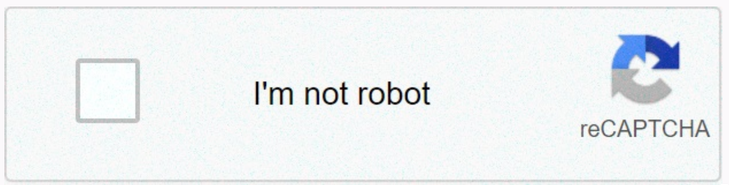

Continue

If statement in ruby

One of the most powerful features present in most programming and scripting languages is the ability to change the program flow as certain conditions change. The easiest form of flow control and logic in Ruby is called an if statement (or technically speaking in Ruby, since everything is an expression, one if expression). These experssions if you essentially check if a condition is true or not. Ruby is created by typing the if keyword, followed by a condition, optionally followed by the keyword then, then the code you want to run if the condition is true, and finally the final keyword to finish the structure. So they must look something like this: 1 2 3 if the condition then #Code to execute if condition is true end As mentioned, however, the then is often removed from the structure as it simply provides separation of condition and code to run – a line if statements require the then to separate these, but multi-line as we will write most of the time you can use a new line to separate the condition and code. 1 2 3 if the condition #Code run if condition is true end From here the obvious question is: How can I formulate a condition?. Well, first of all, it's important to note that any expression you evaluate to a true or false value may be in place for the condition there – which includes Boolean variables and method calls! So the following would always give out Hello as 'some_variable', in which case, will always evaluate to truth: 1 2 3 4 5 some_variable = true if some_variable #In this case, 'if true' would do exactly the same! puts Conditions Hello Real Endings, for example by comparing one variable to another, can be created using conditional or compared operators. These are operators that are made to compare things, and when values are present on both sides of the operator, this will result in a true or false value. The basic conditional operators are: == - Equals != - Not equal to > - It's higher than < - It's less than > = - It's bigger or equal to < = - It's less than or equal to So to put them in actions, let's create a basic script that checks if the user is an administrator of our makeup system - it won't be very safe or really good for anything at the moment, but it will be a good example to make sure we know these things. The easiest way to authenticate a user is probably through a username and password, so the first thing our sample script has to do is ask for that data. Fortunately we already know how to do it using poses and get, however in this example it is more suitable to use print instead of pose – both essentially do the same, however it puts adds a new line to the while printing doesn't. So let's write this part of the script! 1 2 3 4 print Username: username = gets.chomp print Password: password = gets.chomp gets.chomp Here, statements if they are to do their magic. In this case we can only harshly encode the administrator username and password to our script (not a good idea on a real system, but it will work fine here). So first our comparison between the administrator username and what the user actually entered: 1 2 3 if username == foo #foo in this case is our administrator username #Stuff to do if the username is correct end From here we want to check if the password is correct. If the statements and other similar structures can be nested, so in this case it might be a good idea to put another 'if' statement inside it to check if the password is correct, so: 1 2 3 4 5 if username == foo #foo in this case it is our administrator username if password == bar #bar in this case is our administrator password welcome, administrator! final You can see that I added a little puts message so that we can tell when the username and password were correct (foo and bar). If you take our code so far and put it in a ruby file and run it, you should see that our special message only shows when foo and bar are introduced! But what do we do if we want something to happen when the username and password are entered incorrectly? Perhaps we want to show a message telling the user that the details they entered were wrong. We could write a few statements if using the operator is not equal to (!=), but this looks like a waste of processing power, as we have already determined whether the credentials are correct or not. That's where the most statements come in. After a statement if, just before completion, you can enter one more clause. The code in this clause will run if the judgment if it wasn't true (and if none of the others if it's in that chain were true, but we'll get to that in a minute). These are achieved by using the keyword the more where the end would be in general, and the end movement at the end. The syntax is as follows: 1 2 3 4 5 if the condition #Stuff if the condition is true more #Stuff if the condition is false end In our example we could easily use this functionality to generate errors to the user in our sample program: 1 2 3 4 5 6 7 8 9 if username == foo #foo in this case is our administrator username if you password == bar #bar in this case is our administrator password sets Welcome, administrator! more #If the user's password is not bar sets incorrect password. end #If the user name is not foo puts incorrect user name. as previously played however, you can actually have more than one condition in a statement if string, in which case the clause more only if none of the conditions are true. A good way to show this in our example is if our system had multiple users. Maybe we want to add a guest user who doesn't require a password. We may add these additional conditions by using what called if not, written through the keyword elsif. Read 'if' strings out loud often makes some sense of them: If this, do this, if this, do this, do this, do this, if not, do this, this should also make clear the fact that the next element of the string will only proceed if the above is false - so elsifs are only like checking if the main condition is false, and the others only check if everything else is false. The syntax of an 'elsif' in a string is as follows: 1 2 3 4 5 6 7 if the condition #Stuff if the condition is true elsif condition2 #Stuff if the condition2 is true but condition1 is false more #Stuff if the condition and condition2 are false end With this, it should be quite easy to create a guest user: 1 2 3 4 5 6 7 8 9 10 11 if username == foo #If is the administrator if password == bar #If the password is correct puts Welcome, administrator! more #If the password is not correct puts Incorrect Password. final elsif username == guest #If is the guest user puts Welcome, guest! more #If no known user name was entered puts incorrect username. Final It could be argued that asking the invited user for their password is not necessary , and this could be solved by moving the password indicator and entering the administrator's if statement and you cannot see if you can do so, but it is not important for this example. The best way to learn about statements if it's really playing with them. A good idea might be to try to create a script that takes the score of a 100 exam and then comes out a note based on the score entered (within certain grade limits – the higher you and the less that comparisons should be useful here!). The only problem you can encounter when creating this script is the comparison of different types of data. If you try to compare a string variable (for example, something received from gets) with an integer (such as 1), you'll encounter problems. As such, you may want to explore to_i and to_s methods. The first will probably want to be used in this example and is a method to convert strings to integers, while the second is less useful in this example, but may be more useful in others, and converts integers to strings. You can use the to_i method in gets.chomp itself (probably the best option here), or if you want you could use it in all conditions compared to an integer. The first one is basically shown below: 1 2 3 4 5 6 print Enter a score: score = gets.chomp.to_i if the score > 9000 puts It's over 9000! end up moving away from this a little bit, which we hope you now have to understand and be able to use, there is also a little more flexibility and some shorter options in this 'if' functionality. Firstly, one if basic can be in After the line you have to run if true, as well: 1 set 5 is greater than 4! if 5 > 4 Similarly, a basic statement if be compressed into a line if the keyword is then used to replace the new line: 1 If 5 > 4 seeds sets 5 is greater than 4! Ruby endpoint also has something called a ternary operator that provides a shortcut way to make basic comparisons. The syntax of this is a condition, followed by a question mark, followed by the expression to be given if the condition is true, followed by a colon, followed by the expression to be given if the condition is false, as can be seen below: 1 condition ? true_expression : false_expression This is useful when a quick and compact decision between expressions is required, for example the following: 1 2 3 4 print Username: username = gets.chomp puts username == admin? Welcome, Maester! : Hello, #{user name}. In many ways, a computer program is like a journey for your data. Throughout this journey, the data meets many things that have an impact on it and is changed forever. Like any trip, you have to go a certain path. On this road, there are many paths. Some paths are chosen and others are not. Which paths are chosen depends on the ultimate goal. When you write programs, you want your data to make the right decisions. You want your data to do the right thing when they're supposed to. In computer programming, this is called conditional flow. How do we make the data do the right thing? We use conditionals. Conditional A conditional is a fork (or many forks) on the road. The data is approaching a conditional conditional and the conditional indicates the data to go to based on some defined parameters. Conditionals are formed by a combination of whether declarations and comparisons and logical operators (<, >, <=, >=, ==, !=, && ||). They are basic logical structures that are defined by the reserved words if, if not, elsif and final. Note that alsif is missing an e. No more talking, time to code. Create a file called conditional.rb and type the following code. # conditional.rb puts Put in a number a = gets.chomp.to_i if a == 3 sets to is 3 elsif a == 4 set to is 4 more set to not 3, neither 4 end Here we are using gets to let the user enter a number, chomp gets rid of the new line created when the user enters the data, and to_i is a method that can be called in a string to convert it to an integer. We have to turn the entry into an integer because it always gives us a string. Run this code three times and do the following: The first time, type the number 3 and press Enter. The second time, type the number 4 and press Enter. The third time, type any number other than 3 or 4, and then press Enter. You can repeat the third step more than once to see its effect. What making your code is to check, using the operator == you learned earlier, to see if the entry is equal to the number we defined. We have effectively controlled the flow of the program by establishing conditionals in an if statement. Nice Nice The following examples are all valid Ruby conditionals. # Example 1 if x == 3 puts x is 3 end # Example 2 if x == 3 set x is 3 elsif x == 4 puts x is 4 end # Example 3 if x == 3 puts x is 3 more put x is not 3 final # Example 4: you must then use keyword when using the 1-line syntax if x == 3 then set x is 3 end Last, because Ruby is such an expressive language, it also allows you to add the condition if in the end. Example 1 above could be rewritten like this: put x is 3 if x == 3 Ruby also has a reserved word, unless. It acts as the opposite of whether, so it can be used like this: set x is not 3 unless x == 3 Comparisons Go through these comparison operators in a little more depth so you can build some more complicated conditional statements. One thing to remember is that comparison operators always return a Boolean value. A Boolean value is true or false, nothing else. We'll test them at the IRB to see how they work as well. == - The operator equals. Anything to the left of the symbol is exactly the same as anything on the right. We talked about this operator early in our chapter on variables so it shouldn't be totally foreign. irb :001 > 5 == 5 => true irb :002 > 5 == 6 => false irb :003 > 'abc' == 'abc' => true irb :004 > 'abc' == 'abcd' => false irb :005 > 'abc' == 'aBc' => false irb :006 > '5' == '5' => true irb :007 > '5' == '6' => false irb :008 > '5' == 5 => false warning that we can compare strings for > equality. To be equal, two strings must have the exact same value. The specific value does not matter, only that both strings have the same value. If there is any difference at all, the strings are not the same. We launched this latest example as a reminder that two values must have the same type or are not the same. Therefore, the string '5' is not the same as number 5. The comparison is therefore false. != - Operator not equal to. Anything to the left of the symbol does not equal anything on the right. irb :001 > 4 != 5 => true irb :002 > 4 != 4 => false irb :003 > 4 != 156 => true irb :004 > 'abc' != 'def' => true irb :005 > 'abc' != 'aBc' => true irb :006 > '5' != 5 => true As with ==, we can easily compare two values of the same type for inequality and get reasonable results. However, if the two values have different types, the return value is true. < - The minus symbol than anything to the left of the symbol has a value more > low than anything to the right of the symbol. Anything to the left of the symbol has a higher value than anything to the right of the symbol. # Example using less than and greater than irb :001 > 4 < 5 => true irb :002 > 4 > 5 => false irb :003 > 4 < 5 => irb :004 > 4 > 5 => fals irb : 005 > 42 < 402 => fals irb :006 > 42 > 402 => true irb : 007 > 42 < 420 => true irb :008 > 42 < 420 ArgumentError (comparació de String with with failed) irb :009 > 42 > 420 ArgumentError (comparison of Enter with failed string) The numbered examples 005, 006, and 007 are especially difficult! Make sure you understand them. When comparing strings, the comparison is character by character. Ruby moves from left to right on the ropes in search of the first character who is different from her counterpart on the other string. Once he finds a character that differs, he compares that character to his counterpart, and makes a decision based on that. If both strings are equal to the length of the shorter string as in the final example, the shorter string is considered less than the longest string. The last two examples show that < and > cannot be used with values of different types. (The same goes for < = and > = operators as below screenshot shown.) < = - The symbol less than or equal to. Anything to the left of the symbol is less than or equal to anything on the right. > = - the largest or equal symbol to. Anything to the left of the symbol is greater than or equal to anything on the right. irb :001 > 4 <= 5 => true irb :002 > 5 >= 5 => true irb :003 > 4 >= 5 => false irb :004 > 4 >= 3 => true irb :005 > 4 >= 4 => true Of course, <= and > = operators work equally well with strings. By combining chord expressions, you are beginning to get a decent understanding of the conditional flow. It is also possible to

combine multiple conditional expressions to create a more specific scenario. We can do this using &; and || Operators. To see what they mean. && - the operator and. Expressions to the left and right of this operator must be true for the entire expression to be evaluated to true. irb:001 > (4 == 4) && (5 == 5) => true irb :002 > (4 == 5) && (5 == 5) => false irb :002 > (4 == 5) && (5 == 6) => false || - the operator o. Either the expression on the left must be true, or the expression on the right must be true for the whole expression to be evaluated to true. irb:001 > (4 == 4) || (5 == 5) => true irb :002 > (4 == 5) || (5 == 5) => true irb :002 > (4 == 5) || (5 == 6) => false ! - the operator does not. When you add this in front of a Boolean expression, it will change that Boolean value to the contrary. irb :001 > ! (4 == 4) => false What happens here is Ruby first evaluates what is in the brackets and then the ! the operator changes it. We know that 4 == 4 would return true. If we say !true then that comes back false. You may think of !true how to say is not true. This is useful for readability and also helps the computer to more accurately understand its intention. The computer will evaluate parentheses in normal algebraic. Ruby follows a priority order when deciding how to evaluate multiple expressions. Below is a list of from the highest priority order (top) to the lowest (lower). <=, <, >, >= - Comparison ==, != - Equality && - Logic I || - Logic or Knowing this, we can see the following expression and see how it is evaluated. If this statement is true, then the program will run the # do something code in the next line. If the x& statement and it is false, then the z will be evaluated. If z is true, the code for the next line will be evaluated. If the z is false, the code will come out of the if statement. Ternary operator Ruby has a good choice for short and concise conditions if statements. The ternary operator is a common ruby language that makes a quick if/easier statement and keeps everything in a line. Does the ternary operator use a combination of the ? And:. # Example of irb ternary operator :001 > true ? this is true : this is not true => this is true irb :001 > false ? this is true : this is not true => this is not true How does this work? You may have inferred that first the computer evaluates what is to the left of the ?. If the expression to the left of ? is it true, the code directly to the right of the ? runs. If the code to the left of ? is false, then the code directly to the right of the : runs. Ternary operators are definitely helpful as you get familiar if your tax returns. If you feel you are not sure how this works, play with it in irb and try a few other cases out. Nothing can create familiarity more quickly than good exposure and repeated experimentation. Case statement The final conditional flow structure we want to talk about is called a case statement. A case ruling has similar functionality to an if statement but with a slightly different interface. Case statements use the case of reserved words, when, if not, and end. First create one to define a case, and then evaluate the case value and which operation to complete if that case is true. As always, talking about these things is much harder than simply observing how code behaves. We create a file called case_statement.rb to play around with some case statements and see how they work. # case_statement.rb a = 5 case to when 5 sets to is 5 when 6 sets to is 6 plus set to is not even 5, nor 6 end This example is approximately equivalent to the following statement if/elsif/else: # if_statement.rb a = 5 if a == 5 sets to is 5 elsif a == 6 sets to is 6 more set to is neither 5, neither 6 end The main differences are that we only have to specify the variable that we want to test once (as an argument to case) and do not specify a == on the individual when the statements. You can also save the result of a case statement to a variable. Let's refactor the above code to do just that. That way we don't have to write puts so many times. # <-- refactored to = 5 response = case to when 5 to is 5 when 6 a is 6 more to is not even 5, nor 6 final puts answer There is a second form of the statement of the case that does not take an argument: # case_with_no_arg_statement.rb a = 5 case when a == 5 puts to is 5 when a == 6 puts to is 6 more puts to is neither 5, nor 6 final The difference here is that we do not provide an argument to line 5, and we have to fully test each value with a ==; most developers prefer to use if/elsif/else/end, but there are situations where this form is preferred. We're not going to get into this here. As you can see, there are a lot of uses for case statements and they can be very powerful tools when writing Ruby programs. Remember that, if you feel uncomfortable with these, spend some time modifying them and seeing how they respond to the changes you make. Test your limits to see what you are capable of. Curiosity will serve you well on your journey to learn Ruby. There's so much to discover! True and false notice that after if and elsif we have to put an expression that evaluates to a Boolean value: true or false. Ruby, you could even write code like this: a = 5 if one puts how can it be true? more puts is not true end The output is how can it be true?. In Ruby, each expression is evaluated to true when used in flow control, except for false and nile. Try the above code and give a value of 0, (empty string) and even the string 'false' to see the result yourself! Because of this, we could even write code like this: if x = 5 puts how can it be true? more puts is not true end The above code is not testing if x equals 5. It is assigning the variable x the value of 5, which will always evaluate true. Unfortunately, that looks very similar to if x == 5, which is testing if x equals 5. Be careful when reading or writing Ruby; its expressiveness can also be a source of many subtle errors. Summary This chapter covered booleans, comparisons and the ability to control code execution flow conditionally. Here are some of the fundamental tools that you will bring with you as a Ruby developer. We have more exercises to pierce these skills in the head and fingers! Fingers!