



Programming Basics: Statements & Functions: Crash Course Computer Science #12

Crash Course: Computer Science

<https://youtube.com/watch?v=l26oaHV7D40>

<https://nerdfighteria.info/v/l26oaHV7D40>

Hi, I'm Carrie Anne, and welcome to CrashCourse Computer Science!

Last episode we discussed how writing programs in native machine code, and having to contend with so many low level details, was a huge impediment to writing complex programs. To abstract away many of these low-level details, Programming Languages were developed that let programmers concentrate on solving a problem with computation, and less on nitty gritty hardware details.

So today, we're going to continue that discussion, and introduce some fundamental building blocks that almost all programming languages provide.

INTRO

Just like spoken languages, programming languages have statements. These are individual complete thoughts, like "I want tea" or "it is raining".

By using different words, we can change the meaning; for example, "I want tea" to "I want unicorns". But we can't change "I want tea" to "I want raining" - that doesn't make grammatical sense. The set of rules that govern the structure and composition of statements in a language is called syntax.

The English language has syntax, and so do all programming languages. `1:03"A equals 5"` is a programming language statement. In this case, the statement says a variable named `A` has the number 5 stored in it. This is called an assignment statement because we're assigning a value to a variable.

To express more complex things, we need a series of statements, like "`A is 5, B is ten, C equals A plus B`". This program tells the computer to set variable '`A`' equal to 5, variable '`B`' to 10, and finally to add '`A`' and '`B`' together, and put that result, which is 15, into -- you guessed it -- variable `C`. Note that we can call variables whatever we want. Instead of `A, B` and `C`, it could be apples, pears, and fruits.

The computer doesn't care, as long as variables are uniquely named. But it's probably best practice to name them things that make sense in case someone else is trying to understand your code. A program, which is a list of instructions, is a bit like a recipe: boil water, add noodles, wait 10 minutes, drain and enjoy.

In the same way, the program starts at the first statement and runs down one at a time until it hits the end. So far, we've added two numbers together. Boring.

Let's make a video game instead! Of course, it's way too early to think about coding an entire game, so instead, we'll use our example to write little snippets of code that cover some programming fundamentals. Imagine we're building an old-school arcade game where Grace Hopper has to capture bugs before they get into the Harvard Mark 1 and crash the computer!

On every level, the number of bugs increases. Grace has to catch them before they wear out any relays in the machine. Fortunately, she has a few extra relays for repairs.

To get started, we'll need to keep track of a bunch of values that are important for game-play, like what level the player is on, the score, the number of bugs remaining, as well as the number of spare relays in Grace's inventory. So, we must "initialize" our variables, that is, set their initial value: "`level equals 1, score equals 0, bugs equals 5, spare relays equals 4, and player name equals 'Andre'`". To create an interactive game, we need to control the flow of the program beyond just running from top to bottom.

To do this, we use Control Flow Statements. There are several types, but If Statements are the most common. You can think of them as "If `X` is true, then do `Y`".

An English language example is: "If I am tired, then get tea" So if "I am tired" is a true statement, then I will go get tea. If "I am tired" is false, then I will not go get tea. An IF statement is like a fork in the road. Which path you take is conditional on whether the expression is true or false -- so these expressions are called Conditional Statements.

In most programming languages, an if statement looks something like "If, expression, then, some code, then end the if statement". For example, if "`level`" is 1, then we set the score to zero, because the player is just starting. We also set the number of bugs to 1, to keep it easy for now.

Notice the lines of code that are conditional on the if-statement are nested between the IF and END IF. Of course, we can change the conditional expression to whatever we want to test, like "`is score greater than 10`" or "`is bugs less than 1`". And If-Statements can be combined with an ELSE statement, which acts as a catch-all if the expression is false.

If the level is not 1, the code inside the ELSE block will be executed instead, and the number of bugs that Grace has to battle is set to 3 times the level number. So on level 2, it would be six bugs, and on level 3 there's 9, and so on. Score isn't modified in the ELSE block, so Grace gets to keep any points earned.

Here are some examples of if-then-else statements from some popular programming languages -- you can see the syntax varies a little, but the underlying structure is roughly the same. If-statements are executed once, a conditional path is chosen, and the program moves on. To repeat some statements many times, we need to create a conditional loop.

One way is a while statement, also called a while loop. As you might have guessed, this loops a piece of code "while" a condition is true. Regardless of the programming language, they look something like this: In our game, let's say at certain points, a friendly colleague restocks Grace with relays!

Hooray! To animate him replenishing our stock back up to a maximum of 4, we can use a while loop. Let's walk through this code.

First we'll assume that Grace only has 1 tube left when her colleague enters. When we enter the while loop, the first thing the computer does is test its conditional...is relays less than 4? Well, relays is currently 1, so yes.

Now we enter the loop! Then, we hit the line of code: "`relays equals relays plus 1`". This is a bit confusing because the variable is using itself in an assignment statement, so let's unpack it.

You always start by figuring out the right side of the equals sign first, so what does `5:05"relays plus 1"` come out to be? Well, relays is currently the value 1, so `1 plus 1` equals 2. Then, this result gets saved back into the variable relays, writing over the old value, so now relays stores the value 2.

We've hit the end of the while loop, which jumps the program back up. Just as before, we test the conditional to see if we're going to enter the loop. Is relays less than 4?

Well, yes, relays now equals 2, so we enter the loop again! `2 plus 1` equals 3. So 3 is saved into relays. Loop again.



Programming Basics: Statements & Functions: Crash Course Computer Science #12

Crash Course: Computer Science

<https://youtube.com/watch?v=l26oaHV7D40>

<https://nerdfighteria.info/v/l26oaHV7D40>

Is 3 less than 4? Yes it is! Into the loop again. 3 plus 1 equals 4.

So we save 4 into relays. Loop again. Is 4 less than 4?...

No! So the condition is now false, and thus we exit the loop and move on to any remaining code. That's how a while loop works!

There's also the common For Loop. Instead of being a condition-controlled loop that can repeat forever until the condition is false, a FOR loop is count-controlled; it repeats a specific number of times. They look something like this: Now, let's put in some real values.

This example loops 10 times, because we've specified that variable 'i' starts at the value 1 and goes up to 10. The unique thing about a FOR loop is that each time it hits NEXT, it adds one to 'i'. When 'i' equals 10, the computer knows it's been looped 10 times, and the loop exits.

We can set the number to whatever we want -- 10, 42, or a billion -- it's up to us. Let's say we want to give the player a bonus at the end of each level for the number of vacuum relays they have left over. As the game gets harder, it takes more skill to have unused relays, so we want the bonus to go up exponentially based on the level.

We need to write a piece of code that calculates exponents - that is, multiplying a number by itself a specific number of times. A loop is perfect for this! First let's initialize a new variable called "bonus" and set it to 1.

Then, we create a FOR loop starting at 1, and looping up to the level number. Inside that loop, we multiply bonus times the number of relays, and save that new value back into bonus. For example, let's say relays equals 2, and level equals 3.

So the FOR loop will loop three times, which means bonus is going to get multiplied by relays... by relays... by relays. Or in this case, times 2, times 2, times 2, which is a bonus of 8! That's 2 to the 3rd power!

This exponent code is useful, and we might want to use it in other parts of our code. It'd be annoying to copy and paste this everywhere, and have to update the variable names each time. Also, if we found a bug, we'd have to hunt around and update every place we used it.

It also makes code more confusing to look at. Less is more! What we want is a way to package up our exponent code so we can use it, get the result, and not have to see all the internal complexity.

We're once again moving up a new level of abstraction! To compartmentalize and hide complexity, programming languages can package pieces of code into named functions, also called methods or subroutines in different programming languages. These functions can then be used by any other part of that program just by calling its name.

Let's turn our exponent code into a function! First, we should name it. We can call it anything we want, like HappyUnicorn, but since our code calculates exponents, let's call it exponent.

Also, instead of using specific variable names, like "relays" and "levels", we specify generic variable names, like Base and Exp, whose initial values are going to be "passed" into our function from some other part of the program. The rest of our code is the same as before, now tucked into our function and with new variable names. Finally, we need to send the result of our exponent code back to the part of the program that requested it.

For this, we use a RETURN statement, and specify that the value in 'result' be returned. So our full function code looks like this: Now we can use this function anywhere in our program, simply by calling its name and passing in two numbers. For example, if we want to calculate 2 to the 44th power, we can just call "exponent 2 comma 44." and like 18 trillion comes back.

Behind the scenes, 2 and 44 get saved into variables Base and Exp inside the function, it does all its loops as necessary, and then the function returns with the result. Let's use our newly minted function to calculate a score bonus. First, we initialize bonus to 0.

Then we check if the player has any remaining relays with an if-statement. If they do, we call our exponent function, passing in relays and level, which calculates relays to the power of level, and returns the result, which we save into bonus. This bonus calculating code might be useful later, so let's wrap it up as a function too!

Yes, a function that calls a function! And then, wait for it... we can use this function in an even more complex function. Let's write one that gets called every time the player finishes a level.

We'll call it "levelFinished" - it needs to know the number of relays left, what level it was, and the current score; those values have to get passed in. Inside our function, we'll calculate the bonus, using our calcBonus function, and add that to the running score. Also, if the current score is higher than the game's high score, we save the new high score and the player's name.

Finally, we return the current score. Now we're getting pretty fancy. Functions are calling functions are calling functions!

When we call a single line of code, like this the complexity is hidden. We don't see all the internal loops and variables, we just see the result come back as if by magic... a total score of 53. But it's not magic, it's the power of abstraction!

If you understand this example, then you understand the power of functions, and the entire essence of modern programming. It's not feasible to write, for example, a web browser as one gigantically long list of statements. It would be millions of lines long and impossible to comprehend!

So instead, software consists of thousands of smaller functions, each responsible for different features. In modern programming, it's uncommon to see functions longer than around 100 lines of code, because by then, there's probably something that should be pulled out and made into its own function. Modularizing programs into functions not only allows a single programmer to write an entire app, but also allows teams of people to work efficiently on even bigger programs.

Different programmers can work on different functions, and if everyone makes sure their code works correctly, then when everything is put together, the whole program should work too! And in the real world, programmers aren't wasting time writing things like exponents. Modern programming languages come with huge bundles of pre-written functions, called Libraries.

These are written by expert coders, made efficient and rigorously tested, and then given to everyone. There are libraries for almost everything, including networking, graphics, and sound -- topics we'll discuss in future episodes. But before we get to those, we need to talk about Algorithms.

Intrigued? You should be. I'll see you next week.