



Introduction to Computational Thinking

Imagine we have a problem to solve. Perhaps we need to get from Venice Beach to Echo Park lake, or make our sister's favorite type of cake for her wedding, or record the notes and musical phrases in a piece of music.

Thinking computationally entails defining a general **program** that solves the *general* kind of **problem** to which our *specific* problem belongs. A program is just a sequence of instructions: a set of instructions with an order. Recipes, directions, and sheet music are all programs; not all programs are computer programs or programs written in a programming language.

Computational thinking matters because it's *the* way to tackle hard problems with code. You'll solve plenty of challenging problems when you start doing data science in the second half of this course — and throughout your career as a data scientist! Computational thinking can be captured in the acronym (ADTO):

A. Abstraction

D. Decomposition

T. Translation

O. Optimisation

A. Abstraction

★ Abstraction is the stage in which we figure out what is relevant to the problem we're trying to solve as well as the general type of problem we're trying to solve is.

Take an everyday example. The problem at hand might be that we need some general way to cook our sister's favorite cake. We abstract by working out what's relevant to this problem and what isn't. Information about how to cook our sister's favorite dinner, or organize our sister's favorite party, is not relevant; what's relevant is:

- The type of cake our sister likes best
- The ingredients and their quantities
- What we need to do to the ingredients to make the cake

We want a recipe that will do nothing but bake our sister's favorite cake every time we need to do so. By working through the process of abstraction, we remove any irrelevant details that won't help us solve our problem.

Python example: Suppose someone asked us for a piece of code that invites everyone in a list of invitees to a party. With abstraction, we'd work out that this is a *list looping problem* (one that will require us to loop through a list). We'd then be able to write a piece of code that could be used *every time* we needed to solve such a problem.

Using Google to Help Us with Abstraction

We might not know what type of problem we're faced with - for example, we might be unsure whether the problem is one with string slicing or with list updating. Sometimes, we can use Google to help us find out, as we covered previously in the Art of Googling article.

The key principle of **Open-Source** culture is that we can contribute, and benefit from, the community of Internet users to help both the technologies in question and each other. When it comes to programming, we can make the most of the Open-Source world by developing the ability to express our problem generally and write a well-constructed Google search. There's absolutely no shame in the Google search and every senior programmer and data scientist will tell you the same. Typically, if we've expressed our problem well, one of the first pages will be a StackOverflow entry in which somebody else has asked for help with a similar problem. StackOverflow is a hugely useful peer-reviewed and moderated forum where programmers help each other out with programming problems. At the top of the page, we see the question: somebody wants to *get* (one of those crucial programming words) the last four characters of a string. They have a particular string at hand, but we don't really care about that. We want a general solution. So we scroll down.

Like this:

663

```
>>>mystr = "abcdefghijkl"
>>>mystr[-4:]
'ijkl'
```


✓ This [slices](#) the string's last 4 characters. The -4 starts the range from the string's end. A modified expression with `[:-4]` removes the same 4 characters from the end of the string:

```
>>>mystr[:-4]
'abcdefgh'
```


For more information on slicing see [this Stack Overflow answer](#).

share improve this answer

edited Apr 22 '18 at 6:59

 [Graham](#)
744 ● 7 ● 21

answered Nov 2 '11 at 16:29

 [Constantinus](#)
25.4k ● 6 ● 55 ● 80

By the way, if you want to read through the whole entry, click [here](#).

We then see an answer that's been ticked and upvoted over 600 times! This means it's been peer-reviewed and judged to have answered not just the question, but the *type* of question, well. The answer talks about string slicing - this is indeed the kind of problem we have here. Stack Overflow has helped us to abstract!

It's then up to us to extrapolate the general solution, which is what we'll look at in later Computational Thinking stages. Because so many features and abilities are built into Python's default data types like integer, string, bool etc., working out how to code or how to solve a problem is often just a matter of finding out what functionalities are built into those data types. A well-written Google search can find that out for us at the drop of a hat.

Sometimes, though, our problem is too hard for a one-query Google search to return a sufficiently similar question. We have to break our problem down into chunks first. All Abstraction is about is working out how we'd frame our question generally.

D. Decomposition

★ Decomposition is just figuring out what steps we need to take to reach a solution to our type of problem. It's as simple as that, but it's the most important stage in our process.

Decomposition asks: what are the actions (in computer-speak, *operations*) necessary and sufficient for making our sister's favorite cake? These steps could be **piecemeal instructions** – 'put the tray in the oven' – or instructions within control-flow structures: '*until* the egg whites are stiff, beat them'; or '*for each* of the lemons in the ingredients (no matter how many there are) shave their zest into the mixture'. Remember: control-flow structures are so-called because they control the flow of our piecemeal instructions, which are executed, by default, in order from the

first instruction to the last. If in doubt, we work out the piecemeal instructions at the Decomposition stage – for we can always figure out any control-flow structures at the Translation phase.

It's always best practice to try doing the Decomposition on paper. One crucial thing to bear in mind is this: **what counts as a piecemeal instruction partly depends on the toolbox we're using**. For example, 'put the tray in the oven' only counts as a piecemeal instruction if we're giving instructions to a human being (with an oven)! If we're writing instructions to our domestic robot, 'put the tray in the oven' will break down into *many* different, much simpler piecemeal instructions. Indeed if we're doing particularly fancy cooking, we might want to break that instruction down even to a human ('pre-heat the oven to 200 degrees C', 'place the tray on the second-to-highest rung at a 45-degree angle', 'pray in the general direction of Gordon Ramsay', etc.) What counts as a piecemeal instruction depends on our tools and our goals.

Typically, our piecemeal instructions in basic Python programs do stuff with the big triple: **variables**, **functions**, and **logic**. With them, we'll either be **getting** the value of a variable (like a number), **setting** the value of a variable to something else (like changing the time of your morning alarm), **executing** a function, or doing some logic: checking whether something is true (is the number > 3?). As such, '**get**', '**set**', and '**execute**' are all super important words in computer science. At the Decomposition stage, however, we're not worried about translating our piecemeal steps into steps that can be understood by Python. We'll be writing something called **pseudo-code** at the Decomposition stage: a sort of halfway-house between English and Python code. The first use of Python comes at the next stage.

Apply What You've Learned! (Optional Exercise)

Problem

You want to teach your new puppy how to sit and lay down, but all your puppy wants to do is cuddle and chew on your shoes.

Steps

1. Grab a pencil and a piece of paper
2. Write out the piecemeal instructions that you think you'd need to work through to solve this problem
3. Share your work with your mentor during your next call!

T. Translation

★ Translation is just figuring out how we can make the steps we came up with at the Decomposition stage language-precise, that is, precise in the language we're using (in your case, translating your steps from English to Python).

What counts as a precise piecemeal instruction depends on the programming language we're using, just as a precise instruction in German differs from one in English. Even in English, certain instructions can be precise in certain dialects or contexts and imprecise in others. The beauty of programming languages, however, is that any instruction is either precise (and so syntactically correct) or imprecise and so syntactically incorrect.

Translation asks: 'how can we transform the steps we came up with in the Decomposition phase into precise instructions understood by Python (and thereby the computer)? Sometimes, an instruction that was piecemeal and very simple at the Decomposition stage has to be broken down into many different piecemeal instructions in Python. For example, suppose we want to play a game of masked murder mystery at our wedding, and want to give each of our guests a new name made from the letters in their actual names. The new name will be an anagram of their first name and last name combined. One instruction in the Decomposition stage might be:

Give each wedding guest a new name that's an anagram of their name

Sounds simple enough! Suppose further that we've stored our contacts in a table that looks like:

Title	First Name	Last Name
Mr	Barack	Obama
Ms	Jennifer	Love Hewitt
Mr	Larry	David
...

At the Translation Stage, we'd need to write multiple lines of Python to do this, making use of Python's tools (its data types, operations, and the functions and variables attached to its data types). For example, we first might transform the table into a list of lists (i.e, a list, whose elements are themselves lists) called 'guests', so that each element of the bigger list is a row from the table. This would look like:

```
1 guests = [['Mr', 'Barack', 'Obama'], ['Ms', 'Jennifer', 'Love Hewitt'], ['Mr', 'Larry', 'David']]...
```

We then might want to make a new list, at the moment empty, for the new names of each guest:

```
3 newNames = []
```

We'd then want to go through each item in the list guests (where each item is a list with three elements) and apply an anagram function to the second and third elements of each item. This

might involve using Python's range of control-flow structures: abilities built into Python that allow us to do piecemeal instructions repeatedly. We already saw one of these when we were introduced to the *while* loop, but don't worry about learning them all now. In essence, they allow us to do things like execute certain instructions *only if* some condition is true (for example, while we haven't got through all our guests yet).

We won't finish off this task now, as you'll need to learn a bit more Python before you can complete this step, but the important point is this: **Translation is about using the tools available in Python to implement the great ideas you came up with during the Decomposition stage.**

Rest assured: there's not a Python programmer in the world who knows *everything* Python can do. We're always experimenting, figuring things out, and applying tools that we found out about just moments ago with a quick Google search. That's all part of the fun.

0. Optimization

★ Optimization involves figuring out how we can make the steps we came up with in the previous stage as efficient as possible.

Optimization is reached if our Python program already does the job but we then refine the program's work to be as efficient as possible.

For example, do we need to beat the egg whites for 5 hours, or will 5 minutes suffice? How can we make our program as cheap to execute as possible: can we reduce the number of instructions in the recipe or the number of instructions in our directions? Can we reduce their complexity?

While we might have already used Python's control-flow structures at the previous stage, programmers often realize the power of control-flow structures at the Optimization stage. These structures can make things a lot quicker and easier, reducing the number of piecemeal instructions in our code. For example, we might notice that we're repeating an action (say, the heating of water) until some condition is true (water is boiling). We can make a Boolean variable called `boiling` that's by default `False`, and we can heat the water until it becomes `True`.

```
1  boiling = False
2  temp = 0
3  while (not boiling):
4      heat(water)
5      temp = temp + 1
6      if (temp == 100):
7          boiling = True
8  grate(lemons)
```

Let's walk through each of the steps outlined in the above code.

1. The first thing we're doing in line 1 is making our boiling variable, and giving it the value of False: our water isn't boiling at the moment.
2. Then we're making another variable that tracks the exact temperature of our water in the second line. Let's assume the water is currently at 0 degrees centigrade, so we put 0 into that variable.
3. Then we have a while loop. Since the boiling variable's value is at first False, the value of the condition (not boiling) is at first True.
 - a. Remember: the instructions indented in a while loop execute *if and only if* the condition specified after the word 'while' is True.
 - b. Because of this, we go into the indented instructions within the while loop (also known as the while-block).
4. Here we heat the water on line 4
5. In line 5 make the value of our variable-temperature equal to what it was, plus one (we read from right to left here).
6. Once our temperature variable gets to a certain value (specifically, 100), boiling's value becomes True, so (not boiling)'s value becomes False. Then we break out the while loop.
7. Then, and only then, do we get around to grating the lemons.

Consider, by contrast, this code.

Can you see how it's doing the same thing as the first chunk of code we looked at? But look at the number of lines! We've cut the number of instructions from 8 to 5. Count the number of variables. We now have 3, rather than 4. This means we're using less memory because we're storing less information (the values of variables have to be stored somewhere)! Our code here is better. It is optimized.

```
1  temp = 0
2  while (temp < 100):
3      heat(water)
4      temp = temp + 1
5  grate(lemons)
```

Don't worry if this doesn't all make sense yet; thinking in this structured way will take time and practice (happily, you'll get lots of practice while working on this course!)

To finish off our cake-making example: It suffices at this stage to note that putting such control-flow structures into our programs can help generalize our solution (no matter how big the cake we're trying to make is, those egg whites need to be beat until stiff; even if it takes half an hour!)