CS212 Unit 2

57

Gundega

Contents

1 CS212 - Unit 2

This text was auto-generated from subtitles. Please feel free to improve it! Please remove this line when the text has been improved. Thank you very much!

Contents

1.1 1. Zebra-Puzzle

Unit2-1

Welcome. Let's talk about a cliche: the back of the envelope. Much easier to write on the back of the envelope than on a napkin, and it's a really valuable skill. It's a valuable skill in real life to be able to do quick and dirty calculations, and it's especially useful for computer programmers. It allows computer programmers to have the important virtue of being lazy. You don't normally think of lazy as being a virtue, but it is.

It allows us to say we're going to come up with the simplest design we can, validate on the back of an envelope that that design is efficient enough for what we're trying to do, and then we can stop. We don't have to try to make it more complex. This whole class is about managing complexity, and one of the most important ways to manage complexity is to leave it out completely, just go for the simple solution. If we can do that, then we're well on our way to better designs.

In this class we'll learn how to do that, we'll learn how to do the calculations of when you're efficient enough, we'll learn when to stop, and we'll learn how to make programs more efficient.

I'm going to start with a well-known puzzle called the Zebra Puzzle.

Here's a description of it.

- The Zebra Puzzle
    1. There are five houses.
    2. The Englishman lives in the red house.
    3. The Spaniard owns the dog.
    4. Coffee is drunk in the green house.
    5. The Ukrainian drinks tea.
    6. The green house is immediately to the right of the ivory house.
    7. The Old Gold smoker owns snails.
    8. Kools are smoked in the yellow house.
    9. Milk is drunk in the middle house.
    10. The Norwegian lives in the first house.
    11. The man who smokes Chesterfields lives in the house next to the man with the fox.
    12. Kools are smoked in the house next to the house where the horse is kept.
    13. The Lucky Strike smoker drinks orange juice.
    14. The Japanese smokes Parliaments.
    15. The Norwegian lives next to the blue house.

  Who drinks water? Who owns the zebra? Each house is painted a different color, and their inhabitants are of different nationalities, own different pets, drink different beverages and smoke different brands of American cigarettes.

We're going to try to address this puzzle, see if we can come up with a program to solve it, and explore the methodology for how we come up with that solution and the process of deciding what's a good enough solution and whether a brute force solution will work.

You can read the description of the puzzle here.

Now let's start to analyze it. We'll begin with an **inventory of the concepts** in the puzzle just to make sure that we understand what's going on.

The first concept is **houses**. We're told there's 5 of them.

And then there's **properties** of the inhabitants of these houses and of the houses themselves. So there's nationality, colors of the houses, the pets that they own, the drinks that they drink, and the smokes that they smoke.

And then in addition to properties, there's a notion of **assignment of properties** to houses. And you can think of that either way. You can think of it as assigning house number 2 the color red or think of it as assigning red to house number 2.

Then there's a notion of **locations**, the locations 1 through 5 that mention the idea of the first house and the middle house and of the next to relation and of the immediately to the right relation.

And I think that covers most of what was in the specification.

Let's go back to it and see if it works.

IMAGE

So I'm seeing lots of concepts that we've already covered:

There's a few words that we haven't covered, things like *lives in* and *owns the*.

We covered them in a generic sense of saying it's an assignment of Englishman to a house and assignment of the dog to a house, but the question is, do we need to separate out the different types of assignment?

So the question is, are we missing this idea of a property name with a description attached?

So for example, the property name would be nationality and the description is lived in. Do we need to name them like that, or do we just need the notion of a property group to say that there are these properties of Englishman, Spaniard, Norwegian, Japanese, and Ukranian, and the 5 of them go together but we don't need the name for them, or can we ignore this notion of grouping altogether?

This is somewhat subjective --what's a good design choice?-- but tell me which of these 3 do you think are plausible or reasonable design choices and check all that apply.

QUIZ question

1.1.1 1.1. Zebra-Puzzle Answer

Unit2-1 Answer

[Norvig] My answer is defining property names would be a fine design choice. It would probably help us understand what's going on in the definition of the program. Defining groups without names would also be fine.

Ignoring the groups I think would not work, and here's the problem. We have to know that if red is assigned to house number 2, then blue cannot be assigned to house number 2, but orange juice can be assigned to house number 2. So there's this idea that properties within a group are mutually exclusive and properties outside of the group are not. We need to represent that in some way.

1.2 2. Where's the Spaniard

Unit2-2

[Norvig] The crux of the whole matter is doing the assignments. There are 2 ways to do it. We can deduce what assignment must be true - that it must be the case that the Englishman is in a certain house. Or we can just try different possibilities - try to put the Englishman in house 1, then in house 2, then in house 3 and see what works. You'd have to be very clever to figure out how to do all these deductions, and I don't want to try to be that clever. I want to see if I can get by with just trying all the possibilities, make the computer do the work so I don't have to.

So let's approach that and let's say we're going to try all possibilities. And so first we put the Englishman into house 1, and later we'll try him in houses 2, 3, 4, and 5. What about the Spaniard? If the Englishman is in house 1, what possibilities should we try for the Spaniard?

Give me your best answer:

- all the numbers;
- only 1;
- only 2;
- or 2, 3, 4, 5.

1.2.1 2.1. Where's the Spaniard Answer

Unit2-2 Answer

And the answer is **2, 3, 4, 5**, because Spaniard is in the same property group as Englishman, and so it can't be in 1 but it can be in any of the others.

1.3 3. Counting Assignments

Unit2-3

Within a class of 5 properties, the class must cover the 5 houses but in any order.

How many ways can we make that assignment? In other words, here's 5 houses, here's 5 properties in a property group, and here's 1 possible assignment.

How many different assignments are there of assigning these properties and matching them up with the houses?

How many ways are there:

- 5
- 5^2
- 2^5
- 5!

1.3.1 3.1. Counting Assignments Answer

Unit2-3 Answer

And the answer is **5!**.

And you can see that because there's 5 different houses that you could assign red to, and then for each of those 5 assignments there would be 4 that you could assign green to of the remaining unpainted houses and then 3 for blue and 2 for yellow and then only 1 left for ivory. That's 5! or 5 factorial.

1.4 4. Multiple Properties

Unit2-4

But we don't just have 1 property class; we have 5 property classes. We've got the nationalities and the pets and so on. And so if we want to make all the different ways of making assignments of all 5 properties to all 5 houses so each house will have a color, a nationality, a pet and so on in each of the possible ways of making that assignment, how many is that?

Is it:

- 5 * 5!
- $5!^2$
- $5!^5$
- 5!!

1.4.1 4.1. Multiple Properties Answer

Unit2-4 Answer

And the answer is $5!^5$ because for each of the assignments of 1 property we can have the same number of assignments of the second property, the third, the fourth, and the fifth.

1.5 5. Back-of-the-Envelope

Unit2-5

And now, how much is 5 factorial to the 5th?

Just back of the envelope, is it approximately:

- a million
- 20 billion
- 5 trillion

1.5.1 5.1. Back-of-the-Envelope Answer

Unit2-5 Answer

FIXME

And the answer is 20 billion. How do you do that back of the envelope? And we rounded down to get from 120 to 100, so we should round back up, and maybe it's somewhere around 20 billion. It turns out that the actual number is 24.9 billion, approximately, so that's not a bad estimate.

1.6 6. Leaving Happy Valley

Unit2-6

We're in this range where we might be done but we're not quite sure. Our computers can do about a billion instructions per second or so on a good second - that is, one where they're not wasting part of the second with a page fault or a cache miss. If the answer had turned out to be in the millions, we could say, "Oh, great. We're done." If it had turned out to be in the trillions, we could say, "It's totally infeasible." "We need a better solution." But if it's somewhere in the middle with the billions, then we're not quite sure.

We better try to refine the result a little bit to tell if this brute force approach is going to work.

Think of the space of solutions in terms of execution time as like a contour map.

IMAGE

And here deep down in the valley where there's only millions of computations needed, millions of instructions needed to complete the answer, then we're really happy because we know that's going to go really fast. And outside of the happy valley there are these high peaks where we have trillions of computations needed, and there we're going to be sad. If we're stuck out in these domains, we're going to have to somehow find a path back in because we can't just go ahead and calculate when we're out at the trillions. And in the middle here where we need billions of instructions to complete our computation, then we're not quite sure. So maybe we're happy here and we can stay in this domain. Maybe we want to find our path through the wilderness back into the happy valley. It depends on the problem.

And we're going to try to look at problems in terms of this space, try to find our way back in, but know when to stop when we've got close enough.

Now let's keep thinking about what it means to do an assignment. Try to get just a little bit more concrete about it.

If we want to assign red to house number 1,let's think about the ways in which we could implement that.

```
house[1].add('red')
```

- Here's 1 possibility where we say we're going to have an array of houses. We take number 1 and we add red to its set of properties. That means that each house is represented as a **set**.

```
house[1].color='red'
```

- Here's another possibility where we take house number 1, we set its color property equal to red. Here each house would have to be represented as a user-defined custom **class** which had properties for color and nationality and so on.

```
red = 1
```

- Here's a third possibility. We have a variable called red, and we just assign that the number 1. So here we were assigning properties to houses, and in this one we're assigning houses to properties.

I want you to check all the approaches that you think would be reasonable designs for implementing assignment.

1.6.1 6.1. Leaving-Happy-Valley

Unit2-6 Answer

I know there can be matters of opinion here, but from my point of view they're all reasonable. So they would all work. They'd all be fine.

But for the moment, this one - **red = 1** looks just a little bit simpler, so I'm going to go with that until I have some proof that the simple won't work and that we'll have to go to something more complicated.

In this approach we named the properties. We talked about the possibility of whether you have to do that or not. Here we aren't naming them, and so it will be up to me as the programmer to manage the groups of properties.

Here's 1 way I could do it.

I could simultaneously assign the 5 properties that are in the same group of color to the 5 houses. Of course this would only be 1 of the possible assignments. After I tried this one, I'd have to try another assignment - maybe [1, 3, 2, 4, 5] - and I'd have to go through all the possibilities.

Can you think of a good way to do that? What statement or other implementation can you come up with to have the 5 properties go through all the combinations of possible houses?

And if you want, pause the video now and try to think of that on your own.

1.7 7. Ordering-Houses

Unit2-7

So here's 1 approach.

First we'll define the houses. Say there's 5 houses. That was the number 1 thing stated in the problem.

Then we'll say orderings is some collection of possible orderings of the houses, so [1, 2, 3, 4, 5] would be 1 element of orderings,

[2, 1, 3, 4, 5] would be another element and so on.

And then we just use a for statement to iterate the 5 variables in the property group through all the possible orderings and then put in our code here.

The question is, what is this function F that gives us all the orderings?

Is it:

- the permutation function
- the combinations function
- the factorial function
- something else

1.7.1 7.1. Ordering-Houses

Unit2-7 Answer

And the answer is permutations. That's sort of the definition of all possible orderings is called the permutations.

1.8 8. Length-of-Orderings

Unit2-8

It turns out that Python has a permutation function in the itertools module, so you could import that or you could write one yourself.

If you do that, one question is, what would be the length of orderings? Tell me that.

1.9 9. Estimating-Runtime Answer

Unit2-9

Now, suppose we arrange our program like this. So we set up the houses and we have a list of all 120 possible orderings and then we iterate each of the 5 properties in each of the 5 property groups through all those orderings. And so now, 5 levels nested deep into our code we've got an assignment of all 25 properties. Each of them is assigned to one of the houses, and now we can check the constraints, like the Englishman lives in the red house and the Spaniard owns the dog and so on. Those can all go here. If we find something that checks out, we can just report we've got an answer.

Now, I want you to go back to the back of your envelope and tell me if we wrote the program this way and filled in this code with all the constraints and then ran it, about how long do you think it would take to find a solution?

- About a second
- a minute
- an hour
- or a day

Click on the best answer.

1.9.1 9.1. Estimating-Runtime Answer

Unit2-9 Answer

The best estimate is about an hour, and we'll go to the back of the envelope again to figure that out.

IMAGE?

1.10 10. Red-Englishman

Unit 2-10

$5!^5$ was 24 billion. And if we assume we have a 2.4 gigahertz computer, which is fairly typical, then if all this could be compiled into 1 computer instruction, then it would take 10 seconds.

But of course that's ridiculous. You can't do all of that in 1 instruction.

If it was 100 instructions, then that would mean it would be 1000 seconds, which is about 16 minutes, but that seems too little.

It's probably going to be more like 1000 instructions to make it all the way through doing all this and then checking the constraints, and so that would be about 160 minutes. And so an hour to 3 hours. Maybe we'll say 2 or 3 hours, somewhere in there. It might be as little as 1 hour. We can't really tell because this is just an estimate. But it's definitely going to be in the hour range and not in the minutes or day range.

In fact, I actually ran this program, and we'll come back later and see how well this estimate of somewhere in there worked out. Let me just say because we've learned that this will take somewhere in the range of an hour, don't try to run it here in the browser by hitting the Run button because we timeout the calculations after just a few seconds, so it's not going to work here.

You could type it into your own computer if you have Python running on your own computer, set it going, and see how it works.

But first we have to figure out how to do all the constraints. So we already did constraint number 1, which was houses = [1, 2, 3, 4, 5]. Now let's do constraint number 2, which is that the Englishman lives in the red house. We want to write some if statement to check if the Englishman lives in the red house and if so, we go on; if not, we give up.

And so how do we check if it's true that the Englishman lives in the red house? So if, and then I want you to fill in some expression here. We'll just say that this is constraint number 2. Tell me what you could put in here.

IMAGE?

1.10.1 10.1. Red-Englishman Answer

Unit2-10 Answer

And the answer is all we have to do is check to see if Englishman and red have been assigned the same house number.

And we just do that with saying

```
·· 1 if (Englishman == red)¶
¶
```

And I should note just as an aside if you wanted to be sort of clever and make it look more like English, in this case it would be okay to say

```
·· 1 if (Englishman is red)¶
¶
```

because is is checking for whether 2 objects are identically the same object. Equals checks to see if they have the same value. And it turns out that Python handles small integers like 1, 2, 3, 4, and 5 as a single identical object. So

the is check would work as well as the equals check here.

1.11 11. Neighbors

Unit2-11

Now, before we go on to constraint number 3, I want to go backwards a little bit and say there's 2 concepts we haven't talked about yet: the concepts of being next to and immediately to the right of.

Immediately to the right of--well, because we've used house numbers to assign houses, we can say that house 1 is immediately to the right of house 2 if they differ by 1, if h1 - h2 is 1. So house number 3 is immediately to the right of house number 2.

What I want you to do is fill in the code for nextto to say if 2 houses are next to each other.

1.11.1 11.1. Neighbors Answer

Unit2-11 Answer

And the simplest way to do that is just ask if the absolute value of the difference rather than the positive or negative value is equal to 1.

But an alternative way to say it if you wanted to break it down in terms of the other primitive, you could say that nextto (h1, h2) is defined as immediate right of h1, h2 or immediate right of h2, h1.

1.12 12. Slow Solution

Unit2-12

Now let's put the whole thing together. The code below shows the brute force slow solution, followed by a description of how it works.

```
·· 1 import itertools¶
·· 2 ¶
·· 3 def imright(h1, h2):¶
·· 4···· "House h1 is immediately right of h2 if h1-h2 == 1."¶
·· 5···· return h1-h2 == 1¶
·· 6 ¶
·· 7 def nextto(h1, h2):¶
·· 8···· "Two houses are next to each other if they differ by 1."¶
·· 9···· return abs(h2-h1) == 1¶
· 10 ¶
· 11 def zebra_puzzle():¶
· 12···· "Return a tuple (WATER, ZEBRA) indicating their house numbers"¶
· 13···· houses = first, _, middle, _, _ = [1,2,3,4,5]¶
· 14···· orderings = list(itertools.permutations(houses)) #1¶
· 15···· return next((WATER, ZEBRA)¶
· 16·········· for (red, green, ivory, yellow, blue) in orderings¶
· 17·········· for (Englishman, Spaniard, Ukranian, Japanese, Norwegian)
in orderings¶
· 18·········· for (dog, snails, fox, horse, ZEBRA) in orderings¶
· 19·········· for (coffee, tea, milk, oj, WATER) in orderings¶
· 20·········· for (OldGold, Kools, Chesterfields, LuckyStrike,
Parliaments) in orderings¶
· 21·········· if Englishman is red··················· #2¶
· 22·········· if Spaniard is dog····················· #3¶
· 23·········· if coffee is green····················· #4¶
· 24·········· if Ukranian is tea····················· #5¶
· 25·········· if imright(green, ivory)··············· #6¶
· 26·········· if OldGold is snails··················· #7¶
· 27·········· if Kooks is yellow····················· #8¶
· 28·········· if milk is middle····················· #9¶
· 29·········· if Norwegian is first·················· #10¶
· 30·········· if nextto(Chesterfields, fox)·········· #11¶
· 31·········· if nextto(Kools, horse)················ #12¶
```

```
· 32···········  if LuckyStrike is oj·················· #13¶
· 33···········  if Japanese is Parliaments············· #14¶
· 34···········  if nextto(Norwegian, blue)············· #15¶
· 35···········  )¶
¶
```

I've defined zebra_puzzle. It's a function. It doesn't take any arguments because there's only 1 puzzle. There aren't different puzzles that require different arguments.

It's going to return a pair of WATER and ZEBRA, so the 2 house numbers indicating which house drinks water and which house owns a zebra.

I've repeated houses = [1, 2, 3, 4, 5] and I've also defined first and middle. I've repeated the work of figuring out all the orderings. That's constraint #1.

And then I've written the whole function as a generator expression, and I chose to do that rather than sticking with the nested for loops just because the structure is a little bit easier.

What we're doing here is we're asking for the next version. So in other words, the first time through we're asking for the very first solution to this iterator expression where we say iterating through the 5 properties, if each of the tests is true, then return the values of WATER and ZEBRA and then keep on doing that for all possibilities.

But I'm only interested in the very first one, which the next after nothing is the first.

So go ahead and return that. Each of the constraints is very easy to state.

Englishman is red, the Spaniard is dog, coffee is green and so on.

We have some immediate rights and nextto, and that's the whole problem. So in some sense we have a solution; in another sense we know that it's going to take something like an hour or 2 to complete, and maybe we're not happy with that, so we're going to make it faster.

But before I do that, I want to do a little bit of review of these generator expressions, because probably you haven't seen this too much before.

1.13 13. List Comprehensions

Unit2-13

And before I do generator expressions, we're going to do list comprehensions.

You've seen them before. We saw in Unit 1 we said that we could get the list of suits by saying

```
·· 1 [s for r, s in cards]¶
¶
```

And so the format here is that the first 's' is the individual term of the list comprehension and it is followed by a "for clause".

And in general for list comprehensions, we're going to have 1 for clause at least and then, optionally, we could have more "for" or "if" clauses at the end.

And that's what a list comprehension looks like in general.

Now, what does the list comprehension mean?

This expression is similar to saying we're going to write out a loop where we say the result starts as the empty list, then we take the for part and then we take the term and append that into the result. And then when we're all done, result will hold the value of this expression. This would be written as:

```
·· 1 result = []¶
·· 2 for r, s in cards:¶
·· 3···· result.append(s)¶
```

```
¶
```

Now I'm going to show you a more complex list comprehension.

Say we wanted the list of suits for all our face cards. Then we could say s for r, s in cards if the rank of the cards is, say, in jack, queen, king. This would be written as:

```
·· 1 [ s for r, s in cards if r in 'JQK' ]¶
¶
```

I don't know why you'd want that, but there's a possibility. So I've got the term I'm building up, I've got 1 for loop controlling it, and now I've got an if statement, and that's as if we had written this code:

```
·· 1 result = []¶
·· 2 for r, s in cards:¶
·· 3···· if r in 'JQK':¶
·· 4········ result.append(s)¶
¶
```

if we had inserted inside the for loop an if statement and then only done the appending of the term s if the if statement is true.

And in general, we can have any number of for statements, if statements, more for statements, more if statements. We can keep on adding those for clauses and if clauses and intermingling them in any order we want, as long as the for clause comes first. And we can have that 1 big list comprehension, like:

```
·· 1 [ s for r, s in cards if r in 'JQK'¶
·· 2···················· for ...¶
·· 3···················· if· ...¶
·· 4···················· for ...¶
·· 5···················· for ...]¶
¶
```

That corresponds to putting in more fors and ifs here in this loop, like the following:

```
·· 1 result = []¶
·· 2 for r, s in cards:¶
·· 3···· if r in 'JQK':¶
·· 4········ for ...¶
·· 5············ if ...¶
·· 6·············· for ....¶
·· 7···················· for ...¶
·· 8························ result.append(s)¶
¶
```

Now, going back to looking at the form of the list comprehension again (repeated here again for clarity with the text following it):

```
·· 1 [ s for r, s in cards if r in 'JQK'¶
·· 2···················· for ...¶
·· 3···················· if· ...¶
·· 4···················· for ...¶
·· 5···················· for ...]¶
¶
```

The whole thing is read left to right except for the term ('s' in the above example).

So the way to read a list comprehension is to say I'm going to ignore the first 'term' part for now, and then I think of it as saying it's a bunch of nested statements-- for statements, if statements, for statements. They all get nested deeper and deeper. Then when I get to the end, the very last clause, now I'm appending together the term, so now read the term.

And that's why it's okay that term at the start of the list comprehension looks like it's referencing a variable s that hasn't been defined yet. That's okay because s has been defined in the following for clause.

And it looks like it's used before it's defined, but that's not the case, because when it's actually used is right down here at the end.

1.14 14. Generator Expressions

Unit2-14

So that was list comprehensions. Now let's look at generator expressions, which is almost the same idea.

The syntax is the same in that a generator expression consists of a term, a mandatory for clause, and then optional for and ifs clauses - as many of those as you want--0 or more.

There's 2 differences. The generator expression uses parentheses instead of square brackets. Square brackets mean list; parentheses means generator. And then the other difference is that the computation doesn't get done all at once. Instead, a generator expression returns a value, which is a promise to do the computation later.

So if we say g = this and then g is this promise, it hasn't done any calculation yet. It hasn't calculated any of the terms. And then I can ask, give me the next g. Then it starts doing the calculation and it keeps on looping through the for clauses or maybe multiple for clauses until it finds the first term and returns that. And then if I want, I can again ask for the next g and it will give me the second one and so on.

Let's look at an example. Here I've defined the function sq for square of x. It takes in the value x, prints out that it's been called, and returns x * x. Here I've defined a generator from this generator expression that says

- g = (sq(x) for x in range(10) if x%2 == 0).

So that's saying if x is an even number. And notice nothing has happened yet. We didn't get any printing of square was called, so square hasn't been called yet.

The generator function is this promise. We can look at it. It says it's a generator object, but no computation has been done yet. We can ask for the next g and now, finally, square gets called with 0 as an argument, and we return 0 as a result. We can do that again. We get 4, 16, 36, 64. And what do you think is going to happen next?

Now we're getting to the end of the loop. Range(10) means 0 through 9, so there are no more. So now when we ask for the next one, Python raises this condition called StopIteration. So it's saying, "I've gotten to the end." "I have to stop the iteration because there's no more I can give you." "I can't give you the next one." This seems a little bit inconvenient because now I've got these errors and my program has to deal with them, but the idea is that you rarely will be calling next directly. Rather, most of the time you'll be doing this within a for loop. So I can say something like this where I say for x2 in this expression do something, and now the protocol for a for loop arranges to call the generator each time, to call the next function, and to deal with the StopIteration exception and catch that. And so everything works fine.

I can also convert the results. Here I've said I've got a generator expression and I'm converting that into a list. It does all the work and then it returns the result as a list. So I never have to deal explicitly with those StopIterations.

Why do you think I chose generator expression to implement the zebra puzzle?

- Do you think I wanted to confuse students;
- have less indentation so that the code would fit on the page;
- stop early as soon as I found the first result;
- or make the code easier to edit,
- to move around the various pieces of the constraints and so on?

Check all that apply.

1.14.1 14.1. Generator Expressions Answer

Unit2-14 Answer

The answer is no, I didn't try to confuse you. In fact, I'm trying to show you a very useful tool in generator expressions.

Yes, I thought the indentation was important. With all those fors and ifs, I would have run out of space across the page.

Yes, I wanted to stop early, and so a list comprehension would have been a bad idea because a list comprehension would have had to do all the work, whereas a generator expression can stop as soon as it's done.

Having statements rather than expressions also would allow me to stop early.

And yes, it's easier to edit.

If I wanted to move around the order of the clauses, instead of having the indented structure that I would have with statements I have a flat structure and it's easy for me to pick out 1 of the constraints and move it somewhere else in the list without having to worry about getting the indentation right. So editing expressions is easier than editing statements in Python.

1.15 15. Eliminating Redundancy

Unit2-15

I wrote this version of the function, and as soon as I wrote it, as I was recording this video I set it running. And you know what? It's still running. It hasn't completed yet. I've got to admit I've done this before, so I have an idea of how long it's going to take. But we're in a race with it.

Let's see if before it completes - we know it's going to take on the order of an hour or so - can we write a program that's better and faster and maybe even though that program's got a head start, we can catch up with it and finish first?

The problem with this program is it goes through all this work to try all the 5 factorial to the 5th combinations and then they get ruled out really early. And some of the combinations, it seems silly that we're bothering with them. So if the Englishman is not red, we should know that by the time we've got through the second set of assignments here. Here we've assigned red to some house and Englishman to some house. If we didn't assign them to the same house, why are we bothering to go through all the possibilities for the other properties?

And so what we could do is move this constraint up to the earliest time in which both Englishman and red are defined, so there. Now I've moved it up, and now if Englishman is red is false, then we don't even have to bother to go through all 3 of these loops. So we're going to eliminate a lot of work in just that 1 clause.

Now let's consider another of the constraints. Let's look at immediate right of green and ivory. Can we move that up? And where can we move it up to?

Can we move imright(green, ivory) up to here, here, here, or here? IMAGE

1.15.1 15.1. Eliminating Redundancy Answer

Unit2-15 Answer

And the answer is we can move it all the way up to the top because both green and ivory are bound at this point.

1.16 16. Winning the Race

Unit2-16

And if we keep on going, moving every constraint up to the highest point it will go, we get this.

```
·· 1 import itertools¶
·· 2 def imright(h1, h2):¶
·· 3···· "House h1 is immediately right of h2 if h1-h2 == 1."¶
·· 4···· return h1-h2 == 1¶
·· 5 ¶
·· 6 def nextto(h1, h2):¶
·· 7···· "Two houses are next to each other if they differ by 1."¶
·· 8···· return abs(h1-h2) == 1¶
```

```
·· 9 def zebra_puzzle():¶
· 10···· houses = [first,_,middle,_,_] = [1,2,3,4,5]¶
· 11···· orderings = list(itertools.permutations(houses))¶
· 12···· return next((WATER,ZEBRA)¶
· 13········ for(red, green, ivory, yellow, blue) in orderings¶
· 14········ if imright(green, ivory)····· #6¶
· 15········ for (Englishman, Spaniard, Ukranian, Japanese, Norwegian) in
orderings¶
· 16········ if Englishman is red·········· #2¶
· 17········ if Norwegian is first·········· #10¶
· 18········ if nextto(Norwegian, blue)····· #15¶
· 19········ for (cofee, tea, milk, oj, WATER) in orderings¶
· 20········ if cofee is green············· #4¶
· 21········ if Ukranian is tea············· #5¶
· 22········ if milk is middle············· #9¶
· 23········ for (OldGold, Kools, Chesterfields, LuckyStrike, Parliaments)
in orderings¶
· 24········ if Kools is yellow············· #8¶
· 25········ if LuckyStrike is oj··········· #13¶
· 26········ if Japanese is Parliaments····· #14¶
· 27········ for (dog, snails, fox, horse, ZEBRA) in orderings¶
· 28········ if Spaniard is dog············· #3¶
· 29········ if OldGold is snails··········· #7¶
· 30········ if nextto(Chesterfields, fox)¶
· 31········ if nextto(Kools, horse)¶
· 32···· )¶
· 33 ¶
· 34 print zebra_puzzle()¶
¶
```

And now I've checked, and the original version of the puzzle that I set running is still running, and now I can hit Run on this and in less than a second I see my result, (1, 5), meaning water is drunk in house number 1 and the zebra is owned in house number 5.

This is an amazing thing.

It's as if we had a car race and we had the start here and the finish someplace far away and we had this competitor car which was the original zebra puzzle car - we'll call him z for zebra. We said, "Go!" and it started out down the path towards the finish.

We wanted to build a competitor to that, but we just let it go. It's running, and we're thinking and we're analyzing and we're not doing anything, and it's getting farther and farther ahead and closer to the finish line. Maybe we spent half an hour and it's halfway through. It's gotten up to this point. We're still stuck at the Start sign, but what we're doing is we're building a better car. We're building a super fast race car, and we're putting the pieces together using what we know, and eventually this car has gotten a long ways along the way and then we say, "Now we're ready to go."

And when we hit Go, zoom, we're there in less than a second, what took this car who had a half hour head start is not even close.

So it's like the tortoise and the hare, only in reverse.

And by thinking and coming up with a better design, we were able to beat this original design even though we gave him a half hour head start. We won the race. We get the checkered flag. We should be happy at what we've done. And congratulations.

Now, if you're the type who just cares about winning the race and finishing first, we can stop here. But if you're a little bit more analytic and you like to know the exact scores and times, then we've got work to do. Here's 1 thing I can do. There's a module in Python called time. In that module there's a function called clock which gives you the time of day.

And so I can set t0 to be the time before I call the zebra_puzzle, t1 to be the time afterwards, and just return the difference so that will tell me how long the zebra puzzle took. If I hit Run, the answer it came back with was 0.0 seconds. That's because the system I'm running on doesn't have an accurate enough clock. And I know it's accurate down to the thousandth of a second, so all we can say here is that it took less than a thousandth of a

second. I've run it on other systems, and it comes back at 0.00028 on that other system. I was able to do this, but this looks like a good opportunity for generalization.

Why do I want a function that only times the zebra puzzle? It would be great if I had a function that could time the execution of any function. So here I've defined a function called timedcall which takes another function as input, sets up the clock, calls that function, calls the clock again to figure out the time, and tells me what the elapsed time is.

So I could call timedcall of zebra_puzzle to get the answer of how long that took, or I can apply this to any function. So I built a useful tool that I can use again and again. I can make it even more useful by doing 2 things. One is saving up the result and returning that as the second value in case we're interested both in the result and in how long it took, and secondly, allowing functions that take arguments.

You may not have seen this notation before, so let's talk about it for a second.

1.17 17. Star Args

Unit2-17

So the *args notation appears in 2 places:

```
·· 1 def something(fn, *args):· # function definition¶
·· 2 ¶
·· 3···· ... fn(*args)········· # function call¶
¶
```

It appears in the definition of a function, and it can appear in a function call.

In the definition of a function, what it means is this function can take any number of arguments and they should all be joined up into a tuple called args. So if the call to the function looked like this:

```
·· 1 something(f, 1, 2, 3)¶
¶
```

then 1, 2, 3 would be bound up into a tuple and assigned to args. So within the body of something, args is equal to the tuple (1, 2, 3).

And then this part here:

```
·· 1···· ... fn(*args)········· # function call¶
¶
```

means take that tuple and apply it not as a single argument but rather, unpack it and apply it to all those arguments. So fn(*args) is equivalent to writing fn(1, 2, 3).

It's just a way of packing and unpacking arguments with respect to function calls.

One thing I want you to notice here is that we used a function as a variable, as a parameter or something that's passed into another function.

```
·· 1 def timedcall(fn, *args):¶
·· 2···· "Call function with args; return the time in seconds and result."¶
·· 3···· t0 = time.time()¶
·· 4···· result = fn(*args)¶
·· 5···· t1 = time.time()¶
·· 6···· return t1-t0, result¶
¶
```

We're using the property of Python that functions are first class objects just like integers and lists and everything else. Functions are objects that you can pass around.

That's very powerful because that way we can control both what happens and when it happens.

The idea here is that we want to call this timedcall() function, but we want to delay calling of the function until we've already started the clock. We want it to happen between the time we start the clock and the time we end the clock.

And in order to do that, if we tried to write something like this--

```
·· 1···· timedcall(zebra_puzzle())¶
¶
```

--then timedcall wouldn't work.

It would be too late because what we pass to timedcall would be the result of zebra_puzzle.

We don't want to pass the result; we want to pass the function that we're going to call and be able to delay execution of that function until the right time, until we're ready to set the clock.

And so functions allow us to do that--to delay execution until later.

1.18 18. Good Science

Unit2-18

Now, let's get back to this idea of taking timings.

When we're taking timings we're doing measurements. Now we've changed what we're doing from having computer science being a mathematical enterprise to having it be a experimental enterprise-- an experimental science. We're actually doing science.

It's as if we're dealing with the messy world with some chemicals in a beaker, and we're making measurements of those rather than trying to do mathematical proofs.

One thing we know about doing experimental science is that one measurement isn't going to be good enough. That was clear when we got a measurement of 0.0.

We know that wasn't right. We know that there can be errors in measurements.

One way in science that we deal with that is by repeating the measurements multiple times.

I want you to tell me why you think some reasons are that scientists, when they do an experiment, take more than one measurement.

Is it because:

- we want to reduce the chance of some external event effecting the result?
- we want to reduce the natural random variation?
- we want to reduce errors in the measurement process?

Check all that apply.

1.18.1 18.1. Good Science Answer

Unit2-18

The answer is that all three of these are reasons to repeat measurements -- take more than one measurement.

Now, just by itself taking repeated measurements won't solve all these problems. If you have a systematic error, then repeating them will just repeat the same systematic error, but they certainly are required, although we may need more.

1.19 19. Timed-Calls

Unit2-19

Let's build a better tool. Here I built a function called "timedcalls" with the plural s rather than a single timedcall, and it takes a number N, saying repeat this timing N times. Then it takes the function and the arguments to apply to. It builds up a list of the timed calls. It throws away the results and just takes the time. Then it returns three values, the minimum of the times, the average of the times, and the maximum of the times. From those you can do whatever statistical analysis you want to do in order to get a better feeling for what the timing is like.

Now, if the function runs very quickly-- say, if the function took 100th of a second then you might want to give an N of 1000 or so. If the function takes about a second to run, maybe you don't want to wait that long, or maybe you want to give a smaller value of N. Part of the problem is if you have a good idea how long the function takes, then you can be precise about what a good value of N is. If you don't, you don't know.

I'm going to propose a different version of timedcalls. This version has the same signature as the three inputs, and returns the min, average, and max, but this time it treats N two different ways. What we're going to do is say if N is an integer, then do the same thing we did before, but if N is a floating point number, then what we want to do is keep on repeatedly call timedcalls for the number of trials it takes until we've added up to that number of seconds.

If N is equal to the integer 10, we repeat 10 timedcalls. If N is equal to the floating point number 10.0, then we repeat it until 10.0 seconds have elapsed.

Here is a hint. We can ask if N is an integer and then do one thing. If N is not an integer, then to the other. See if you can fill in that code.

1.19.1 19.1. Timed-Calls Answer

Unit2-19 Answer

Here is my answer.

If it's integer, we do what we did before. Otherwise, we go into a loop until the sum of the times is greater than or equal to the floating point number that passed in. Now I think we've got a pretty good handle on timing our functions.

Another interesting question would be how many assignments did we take along the way, not just how long did they take?

Now, if our program had been structured using for statements rather than using the big generator expression, we could do it something like this. We could say we're going to count the number of assignments. We're going to start the count at 0, and then every time we go through an ordering we're going to increment the count by 1.

You might say we want to increment the count by 5, because we've assigned each of the 5 houses one of the colors, but I'm counting that as 1. You can choose which one you want to go on. Then we just have a structure like that, and we can keep track of the counts. That will work fine, and it's not too bad, but it bothers me a little bit that we had to do such violence to this program. We had this program, and we had to go in in so many places, interrupt it, and put in so many statements. I wanted to see can we separate that out a little bit?

1.20 20. Cleaning-Up-Functions

Unit2-20

Now, when we're designing a program, we're thinking at multiple levels. We're thinking about different aspects of that program.

For example, we always have to think about is the program correct. Does it implement what we want it to implement?

We saw in the zebra puzzle that we also want to worry about is it efficient. Maybe we have a correct program, but it takes an hour, and we'd rather have a program that takes thousandths of a second.

We also have to worry about the debugging process of as we're developing the program, we're building up all sorts of scaffolding to make it run properly that maybe we won't need later.

Traditionally, we write our program. Some of what we're written has to deal with having it be correct. Then some of the code interspersed in there deals with efficiency. Now we've talked about adding further code that deals with debugging it. We end up with this mess that's all interleaved, and wouldn't it be nice if we could break those out so that some of the debugging statements were separate, some of the efficiency statements could live someplace else, and the main sort of correctness program could, say, stay distinct from the other parts.

This idea is called "aspect-oriented programming." It's an ideal that we can strive for. We don't get it completely, but we should always think of can we separate out these different aspects or concerns as much as possible.

What I want to ask is if I've written the function this way as this big generator expression rather than as nested statements, I can't have count equals count plus 1 statements inside here, because there's no place to put something inside a statement, and I'd like to separate the counting as much as I can from the puzzle to do the minimum amount of damage or editing to allow me to insert these bookkeeping nodes that do that counting. Let's think about that. What would be the right way to do that?

I'll tell you what I came up with. This is the second-best answer I came up with. That's to say, well, what am I doing? I'm counting iterations through the orderings, so maybe the place to insert my annotations is right here. I want this to be a debugging tool.

For debugging tools, I'm willing to have shorter names, because they're things that really aren't going to stick around. I'm going to use them for a while, and then I'm going to get rid of them. I'm willing to have a one-character name here, although it bothers me a little bit. I can always rename it as something else.

What I'll do is I'll just insert this call to the function c around each of my uses of orderings. I'm going to insert 1, 2, 3, times 5, 15 characters. There we are. I think that's a lot less intrusive than putting in a lot of count equals count plus 1 statements. I'm pretty happy with that.

What I've done here is defined a function called "instrument function," which will count the calls that it takes to execute the calling of a function with the arguments.

I haven't shown you the definition of c yet, but when I show it to you, you'll see that it keeps track of two counts--the number of starts, the times we started an iteration, started the for loop, that was measured with the c function, and the number of items that we got through.

How many times did we go through that for loop? We initialize those counts to 0, we call the function, and then we say what do we get back. With the zebra puzzle, it only took us 25 iterations over 2,700 items.

Puzzle2--this was the definition for when we took the original definition and then moved only one of the constraints up in the ordering. That gave us this number of iterations and items.

I didn't show the puzzle where none of the constraints are moved up. That would've taken even more. We see even here quite a reduction in the number of iterations in the counts, and this tells you how efficient we are.

1.21 21. Yielding Results

Unit2-21

To implement the function C, I have to tell you about a new tool called a generator function, which is almost the same as a generator expression, and we'll show you what it looks like. I'm going to define a function to iterate over the integers from some start to some end. This is going to be just like range. Range is a great function, but sometimes it's annoying, because you really don't want to 1 less than the end. You want to go up all the way to the end.

That's what ints is going to do. I'm going to implement it as a generator function. I'm going to start off with an integer I that starts at start number that you told me. Then while i is less than or equal to the end, I'm going to introduce a new type of statement here called yield, and I say yield i, and then I'll say i equals i plus 1. Now, the yield is something new, and it makes this definition a generator function, rather than a regular function. What that means is what's going to happen is it's going to execute. As soon as it sees this yield statement, it's going to generate that value i, but keep it's place and when asked for the next, it will continue incrementing i and then continuing through the loop.

```
·· 1 def ints(start, end):¶
·· 2···· i = start¶
·· 3···· while i <= end:¶
·· 4········ yield i¶
·· 5········ i = i + 1¶
¶
```

When we call this function, if we say L equals the integers from 0 to 1 million, now L won't be a list right away. It'll be a generator. Now L is equal to this generator, some funny object. Then you call next of L to get the next object, and you keep on going through. More commonly, instead of calling next explicitly, you'd say for i in L.

And so generators obey this iteration protocol just like other types of sequences. The great thing is we can yield from anywhere inside this function. We get a very complicated logic here and then yield a partial result, and then continue getting the next result right where we are. This is a convenient feature for writing functions.

Another great thing about generator functions is it allows us to deal with infinite sequences. Let's say we wanted to allow the possibility of an open ended interval of integers. We'll make end be an optional argument, which can be None. The idea is that if we have a call, say ints, starting at zero, but I don't give it a end, then that means all the non-negative integers--0, 1, 2, 3, going on forever and never stopping.

The question for you is how would you modify generator function in order to make it obey this idea of never stopping when end is equal to None. You should be able to modify just this line here to make it obey this idea that it should keep on going forever when end is None.

1.21.1 21.1. Yielding Results Answer

Unit2-21 The answer is we want to continue with this while loop while i is less than end or end is None. In the case where end is None, that will give you an infinite loop.

```
·· 1 def ints(start, end=None):¶
·· 2···· i = start¶
·· 3···· while i <= end or end is None:¶
·· 4········ yield i¶
·· 5········ i = i + 1¶
¶
```

1.22 22. All ints

Unit2-22

Let's give you practice with one more example. I want you to define for me the function all_ints, which generates the infinite stream of integers in the order of 0 first, then +1, -1, +2, -2, +3, -3, and so on. Put in your definition.

1.22.1 22.1. All ints Answer

Unit2-22 Answer

Here is my answer. It would start by yielding 0. Then for i in all the positive integers yield first +i and then -i.

```
·· 1 def all_ints():¶
·· 2···· "Generate integers in the order 0, +1, -1, +2, -2, +3, -3, ..."¶
·· 3···· yield 0¶
·· 4···· for i in ints(1):¶
·· 5········ yield +i¶
·· 6········ yield -i¶
¶
```

If you didn't want to use ints here, you could do an alternative solution that looks like this. While True means it's an infinite loop, and I'm yielding +i and -i and then incrementing.

```
·· 1 def all_ints():¶
·· 2···· "Generate integers in the order 0, +1, -1, +2, -2, +3, -3, ..."¶
·· 3···· yield 0¶
·· 4···· i = 1¶
·· 5···· while True:¶
·· 6········ yield +i¶
·· 7········ yield -i¶
·· 8········ i = i + 1¶
¶
```

1.23 23. Nitty Gritty For Loops

Unit2-23

Now, after all that, I think you're finally ready --you're mature enough-- to learn the whole truth about how for statements actually work. You've been using them all along, but you may not have known the inner details, the gory truth, about what's inside the for statement.

Now, when I say:

```
for x in items:
    print x

```

Assuming items is a list or a tuple or a string, you think of this code probably as something like:

```
i = 0
while i > len(items):
    x = items[i]
    print x

```

That's a good model as long as items is one of these sequence types like lists, but items can also be other things, as we've seen. It can be a generator expression, for example.

Overall, Python calls the thing that can be iterated over in a for loop an iterable. Strings and lists are examples of iterables, and so are generators. What actually happens when we implement this piece of code, it's as if we had written this code.

```
it = iter(items)
try:
    while True:
        x = next(it)
        print x
except StopIteration:
    pass

```

In the above code, first we take the items and we make an iterator from them by calling the built-in function "iter." I'm going to call that "it." Then we're going to have a while loop that goes forever. "It"s loop control says while True we're going to assign x to be the next item off of the iterator, then do whatever was in the body of the for loop, in this case print x.

We're going to keep on doing this sequence as long as we can get an x value. But when next stops then we'll have an exception, which is a stop iteration exception, and we don't need to do anything more. We're done.

That's what a for loop really means in Python.

We take the items, we create an iterator over them, we call that iterator until we're done, and then we pass through to the next statement. We're finally ready to define this c, this counting function.

```
def c(sequence):
    """ Generate items in a sequence; keeping counts as we go.
c.starts is the
    number of sequences started; c.items is the number of items
generated. """
    c.starts +=1
    for item in sequence:
        c.items += 1
        yield item

```

What it does is it takes a sequence, it says this is the first time I've been called. I'm going to initialize my starts to one. Then I'm going to enter into a loop and this means that c is a generator function. The generator function will be returned, and as part of the for protocol, we'll call that generator function each time we want the next item in the sequence and each time we do that, our count of items will be incremented.

When we're done, when the for loop doesn't want any more, we'll stop incrementing.

We don't necessarily go through every item in the sequence. We'll just have just the right number of counts for item. This will give us the right number of starts and the right number of items. We can do that because we control the time of execution, because this is a generator function and not a regular function.

1.24 24. Zebra Summary

Unit2-24

In summary, what have we learned from our friend the zebra?

- Concept inventory
- Refine ideas
- Simple implementation
- Back of the envelope
- Refine code

Well, we learned that we took our approach of making a concept inventory and then refining the ideas and choosing the simplest implementation we could think of and then doing a back of an envelop calculation to say how long is it going to take to run this simple implementation and then refining the code as necessary, making it a little bit faster by swapping around some of the clauses.

We also learned the idea of building tools. We built tools for timing and counts.

In general we learned this idea of separation of aspects from the program to try to keep the design as clean as possible, so that you could tell what was working on getting the problem right and what was working on making it more efficient.

In the end you might say, wow, that was a clever solution. It was great to see how it works out, but we can see if we just follow the steps we can arrive at a solution like that every time.

1.25 25. Cryptarithmetic

See Unit2-25

Now we're going to turn our attention to a different type of puzzle.

This is called a cryptarithmetic problem.

"Crypt-" for cryptography--secret writing---and arithmetic for arithmetic-- doing addition and other types of problems, and the idea here is that each of these letters of the alphabet stands for some digit from 0 to 9.

The problem is to figure out which digit stands for which such that the equation will be correct.

Some people call these alphametics--is another name for them. They're a puzzle for humans to work out because there are so many possibilities, and humans are limited in the speed in which they can mark them out. But there are some types of inferences that they can make.

For example, in this addition problem here we have an addition problem of of two 3-digit numbers, and the result is a 4-digit number. E is that fourth digit, so what does that tell us about what the letter E must stand for?

What digit it must stand for?

Do you think E should be?

- 1
- 2
- 3

- 4
- 9

1.25.1 25.1. Cryptarithmetic Answer

Unit2-25

The answer is that E has to be 1 because E is the carry digit.

After adding up these two 3-digit numbers we get a little bit more, and the most that the carry digit could be is a 1.

If O stood for 9, even 9 plus 9 is 18, and we might carry over a little bit. But still that's 19. It's not 28. So the E must be a 1.

We could go on from there and figure other things.

1.26 26. Odd-or-Even

Unit2-26

Is this digit the digit that replaces the N? Do you think that should be odd or even?

1.26.1 26.1. Odd-or-Even Answer

Unit2-26 Answer

The answer is N must be an even digit because D plus D, no matter what D is, that's equal to 2 times D, so N must be even.

Human problem solvers would continue on like that, making inferences from what they know about the rules of arithmetic to figure out what each letter should be.

In the end we come up with one of two possible solutions.

Either '655 + 655 == 1310', or '855 + 855 == 1710'.

1.27 27. Brute-Force-Solution

Unit2027

Here is one possible design for coding up a solver for these types of problems.

That design would be to write down all the rules of arithmetic in terms of carry digits, in terms of odd and even and so on.

Now, that seems like a challenging task.

There's a lot of complexity involved in understanding all the rules about arithmetic.

Even if we figured out everything about addition, there's also subtraction and multiplication and other operators.

So what we really want is a short cut that'll allow us to eliminate this complexity.

Let's go back to the back of the envelope and see if we can figure out a shortcut.

One possibility would be to try all possibilities.

There are 10 digits.

All combinations of digits--all permutations of the digits, rather-- would be only 10 factorial, which I happen to know is about 3 million. That's not so many. It seems like it's feasible to try all the possibilities. It's not going to be super quick. We would rather have this be thousands rather than millions because there seems to be a fair amount of work in substituting in all the letters with digits. But we can expect to be able to try all millions, not within a second, but within about a minute or so.

Now we have an approved design, which is we represent our formula as a string, and we'll use official Python notation here with the double equal sign.

Then we fill in with all permutations of the 10 digits for each of the letter, and if there's fewer letters, we have to account for that.

For example, we might substitute a 1 for the Os, and a 2 for the Ds, and a 3 for the Es, and a 4 for the Vs, and a 5 for the Ns.

Otherwise just copy the equation. Then evaluate that and check if that's equal to True. If it is, then we have a solution. If it's not, we'll go back and we'll try another combination-- maybe 1, 3, 3, and so on. We'll keep on going through until we find some permutation that works. That's the design.

Now, let's take an inventory of all the concepts we're going to need. First we have equations. There's two types of those--the original and the filled-in. The original has letters. The filled in has digits. Letters and digits are concepts we have to deal with. The assignment of a letter to a digit or set of those is also a concept we have to deal with. I think that's pretty much it.

Maybe one more concept is evaluation or validation that the filled in equation is an accurate one.

Now let's come up with some choices to represent each of these concepts. The original equation can be a string. The filled-in equation can also be a string. The letters will be single character strings like D. The digits will also be single character strings--like 3.

The assignment of letters to digits will be some sort of mapping or a table that consists of this type of mapping that says D would be replaced by 3 and so on. It turns out that there is a facility built into Python that's part of strings called the translate function. We can call the str.translate method in order to enact that type of assignment or substitution.

Then for the evaluation, there is a function called "eval" in Python, which takes a string and evaluates it as an expression.

You may not be that familiar with these last two items, so let's go over them.

Eval is pretty simple. If we asked for eval of the string "2 + 2," then that would give us a value 4. If we asked for eval of the string "2 + 2 == 3," Python would evaluate that using it's normal rules and tell us that that's equal to False.

1.28 28. Translation-Tables

Unit2-28

Now I'm going to show you how these translation tables work.

I'm going to define a variable called "table" that's using the string.maketrans function, which makes a translation table, and I'm going to tranlate from the characters 'ABC' to '123.' I can give any number of characters here-- the characters I want to replace and the ones I want to replace them with. I should say that this is using the string module, so somewhere we have to say import string before we start doing any of this. You only have to do that import once, of course.

Now I'm going to define a formula f to be a simple formula A plus B equals C. Then I'm going to call the translate method of the formula f and pass it this translation table. That will evaluate to the string 1 plus 2 equals 3. It has taken each of the elements in the table, and they correspond A to 1, B to 2, C to 3, substituted those into f and given me back a brand new string. Now if I go ahead and evaluate f.translate of table, which is 1 plus 2 equals 3, then that will give me the result True, because 1 plus 2 is 3, and that's a legal Python expression.

Now what I want you to do is to define for me the function "valid," which takes a filled-in formula like 1 plus 2 equals 3, filled-in formula f, and returns True or False. True if the formula is, in fact, valid. If it represents a true equation like this.

And False if it represents an invalid equation like 1 plus 3 equals 3. Or it should also return False if it represents a error like 1 divided by 0 equals 3. That wouldn't return True or False, that would signal an error, and I want you to handle that within the code for valid.

I'll give you a hint, which is you should consider using a try statement. Try, do something, and then you can say "except ZeroDivisionError" something. What that does is it executes the main body in which you can test if evaluating this expression f is true or not and return appropriately, but if evaluating the expression f causes a zero

division error, then this clause will catch it, and then you can do the appropriate thing here.

You should also think about if there's anything else that can go wrong within the execution of valid.

Here's my version of the solution.

```
·· 1 import string, re¶
·· 2 ¶
·· 3 def valid(f):¶
·· 4···· "Formula f is valid iff it has no number with leading zero, and
evals true"¶
·· 5···· try:¶
·· 6········ return not re.search(r'\b0[0-9]', f) and eval(f) is True¶
·· 7···· except ArithmeticError:¶
·· 8········ return False¶
¶
```

I'm defining valid, takes filled-in formula f, and it's going to return True.

The main part is if we evaluate f and if that's true, then we should return True, but I also had to check for the zero division error and even to be a little bit more sore here, I ended up checking for arithmetic error, which is a super class of zero division error. It covers a few additional things like overflow of really big numbers. You didn't have to do that. I would've been fine to just catch the zero division error. If there is such an error, then you should return False.

But I did one more thing here, and it looks kind of complicated. I'm using a regular expression search, and let's look at exactly what's going on in this confusing part of the clause here.

1.29 29. Regular-Expressions

Unit2-29

Let's say we have a formula like ODD plus ODD equals EVEN.

Now, one of the rules of the way we form numbers is that we don't want to have a number that starts with 0. EVEN could be, say, 3435. That would be a perfectly valid set of digits to fill in for the word EVEN. It shouldn't be 0405, because we just don't normally write numbers with a leading zero. I excluded that. How did I do it? I did a regular expression search for the regular expression slash b, which matches at a word boundary--b stands for boundary. Then I'm looking for a zero followed by any digit 0-9. I'm looking for that within f, and I want that to be not true.

If I took this string here--something plus something equals 0405, and I did this regular expression search, it's saying find me a word boundary, a boundary between a letter and/or a digit and something that's a punctuation or something else, a word boundary followed by a 0, followed by another digit, it would say, yes, indeed, that search does succeed. It succeeds right here. The 0 and the 4 match. I want that not to succeed, so I would return False.

I would rule out this case where we have a word starting with 0 and 4. I did that, one, just because it's good form that normally you don't allow valid numbers to start with a zero, and also because in Python we could come up with an error because of that. Here is the problem. In a Python program, the string 012 corresponds to an integer, but the integer it corresponds to is not 12. It's 10. Why is that?

Well, it's an old historical accident. Way back in the 1970s, the C programming language defined it that way, where they said any number that starts with a 0 is going to be interpreted as an octal number, which means base 8. So 012 means one 8, zero 8-squared, and 2 more, so 8 + 2 is 10. So that would give us the wrong answer if we allowed octal numbers to sneak in where we were expecting decimal. It would also give us a possible error. If we had the string 09 that gives you an error, because 9 is not an octal digit.

To avoid all those problems, I put in this regular expression that says any time you see a lead zero just rule it out.

1.30 30. Solving Cryptarithmetic

Unit2-30

Now what I want you to do is write for me the code for the function solve. Solve takes an unfilled formula with letters in it, and if it can find a solution in which you can fill in all the digits, it returns the string with the digits

filled in properly. If not, it should return None.

Put your code here.

You can assume that we already have the correct definition of what a valid filled-in formula is. You can assume that we have a function called "fill_in" that I'm not showing yet.

Fill_in takes a formula and generates a collection of all the possible filling ins of the letters. You pass it the string ODD plus ODD equals EVEN, and it passes you back first string filled in with, say, a 1 for the O and a 2 for the Ds and so on. Then the next string filled in with different numbers filling in each of the possible digits, and so on and so on.

1.30.1 30.1. Solving Cryptarithmetic Answer

Unit2-30 Answer

Here's a code that's pretty simple.

```
·· 1 def solve(formula):¶
·· 2···· """Given a formula like 'ODD + ODD == EVEN', fill in digits to
solve it.¶
·· 3···· Input formula is a string; output is a digit-filled-in string or
None."""¶
·· 4···· # Your code here¶
·· 5···· for f in fill_in(formula):¶
·· 6········ if valid(f):¶
·· 7············ return f¶
¶
```

All we do is we iterate the variable f over all the possible values of filling in the formula. If we say that filled-in formula f is valid, then we should go ahead and return it. Otherwise, when we get through the end of the loop, we automatically return None.

1.31 31. Fill In Function

Unit2-31

Now I want to talk about a strategy for defining the function fill_in, which takes an unfilled formula and returns all the possible filled in formulas.

What do we have to do?

Well, let's consider a formula, and I'm going to take one that's simpler than one we've seen before. I'm going to take the formula:

```
·· 1 'I + I == ME'¶
¶
```

What we have to do then is find out what all the letters are in this formula and fill in all possible values for digits for those letters.

It seems like a good thing is to collect all the letters, and in this particular formula the letters should be 'IME'.

What are the possible digits that we want to replace these three letters with?

Well, we can just iterate through all the 3-digit numbers, but make sure that we're not duplicating any of those numbers. So maybe we'll start with 123, then 124, and so on, 120, and then 13.

We wouldn't do 131, because we already have a 1 there, so 132. We just have all these possibilities.

How many possibilities are there?

Well, there's 10 possible first digits. Then we don't want to repeat a digit, so 9 possible second digits, and 8 possible third digits, so that's 720 possibilities. Not very many at all. It should be really fast to go through all these possibilities.

Now, what function gives me all the possible sets of numbers in order (so order matters)?

What function is it that we give it the list of digits?

So we want some function F, we give it the list of digits, and we give it the number that we want, and there are three letters in this particular formula, so we want three of them. What function F takes that and then returns this sequence of all possible sets of numbers?

Maybe it returns it as a tuple or a list or whatever.

The question is what function can do that?

Is it combinations, permutations, factorial, or some other function?

1.31.1 31.1. Fill in Function Answer

Unit2-31 Answer

The answer is that it's permutations.

That's what the definition of permutations is. We take some collection, and then we pick out some number of them. The order matters, and we have all possibilities.

So there is a built-in permutations function. It happens to be in a library called "itertools."

1.32 32. Filling In Fill In

Unit2-32

Now with that in mind, I've given you the template of most of the the fill_in function, and I want you to just put in two missing pieces.

What does the fill_in function do?

First, it finds all the letters in the formula, and I want you to fill in that piece. Then it iterates a collection called the digits, a permutation of the digits taken from all the digits n at a time, and you have to fill in the right value of n. Then it builds a table from taking the letters.

That means you want to make sure that the letters up here are represented as a string and not as some other type of collection. It makes a translation table from the letters and a string of all the digits that we got out of the permutations.

The function itertools.permutations returns a tuple of results, and so we want to join together that tuple into one big string, make up the translation table, and then call formula with the translate method with that table to translate all the letters into the appropriate digits and yield that result.

In other words, fill_in is a generator function not a function that returns a list of results.

Why did I do it that way? Well, because you might get lucky. It might be that the very first formula you try or one of the few first formulas you try is the correct one.

If I do for f in fill_in a formula, I ask for the first formula, and if it is valid, then I want to return it right away. I don't want to waste time calculating all the other possible fill_ins. That's why a generator makes more sense here.

See if you can fill in these two missing pieces.

1.32.1 32.1. Filling In Fill In Answer

Unit2-32 Answer

```
·· 1 from __future__ import division¶
·· 2 import string, re, itertools¶
```

```
·· 3 ¶
·· 4 def solve(formula):¶
·· 5···· """Given a formula like 'ODD + ODD == EVEN', fill in digits to
solve it.¶
·· 6···· Input formula is a string; output is a digit-filled-in string or
None."""¶
·· 7···· for f in fill_in(formula):¶
·· 8········ if valid(f):¶
·· 9············ return f¶
· 10 ¶
· 11 def fill_in(formula):¶
· 12···· "Generate all possible fillings-in of letters in formula with
digits."¶
· 13···· letters = ''.join(set(re.findall('[A-Z]', formula))) #should be a
string¶
· 14···· for digits in itertools.permutations('1234567890', len(letters)):¶
· 15········ table = string.maketrans(letters, ''.join(digits))¶
· 16········ yield formula.translate(table)¶
· 17 ¶
· 18 def valid(f):¶
· 19···· """Formula f is valid if and only if it has no¶
· 20···· numbers with leading zero, and evals true."""¶
· 21···· try:¶
· 22········ return not re.search(r'\b0[0-9]', f) and eval(f) is True¶
· 23···· except ArithmeticError:¶
· 24········ return False¶
¶
```

The first part here was easy. We want digits to be the same length as the number of letters, because we want to substitute them one for one--123 for ABC or whatever. We want to ask itertools.permutations to take permutations of the letters taken n at a time where n is the number of letters, the length of the letters. That part's easy.

This part's a little bit more complicated--how do we find the letters. Here's what I did. I used the regular expression, the "re" module. I used the findall function so that says find all letters from A to Z, and I didn't specify here capitals before.

My interpretation and the general rules for this type of cryptarithmetic problems is that they should be capitals, but if you allow lowercase as well, that's fine. Find them all within the formula, make a set of those, because if a letter occurs more than once we don't want multiple versions of it. We just want individual ones. Then join them together into a string.

Now, I should say that I snuck in one piece of code here that you haven't seen before that may seem wild at first. That's the very first line. It says from _ _future_ _ import division. That seems pretty amazing. Wow. Python has a built-in time machine.

What does it mean to import division from the future?

1.33 33. Future-Imports

Unit2-33

In Python 2.x, version 2.6, 2.7, and so on, if you do integer division in Python 2.7, say, 3 divided by 2 evaluates to 1. And 1 divided by 2 evaluates to 0. The reason is in their wisdom the designers of Python said, well, if you're going integer division, you probably want an integer answer, and we'll do the best we can, and we'll have to truncate to give you the answer.

In Python 3, the designers decided, well, that's really confusing. What you really want when you divide 3 over 2 is 1.5, and when you divide 1 over 2 you want 0.5. That's what Python 3 does, so it says let's change the result from an integer to a floating point number if that's the best you can do.

Now, if you want this kind of behavior in Python 2, then you say from the "_ _future_ _" with two underscores on either side of it import division. I want that because in my cryptarithmetic problems I really prefer this type of answer and not that type of answer.

Now I'm done, and I got a pretty concise program. It followed by plan very nicely and easily. I like this design of the three pieces that I've tried to divide it up into.

Look how simple the solve function is. It just says iterate over all possible ways of filling it in, ask if it's valid, and if it is, return. Can't get much more clear than that.

Valid is pretty simple. Valid is almost like asking is eval f True. If it were that simple, I would just put it in line up here, but it's not quite that simple. I need the try and except, and I need to catch the arithmetic error, and then there's this little trick with the leading zero digits.

I like having a separate function for valid to break that out. Then I like having fill_in as a generator. To make the logic of the main function simple but not to slow me down by requiring me to generate all the possible answers at once, the generator comes up with up the answers one at a time. It's separating out the flow of control from the logic.

Now, am I done at this point? Well, no, because what I have to do next is convince myself that I've got it right.

1.34 34. Testing

Unit2-34 Here I've defined some test examples, and here I've written a simple test function.

```
·· 1 from __future__ import division ¶
·· 2 import time¶
·· 3 ¶
·· 4 examples = """TWO + TWO == FOUR¶
·· 5 A**2 + B**2 == C**2¶
·· 6 A**2 + BE**2 == BY**2¶
·· 7 X / X == X¶
·· 8 A**N + B**N == C**N and N > 1¶
·· 9 ATOM**0.5 == A + TO + M¶
· 10 GLITTERS is not GOLD¶
· 11 ONE < TWO and FOUR < FIVE¶
· 12 ONE < TWO < THREE¶
· 13 RAMN == R**3 + RM**3 == N**3 + RX**3¶
· 14 sum(range(AA)) == BB¶
· 15 sum(range(POP)) == BOBO¶
· 16 ODD + ODD == EVEN¶
· 17 PLUTO not in set([PLANETS])""".splitlines()¶
· 18 ¶
· 19 def test():¶
· 20···· t0 = time.time()¶
· 21···· for example in examples:¶
· 22········ print; print 13*' ', example¶
· 23········ print '%6.4f sec:·· %s ' % timedcall(solve, example)¶
· 24···· print '%6.4f tot.' % (time.time()-t0)¶
· 25 ¶
· 26 def timedcall(fn, *args):¶
· 27···· "Call function with args; return the time in seconds and result."¶
· 28···· t0 = time.time()¶
· 29···· result = fn(*args)¶
· 30···· t1 = time.time()¶
· 31···· return t1-t0, result¶
¶
```

What does it do? It iterates through each of the possible examples. It prints out what the example is, the original formula. Then it uses the time call function that we defined in the previous lesson. It calls solve on the example and gets back the time it took and the possible results and prints that out. Then in the end it prints out the total elapsed time and does that by starting a timer at the start and taking a time at the end and just calculating the difference.

It's going to tell me for each example what the answer is, how long it took for that example, and how long it took for all of them together.

Here is the results of the test:

```
·············· TWO + TWO == FOUR¶
0.0576 sec:·· 734 + 734 == 1468 ¶
¶
·············· A**2 + B**2 == C**2¶
0.0030 sec:·· 3**2 + 4**2 == 5**2 ¶
¶
·············· A**2 + BE**2 == BY**2¶
0.0326 sec:·· 5**2 + 12**2 == 13**2 ¶
¶
·············· X / X == X¶
0.0000 sec:·· 1 / 1 == 1 ¶
¶
·············· A**N + B**N == C**N and N > 1¶
0.0247 sec:·· 3**2 + 4**2 == 5**2 and 2 > 1 ¶
¶
·············· ATOM**0.5 == A + TO + M¶
0.0037 sec:·· 1296**0.5 == 1 + 29 + 6 ¶
¶
·············· GLITTERS is not GOLD¶
0.0000 sec:·· 35499187 is not 3652 ¶
¶
·············· ONE < TWO and FOUR < FIVE¶
0.0580 sec:·· 351 < 893 and 2376 < 2401 ¶
¶
·············· ONE < TWO < THREE¶
0.0000 sec:·· 341 < 673 < 62511 ¶
¶
·············· RAMN == R**3 + RM**3 == N**3 + RX**3¶
0.4187 sec:·· 1729 == 1**3 + 12**3 == 9**3 + 10**3 ¶
¶
·············· sum(range(AA)) == BB¶
0.0001 sec:·· sum(range(11)) == 55 ¶
¶
·············· sum(range(POP)) == BOBO¶
0.0007 sec:·· sum(range(101)) == 5050 ¶
¶
·············· ODD + ODD == EVEN¶
0.2935 sec:·· 655 + 655 == 1310 ¶
¶
·············· PLUTO not in set([PLANETS])¶
0.0000 sec:·· 63894 not in set([6315297]) ¶
0.8982 tot.¶
```

For each example, we see the example printed, then the filled in example found, and you can verify that that is, in fact, correct.

It took a tenth of a second or so. Some of them take less. When there's only one variable it's really fast.

Here we find these Pythagorean triplets of 3-squared plus 4-squared equals 5-squared, and 5-squared plus 12-squared equals 13-squared. Notice that we've gone beyond the most simple of cryptarithmetic problems, which were just word plus word equals another word. Here we can allow division and exponentiation. Over here I've been even more complicated where I've come up with A**N plus B**N equals C**N, and N is greater than 1.

The lowercase letters I have not translated, because I said the only letters that count as being translated by digits are going to be the upper case ones, so I can use any of this Python syntax like the and operator.

Now, if you can find a solution for this when N is greater than 2, I want you to let me know.

But here I found one when N is greater than 1.

I've split the atom. I've determined that all that glitters is not gold.

I've determined that 1 is less than 2 and 4 is less than 5.

You can see a bunch of other equations that I've been able to process, including ODD plus ODD equals EVEN.

And for these 14 different examples it took 2 seconds. Not too bad.

1.35 35. Find-All-Values

Unit2-35

Now I want to ask you one quiz question. What if we wanted to return or to print all values of the filled-in formula rather than just the first one? For some of these formulas there are multiple possibilities of digits to fill in that would all work equally well.

If we wanted to see them all, if we wanted to have them all printed out, what should we do?

- Should we change the function fill-in to return a list rather than generate results?
- Should we change the single word "return" to "print" in the function solve?
- Should we change the function valid to do something else to validate each of these possible formulas?
- Or do we need to add a new function to do something that we haven't done before?

1.35.1 35.1. Find-All-Values Answer

Unit2-35 Answer

The answer is all we need is this one word change. So far we're returning the first value we found, but if we print it, then we can see that and the loop will keep on going, and we can see all the possible Fs--the filled-in formulas that are valid solutions.

1.36 36. Tracking-Time

Unit2-36

Now if you have Python installed on your own machine, and you have some kind of a terminal or shell in which you can get a command prompt, you can type the command Python, and then give it the -m flag, which says module, and then load what's called the cProfile module-- lowercase c, uppercase P. Then the name of your file that you want to profile. I called my cryptarithmetic file crypt.py. Execute that, and you'll get a nice table of where all the time goes.

If you don't have Python installed or you can't run a command like this, you can do it from within Python. What you'll have to do is say import the c profile module, and then do cProfile.run and then a string to be evaluated, which is the code that you want to run. Then you'll see output that looks like this.

- In the right had column we see the various functions that are being called.
- Within my crypt program there are three main functions--solve, fill in, and valid.
- Then within the regular expression module, I was calling search and compile, and then there's various other built-in methods of Python, such as the string maketrans function, the eval function, and so on.
- These other columns tell us the number of times each of these functions was called, the total time spent, some percentage--we won't worry about that.
- That's time per call.
- Then the cumulative time, the total number of times spent there.
- Mostly I was just called solve. That was about 75 seconds.
- Within that 62.7 seconds when to valid, so that's where most of the time is going.

I should say that most of these results are pretty much what I was expecting. I was a little bit surprised that the re search too so much time--12 seconds out of the 75. I was also a little bit surprised that this maketrans and these other methods, the translate methods, took so little time--just about 3 seconds all together.

If we want to make our program faster, it seems obvious that we'd better look where the time is. Out of that 75 seconds, 63 of it is within valid. That's where we have to look. Of valid, 47 seconds is within eval. If we want to make our program faster, it makes sense to concentrate our efforts on the parts where most of the time is. It's not just a good idea. It's the law.

It's called the Law of Diminishing Returns. The way the law works, if we imagine our total execution time as being this bar here, and if we said that goes up to 75 seconds, and 10 seconds was taken up by fill in, and then all the rest by everything else.

We can see if we want to make things faster, we'd better make this bar shorter or maybe this bar; and it won't help much to make these other bars shorter.

For example, if we made fill in and everything else vanishingly small, then if we didn't touch valid we'd still have an execution time of 63 seconds, even if we could improve these infinitely fast and we wouldn't have helped all that much.

1.37 37. Increasing-Speed

Unit2-37

Here is a quick quiz on diminishing returns.

Say if we have an execution time that's this long and function A takes up 90 seconds, and function B takes up 10 seconds for a total of 100 seconds execution time. Let's saw we're able to make an improvement to our code. We speed up B by a factor of 10. We make B ten times faster.

The question is how much faster do we make the overall execution of our whole program if B is made 10 times faster?

Do we make the whole program 9 times faster,

These are approximate number, not exact numbers.

1.37.1 37.1. Increasing-Speed Answer

Unit2-37 Answer

The answer is even though we sped up B by a factor of 10, we've only made the whole program 1.1 times faster--that is, 10% faster. The problem is we've left A unchanged, so our new program would have an execution time that looks like this. B would be this tiny little slice here. It would just take 1 second rather than 10 seconds, but the whole program would still take 91 seconds.

1.38 38. Rethinking-Eval

Unit2-38

If we want to make our program faster effectively, we'd better concentrate on the eval function, because that's taking up about 47 out of our 75 seconds or about 63% of our time.

The problem is that eval is a built-in function. We can't go editing eval to try to make it faster, but if we can't touch eval itself, the only choices we have are we could make fewer calls to eval. We call it fewer times. We'll spend less time in it.

Or we can make easier calls to eval. Pass it an argument that's easier for it to evaluate. Let's concentrate on easier first. Do you see a way to break up the problem of evaluating a formula into smaller pieces in such a way that we could make the resulting program, say, What do I mean by making it easier or breaking it up into pieces? We could do eval of ODD plus ODD and then do eval of EVEN and do those separately rather than do them all together in one equation. Of course, I'd probably substitute in the numbers here rather than eval the letters themselves. That might be one way of breaking it up into smaller pieces. Often this idea of divide and conquer is a good idea for program design.

Do you think that that approach would work here to make the calls to eval easier so that we could cut down on this 47 second execution time.

Yes or no?

I guess this is an matter of opinion. You might have some ideas. I might have different ideas, so don't worry if you agree with me too much, but think about it and give me your answer.

1.38.1 38.1. Rethinking-Eval Answer

Unit2-38 Answer

Well, my answer was no. I couldn't think of a good way to break these up that would make the calls easier and still get the job done. If you've come up with a good way, great. I want to hear about it. Let's talk about it in the forums.

1.39 39. Making-Fewer-Calls

Unit2-39

But if we can't figure out a way to make the calls easier, then we're going to have to concentrate on having fewer calls.

Let me write down some possibilities for fewer calls and see which ones you think make sense.

- One possibility would be for each equation, like ODD plus ODD equals EVEN, rather than evaluating them all N factorial times, maybe we could combine all of those into one big equation. Certainly that would result in fewer calls if we could figure out a way to do that and still get the right answer.
- Another way might be to, say, could we figure out a way to fill in one digit at a time? Rather than do all N factorial permutations of digits, if we could do one digit, see if that works, and if it does then do the next digit. That would certainly help us do a smaller number of calls. That's the approach we took with the zebra function where we started out by doing all the permutations, and then we figured out let's go through and if there's a contradiction, let's stop and not do the remaining ones. The question is can we figure out a way to take that general approach for this problem.
- Then the third approach would be to eval the formula once but eval it as a function with parameters. To do all the work of figuring out how to understand number plus number equals number, do that just once and then call that function repeatedly with all the permutations. There'd still be lots of calls to the function, but there'd be fewer calls to eval.

Tell me which one of these three do you think is the most promising. Again, it's a matter of opinion. You might have better ideas, but I want you to think about it and tell me what you think.

1.39.1 39.1. Making-Fewer-Calls Answer

Unit2-39 Answer

My feeling was that this was the most promising.

I couldn't figure out how to fill in one digit at a time and make that work, because if we had something like the word EVEN, and we fill in 1 digit, and we have, say, 8V8N, I didn't figure out how to make sense of that. We have this mixture of some letters that are filled in and some that aren't. That didn't work for me.

This approach, one big formula--yes, we could figure out a way to make one big formula, but the problem is I don't think that would be any faster, because it would take eval a long time because it would be a very long formula.

This approach I think is the most promising, so let's talk about it.

1.40 40. Lambda

Unit2-40

Let's consider a formula--I'm going to write a new one. Let's say You equals Me-squared. We're treating these formulas as strings. Now when we substitute numbers into this, we get something like 123 equals 45-squared.

What happens when we call eval on this string? What eval has to do is it takes it's input, which is a string, and then it has a process of parsing that string into a parse tree. The parse tree would say something like this is a comparison, and it has an operator, which is a number, which is 123, and another operand, which is an expression which has the number 45, and then the exponentiation operator, and then the number 2. Python builds up a data structure that looks something like that.

There's another operation of code generation in which Python takes this tree and says in order to evaluate this tree I'm going to do something like load the number 123 and then load the number 45 and then do an operation on that and so on and so on and then return the result.

That's a lot of work to build up this tree, generate the code, and then finally, the final operation after we've come up with this, is to execute this code and come up with an answer, which in this case would be false.

Now, this is a lot of work for eval to do, and it seems like there's a lot of duplicate work, because we're going to do this for every permutation of the digits, but each time we go through this part of the work, the parsing, is going to look exactly the same with the exception of the specific numbers down here at the bottom, but the rest of the parse tree is going to look the same. Similarly, this part of the work, the code generation, will also look the same

except these numbers will differ. We're going to have to repeat that over and over again.

What we'd like is an approach where we can only do these two parts once and then pass in the specific numbers and then get our final result back.

But one thing I should say is that the eval function doesn't take a statement, like this function definition. It takes an expression. Furthermore, we don't really need this name F. That was kind of arbitrary. We'd be fine with having an expression that represents a function but doesn't have a name associated with it. Then we can pass that directly to eval.

It turns out there is a way to do that in Python. There is a keyword that creates a function as an expression. That keyword is lambda. It's kind of a funny keyword. I would've preferred them to use function or fun or maybe just def and leave the name out, but they chose lambda. It's based on the lambda calculus, the Greek letter , which is a branch of formal mathematics defining functions.

The syntax is fairly similar to the syntax of a definition of a function, except we leave the name out. It's an anonymous function with no name. It also turns out for some reason we leave out the parentheses. We just say lambda Y, O, U, M, E, then a colon, and we leave out the return, and then we just put in the rest of the code--100 times blah, blah, blah, blah.

1.41 41. Compile-Word

Unit2-41

Here's an example of how it all works. This I've actually typed into the Python interpreter. I've defined a function f as a lambda expression. It looks like this.

Then I've asked what F is. All Python prints out is it says that it's an object of type function, which doesn't have a name other than the lambda and the address and memory where it's stored.

Then I'd say, is F true of the sequence 1, 2, 3, 4, 5. That is, Y is equal to 1, M is 2, E is 3, U is 4, and O is 5. The answer is no, it's False. This is not equal.

Then I asked is it true for 2, 1, 7, 9, 8. Yes, that's true. The reason is because it works out to this expression 289 equals 17-squared.

Now I'm thinking of a design where we have some type of a solve function that's going to solve a formula, and we're going to have a compile formula function that's going to take a string formula as input and return a lambda expression function as the result of compiling the formula.

As part of that, I want to have a function that I'm going to call compile_word, It's going to take a word like ME and compile that into something like 10M plus E. You could have some variation on exactly how you want to express that.

It will also take a word like equals and compile that into itself, into equals, and a word like 2 and compile that into 2 itself.

This is the function I want you to write.

Compile_word where compile_word of YOU is something like this-- don't worry about the communicativity and associativity. You can write this any way you want as long as it is a code that would compute the right answer. It's important to put parentheses around it. Anything that's not an uppercase word you should leave alone.

1.41.1 41.1. Compile-Word Answer

Unit2-41 Answer

Here is what I did.

I used the isupper method of the word, which is a string. String.isupper method to check if it is in fact an uppercase word. If it is, then I enumerate all the letters in the word.

I reverse it, so this slice says reverse it.

The missing numbers on either side says take the whole word, and the -1 says go from the back forward, so reverse the word, enumerate it give me pairs of indexes from 0 to N along with the individual digits.

Then I'm going to just say 10 to the ith power and the digit.

That gives us 1 times U plus 10 times O plus 100 times Y.

It works out backwards from the normal YOU order, but that doesn't matter.

I take those results, I put a plus sign in between them, and I wrap parentheses around them.

If it's not upper, I just return the word itself.

Now I explained the whole program.

I'm calling it faster_solve, and I take a formula.

The first thing I do is compile the formula. That gives me back the function that represents it, and while I'm there, I'm also returning the letters that are within the formula. That evaluation or compilation of the formula is done once.

Then I go through all the permutations of the digits taken length of letters at a time. Notice that I'm using actual integers here, not strings, for the individual digits.

If applying the function to the digits is true, then I did the same thing that I did before of making up a translation table and calling the formula.translate to get my final result. Note I only have to do this translation once.

On the specific permutation that I know works, I don't have to do it every time like I did in the previous version of solve. Then if there's an arithmetic error, I catch that and go on to the next permutation.

Now, the function compile_formula takes a formula as input. First I figure out all the letters, so that's going to be the Y, M, E, U, and O. Then I figure out that the list of parameters I want to put into my function is going to be a string consisting of the letters with a comma in between them.

I figure out all the tokens. A token is an individual term like the U or the equal signs or the ME and so on.

1.42 42. Speeding-Up

Unit2-42

Now let's take a look at what we did here.

So we had re.split and then this regular expression and then out of the formula. So what does this say? This says I want to split up the original formula, and the way I'm splitting it up is I'm taking sequences of the characters A-Z in a row. The plus means one or more, and the parentheses means keep that result.

So when I have YOU = ME squared, this part and this part will match, and they'll be returned as individual elements in this split, and so will the other parts. Re.split will return the list consisting of the parts that it found in the split plus everything in between. And it will be this type of list, and then I'm going to map compile word-- it didn't quite fit on one line-- to re.split, and that will give me-- these tokens will remain unchanged, and these will be converted into the form that multiplies out the digits. Then I'm going to assign that to the variable tokens. Now I've got my tokens.

The body of the function we just formed by concatenating all the tokens together. If I wanted to I could put spaces between them; it doesn't matter. And the function is lambda created with the parameters in the body, and then I just return the evaluation of the function, so that complies the function, does it only once rather than once per permutation.

And I return the letters that I found.

As a convenience, because when you're debugging this function you may want to say, "Well, what did this function look like?" "What did I come up with? What did I come up with?" I have an optional parameter here that says if you pass in verbose equals true, then it just prints out what it found. It just makes it a little bit easier to debug this function.

And now, when I run profiling again, look what I get. The time has sped up quite a bit on the simple list of 14 examples that we saw before. This version of the code is 25 times faster. On a more complex list, it's only 13 times faster. But it's at least an order of magnitude faster.

And we did that because we eliminated calls to eval. So we found the bottleneck. We found that eval was the function that was taking up the most amount of time in the previous profiling. We figured out a way to call eval less often by precompiling, and now we've done away with that difficulty.

1.43 43. Recap

Unit2-43

Let's do a recap of what we learned in this unit.

First, we talked about some Python features that were maybe new to you.

We used complex list comprehensions. That's something like x-squared for x in blah, blah, blah, if something.

We showed generator expressions, and that's similar but with parentheses.

We talked about generator functions or just generators, and we recognize those with the yield statement.

We talked about the idea of handling different types. This has the fancy name of polymorphism, meaning different forms.

We saw an example of that in timed calls where we said that the input n-- and then there are other inputs there--n could be either an integer, in which case we would do one thing, or a float, in which case we would do something else. We checked which is which by using "isinstance."

We talked about the eval function and how we can use that to map from a string to a Python object, which is the result of evaluating the string.

In particular, the case of evaluating to a function. Eval is a way of making the computation be done once and getting all that work over with so that we can then use that work repeatedly.

We also talked about instrumentation, and we did timing with the time.clock method that's built in, and then we built up timedcall and timedcalls routine.

And we talked about counting number of invocations of functions or assignment statements or whatever. There we came up with our own routine that we called c.

I guess I should say a little bit about variable naming conventions. Why did I use a short name like c here, whereas other places I had long, more expressive names? I guess the reason is c was used only for debugging purposes. It wasn't intended to be part of the final part of the program. I felt justified in having that be short, because I was going to be typing it and deleting it frequently. Things that are going to persist for longer have longer names

For example, it's fine to say "for i in range something" where there we know that i is an index integer, and it only persists over this short loop. It's okay to have a short name. If something lasts longer, we probably want it to have a longer, more descriptive name.

1.44 44. Zebra Puzzle Code with Additional Output

```
·· 1 def find_water_zebra():¶
·· 2···· import itertools¶
·· 3 ¶
·· 4···· houses = [first, _, middle, _, _] = [1,2,3,4,5]¶
·· 5 ¶
·· 6···· orderings = list(itertools.permutations(houses)) #1¶
·· 7 ¶
·· 8···· def imright(h1,h2):¶
·· 9········ "House h1 is immediately right of h2 if h1=h2 = 1"¶
· 10········ return h1-h2 == 1¶
· 11 ¶
· 12···· def nextto(h1,h2):¶
· 13········ "Two houses are next to each other if they differ by 1"¶
· 14········ return abs(h1-h2) == 1¶
· 15 ¶
· 16···· return [result for result in (¶
· 17············ (·· ('Drinks',·
```

```
{'coffee':coffee,'tea':tea,'milk':milk,'WATER':WATER,'oj':oj}),¶
· 18·····  ('Nations', {'Englishman':Englishman,
'Spaniard':Spaniard,¶
· 19·····  'Ukranian':Ukranian,
'Japanese':Japanese, 'Norwegian':Norwegian}),¶
· 20·····  ('Colours', {'red':red, 'green':green, 'ivory':ivory,
'yellow':yellow, 'blue':blue}),¶
· 21·····  ('Pets',··· {'dog':dog, 'snails':snails, 'fox':fox,
'horse':horse, 'ZEBRA':ZEBRA}),¶
· 22·····  ('Smokes',· {'OldGold':OldGold, 'Kools':Kools,
'Chesterfields':Chesterfields,¶
· 23·····  'LuckyStrike':LuckyStrike,
'Parliaments':Parliaments}),¶
· 24·····  )¶
· 25 ¶
· 26·····  for (red, green, ivory, yellow, blue) in orderings¶
· 27·····  if imright(green, ivory)·····  #6¶
· 28·····  for (Englishman, Spaniard, Ukranian, Japanese, Norwegian)
in orderings¶
· 29·····  if Englishman is red·····  #2¶
· 30·····  if Norwegian is first·····  #10¶
· 31·····  if nextto(Norwegian, blue)·····  #15¶
· 32·····  for (coffee, tea, milk, oj, WATER) in orderings¶
· 33·····  if coffee is green·····  #4¶
· 34·····  if Ukranian is tea·····  #5¶
· 35·····  if milk is middle·····  #9¶
· 36·····  for (OldGold, Kools, Chesterfields, LuckyStrike,
Parliaments) in orderings¶
· 37·····  if Kools is yellow·····  #8¶
· 38·····  if LuckyStrike is oj·····  #13¶
· 39·····  if Japanese is Parliaments·····  #14¶
· 40·····  for (dog, snails, fox, horse, ZEBRA) in orderings¶
· 41·····  if Spaniard is dog·····  #3¶
· 42·····  if OldGold is snails·····  #7¶
· 43·····  if nextto(Kools, horse)·····  #12¶
· 44·····  if nextto(Chesterfields, fox)··  #11¶
· 45·····  )¶
· 46·····  ]¶
· 47 ¶
· 48 ¶
· 49 hps = find_water_zebra()¶
· 50 print 'Result count:',len(hps)¶
· 51 if len(hps) < 20:¶
· 52····  for hp in hps:¶
· 53·····  print 'House ',''.join('%15s'%(num if num else 'House') for
num in range(1,6))¶
· 54·····  for item, props in hp:¶
· 55·····  propKeys = sorted(props.keys(),key=lambda k:props[k]) #
sort on props[k]: house¶
· 56·····  print '%-7s'%item, ''.join(['%15s'%propKey for propKey in
propKeys])¶
· 57·····  print¶
¶
```

The result of running this code is:

```
Result count: 1¶
House·············· 1·············  2·············  3·············
4·············  5¶
Drinks···········  WATER···········  tea···········  milk···········
oj·····  coffee¶
Nations·····  Norwegian·····  Ukranian····  Englishman·····  Spaniard·····
Japanese¶
Colours·····  yellow·····  blue·····  red·····
ivory·····  green¶
```

```
Pets·············· fox········· horse········ snails··········
dog········· ZEBRA¶
Smokes·········· Kools· Chesterfields····### OldGold··· LuckyStrike···
Parliaments¶
```