Interpreting A Language

2019

We've learned lots about programming computers, but so far we haven't really covered how computation actually *works*! One fun way to learn about how computers work is to build one. In this project, you'll build a model computer, define a language for that computer, and then write programs for it.

Today, we're going to use Python to *define our own language*! Writing a language can sound intimidating, but it's actually a fun experience which tells you a lot about computers and computer programs. Tom's own PhD supervisor adviser recently told him that writing an interpreter similar to the one you will — during his PhD! — taught him lots about what programs really are and how to think about computation. Tom did the same and agrees. About 50 years ago, somebody prove what the simplest computer language which can compute *anything* needs to have. We're going to write a version of that![1]

Computers need a couple of things to perform calculations. They need to know what they're performing calculations *on* — they need data, which they store in memory! — and they need to know what to *do* with the information — they need some sort of program, the simplest of which is really just a set of instructions. [2]

[1] We're not going to implement the actual original language, because it's messy and complicated; instead, we're going to define a nicer version and use that.

[2] Different people have different perspectives on whether the data and the instructions can be in the same place, or whether they should be stored in two different places. We will be storing data and instructions in two separate places. This is called a "Von Neumann" architecture!

Data

You've learned about a few data structures in Python, but because this is the first language we've interpreted, we want to keep things quite simple. What sort of data do we want our language to process?

The simplest kind of information, really, is a "bit": a `0` or a `1`. However, manipulating `0`s and `1`s directly will take a very long time! To make things easier for ourselves, let's represent information as numbers — as integers, specifically. Then, the data is still very simple, so our language can still be simple[3], but we make it easier to manipulate the data we have.

[3] If we wanted to perform computation on strings, we could represent this kind of information too, but as you will see, this would make our language much less simple.

Representing just one number isn't enough for us, though. What if we want to perform a computation which finds out whether one thing is bigger than another? We need at least a few numbers. We'll therefore store all of our information in one big list of numbers. You might think of it as writing all of our numbers down in a list on a piece of paper. How long should our list be? For our purposes, let's make the list 1024 items long[4], and let's set every item in the list to `0` initially — just so it has some value to begin with.

[4] That's 2^{10}.

We've now defined the memory for our computer. Fantastic!

Data Pointers

There's one thing missing, though. Our computer has a big list of numbers to store information in, but it needs to remember what part of the list it's looking at. We can represent this as some number indicating which item in the list we're looking at. We only need one! This is called a *data pointer*. The data pointer lets the computer look at a part of the list — we'll call each space on the list an element, as if the list was an array or a Python list — and by changing the data pointer, we can move around our elements. . . but to "move around" memory, the computer is going to have to run a program. It's time to define the instructions our computer understands!

Instructions

To calculate things, our language needs to describe a few things. Once we've worked out what they are, we can design the language itself.

First: we need to be able to manipulate the integers in our lists! We could build in really complicated calculations in our language, but we might as well keep it simple for now; let's start with some command that can just add or subtract `1` from element we're pointing to.

To change the element we're pointing to, we also need to change the data pointer! So we need some kind of instruction in our language that will change our data pointer. We could just add or subtract one to that, too.

It actually turns out that this is almost enough to perform any calculation! Only one thing is missing, which is that the language needs some way to make decisions. We'll do this with a *conditional jump*. Specifically, if the element our data pointer is on is non-zero, then it will jump to some point we mark in our code.

We'll represent adding and subtracting the value in our current cell by `1` with `+` and `-` respectively. We'll represent adding or subtracting `1` to the data pointer with `>` and `<` respectively. For conditional jumping, we'll mark our code with `[` for the point we might jump to, and `]` for the jump. This way, the brackets represent a *block* of code that will repeat if it ends in a non-zero value. Using this, the following program would move a number from one element to the one next to it:

```
[>+<-]
```

Try starting with the number `3` in an element of our list, and see whether you can follow what this program will do to shift it into the next element.

Implementing the Machine

We now have a simple language made of six symbols:

Symbol	*Meaning*
+	Increment current element
-	Decrement current element
>	Increment data pointer (shift to right element)
<	Decrement data pointer (shift to left element)
[	Begin conditional jump block
]	End conditional jump block

Your mission is to write a program which will take a program in this language as input, and execute it! You can use whatever Python tools and data structures you like.

Once you're finished, try writing some programs in our language. Can you write a program that will multiply two numbers together, for example?

Once you have written some programs in the language, try extending the language. Can you think of different kinds of jump that you might want to implement? What about different sorts of incrementation? Are there symbols you could define in your langauge that would have made your earlier programs simpler? Implement these new symbols; then, rewrite your earlier programs to use them.

Appendix: fun problems

If you've finished this and want something fun to play with, try writing the simplest Python program you can think of that will print itself to the console. This is a fun exercise with a simple solution, but can be difficult to think of! Once you've solved the puzzle, try writing it. **DON'T TELL OTHER PEOPLE HOW TO DO IT** — the fun is in trying to work it out, and once the secret is explained the lesson is lost!