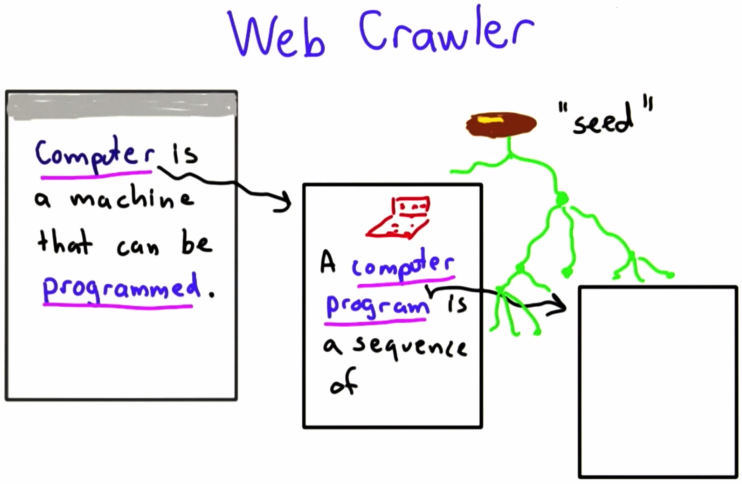
DE: Let's get started by asking Sergey Brin, the co-founder of Google, what the most important thing is in building a search engine.

ST: So what's your take on how to build a search engine? You have built one before, right?

SB: Yes, I think the most important thing if you want to build a search engine, I think is to have a really good, fun corpus to start with. In our case we used the world wide web which at the time was significantly smaller than it is today. It was also very new and exciting there were all kinds of unexpected things there.

Overview of the Unit

The goal of the first three units in this course is to build a Web crawler that will collect data from the Web for our search engine. And to learn about big ideas in Computing by doing that. In Unit 1, we'll get started by extracting the first link on a web page. A Web crawler finds web pages for our search engine by starting from a "seed" page and following links on that page to find other pages. Each of those links lead to some new web page, which itself could have links that lead to other pages. As we follow those links, we'll find more and more web pages building a collection of data that we'll use for our search engine. A web page is really just a chunk of text that comes from the Internet into your Web browser. We'll talk more about how that works in Unit 4. But for now, the important thing to understand is that a link is really just a special kind of text in that web page. When you click on a link in your browser it will direct you to a new page. And you can keep following those links as a human. What we'll do in this Unit is write a program to extract that first link from the web page. In later units, we'll figure out how to extract all the links and build their collection for our search engine



First Quiz

So we're going to have many quizzes throughout each unit. The point of a quiz is to check that you understand what we've covered. Some of the quizzes will be fairly straightforward just to see if you've followed what we said. Other quizzes will be more challenging and require you to put together several ideas that we've covered. The quizzes don't count towards your grade. You'll be able to try them as

many times as you want until you hopefully get the answer right. It certainly will be valuable to try to get the answer right the first time. But they shouldn't be stressful. They're meant to keep you engaged in the lecture, make sure you're understanding things. After each quiz there will be an explanation of the answer that will often go in more depth than just answering the quiz correctly. So for the first quiz and to get started practicing doing quizzes, our question is: "What is the goal of Unit 1?" There are 4 choices. You can check all the choices that you think are good answers. So the goal is to get started programming, the second choice is to learn some important computer science concepts, the third choice is to write some code that extracts a link from a web page, and the fourth choice is to write code to rank web pages. So that's the first quiz. Check as many answers as you think are correct.

[x] Get started programming
[x] Learn some important computer science concepts
[x] Write code to extract a link from a webpage
[ ] Write code to rank web pages

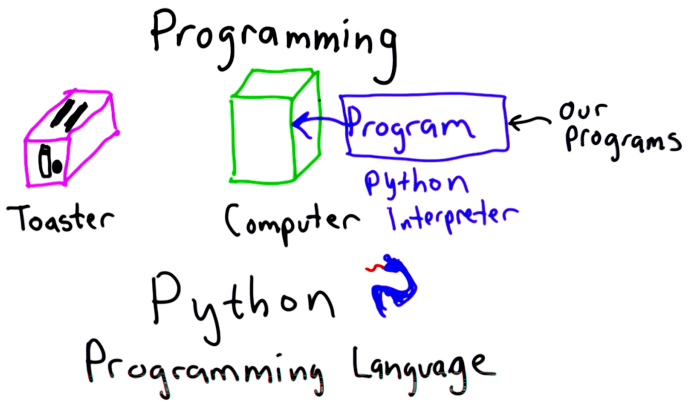
Answer:

So the answer is the first three are correct. We want to get started programming, and we'll get started programming very soon. The most important thing is to learn some computer science concepts, and we'll do that as we get started programming as well as along the way. And the third goal is to write code to extract a link from a web page. That'll be the first thing that we do to start building our search engine, and collecting the corpus that we'll need for the search engine. The fourth choice is not true. We're not going to get to the point where we write code to rank web pages this unit. That's something that we'll be covering in unit six.

Programming

So let's get started with programming as it is the core computer science. Most machines are designed to do just one thing. The drawing here is supposed to be a toaster, more of a representational drawing than an accurate one. With the toaster here we can do more than one thing. We can put different things in it. We can toast bread and muffins. Maybe it has something we can use to change its behaviour. We can change the setting to make it toast for longer or shorter. But it is pretty limited what it can do, everything it can do is a variation on this basic functionality that it was designed for. The

basic process of putting toast in, heating it up and getting the toast to pop back out. If we want to change it's behaviour to do something really different we would have to physically alter the machine. We could maybe take the parts out, and put them together in a new way. If we are really creative we could make a bicycle from the toaster. That would be a pretty big project though. So without a program a computer is even less useful than a toaster. You can't do anything without a program. The program is what tells the computer what to do. And the power of the computer is, unlike a toaster which is only designed to do a few things, a computer can do anything. A computer is a universal machine. We can program it to do essentially any computation.



So anything we can imagine, anything we can figure out how to write a program for we can make the computer do. And what the program needs to be is a very precise sequence of steps. The computer by itself doesn't know how to do anything. It has a few simple instructions that it can execute. And to make a program do something useful we need to put those instructions together in a way that it does what we want. So we can turn the computer into a web browser, into a server, into a game playing machine, into a toaster, without anywhere to put the bread. But it can do anything we can imagine and at least any computation we want to do. The power of the computer is that it can execute the steps super super fast. So we can execute billions of instructions in one second. The program gives us a way to tell the computer what steps to take.

So there are many different languages for programming computers. We are going to learn the language Python, like the snake. It is named after Monty Python. The important thing about Python is that it gives us a nice high level language that we can use to write programs. Instead of our program running directly on the computer, the programs we write will be an input to the Python program which runs on the computer. What Python is is called an interpreter. That means it runs our programs, it interprets them, executes the programs that we wrote in Python language by running a program in a language the computer understands directly.

Quiz: What Is a Program

Now it's time for a quiz to see if we understand what a computer program is. So which of the following are computer programs? Check all that apply. A web browser like Firefox or Chrome or Internet Explorer. The Python interpreter. The Python code that you'll write in this class, and you'll start writing Python code very soon. A slice of toast. The calendar application on your mobile phone. Check all of the ones that are computer programs.

[x] A web browser
[ ] A slice of toast
[x] The Python interpreter
[x] Calendar app on a phone
[x] The python code you will write in this class, print ‘Hello!’

Answer:

The answer is, all of these are computer programs except for the slice of toast. In general, if you can eat something it's not a computer program. The programs that you write in this class will be Python code. Those will be input to another program, which is a Python interpreter that follows instructions in your code, and it does that by following the instructions in its code. And you'll be able to run all that using your web browser.

Getting Started with Python

Now it is time to get started programming in Python. For this class, you don't need to install any software. You can run your Python programs directly using the web browser. After the video finishes, you'll see a window that looks like this that you can use to write and execute Python code. In the top part, which we call the editor, you can enter Python code. For example, we'll enter the code print 3. Print is the Python command that prints something out. After the print we have an expression and the value of that expression is what gets printed. This could be a simple thing like the number 3. It can also be a more complicated expression, doing some arithmetic. We write our code up here, when we click run... It will run the code, we'll see the output. And here, the first line ran and printed 3. The second line printed the result of this computation which is 101. If you don't believe me, you can try it yourself. We'll talk lots more about Python expressions soon but I think you actually know enough now to already be able to write your first program. So it's time for the first programming quiz.

Programming Quiz: First Programming Quiz

So, now let's get started running code in the Python interpreter. For this class you don't need to install any new software. You can run Python right in your web browser, and you should see something like this in your web browser once the video stops. There are two parts to this, there's the environment where you can edit code, so let's type code to print 3. So print is a Python command that prints something out, and then after the print, we can have any expression that we want that is valid Python. We'll talk more about what expressions are in Python. Now we're just going to print the number 3, and when we click Run, it will run this code and show us the result down here. And the result of printing 3, we see the output 3. We can do more interesting things. We can print an arithmetic expression, so I've

got 1 plus 1.

Now we run this. We see both outputs, so first we printed 3, we see the the result of 1 + 1 is 2. We can write this more clearly by having spaces. We can have spaces between the parts of our expressions. So we can have one plus one with spaces between there. that's a little easier to read, when we run that, we see the same result. The result is still 2. And we can make more and more complex expressions, so let's print the result of 52 *3 + 12*9. This will print the result of multiplying 52 times three and adding that to multiplying 12 times 9. And we get 264. You can check that yourself to see that Python got the right answer. We can use parentheses to group expressions.

So, if we use parentheses, we can do what we did before, but putting parentheses around grouping the multiplications and grouping the other multiplication. And we run this. We see the same result as without the parentheses. That means the same thing as we had when we add the parentheses like this. If we put the parenthesis in different places it means something different. Now what it means is 52 times the result of adding 3 plus 12 which is that we want like this. For example if we wanted to compute the number of seconds in a year, we can compose many multiplications. So we'll multiply 365 days times 24 hours in a day times 60 minutes in an hour times 60 seconds in a minute. We can do all those multiplications together and we get this result. Which is about 31 and a half million seconds in a year.

So now it's time for your first programming quiz. You've seen enough to be able to write a Python program. And your goal is to write a Python program that prints out the number of minutes there are in seven weeks, which is the amount of time we have for this course. You'll do that by entering your code in here. And then you can try different things. You can try running the code. See the result. And then when you've got an answer, you can click to submit that answer and see if it's correct.

Answer:

So there's lots of different ways you could have solved this. You need to use the print command to print out the result. And then we want an expression that calculates the number of minutes in seven weeks. There's seven weeks, each week has seven days, so we can have seven times seven for the number of days. Then to get the number of minutes, we need to multiply that by 24 to get the number of hours And then multiply again, by 60. That should give us the number of minutes. So let's see that in the Python interpreter. And now we hit Run, and we see that we have 70,560 minutes. Seems like a lot of time. It's going to go pretty quick, and we hope by the end of the seven weeks, all of you will be accomplished Python programmers.

Congratulations

Congratulations, you've passed the quiz and you've actually written your first program. It might not seem like much of a program but that's all a program is. It's just instructions to tell the computer what to do. To be able to write something more interesting, like a web crawler well, we are going to need to learn to write more complex programs.

Quiz: Language Ambiguity

So now that you've written your first Python program, you might be wondering why we need to invent new languages like Python to program computers and spend all the effort into learning a new language, rather than using some language like English or Mandarin that we already know. And there lots of reasons for that. That natural languages like English and Mandarin are great for people who grew up speaking them for day-to-day conversation. But there are many reason that they wouldn't work well for programming computers. The first one is ambiguity. Natural languages are inherently ambiguous. Different people can interpret the same sentence to mean many different things. When we program computers, the computer needs to interpret that program. And we want to make sure that the computer interprets the program the exact same way that the programmer, who wrote the program, intends for it to be interpreted. So here's an example of the kinds of ambiguities that come up in natural languages like English, and we'll do this in the form of a quiz. So the question is, would you rather be paid $100 weekly or biweekly? So these are the choices, you can prefer to be paid $100 weekly; you can prefer to be paid $100 biweekly. It depends, or you are an altruist and you don't like to be paid at all

[ ] Prefer to be paid $100 weekly
[ ] Prefer to be paid $100 biweekly
[x] It depends
[ ] Prefer not to be paid at all

Answer:

So there is no real correct answer to this quiz if you really don't like money, I guess you could prefer not to be paid at all. The closest thing to a correct answer is that it depends, and it depends on how you interpret the meaning of biweekly. We could look that up in the dictionary to decide how we should interpret it, and if we look up biweekly in the dictionary, what we see is the definition of biweekly, which is right here. It says happening every two weeks is the first definition. The second definition is happening twice a week. If you have the first definition, you'd rather be paid weekly, assuming you'd like more money. If you have the second definition, you'd rather be paid biweekly. Maybe that's the problem with paper dictionaries, that they're not very precise, because they're old things. Let's look up a more modern dictionary, let's see what Wiktionary has to say about this. So, here's the Wiktionary entry for biweekly, and if we scroll down, we can see that it has two meanings. Occurring once every two weeks, occurring twice a week, this is chiefly in the UK the

American dictionary didn't seem to think that. It seemed to think both definitions occurred in both countries. So, this is a pretty big problem. If we wrote a computer program and the program was doing our payroll and had to decide what it meant to pay someone biweekly, we want to make sure it understood that the same way the programmer did.

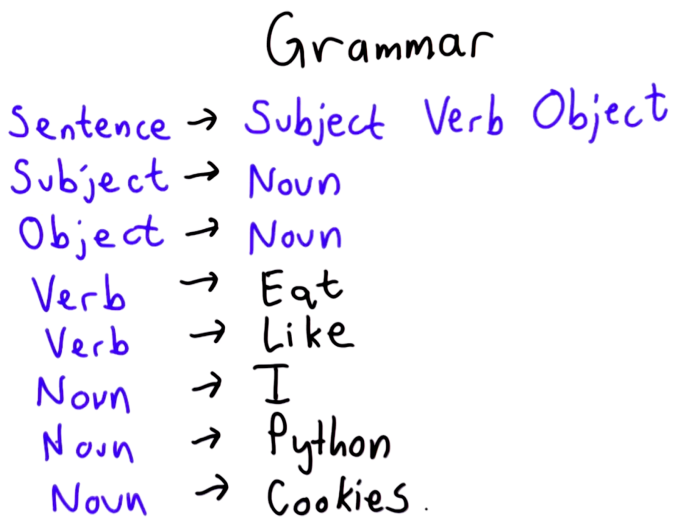
So that's one important reason why we need to invent new languages like Python to program computers, rather than using natural languages that we already understand. And that's because we don't really understand languages like English or Mandarin, that there's lots of things that different people will understand different ways. And this happens at the granularity of words like biweekly. It also happens in complete sentences where there are many different ways to interpret the same phrase. Another reason we don't use natural languages for programming is that natural languages are actually very verbose.

To write a program, we need to describe exactly what the computer should do in very precise sequence of steps. If we had to describe all those details using a natural language, that would require a huge amount of text. So we'll see soon that with a programming language like Python, we only need a few lines of code to describe a complicated thing and describe it in a very precise, step by step way. So I want you to think about as we write programs in this class, how much more detail you need to actually describe in English, precisely how to do what we're describing with a few lines of Python code.

Grammar

So in order to learn about programming, we need to learn a new language. This will be a way to describe what we want the computer to do in a much more precise way than we could in a natural language like English. And it's a way to describe programs that the Python interpreter can run. One of the best ways to learn a programming language is to just try things. You can try things in the Python interpreter that's running in your browser. Let's, for example, try running print 2 + 2 +. In English, someone could probably guess that the value of 2 + 2 + should be 4. In Python when we try running this, we get an error. And the reason we get an error is that this is not actually part of the Python language. The Python interpreter only knows how to evaluate code that's part of the Python language. If you try to evaluate something that's not part of the Python language, it will give you an error. Errors look a bit scary, the way they print out.

But there's nothing bad that can happen. It's perfectly okay to try running code. If it produces an error, that's one of the ways to learn about programming. The error we got here is what's called a syntax error. That means that what we tried to evaluate is not actually part of the Python language.

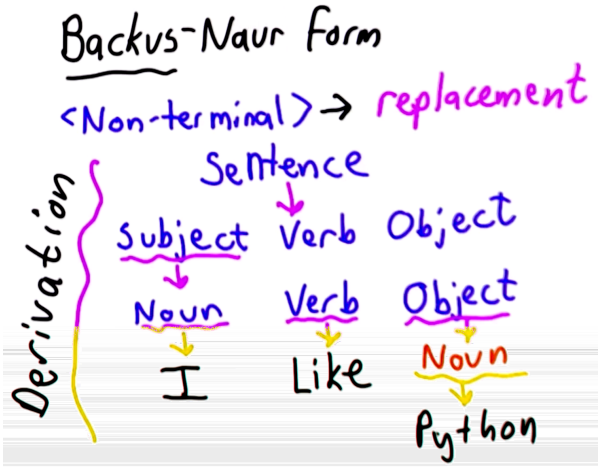


Like English, Python has a grammar that defines what strings are in the language. In English, we can make lots of sentences that are not completely grammatical, and people still understand them, but there's some underlying grammar behind the language. Those of you who are native English speakers, might have learned rules like this in what was once called grammar school. Those of you who learned English as a second language, probably learned rules like this when you were learning English.

So, English has a rule that says you can make a sentence. By combining a subject with a verb, followed by an object. Almost every language has a rule sort of like this. The order of the subject and the verb and the object might be different, but there's a way to combine those three things to form a sentence. The subject could be a noun. The object could also be a noun. And then each of these parts of speech, well, we have lots of things they could be. So a verb could be the word eat. A verb could also be the word like, and there are lots of other words that the verb could be. A noun could be the word I, a noun could be the word Python, a noun could be the word cookies. The actual English grammar is of course, much larger and more complex than this. But we can still think of it as having rules like this that allow us to form sentences from the parts of speech that we know, from the words that make those parts of speech. The way we're writing grammars here is a notation called Backus-Naur Form. And this was invented by John Backus.

So John Backus was the lead designer of the Fortran programming language back in the 1950s at IBM. This was one of the first widely used programming languages. And the way they described the Fortran language was with lots of examples and text explaining what they meant. And this is a shot from the actual manual for the first version of Fortran. This works okay, many programmers were able to understand it and guess correctly what it meant, but was not nearly precise enough. And when it came time to design a later language, which was the language called ALGOL, it became clear that this informal way of describing languages wasn't precise enough. And John Backus invented the notation that we're using here to describe languages.

Backus Naur Form



The purpose of Backus-Naur Form is to be able to precisely describe exactly the language in a way that's very simple and very concise. So each rule has the form like this where on the left side there's a non-terminal. Non-terminal means something that we're not finished with. All the words written in blue in the grammar are non-terminals. Sometimes they're written with brackets around them. I've just used blue to distinguish the non-terminals like this. Then there's an arrow, and then on the right side there's a replacement. The replacement can be anything. It can be a sequence of non-terminals like here. Sentence can be replaced with subject followed by verb followed by object. It can be one non-terminal like here. It can also be a terminal, and the terminals I've written in black. What's special about the terminals is they never appear on the left side of a rule. Once we get to a terminal, we're done, we're finished. There's nothing else we can replace it with.

So all the rules have this form. We can form a sentence by starting from some non-terminal, usually whichever one is written at the top left-- in this case the one I called sentence. And then by following the rules we keep replacing non-terminals with their replacements until we're left with only terminals. Here's an example starting from sentence using the grammar above. We can start with sentence. We only have one rule to choose from where sentence is on the left side, so we're going to replace sentence with subject, verb, object. Now we have a lot of choices. We can pick any of the non-terminals we have left. Find a rule where that non-terminal is on the left side. We can pick any of the rules where it's on the left side and do the replacement. So I'm going to start with the left one. We'll pick subject. We only have one replacement rule for subject. We can replace subject with noun. The others stay like they are, so we still have verb and we still have object. Now we can keep going. We can pick the first one again. It's still a non-terminal, so we can still do replacements. With noun we have three choices. We can pick any one of those choices. I'm going to pick the first one. We'll replace noun with the terminal I. Now we've got a terminal. We're done with that replacement. Verb and object stay the same. I'm running out of space, so I'm not going to write them again because now we can use rules to replace verb. As a separate step we're going to find a rule that matches verb. We have two choices. I'll pick the second one and replace verb with like. We still have object. Object is a non-terminal, so we have to keep replacing it until we're done. We have one rule for object. We can replace object with noun. Now we have three rules for noun. I'm going to pick the second rule and replace noun with Python. What I've done here is what's called a derivation. A derivation just means starting from some non-terminal, follow the rules to derive a sequence of terminals. We're done when we have only terminals left and we can derive a sentence in the grammar. In this case we produced the sentence I like python, but there are lots of other sentences we could have produced starting from the same non-terminal if we pick different rules to follow.

Quiz: Eat Quiz

So now it's time for a quiz to see if you understand how replacement grammars work and can follow the BNF grammar we have here. So the question is, which of these sentences can be produced from the grammar here, starting from the non-terminal sentence? So check all of the sentence that are in the grammar. The choices are Python eat cookies, Python eat Python, and I like eat.

[x] Python eat cookies
[x] Python eat Python
[ ] I like eat

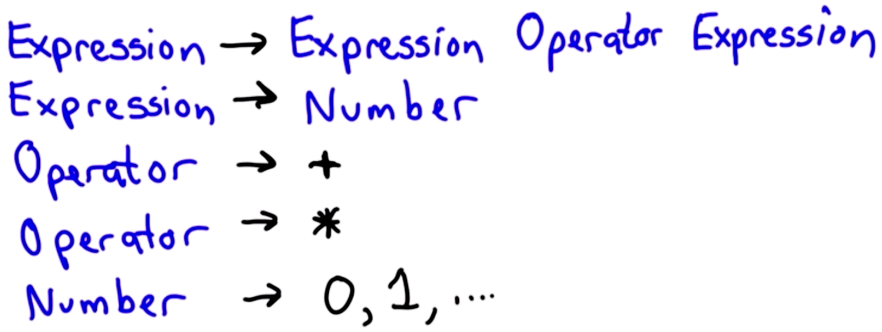
Answer:

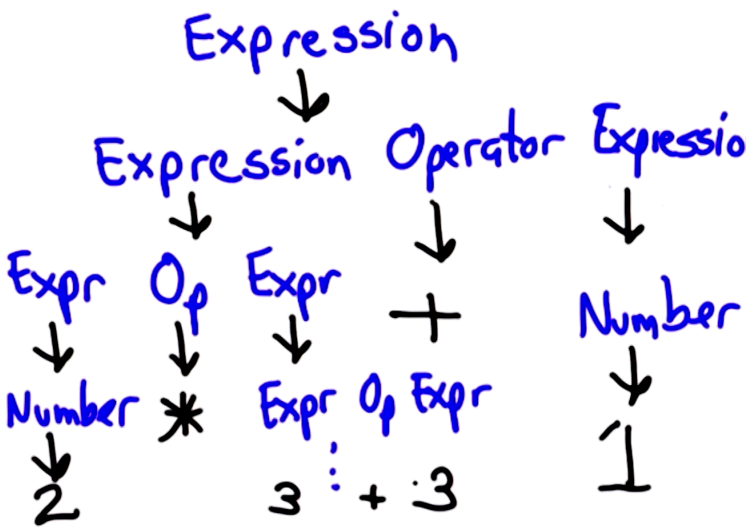
The answer is the first two can be produced. The third one cannot. And the way to see that is to follow the rules. So if you start from sentence, well, we only have one rule from sentence, so we're always going to end up with subject verb object. We also only have one rule from subject and one room, rule from object. So we're always going to end up with noun verb, followed by object, which also is replaced with noun. From here, we have lots of choices, so we can replace the noun with Python. We can replace the verb with eat, and we can replace the second noun with cookies. That will derive the first sentence. We could replace the second noun with Python, using instead the second choice, that would derive the second sentence. There is no way to produce the third one. We can get close. We can turn the noun into I, following this rule. We can turn the verb into like, following this rule. We can't turn the noun into eat. The noun can either by I, Python or cookies. So that's why we can't derive the third sentence. The important thing about replacement grammars is we can describe a large language. In fact, an infinitely large language with a small set of rules. This language itself is quite small. But we'll see how to describe much bigger languages, still using a small number of rules soon. And it is very precise; we can figure out exactly what sentences are in the language, just by following the rules here, doing replacements.

Quiz: Python Expressions

So, the Python grammar is much stricter than the English grammar or most natural language grammars. In English, if we say something non-grammatical like me go to the store, another English speaker might laugh at us.
But they can probably
understand what we
meant. In Python, the code
must match the language
grammar exactly. Here we
saw when we tried to

evaluate 2 + without the extra operand, we got a syntax error, that means it's not grammatical. If we said 2 plus to someone in English well they might be able to figure out that that means the same thing as 2. In Python it needs to match the grammar exactly. So now we're going to look at the Python grammar for arithmetic expressions. We won't see the entire Python grammar here. But enough to get an idea of why the expressions we've seen were valid and why the 2 plus is not. So first of all, an expression is something that has a value. We're going to start our grammar with an non-terminal expression. And we can make an expression by combining two expressions with an operator. This is sort of like the sentence rule we have for English where we could make a sentence by combining a subject, a verb and an object. This is a lot more interesting though, because we have expressions both on the left side and the right side. This looks kind of circular. Because we have other rules, it's not completely circular. This is what's called a recursive definition. We'll talk a lot more about that in a later class, in unit six we will, but for now, we're going to just see how we can use that to make the Python grammar. And to make a good recursive definition, we need at least two rules. We need one where we can keep going, using the same thing on the right side as the left side. And we need one where we can stop. So, another thing that an expression can be is a number. And this is why we can make expressions like 1 + 1. The operator will become plus, and the other expression will become one. So let's write a few more rules to see how that works. So we can replace the operator with plus. We can also replace operator with times. And there's several more operators in Python. Those are enough for now, we just need those two. We can also make numbers. And the complete rule for making numbers is sort of complicated, but let's assume we know what numbers are. We can make numbers any number of digits. There different rules to make all those numbers, but we know what they are. So, this looks like a very simple grammar. It's quite small, but it can express infinitely many things. And the reason for that is because we have expression on both sides here. That we can replace an expression with a derivative expression, and keep going.





So here's an example derivation. We can start with expression. We can follow the rule, replacing that with expression operator expression. And then, we can follow the rules again. We can replace the first expression with a number and replace that number with the actual number one. And we notice the actual numbers are terminal. So, once we get to the number one, we're done. There's no more replacements to do. We can replace the operator with plus. Plus is also a terminal, and we can replace this expression with a number, and replace that number with the number one. So this is how we got the expression 1 + 1 in the Python grammar and that's why that was valid. We can do a lot more than this though. Instead of replacing this expression with a number, we can replace this expression with another expression, operator expression.

So we're going to use the first rule replacing the expression with expression, operator expression here. I'm going to start abbreviating so Expr is an abbreviation for expression. Op for operator and Expr but, we're just using this rule. And now we can do the same thing again, replacing this expression with a number and replacing that number, let's say with number two. We could replace this operator with the multiplication operator. And we can replace this expression with a number. Or you could replace it with an expression, operator, expression. Let's do that, and build increasingly complicated expressions this way. We can keep doing this. We are not done until everything is a terminal. So lets say, we skipped a few steps here, but this could be 3 + 3. So this is how we can build up complicated expressions from very simple rules. So I am going to add one more rule to our python grammar, that will be enough to produce all the expressions that we have used so far. And that rule that we need is for parentheses. And this is actually quite a simple rule. It just says we can make an expression by taking any expression we have and putting parentheses around it. So, this is a pretty small grammar, but this is actually grammar that's powerful enough to produce all the Python expressions we've seen so far. As well as infinitely many different expressions. We're not going to cover the entire Python grammar this way, but this should give you an idea of the power of writing our grammar this way.

So we're going to have a quiz to see that everyone understands the grammar. So the question is which of the following are valid Python expressions that can be produced starting from expression using the grammar rules. So here are the choices; there are five possibilities, check all the ones that are valid expressions that can be produced from this grammar starting from expression. You can try your answers in the Python interpreter if you want, but first see if you can figure it out for yourself, which ones are valid.

[x] 3
[ ] ((3)
[x] (1*(2*(3*4)))
[ ] + 33
[x] (((7)))

Answer:

So, the answer is the first one. This is just a number three. And we can get three from expression by following this rule. Starting from expression, replacing with number, and then replacing number with the number three. The second one is not valid. And the way to see that is if we start from expression, we only have three choices. We can replace it with expression operator expression. We can replace it with number, or we can replace it with a left paren, expression, right paren. Since the expression we're trying to match starts with a left paren, the only way to get a left paren is to eventually have this rule that replaces expression with left paren. When we use that rule, what we get is the left paren followed by expression. Now we have expression here. From here, we've matched the far left paren and the far right paren. What we have left is this. We can replace this expression with paren expression, and we have paren that's replacing this expression using this rule. And then we still have the second right paren that we had before. This isn't going to match here because we need two right parens to match. And the essence of this rule is that every time we open a parenthesis we have to eventually close it.

So we can't produce expressions where the parenthesis aren't balanced. So this is not a valid Python expression. We can produce the third one. This has many nested expressions. We have three parens that are open, but we have balanced and closed parens. And the way we can derive this expression, we'll start with one expression that's our starting point. We use this rule to replace it paren expression, paren. Now, we have an expression that starts with a one, so we want to replace this expression with expression operator expression. We still have the outer parens that we had before. Now, we're working on this part. We want to to replace this expression with a number and then with the number one. We're going to replace the Op, the following operator goes to times rule. That's going to be replaced with times. And now we've got to replace this expression. We're trying to produce this whole thing, so we're going to replace this with, using the parentheses rule. And we can keep going, we're going to replace this expression with expression operator expression. Eventually get the two times and then do that replacement with parens again, to get the three times four. And not going to fill in all the steps here because they won't fit on the screen. But I hope you get the basic idea that we can keep doing the replacements, making complex expressions like this one. So the fourth possibility is not a valid expression. There's no way that we can get an operator in front with this rule. There actually are rules in Python that allow us to have an expression go to operator expression. But we have two expressions here. There's a three space three. If it was just the number 33, this would actually be a valid Python expression. Not something that we can produce with this grammar though. And the first one is valid. It's not sensible, or necessary, to have nested parentheses like this, but we could always keep using this rule, so we can derive that expression starting from expression using the last rule with the parens, and we're going to end up with paren, expression, paren, close paren. We can use that rule again, we're going to replace the middle expression with expression paren expression. Replacing this expression with the last rule. That'll give us expression surrounded by parens, and we still have the two outer parens on each side. So we can have as many necessary parens as we want. It's not necessary, it doesn't make a lot of sense. We would never actually want to write code that way. But it's still a valid Python expression. So recursive grammar rules like these are very powerful. We only need a few simple rules to describe a big language, and the whole Python

language that we're going to learn in this course can be described this way. We're not going to describe the entire Python language this way, but what I hope is that you have a good sense for how grammars work, and you can see that, even as we introduce constructs somewhat more informally as we go on, that they could be broken down into these kinds of formal or placement rules.

Programming Quiz: Speed Of Light

So before we go on to the next major computer science topic we're going to introduce. I want to give you one more quiz, to see if you can write a python expression that's going to give you some idea how fast a computer executes. So your goal for this quiz is to write some python code that will print out how far light travels in one nanosecond. Let me give you some information that will help with this. So, the speed of light is 299,792,458 meters per second. So, almost 300 million. One meter is 100 centimeters. One nanosecond is one billionth of a second. Which is 1 divided by 1,000,000,000. So, your goal is to compute how far light travels in one nanosecond, and to get that answer in centimeters. And I don't want this to be an algebra quiz. So, all you need to do is multiply these three values together, and you'll get the answer we want.

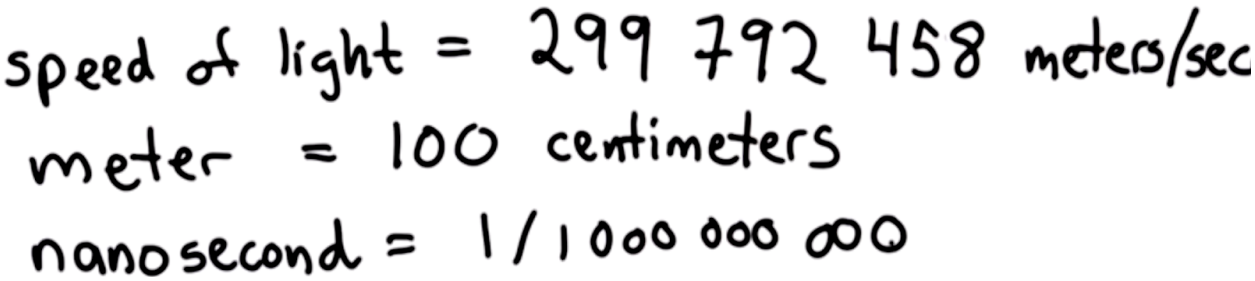
Answer:

So here's the Python expression to compute that. We're multiplying the speed of light times 100 cm in a meter, times a second. You'll note that I can't have space in the numbers. It's convenient when I write out the numbers to put spaces in them, so we can see how big they are. Python doesn't allow that, that looks like separate numbers if we have spaces there that wouldn't be valid in the Python grammar. So we can't have the spaces there, so when we run this we get the result 29. That says it's about 29 centimeters that light travels in one nanosecond. This is a little surprising that is an exact number, and it's an integer, and the reason it's an integer is because of the way Python does arithmetic. If all the numbers here are integers, Python will truncate down to that integer. If we want a more accurate result we should turn one of these numbers into a decimal number. So now we've changed the one to 1.0, now we run it we get 29.979. Two, four, five, eight, so almost 30 centimeters, which is a better answer than 29 was.

Processors

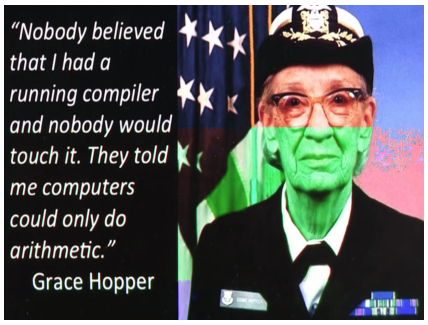
So why do we care about how far light can travel in one nanosecond? If you know what kind of processor you have, and if you have a Mac, you can find this by selecting from the Apple menu, About this Mac. They'll be instructions on the website how you can do this for other operating systems. But you'll see a window like this appear that will tell you what kind of processor you have. And if you zoom that a little bit you can see that we have a 2.7 GHZ Intel Core processor. What GHZ stands for is gigahertz. Which means that we can do 2.7 billion cycles in each second. So that means the time we

have for one cycle. Is actually less than a nanosecond, and you can think of a cycle as the time that computer has to do one step. So it does one step 2.7 billion times in a second, that means the time for each cycle is in the time that computer has for one cycle, light travels 11.1 centimeters. So how far is that? So, let's have a little scale here. If we have a dollar bill, that's actually quite a bit longer than 11.1 centimeters. 11.1 centimeters is about three quarters of the way across the bill. So within the time light can travel that distance, the computer's gotta finish processing one cycle, finish at least part of an instruction. This should give you some idea of how fast the computer is operating, and this is part of the reason the processor has to be so small. If the processor was bigger than the processor in the time for one cycle.



Grace Hopper

One of the pioneers in computing was Admiral Grace Hopper. She was famous for walking around with nanosticks, which were pieces of wire that were the length light would travel in a nanosecond-- Grace Hopper wrote one of the first languages, and the language COBOL, which she is seen holding here next to UNIVAC, was for a long time the most widely-used computer language. She was one of the first people to think about writing languages this way, ["Nobody believed that I had a running compiler and nobody would touch it. They told me computers could only do arithmetic." Grace Hopper] and you have this quote when she talked about writing compiler, and a compiler is a program that produces other programs, like Python. The difference between a compiler and an interpreter like Python is the compiler does all the work at once and then runs the new program whereas with an interpreter like Python you're doing this work at the same time. But she had this great quote talking about no one believing that she could do such a thing, and they told her that computers could only do arithmetic. So far we've only seen computers do arithmetic. We're going to see lots more interesting things in our Python program soon. There's a link on the website to a video of Grace Hopper's appearance on the David Letterman Show where she gives him a nanostick. I hope you'll enjoy watching that.



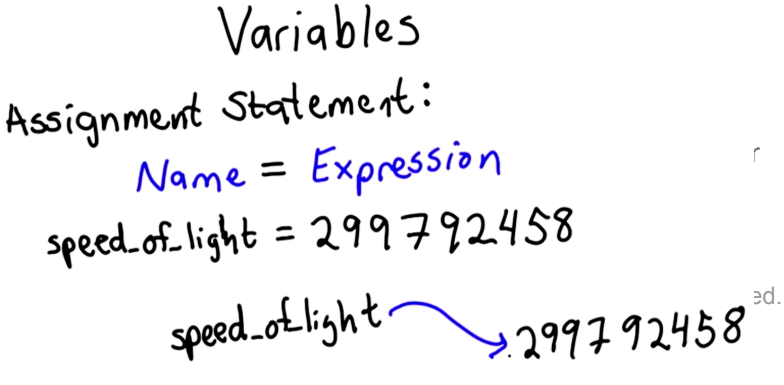
Variables

So our answer to the last quiz would've been a lot easier to read and a lot more useful if we used names to keep track of values, instead of writing out those big numbers, especially numbers as big as the speed of light. Python provides a way to do it. It's called the Variable. We can use the variable to create a name and use that name to
refer to a variable. So the way to introduce a variable is using an assignment statement. And an assignment statement looks like this. We have a name, followed by an equal symbol, followed by an expression. After the assignment statement, the name that was on the left side refers to the value that the expression has. The name can be any sequence of letters and numbers, as well as underscores, as long as it starts with a letter or an underscore. So here's an example, we could create the name, speed_of_light, and we can assign to it the value of the speed of light in meters per second. So after that assignment, the name speed_of_light refers to that value. One way to think of that is to have an arrow, so we can have the name speed_of_light, and that's a name which refers to a value. And the value it refers to is this long value, which is the speed of light in meters per second. So once we've done the assignment, we can use the name and the value of the name is the value that it refers to. In this case it's the speed of light in meters per second. So let's try that in the Python interpreter. Here we've introduced t to it the value 299,792,458, the speed of light in meters per second. And now we've got that, assign it a variable. Instead of having to type out that whole number, we can use it directly. When we print out the speed of light, it will be the value that that name refers to. So we'll see, instead of seeing speed of light, we'll see the 299 million value here. We can use in expressions as well.

Run

```
1
2 speed_of_light = 299792458
3 billionth = 1.0 / 1000000000
4 nanostick = speed_of_light * billionth * 100
5 print nanostick
```

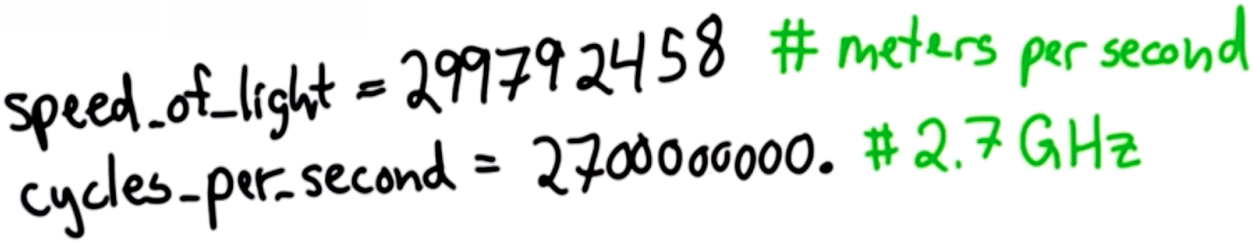
So if we want to convert it into centimeters instead of meters, we can multiply by 100 and now we see the result is the speed of light in centimeters per second. So let's define another variable. This one will define billionth, which means 1 divided by 1,000,000,000. That's hard enough to remember how many zeros to type, so it's nice to have that in a variable. And now we can define nanostick, which is the length of Grace Hopper's nanostick, a
of the nanostick in meters. If we want
defined a variable, nanostick, which is
we can print that out. So, variables are



to understand. They also mean that we can use the same expression, changing the values of the variables to compute different things. So, now it's time for a quiz to see if you understand variables.

Quiz: Variables Quiz

So for this quiz, the question is, given the variables that are defined here, and you'll see the variables already defined in the code in your web browser. Your goal is to write Python code that prints out the distance, in meters, that light travels in one processor cycle. Do the first variable as speed of light, and we've assigned to the variable speed_of_light the number of meters that light travels in a second. It might be hard to remember that. So we also have a comment there. We haven't used comments yet but they're a very useful thing to add to our programs. We can write a comment by starting with a hash symbol, that's usually the Shift+3 on most keyboards. Everything from the hash to the end of the line is a comment. That means it's ignored by the Python interpreter, but it's useful for the programmer to be able to see it.

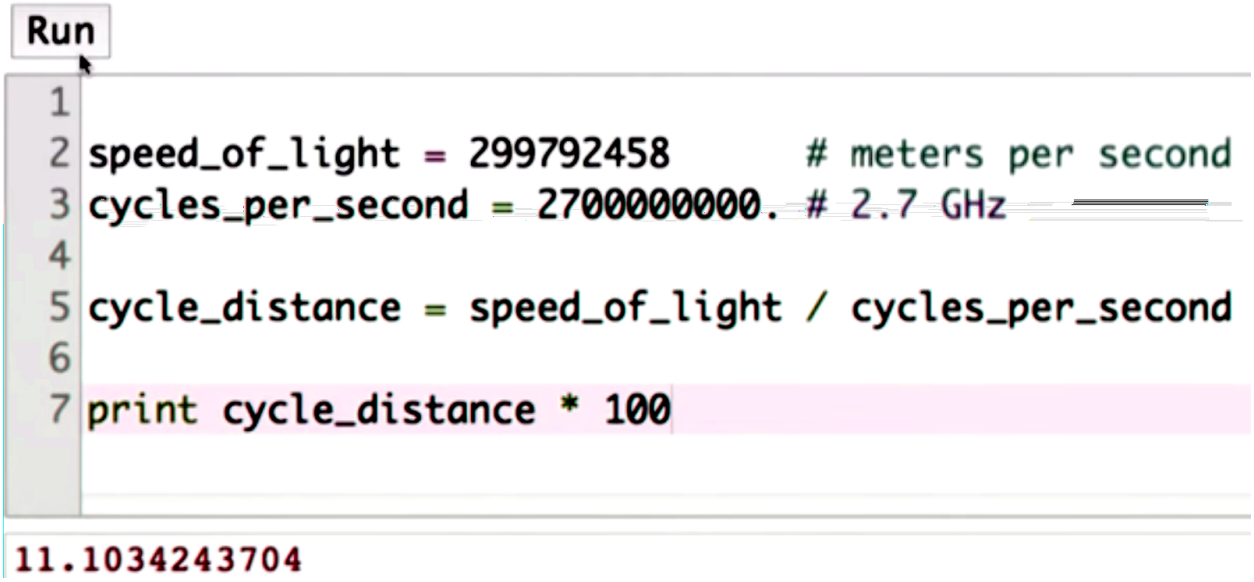


So we'll have a comment here that says that's meters per second. We can put anything we want in our comments. They are not interpreted by the Python interpreter. But it's a good idea to have comments that are helpful for us to remember what we did if we need to go back to the code a few months from now and understand it. It's also a good idea if someone else is reading the code, the comments will help them understand it. So the second variable, we'll call cycles_per_second, and we'll give that the value 2 billion 700 million. And I'm going to give it a decimal point so when Python does division, we'll get exact results. And we'll also give a comment for this one to indicate that this is the 2.7 Gigahertz speed that the processor in my computer has. So now, given those two variable definitions, your goal is to write some Python code that prints out the distance, in meters, that light travels in one processor cycle. And we can compute that by dividing the speed of light by the number of cycles per second.

Answer:

So, here's one way to answer this question. We have our two variable definitions. We can print out the distance light travels in one cycle by dividing speed of light by cycles per second using the variables, and when we run that we see the result is 0.11. So, what might be better would be to introduce another variable. So, instead of just printing the result, we can store it in a variable. We'll call it cycle_distance. Now, when we run it, there's no result. We haven't printed that out yet. But

we've stored it in a variable. And now, we can print out the result of cycle_distance. Which gives us 11, which is 0.11 meters. If we want the result in centimeters, well since we've already stored the result in meters in a variable, we can compute that by just multiplying that by 100. And now we get the result in centimeters.



Variables Can Vary

So the speed of light is a constant. But the important thing about variables in Python is that they can vary. That's why they're called variables. Once we define the variable, we can change the value. And then when we use that name again it refers to the new value. So let's see that in an interpreter. We could change the value of cycles_per_second. Suppose we have a faster processor. Now we've upgraded, we've got a 2.8 Ghz processor. Now, that doesn't affect the value of cycle distance. This was already computed with the old cycles_per_second. We still get the result, 0.111. We'll compute it again. This time the value of cycles per second is the new value since we changed what cycles per second refers to. And now when we print cycle_distance we see that the values changed. So we have a faster processor the second time we print cycle_distance now we only have 0.107 meters, less than 0.11 centimeters per cycle. So what happens now that we have assignment, the same expression can have different meanings at different times we evaluate it. The value of the speed of light divided by cycles per second depends on what the current value of cycles per second is. When we evaluate the first one, the value 2,700,000,000. So we got a different result which, was which was the 0.111 result we got the first time. When we evaluate the same expression the second time, the result is different, because the value that cycles_per_second refers to is different. And that's why we get the smaller cycle_distance the second time. So, let's see what's going on there and make sure we understand assignment. So, suppose we have a variable, days. And we'll initialize it to the value 7 times 7.

So what that does is introduce a name days. And it refers to a value, which is the result of that expression. So it refers to the value 49, and that means when we look at the name days, we see what it

refers to and we get the result, 49. If we do another assignment. Let's say we have one less day. And in this case we'll assign 48 to days. Well, that's a new assignment. We already have a name days. It used to refer to 49. But after the new assignment, it's going to refer to this new value. Now it's going to refer to the value 48. The number 49 still exists, but days no longer refers to it. Now days refers to 48. Where things get more interesting is where we use variables in their own assignment statements. So here we have an assignment statement where we have the value days minus happens with that assignment? Well, we evaluate the right side first. We look for the value of days and we see that it refers to 48. We compute days minus 1 and we get the value 47. Then we do the assignment that will assign to the variable days.

So now the value days refers to the value 47, no longer refers to 48. So we could keep doing that, if we did another statement, same exact one, that's going to change the value again. This time, the first time, the value days is 47, we'll subtract 1, we'll get the value 46. And then we do the assignment, that'll change the value, so now days refers to the value 46. So the important thing to notice, this is not an equal symbol. This looks like an equal symbol. If you studied algebra you would think an equation like this looks like an equality, and there is no way to solve an equation like that. In Python and in most programming language, equal does not mean equal. What equal means is assignment. You should really think of it as an arrow. It's an arrow saying put whatever value of the right side evaluates to, into the name on the left side. We don't write it as an arrow in most programming languages. There are some that do. But an arrow is harder to type, and lots of programs have lots of assignments. So you should think of the equal sign as not meaning equal. It means assignment.

Quiz: Varying Variables Quiz 1

So now, we're ready for a quiz to see that you understand the meaning of assignment. So the question is, what is the value that the variable hours refers to after running this code? And the code is below. First, we have an assignment statement assigning the value 9 to the variable hours. Then, we have another assignment statement where the right side is hours plus 1 and the left side is hours. And then we have another assignment statement where the left side is hours, and the right side is hours times 2. So the possible answers, the value of hours is hours is 20, the value of hours is 22, or it's an error. So try to figure out the answer yourself without evaluating this code in the interpreter. If you want to evaluate code in the interpreter, though, it's certainly a good idea to try that.

[ ] 9
[ ] 10
[ ] 18
[x] 20
[ ] 22
[ ] Error

Answer:

So the answer is 20. And here's why. So initially, we have created the name hours, and assigned it the value 9. So, I will refer to the number, 9. That is what we have after the first assignment statement. For the second assignment statement, first we evaluate the right side. And that says hours plus 1. That will produce, 10, heading 1 to 9. And now we assign that new value to hours. So that will change what the name hours refers to. Now ours will refer to the number 10. Then we do a multiplication. The third assignment statement has hours on the right-side, hours times 2. So that will evaluate to, the value hours refers to which is 10 times two, so we get 20. That will change the value of hours to now refer to 20. So, at the end of these statements, the value of hours is 20.

Quiz: Varying Variables Quiz 2

So we'll have another quiz about variables. This one's a little tricky. So the question is, what is the value of seconds after running this code? The code has two assignment statements. The first one assigns to minutes the value of minutes plus 1. The second one assigns to seconds, the value of minutes times Like the previous one, see if you can guess what the answer is yourself. You can always try running the code in the Python interpreter.

[ ] 0
[ ] 60
[ ] 120
[x] Error

Answer:

So the answer is, it is an error. And the reason why is this first assignment has minutes on the right side, but we didn't define minutes previously. So, if we use a variable that's not defined, there's no meaningful value for this. The Python interpreter will give us an error. Let's see that in the Python interpreter. So here, we have just the first assignment. When we run this, we get the error that says the name minutes is not defined. Minutes is not defined because we didn't introduce that variable yet. It looks like we're defining it. We've gotten in on the left side, but the right side needs to execute first. So without knowing the value of minutes, there is no sensible value this code can produce, so running the code produces an error. If we have a statement before this that gives minutes a value, then the code will be fine. We can run this. There's no error. And we can do the assignment that we had in the quiz, assigning to seconds minutes times 60, and print the value of seconds. This works now, but only because we added this definition. We need to always introduce a variable before we use it.

Programming Quiz: Spirit Age

So we're going to have one more quiz about variables. Variables are a really important concept. We're going to use them all the time in the programs that we write. So this quiz is going to test if you can define a variable.yourself, and use that to do a computation. So your goal for this quiz is to write Python code that defines the variable age, gives it the value of your age in years, and then prints out the number of days you've been alive. You don't have to use your real age. If you don't want to include your real age, you can make up whatever age you are or you can use your age in spirit. The important thing is that you can define that in a variable and then do the computation.

Answer:

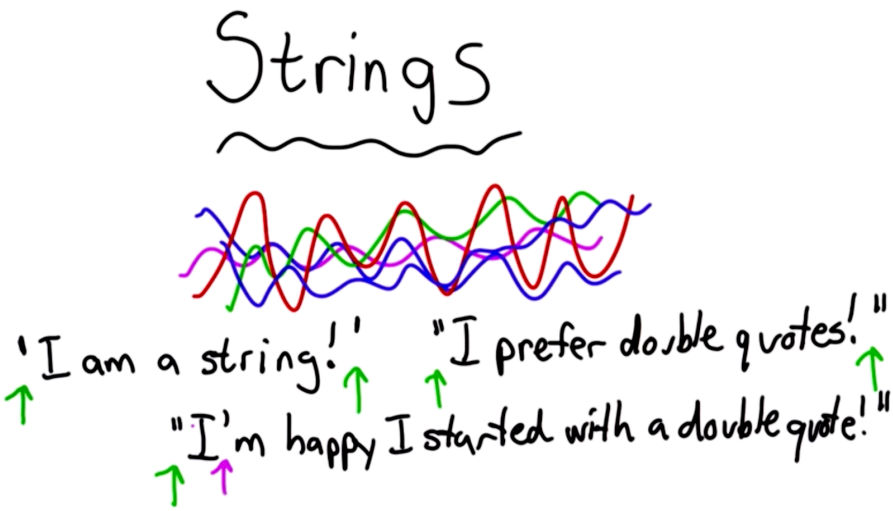
So there are lots of different ways to answer this, especially, depending on your age. Here's one way. So, I'm going to define the variable, age, and I'm going to use my age in spirit, which is 7. And then, we'll multiply that by the number of days in a year. But let's also define a variable for the number of days in a year. And we could use 365. If we want to account for leap years, it would be more accurate to use 365 and a quarter, since one quarter of the years are leap years. And to compute the number of days that I've been alive, we can multiply those two values and print out the result. So let's try that in the Python interpreter. So here, we've defined our two variables. We have age is 7, not my real age, in case you're wondering. And days_in_year is 365 and a quarter. So we'll multiply those two to print out the result, which is 2556 and 3 quarter days. Of course to do this more accurately, we'd need to use a real age. And our actual age is probably not an exact number of years, unless it's your birthday today, in which case, happy birthday. But otherwise, you'd need to add the number of days since your birthday to get a more accurate number. I'll change my age to my real age. And I will add the number of days since my birthday. Which is 268, if you're watching this on the day it's released. And then we got my age, in days, is 14,878. Which seems pretty old.

Strings

So far all the computation we've done is operated only on numbers, and in the early days of computing, people thought of computers as super powerful calculators, for doing computations like simulating nuclear weapons, computing ballistic tables, or breaking encryptions, which was a little more than just arithmetic. But still was mostly about counting and doing simple arithmetic. We saw this quote from Grace Hopper earlier talking about computers could only do arithmetic, and this is what people thought about computers in the 1940s and can operate on any kind of data we want, and it gets much more interesting when we operate on data besides just numbers. If we're going to build a search engine, most of the data that we want to deal with is not numbers. It's the letters that are contained in web pages, and in Python, that's what we call a string. A string is just a sequence of characters surrounded by quotes. So here's an example of a string in Python, it starts with a single quote, has a sequence of characters and anything that we can type on the keyboard can be in a string, and ends with another single quote. The string is the sequence of characters between the single

quotes. If we want, we can use double quotes instead. If we use double quote, then the double quote starts the string. We can have a sequence of characters and a double quote that ends the string. The only requirement is that if we start the string with a single quote, it has to end with a single quote. If we start the string with a double quote, it has to end with a double quote. And that's actually a handy property. Because that means we can have the other kind of quote within our string. This string starts with a double quote. It contains a single quote inside it. But because we started with a double quote, that single quote doesn't end the string. That single quote is just like another character in the string. The string continues. Until the closing double quote. So let's try some things in the Python interpreter. So, we can print a string, just like we can print a number. So here I'm printing the string hello. And when we run this, we see the output hello. It's printing hello. We don't see the single quotes as it prints, but we know that it's a string that was printed. We can print a string with double quotes, and one thing to notice when you enter strings through the interpreter, the color is black now, since it's an open string, it hasn't been finished. Once I type the final quote, that closes the string, the color changes to blue.

So now when we run this, we have two prints, both that print hello, it looks the same both times. It doesn't matter if we use single quotes or double quotes around our string. Just to check everyone's paying attention, I'm going to try one more thing. And now I am printing hello without the quotes. You can guess what that will do. We won't make a quiz of this, but try to guess before I run it. Now that I run this, we see the result. We get a name error. The name Hello is not defined. Without the quotes, this looks just like a variable. It's a name, but it's a variable that we didn't define. So when I try to use it, I get an error that the variable name Hello is not defined. If I wanted to I could define a variable named hello. Let's make hello refer to the string howdy and now when I print hello, it works. I see the first two prints that printed hello, now when I print the variable hello, well that refers to the string howdy and I see howdy as the result. We usually don't want our variables to start with capital letters. That's just a convention, so I'm going to change this back to a lower case hello because it makes me feel uncomfortable to have a variable with a capital letter. There's no rule against that in Python. It's just a convention that we like to follow.



Quiz: Valid Strings

So now we're ready for our first quiz about strings. So the question is, which of the following are valid strings in Python? So here are the choices. Check all of the choices that are valid strings.

[x] “Ada”
[ ] ‘Ada”
[ ] “Ada
[ ] Ada
[ ] ‘“Ada’

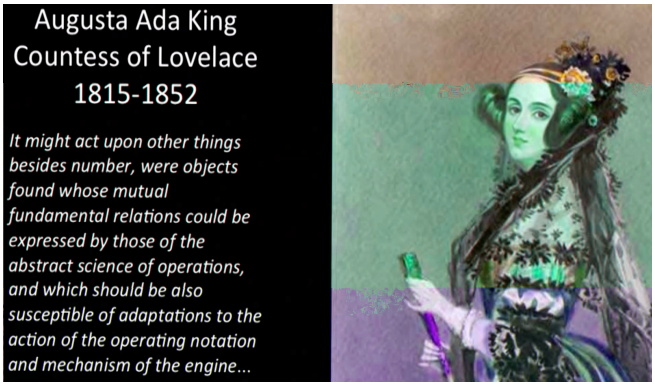
Answer:

So the answer, the first one is a valid string; it has a double quote followed by another double quote, and a sequence of characters between the double quotes. The second one is not a valid string, and the reason it's not is because it starts with a single quote, but ends with a double quote. A valid string must start with the same kind of quote it ends with. The third one is also not a valid string, and the reason for that is it starts with a double quote but there's no double quote that closes the string. The fourth one is a name, this is a variable, it's possible it could be defined to be a valid string, but without defining Ada as some string, it's not a valid string by itself. The surprising one, might be the last one. This is a valid string, it starts with a single quote, it ends with a single quote and it's perfectly okay to have a double quote in the middle.

Ada

If you're wondering who "Ada" is in all of those strings, Ada is Augusta Ada King, who is arguably the world's first computer programmer. Back in the 1840's, in England, she started to think about how to program mechanical computers-- Charles Babbage was proposing to build such machines-- and Ada was the first person to really think about how to program them. Grace Hopper wasn't the first person to think about doing things other than arithmetic with computers. Back in the 1840's, Ada was already thinking about that. You can see this quote here from her: "It might act upon other things, besides number, were objects found whose mutual fundamental relations could be expressed by those of the [abstract] science of operations..." The "it" that she's talking about is the analytical engine that Babbage was proposing to build. He never succeeded in building it-- in the 1840's, the

technology for building it was not quite good enough to make a machine that precise. But he had a design for it, and Ada was thinking about programming it to do more interesting things. The quote goes on to talk about actually using it to compose music. ["...and which should also be susceptible of adaptations to the action of the operating notation and mechanism of the engine..."] People do use computers to compose music today. It's debatable whether or not you actually want to listen to that music, but certainly it's something computers can do.



Quiz: Hello!!!

So, now we're going to have a quiz to see if you can actually use strings. Your goal is to define a variable, the name, called name and assign to it a string that is your name.

Answer:

So, assigning a string to a variable is just like assigning a number to a variable. We can do that with an assignment statement. My name is Dave. So, I'll assign the string Dave, to the name, to the variable name. The nice thing about using a variable like this, is now we could have code. Say print Hello plus name. Here hello is a string. Name is a variable. If you assigned name to mean some other string, this will print Hello whatever the value of name is. I've introduced a new operator here. We're using the plus operator. We already saw plus on numbers. Now, we're using plus on strings. It means something different. What plus means with strings is concatenation. So we can put strings together by using the plus operator. The value of the string plus another string is the concatenation. That's a new string that's a result of pasting those two strings together. So let's try that in the Python interpreter.

So here I've initialized the variable name to be the string Dave, and I'm going to print out hello plus name. You'll see I won't get quite what I want from this? Here's what happens, it concatenated the two strings together, it doesn't add spaces or anything automatically. So now we've got, the output is hello Dave, without any space in it. We need to add a space here. Now we'll have hello space Dave, which looks more like we want. We can keep doing this just like we could use the plus operator with numbers more than once, we can use it with strings more than once. Adding another string to the end, now we've got the exclamation point I need. Maybe one is not enough. We can keep adding more exclamation points. Now I have a more pleasant reading.

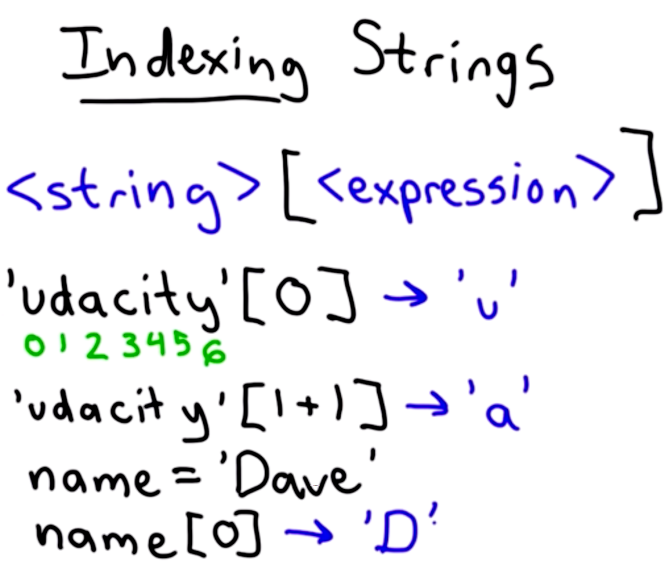
Strings And Numbers

So, it seems we can use plus with numbers and we can use plus with strings. What if someone tries to use plus with a string and a number at the same time? So, suppose my name is actually the number 9. Well, when we try and run this, what we get is an error. Here's the error we get. This is a little different from the syntax errors we've seen before. This is still in the Python grammar. But it's something that doesn't make sense. What we tried to do was concatenate a string, which is the string my name is, and the number, which is the integer 9, and it doesn't make sense to paste those together. At least Python doesn't know what it means. So this produces an error, we can't add numbers and strings together. We can multiply strings though, this is a little strange. And so, here I'm doing the exclamation point string times 12, and what I get is 12 exclamation points. This is better than having to type them all out. So I can change, instead of doing Hello name, exclamation point plus exclamation point, I could do exclamation point times 3 here. And this would be the Hello Dave with exclamation point times 38. And then I'll get Hello Dave, followed by 38 exclamation points.

Indexing Strings

So one of the things we can do with strings that we can't do with numbers, is we can extract subsequences from the strings. So remember what a string is. It is a sequence of characters. If we have a string, we can use the square brackets to extract parts of that string. So if we have the string udacity, and we use the square bracket, with the value 0, the characters of the string are indexed starting from 0, so the result of index 0, is the string with just the letter u. The expression inside the square brackets can be anything that evaluates to a number. So we could have, 1 plus 1 in here. 1 plus 1 evaluates to the number 2. And at position 2, we find the letter a, so the value of this would be the string containing the single letter a. This looks a little strange and we wouldn't normally use indexing with the string literal like this, but it does work. It looks a lot more normal when we're doing it with a variable. We've initialized the variable name to the string Dave. Then when we do name index 0, whatever the variable name is, that will give us the first letter of that string, in this case it's the uppercase D. Let's try that in the Python interpreter. So here, I've defined the variable name, with the value, the string Dave. And I'm printing name index print name index 3, that will give us the fourth letter, the e. Suppose I tried to do name index 4. Well, there's no character at position 4, remember that the

indexes start at 0, so this is 0, 1, 2, 3. When I run this, what I get is an error. I get an error because I've asked for position 4 of the string; that's out of range, so the error I get says that that string index is out of range. What I can do is use negative numbers. When I use negative numbers in my index it starts counting from the back of the string. So name index negative 1 will go as the last character in the string. If I do name index negative 2, that will give us the next to last character which is the v.



Quiz: Same Value

So the question is, given any variable, we're going to introduce the variable S, and we're going to assign it any string. Which of the pairs below are two things where both elements of the pair are the same exact value? So here are the choices. Check all the choices where the first thing and the second thing have exactly the same value. So the first thing is s index 3, s index 1 plus 1 plus 1. So, for any string that we choose for s, those two always have exactly the same value, then you should check this box. Second, we have s index 0 paired with s plus s, index 0. For the third choice, we have s index 0 plus s index 1 paired with s index 0 plus 1. For the fourth choice, we have s index 1 paired with s plus the string 'ity' index 1. And for the fifth and final choice, we have s index negative 1, and remember, the negative indexes go from the back of the string, paired with s plus s index negative 1. So for this question, you should check all the boxes where the two items in the pair have exactly the same value, no matter what value s has, as long as s is a valid string.

[x] s[3], s[1+1+1]
[x] s[0], (s+s)[0]
[ ] s[0]+s[1], s[0+1]
[ ] s[1], (s+ 'ity')[1]
[ ] s[-1], (s+s)[-1]

Answer:

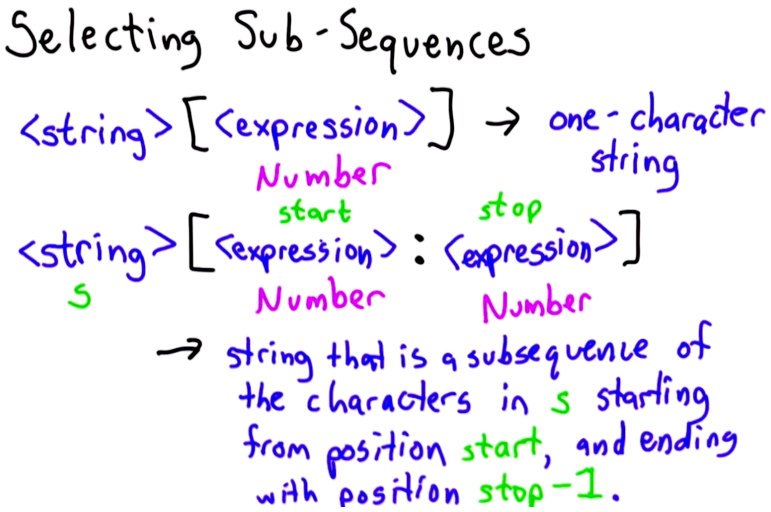
So here's the answer. The first one is true. s index 3 is exactly the same thing as s index 1 + 1 + 1. If S is a string with at least 4 characters in it, this will get the fourth character from s. If s has fewer than 4 characters, both of these will produce an error. The second choice, both of these actually also have the same value. s index 0 will get us the first character of s as long as s has at least one character; otherwise it will produce an error. s plus s will produce the string, concatenating s with itself but its

first character is still the same as the first character of s, so these two are equivalent. The third option is not equivalent. So here we're taking the first character of s, concatenating that with the second character of s. Here we're taking from s, the 0 plus 1th character, which is, evaluates to 1 so that's the second character of s. This is just going to be one character, this is two characters. They're never equivalent. The fourth option is sometimes equivalent. If s is a string with at least two characters then s plus 'ity' index 1 is the second character for s. The problem is that if s has less than two characters. So suppose our value of s was just a one letter string. s index 1 will be an error. There is no second character from s. s plus ity index 1 would be the second character of the string that resolves from that concatenation which would actually be the I. So these are not always equivalent. They are equivalent in the case where s has at least two characters. The final option, we have s negative 1, so that means the last character of s. And we have s plus s, which produces the string which concatenates s with itself index negative 1. That's going to be the last character of concatenating s with itself, which is the same as the last character of s.

Selecting Sub Sequences

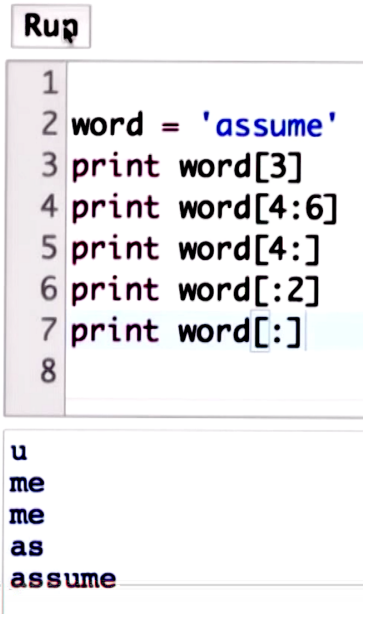
So there are lots more things we can do with strings. The next one I am going to talk about, is selecting sub-sequences from strings. So what we have seen so far, we have used indexing, where we have a string, where we have our square bracket. We have some expression that produ square bracket, and that gives us a one-character stri to in the string here. The other thing we can do with s the string. Instead of just having one expression here also something that should evaluate to a number, fol expression. Both of these expressions are numbers. A characters in the input string. So I called the string s. value of this number will call stop. And what the resu characters in s, and the string that we had here. Start colon, and ending with position stop, but not includi include the characters from position start, up through from any string A subsequence of continuous charact Python interpreter. I'm going to initialize the variable single index operator we saw initially, we can select a character from word. So, if I select word index 3. That will give us u, and when we run that, we see the result is u. With the new operation, that's the same as selecting word 3:4, that's going to select starting from position three, going to just before position four. So that will end up being just the one letter u. Just to make this clear, if I selected single empty string which prints out in a way we can't see it. There's no characters between 3 and character 3. We'll go back to just selecting index 3. I could select from position 4 through position 6. That would give us a string which is a subsequence of letters at position 4 and position first one prints out to u. The second one prints out me which is position 4 and position 5. The last two

characters in the word. I'll show you one other thing we can do. Which is, leave one of the sides of the colon empty. So that will select from position 4 to the end. We don't need to actually count the characters to know where the end is. So that will produce the same thing as we did before. We can also from the beginning. So if we do :2, that will select fro letters of the word, as. And we could leave both sides either side, well, it starts from the beginning, goes to word. There's no good reason to ever want to do this can leave one side of the colon empty and that mean e



Programming Quiz: Capital Udacity

So now we'll have a couple quizzes to see that you understand how to index subsequences from strings. So your goal for this quiz is to write Python code that will print out Udacity with an uppercase U, given that we've already defined the variable S to have the value, the string audacity. And we want you to do this in a way that uses the string indexing operators up. It wouldn't be a good way to solve this by just printing out Udacity with an uppercase U as a string. But see if you can find a way to write Python code that uses s and has as little extra code as possible but prints out the string Udacity starting from s.



Answer:

So the answer is, we want to select the part of the word that we can still use in our solution. So we can't keep the a, and we can't keep the lower case u, so those are positions 0 and 1. From position 2 to the end is good, so we can produce that using the index selection. We're going to select from s, starting from position 2. If we wanted, we could count all the letters and select from s to the final position. We can also just use a closed bracket. We don't need to count the letters that's going to select from position 2 to the end of the string. Now, we want to get the uppercase U in front. So, we're going to have an uppercase U added to that. And finally, we need the print to print that out. So, here that is in the Python interpreter. We have the definition of s that was provided. And what we're going to do is print out capital U plus s[2:]. And as expected, we get our result: Udacity.

Quiz: Understanding Selection

So selecting sub-sequences from strings is a very useful thing. We are going to use it lots of times in many of the programs we write. So we're going to have one more quiz about this. This one's going to be a little more abstract, so for this one, you're given any string s, so s is a variable. It holds a value of any string, and the question is, which of these are always equivalent to the string s, no matter what s was at the beginning? So here are the choices. We have s index colon. We have s concatenated with s index 0 colon negative 1 plus 1. We have s index 0 colon. We have s index colon negative 1. And we have s index colon 3 plus s index 3 colon. So your goal is to check all of the expressions here that have the same exact value as s, no matter what string s is initially. Feel free to try evaluating things in the Python Interpreter. Try to think about it without doing that first. But you're welcome to do experiments running code in the Python Interpreter to try and answer this. For remember, for your answer to be correct, it has to work for any string s, not just the one that you try.

[x] s[:]
[x] s+s[0:-1+1]
[x] s[0:]
[ ] s{:-1]
[ ] s[:3]+s[3:]

Answer:

So here's the answer. The first one is always equivalent to the string s, remember what it means to use the colon when there's nothing on the left side. It means, start from the beginning, and when there's nothing on the right side, it means go all the way to the end. So if there's nothing on on either side, that's selecting the entire string. So the second one's a little trickier, but this also always has the same value as the string s. And the reason is, we have the original string s, and we're concatenating to it the string s index 0, colon negative value 0. When we index from 0 colon 0, that's an empty string. And the reason that's an empty string is because we're starting from 0. And we're stopping just before 0, there are no character between 0 and 0 so that's an empty string. Adding the empty string to itself, leaves the string empty. And the interesting thing about this is that this works even when s is an empty string. If s was an empty string, s index select the first character from s, and there is no first character since s was an empty string. But s index 12 colon 12, even though there is no characters between there, it's okay that s doesn't have them. That's going to valuate to the empty string. So, the third one is also always equivalent to s. It's selecting from the beginning of the string, position 0, all the way to the end.

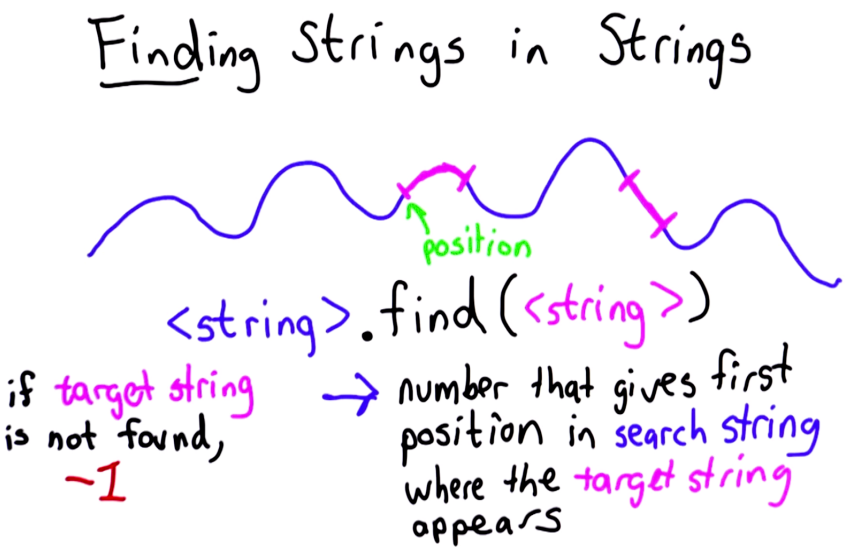
Again, as we saw here, even if s has no characters, this is still okay. It will give us the empty string, if s has any characters, it will give us all the characters in s. This first one is not equivalent, and this might be a little surprising. It sounds like, well, it's selecting all the characters from the beginning to the end, because negative 1 is the index of the last character in s. But remember that the selection stops

before the last character. So what this does, will be to give us every character besides the last one. So let's see that in the Python interpreter. Were going to print s index, colon, negative one, and this will give us all the characters except for the last one. So the final one is the most surprising. That this actually is always equivalent to s. It's equivalent even if s has fewer than 3 characters. So the first part of selecting all the characters from the beginning up to position three, not including position three, and then we're concatenating that with all the characters of s from position 3 to the end. If s has less than 3 characters, well this is going to be empty. There are no characters from position 3 to the end. But that's okay, we're going to add the empty string to the characters before position 3. So here's how that works in the python interpreter. We'll do s from position beginning to index 3 and add that s to index 3 to the end. That produces the same string. And if we do this with a shorter string. Let's initialize t to be the string, Hi. We can print from t to position 3 plus from t to position 3 to the end. We don't get an error. We get the string Hi out. So, this indexing subsequences is very resilient. It works even when we index positions that don't exist in the string. This is different from when we were doing indexing of particular positions by themselves. If we did index t, index 3, that will give us an error. Because there's no character at position 3.

Finding Strings In Strings

So I want to introduce one more operation on strings, which we'll find very useful, which is the Find operation. It gives us the way in a big string to find some sub-string that we're looking for. The way we use Find is a little different
from the way we've used
other operators so far.
Because Find is actually a
method, and what that means
is it's a built in procedure
provided by Python. We'll be
able to define our own
procedures soon, we'll get to
that in unit two. Find is a
procedure that operates on
strings, so we use it by having
a string followed by .find,
followed by a parentheses,
then we pass in another string. Which is the string that we want to find in the first string. And the output of Find is the position in the string where that sub-string is found, the first occurrence of the string. So, if that string happens to occur in more places than one in the input string, the

result of find is always going to give us the position. That's the number where the first occurrence of the sub-string occurs. So the output of using Find will be the first position in the search string, which is this blue string right here, where the target string, which is the purple string, occurs. So that will be a number. If the target string is not a found anywhere in the search string, then the output would be negative that works and we'll do this in the Python interpreter. Here I've initialized the variable Pythagoras to hold the string here that's been attributed to Pythagoras. We don't know if he really said it. But it says there's a geometry in the humming of strings, there is music in the spacing of spheres. So now, we have that variable initialized, so I'm going to invoke find, using Pythagoras as the string that we're searching in. And that's the value that we initialized it to with a string, passing in as the search string the string string. When we run this, we see that we get 40 as the result. If we counted, this is position 0, we would see string starting at position 40. Since I don't want to count that far, we can use our indexing to see if that's right.



Run

```
1
2 pythagoras = 'There is geometry in the humming of the strings, there is m
3 print pythagoras.find('string')
4 print pythagoras[40:]
5 print pythagoras.find('T')
6 print pythagoras.find('sphere')
7 print pythagoras[86:]
8 print pythagoras.find('algebra')
```

```
40
strings, there is music in the spacing of the spheres.
0
86
spheres.
-1
```

So let's print Pythagoras starting from index 40, we could print all the way to the end using a colon. And, when we run that, we see that it starts with string which is what we found with the find. We can search for other positions if we search in Pythagoras for the single letter T. Well that matches the beginning, so we should find the result at position 0. We can look for sphere. That will match sphere at the end. We get position 86. Let's print the quote from position 86. And we see the end of the quote starting from sphere. If we search for a string that's not in the string that we're using as the search string, so let's look for say, algebra, which was not in the quote from Pythagoras, we get the output negative one. That means the string was not found.

Quiz: Testing

The question for this quiz is, which of the following will evaluate to negative 1. Here we have the string test invoking find, passing in the string lowercase t. Here we have the string test in double quotes, using find to search for the string st. Here we have the string test in double quotes with a capital T. Searching for the string using find lowercase te. And here we have the string west, finding the string, test.

[ ] 'test'.find('t')
[ ] "test".find('st')
[x] "Test".find('te')
[x] 'west'.find('test')

Answer:

The answer is that find will return negative one when the target string is not found in the search string. Of the four examples here there are two where that's the case. For both of these, it is found, so in the first example we're looking for t within the string test. It occurs in two places. But it'll find the first one, so the result of this find will be zero. For this one we're looking for the string st. Strings are the same whether they have single or double quotes. It doesn't matter what we use, as long as we're consistent with the first and the end quote. And in this case we find the string st at position two, so that would return two. For this one we used an uppercase t, a lower case here. The find has to match exactly, it has to be the same character and case matters. So this would return negative one, and would be correct that this is one of the examples that returns negative one. And for the final one, we're searching for the string test, within the string west. They don't match exactly; so this would also return negative one.

Quiz: Testing2

So here's a more challenging quiz about find. This one will require you to think pretty carefully about how find is defined. For any string, so we're going to have a variable s that we've initialized to any string. Your goal is to determine which of these will always have the value zero. So the first choice, s.find (s). Notice there are no quotes around s. The second one is s.find('s'). The third one is 's'.find('s'). The fourth one is s.find(''). And the final one is s.find, passing in s concatenated with the string consisting of three exclamation points, one of my favorites, plus one. So check all of the answers where the expression here evaluates to zero, no matter what value the string s starts with. So try to answer this, just based on thinking, understanding the description of find. You can always also try things in the Python interpreter after that. And see if your interpretation is correct. But remember our goal is to answer the question where no matter what the string s is, the resulting expression is always zero.

[x] s.find(s)
[ ] s.find('s')
[x] 's'.find('s')
[x] s.find("")
[x] s.find(s+'!!!')+1

Answer:

So here's the answer. The result of the first expression is always 0. No matter what string s is, we always find s at the beginning of itself. So, think of s as any string. If we search for the string hello, in hello, well, we're going to find hello at the beginning of hello, and that's position 0. That's what find will return, no matter what string s is. The result of s.find(s) will always be 0. The result of s.find passing in the string 's', will only be 0 if s started with the letter s. So this is not always true. The third example is always true. Here we have the string s, that's a string with one character s, and what we're trying to find in it is the string s. Well, we find that at position 0. So that's the result find will produce. The fourth one gets a little more tricky. So here we're looking in the string s. We're looking for the empty string. Remember the definition of find says it will give us the first occurrence where that string appears. Well the empty string appears in s. We don't actually see it, but it's there. We found the empty string. There is no string there no matter what s is. We can find the empty string in it, because we don't need any string to find the empty string. So the result of s.find passing an empty string, will always be 0, no matter what the string s is. So the final one is also always in s for the string s plus !!!, well, no matter what s is. Even if it was a string that already included !!! . So let's suppose that's the value of s. What we're trying to find is now s plus !!! . So that's going to be what we had before, plus three more exclamation points. Searching for that within the string s will never be found. And we know it can't be found because this string is actually longer than the string s that we started with. So the result of this expression is always going to be -1. That means the string was never found. We add 1 to that, we get 0. That's why this expression will always evaluate to 0, no matter what the value of s is.

Quiz: Finding With Numbers

So, there's one other interesting thing we can do with find. And going back to the original description of find, we said that find returns the first position. There might other occurrences. And we might want to find those other occurrences. The way to find those other occurrences is, we can actually pass in an extra parameter. So, instead of just passing in one string, we can pass in the second parameter, which is going to be a number. Then when we pass in a number what find will output is the number of the position in the search string, where the target string appears, the first occurrence after that position.

So, it'll give the first occurrence where the target string appears in the search string, but starting from whatever position we pass in this number. So, if we pass in zero, well, it would start from the beginning, that would mean the same thing as the original find. If we pass in the position here, it would start from there and would still output the same value we found before. If we start from here, well then, it wouldn't find this occurrence, because this occurrence starts after that position, it would find this one. So, here's an example. We'll use this quote attributed to George Danton during the French Revolution. Translated loosely, it means audacity, more audacity, always audacity. I should mention that Danton was actually executed, so whether you want to follow this advice or not depends on how you think things worked out for him. But if we do a find with danton, passing in audace with no extra parameter, we'll get position five which is the position where audace starts. The first occurrence of audace is found. If we pass in a number as well, well if we pass in zero, we'll also get five. That's because starting from position zero, we find audace at position five. If we pass in the number five, we'll still get five as the result, starting from position five, we find audace at position five. If we pass in a number higher than five, let's pass in six, well now find is looking for an occurrence starting from position six, which is here. Instead of finding the first occurrence of audace, it'll find the first one that occurs after position six, which is the one here at position 25. Let's just print out the string from position six to confirm that.

So, printing from position six to the end, we see that the quote is udace, encore. And so on. I'll spare you from my poor French pronunciation, and that's why it found audace at this location, which is position 25. The result always counts from the beginning of the original string. Position 25 of the original string, which is where it found the second occurrence of audace. So if we pass in 25. We'll get 25 as the output, finding the position where this occurrence occurs. If we pass in 26, now it will start searching at position 26 which is the u here. We'll find at position 47. The final occurrence of audace. So, seeing the string from position 47 to the end, starting from here, we see that. Now, we're starting from position 48, which is this u. We'll get as our output, negative one. And that's because, although the string occurs many times in the string danton, it doesn't occur at all starting from position 48. And so, the output of find, when the target string is not found in the search string, is negative one and that's true if we start from position 48, we don't find the target string at all in the search string.

[ ] s[i:].find(t)
[ ] s.find(t)[:1]
[ ] s[i:].find(t)+i
[ ] s[i:].find(t[i:])
[x] None of these

Quiz: Finding With Numbers Quiz

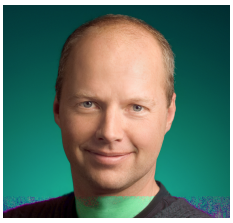
So now we're ready for a quiz to see if you understand the find command with the number as the second parameter, as well as indexing and selecting sub-sequences of strings. And the question is given any string variables, so we're going to initialize the variable s to hold any string. T to hold any

string, and i to hold any number, which of the expressions below is equivalent to s.find passing in t as the target string ni as the number? So here are the choices, s index i colon, find t. S.find t index colon i. S i colon.find t Plus i or s, i colon, find t, i colon. So check all the ones that are always the same as s dot find t, i no matter what s, t, and i are. And for this quiz, this is pretty tricky. Try to think of the answer yourself before trying to run anything into the Python interpreter. But feel free to also run things in the Python interpreter to try to confirm your answer and to see if things behave the way you expect.

Answer:

So this was a really tough question, and I'll admit I got it wrong twice myself in trying to answer it. But I'm pretty sure I know the correct answer now. Which is that none of these are actually equivalent to s.find(t,i). If you didn't get that, that's okay. Two of them are pretty close, and the two that are pretty close are the first and the third. But none of them are exactly correct and let's go through an example to understand why. So let's suppose we started with s as udacity. Let's make t city and make i three. If we do s.find(t,i), what we're going to get starting from position i which is in s and looking for city, we're going to find it right there. The output of s.find(t,i) with these variables is going to be three. So search is starting from position three, but it's still going to give us the output where that is position three. It gives us output in terms of the positions in the original string, s. So the first one, s[i:], where i is three, is going to be this string. So when we do the find where that's the string, we find city at position zero, and the result of this is zero. That's not the same as the result we expect, which is three. The second one doesn't actually make any sense. So here s.find evaluates to a number, and then we're trying to index from a number, that doesn't make any sense. We can only index from a string, select a sub-sequence of characters, there's no way to select a sub-sequence of characters from a number. So this is invalid, definitely doesn't produce the same output as s.find(t,i). The third one's the trickiest one. This is one that I originally thought would be equal and for this particular case it is. In this case when we do s[3:].find(t). That's going to produce the result zero we that we saw before. And then when we add i to it, we're going to get the value three. So in cases where we find the string t, in s starting from position i, this will be equivalent. If we made t something else, so let's suppose we made t dog, well then what's going to happen is that the result of s.find(t,i) when t is dog would be negative one. Find always returns negative one when the string we are searching for is not found. In this case, we would have negative one as the result for this, add i to it. That would give us the number two. That's not the same as negative one because this also is not equivalent. In cases where t is found, it is equivalent, but in cases where t is not found, it's not equivalent. The fourth one is also not equivalent. For the fourth one we're starting from selecting the sub-sequence of s, starting from position i. And we're looking for the string t where we sub-sequence from t starting from position i, this won't give us the same result as s.find(t,i). So this was kind of a tricky question. You are not upset if you didn't get it right but the answer is none of these are actually equivalent to s.find(t,i).

String Theory

DE: Great, so you've passed the last quiz on strings.

ST: And Dave, I think you've explained strings better than I've ever seen it. I mean, I've always wondered what string theory's all about. And, and physicists work really hard on it, and it looks so easy. So do you know what, what

DE: I have no idea what the physicists are doing. They must have very long and complicated strings to deal with.

ST: I hear you can get a PhD in string theory, and now you know all about strings.

DE: Well, I think we know enough about strings now to move on to the part where we're building the web crawler by extracting links from the page.

Programming Quiz: Extracting Links

So now you know enough about Python to be able to solve the problem that we started with at the beginning of this unit, which the problem of extracting a link from its page. Before we get to the code, I want to describe a little more carefully what's going on in a webpage. So we've talked about strings in Python and all a web page really is, is a long string. When you see a web page in your browser, it doesn't look like that. So here's an example web page, one of my favorite XKCD comics. And hopefully, you're starting to learn enough about Python to appreciate the power of Python to make you fly. Probably the rest of the comic, if you haven't done anything other than using Python, is a little hard to relate for now. But it's making fun of other languages where there's an awful lot of work to do something simple, like we've seen here, just being able to print out a string. But with Python, we can fly quickly, and you're going to learn to fly very quickly in this class. This doesn't look like just a string. We've seen just a string is a sequence of characters. When we look at a webpage like this, well, we see images. We see buttons. We see some text. We see things that are links and you can see the underlines these are all links. And the browser renders the webpage in a way that looks attractive. What actually was there though, started just as a stream of text. If you right-click on the webpage, one of the options you see is View Page Source. When you click on that, you'll see the actual source code. This is what came into the browser.

So, your browser sent a request, the URL is what's shown in the address bar. So, it's sent a request to xkcd.com/355. It sent that request and this is what came back. What came back is just a stream of text. We can look at that text and some of it is fairly hard to understand. So what's important is the links. Here's an example of a link. So, the link starts with a tag like this. The language HTML uses these angle brackets. And the angle bracket a href equals is how we start a link. That's followed by a string which is surrounded by double quotes, similarly to a string in Python. So, we have a double quote. Between the double quotes is a URL. The URL is the way of locating content on the web so here we have the URL http colon, that means it's a web request. We'll talk more in a later class about what http means and the protocols used to request web pages. What's important now is, that's a location. If we open that in a web browser, that will give us another page. What I'm looking at here is the link that is underneath the text for News/Blag. If we click on that link, that will take us to the page blag.xkcd.com. That was the page that we saw in the link here it said blag.xkcd.com. When we click on the link, that's where we went.

So to build our crawler, what we want to do for each web page, we want to find these links in the page. We're going to keep track of those links and we're going to follow them to find more content on the web. This is similar to what someone would do if their browsing. If they're clicking on every link of a page, following all the links they find, looking at all that content. That's a really good way to waste a horrendous amount of time if you do that yourself. We're going to build a web crawler that can do that automatically. So our goal is to take the text that came back from a web request, find a link in that text, which is going to be a tag that starts with a href equals and then extract from that tag the URL of the webpage that it links to. Those are the URLs that we're going to use in our crawler to make progress.

So by using what we've learned about strings, and what you've learned about variables, you know enough to be able to do that. What we want to do is find the beginning of a tag. And what the beginning of a tag is is the beginning of this text. We're looking for something that matches exactly the a href equals part. That's what the tags were here they all start with a href equals. Not all web pages have the same structure. There are lots of other ways to make a tag. The A could be a capital letter for example. There could be more spaces between the a and the href. The double quote doesn't actually need to be there. For what we do now, we're going to assume that all our web pages follow the same structure that we're seeing here. That each link starts with an a href without any funny spaces or anything else, has an equal, has a double quote, has the URL following that, and then another double quote. So that means we're looking for strings like this, we're looking to find the a href; that's followed by a double quote. After the double quote is the URL. This is what we actually care about; we want to find the URLs on the Web page. That's followed by a closing double quote and then, there's more that closes the tag. And there's lots of other stuff on both sides of this. But this is what we want to do. We want to find the tags that are links and then, within the tags that are links, we want to find the URLs. So we're going to assume that we start with the page contents in a variable. We'll call that page, and we're not going to worry today about how we got those page contents. We're going to provide a

function that does that. For the code that you have today, let's going to assume the page is already initialized. That it contains the content of some web page stored as a string and our goal is to find the URL of the first link in the page. That's going to involve a couple steps.

So what we want to do is find the start of the link. We want to find where we have the a href equals. We can't just look for the first string we find, because there's lots of other strings on the page that aren't URLs. So I think you know enough to do that, so we'll make it a quiz. So your goal for this quiz is to write some Python code that will initialize the variable start link to be the value of the position where the first a href equals. So the first tag that starts a link occurs in page, so you should assume that page starts with the content of some web page, and what we're doing is looking for the place where the first a href equals occurs, and that's the first link on the page.

Answer:

So here's the answer. We can do this using the find method. What we want to do is find in the search string page the target string a href. So we can do that with code like this. We're going to have start link as the variable that we're initializing. That's going to be the left side of the assignment. And we're going to set its value to the result of calling find where page is the search string and the target string is a href equals. And that will give start link the value of a number, which is the position where the first link tag is found on the page.

Programming Quiz: Final Quiz

Now what we want to do is extract the url. So we found this position, that's going to be the value of start_link. Now our goal is to extract this url. And the url starts from the first double quote that we find after start link, and it ends with the second double quote. So for the last quiz of this unit, your goal is to write all the code we need to end up finding that url. So this is going to be the hardest quiz that we've had so far. It's going to involve you thinking about variables, thinking how to use find in different ways. And write several lines of code yourself. Think carefully about how to do this. But I think if you understand what we've covered so far, you'll be able to do this yourself. So to be specific, here's the quiz, your goal is to write Python code that assigns to the variable url. These are all lowercase letters, a string that is the value of the first URL found on the page, in a link tag.

So you should assume that we have the variable page that's already initialized to the contents of the page. And in your browser this will be initialized to the contents of an example page. From the previous quiz we also have the value of start-link already worked out. It said start-link is the result of calling find on page, passing in the string, a href equals. And your goal is to finish this code, so at the end of this code, the value of url is the first link on the page. And if you print the value of url, and you've done all the code correctly for the example that we provided, you should get the result http://

udacity.com, which is the url that finds the Udacity webpage. So this is pretty tough quiz. There's a lot of steps. Think about it carefully. So it uses almost everything that you've seen in the unit today. You'll have to think about how to use the find method, possibly passing in numbers as well as strings, as well as variables to keep track of things. To find the result, you'll also need to think about how to use the string indexing operators to select a sub-sequence of a string.

Answer:

So here is the answer. What we have to do is find these two double quotes. So we know that we want to start looking from start_link, we can't start from the beginning of page. So we want to use find on page, passing in the double quote, which is what we're looking for, starting from the start_link position. So here's how we need to do that, we're going to initialize the variable, start_quote, which is where that double quote that starts the url is. And what we want to do is use page.find, to find in page. The string that we want to find is that double quote. To make a double quote as a string, we need to use a single quote, followed by a double quote, followed by a single quote. We also need to pass in a second parameter. We don't want to find the first double quote in the page, because that might not be part of a link tag. So we also need to pass in the variable start_link. So that makes find start from the position of start_link look for the double quote, and it will return the location where that start quote appears. We also need to find the end quote. We'll call that end_quote. To find the end_quote we don't want to start from the start_link. If we started looking from the start_link we'd find the start_quote again. To find the end_quote we need to look starting from after the start_quote. So we're going to use page.find again. Again looking for a double quote, so passing in the double quote as the string we're searching for. We need to start from the position of start_quote. We can't start from just the position of start_quote. If we started from start_quote, we'd find just the start_quote itself. We knew to start one after that position, so we need to add 1 to start_quote. And if we start looking from there, we'll find the first quote after that. That's the quote that ends the string. So the final thing we want to do, is set the variable url. So we will initialize url, to the string that we find between the start quote and the end quote. And so we can do that using the string selection. We're selecting from the string page. We have to think carefully about where we start. We don't want to start with a start_quote because that would include the double quote in the url. We just want the url not including the double quote, so we're going to start with the start_quote plus 1. And we're going to go from there until the end_quote. We do not want to include the closing double-quote. We don't include that one in the string. So we don't need to subtract 1 from the end_quote. And that will get us the url, extracted from the first link on the web page.

Great Job!

So congratulations, you have made it to the end of unit one. You are learned a lot of computer science already, you know what a program is, you've learned about variables, you have learned about

expressions and grammars, you have learned about strings in Python. So now it is time for you to work on homework one on your own. And that will check that you understood everything from this class and prepare you to get started on unit two. And we're well on our way towards learning a lot of computer science, as well as building our web crawler and then building our search engine.