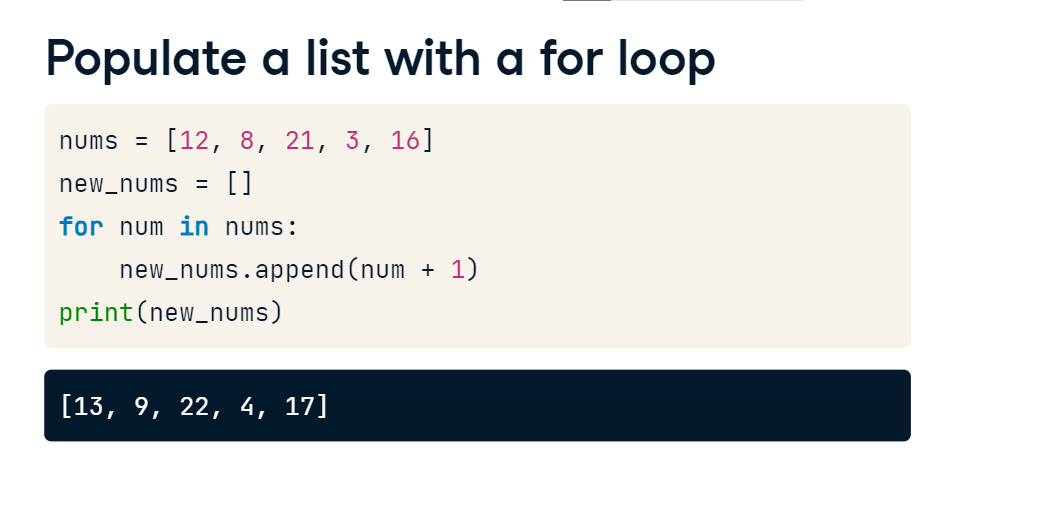
**List comprehensions**

Let's say that you have a list of numbers and you want to create a new list of numbers that's the same as the old list, except that each number has 1 added to it.

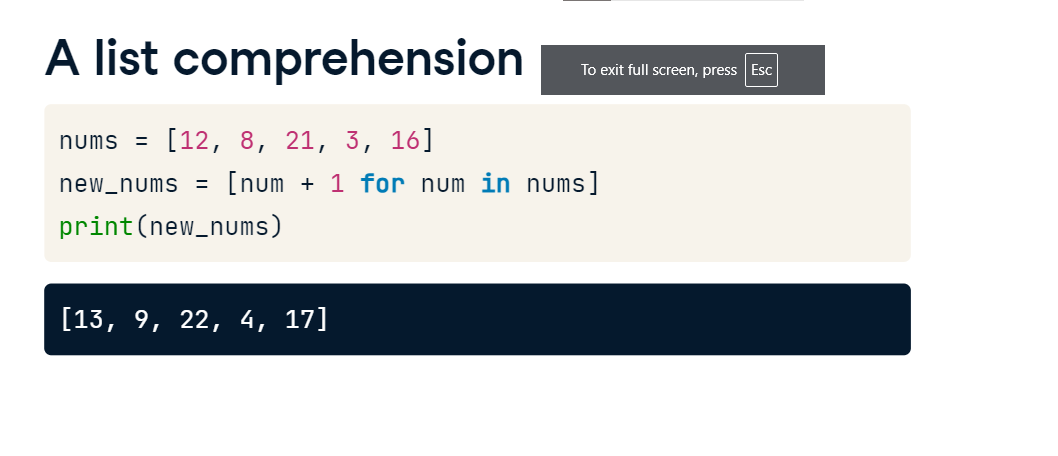
**Populate a list with a for loop**

You could initialize a new empty list, loop through the old list, add 1 to each entry and append all new values to the new list, but for loops are inefficient, both computationally and in terms of coding time and space, particularly when you could do this in one line of code. "One line of code?" I hear you asking.



**A list comprehension**

Welcome to the wonderful world of list comprehensions! The syntax is as follows: within square brackets, you write the values you wish to create, otherwise known as the output expression, followed by the for clause referencing the original list. So in our case, you open the square bracket, followed by num + 1 for num in nums and then you close the square bracket. This is a list comprehension and creates precisely the desired list!

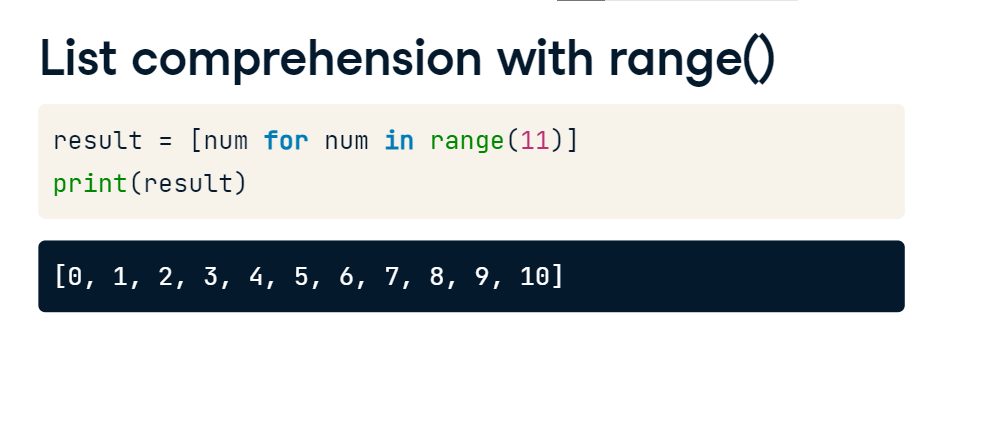


**For loop and list comprehension syntax**

See here the relationship between the for loop syntax and the list comprehension syntax. The power of list comprehensions is not merely relegated to the world of lists, however, you can write a list comprehension over any iterable.

**List comprehension with range()**

Here's an example of a list comprehension using a range object. To summarize,

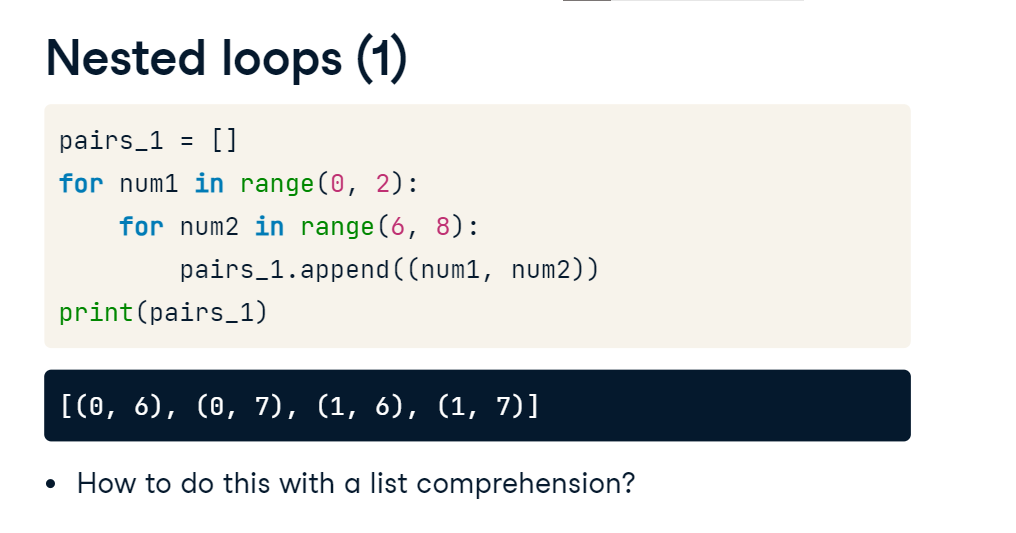


**List comprehensions**

list comprehensions collapse for loops for building lists into a single line and the required components are 1) an iterable, 2) an iterator variable that represents the members of the iterable and 3) an output expression. That's it. You can also use list comprehensions in place of nested for loops.

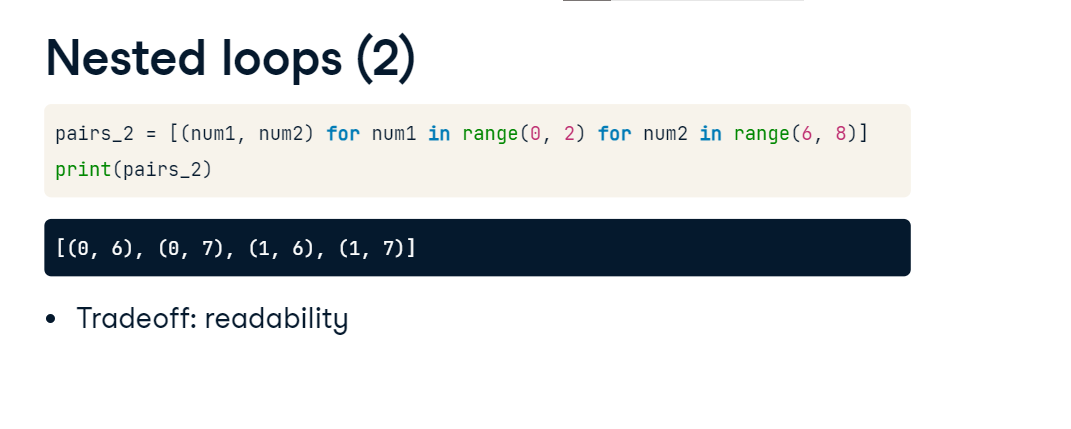
**Nested loops (1)**

For example, lets say that we wanted to create a list of all pairs of integers where the first integer is between 0 and 1 and the second between 6 and 7. This nested for loop would produce the required result. The question is, can we do the same with a list comprehension? And the answer is, yes, as follows.



**Nested loops (2)**

Once again, within the square brackets, place the desired output expression followed by the two required for loop clauses. You may observe that while it keeps to a single line of code, we sacrifice some readability of the code as a tradeoff, so you'll have to consider if you'd like to use list comprehensions in cases such as this. The more often you use this, the more you get used to reading list comprehensions, so readability may not be a problem for you later on. But do remember that others may have to read your code as well!

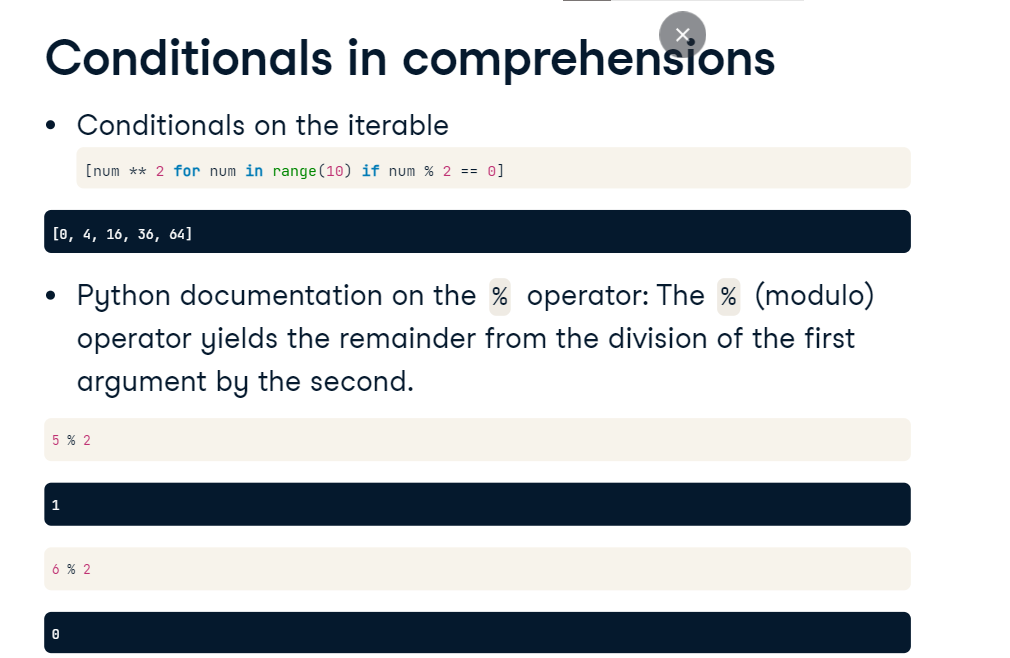


**Advanced comprehensions**

Now that you know the basics of list comprehensions, lets check out some more

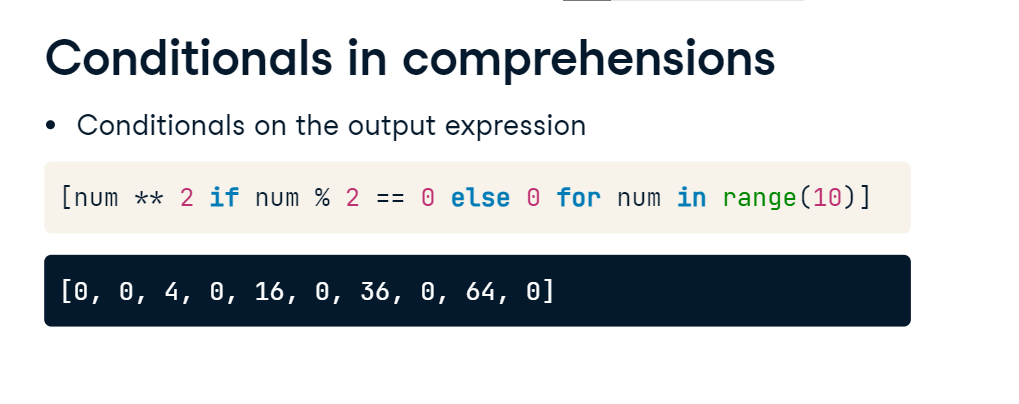
**Conditionals in comprehensions**

advanced comprehension capabilities, such as conditionals! Here we see that we can filter the output of a list comprehension using a conditional on the iterable: in this example, the resulting list is the square of the values in range(10) under the condition that the value itself is even. If you have not seen it before, the percent operation that you see being used in the comprehension is called the modulo operator. We can look at the Python documentation to see how the modulo operator is used and it shows that it produces the remainder from the division of the first argument by the second. Thus an integer modulo two is equal to zero if and only if the integer is even.



**Conditionals in comprehensions**

We can also condition the list comprehension on the output expression. Here, for an even integer we output its square. In any other case, signified by the else clause, that is for odd integers, we output 0.



**Dict comprehensions**

Now we can also write dictionary comprehensions to create new dictionaries from iterables. The syntax is almost the same as in list comprehensions and there are 2 differences. One, we use curly braces instead of square brackets. Two, the key and value are separated by a colon in the output expression as we can see here. In this example, we are creating a dictionary with keys positive integers and corresponding values the respective negative integers.

