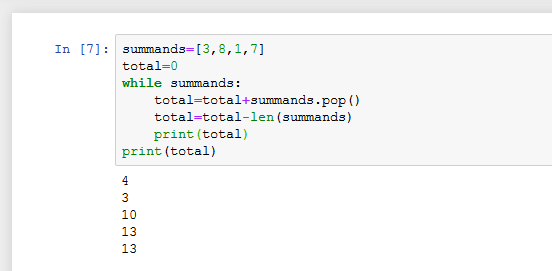
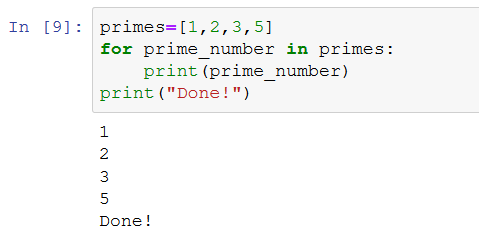
Control Flow in Python

* Expressions
  + An **expression** is any valid piece of code that evaluates to something. Notice that we can convert any expression to a particular data type. For example, let's convert some expressions to Booleans using the built-in bool() function:
    - bool(1 + 1) -> True # Any non-zero integer is True.
    - bool(0) -> False
    - bool('Hi' + ' there') -> True # Any non-empty string is True.
    - bool('') -> False
  + Experiment with other data types. Can you guess what evaluates to True/False for dicts, sets, and tuples?
* The If Statement
  + Here is an example of using if:
    - if answer.lower() == 'yes': # Line 1
    - print("You are given gold.") # Line 2
    - player\_gold = player\_gold + 10 # Line 3
    - print("You put away your bag.") # Line 4
  + This code is evaluated line by line, top to bottom. Note that lines 1–3 are all part of the same if code block. Line 4 comes after the code block.
  + Starting on Line 1, we evaluate the expression answer.lower() == 'yes'.
    - If this expression evaluates to True when converted to a bool, then the indented Lines 2 and 3 are evaluated. Now that we completed the if block, Line 4 is evaluated.
    - If the expression is False, we skip the if block and only evaluate Line 4. (It is the next line on the same level as Line 1.)
  + The if statement has a particular structure. Every if statement must begin the line with if, followed by any expression, followed by a colon, followed by one or more indented lines of code.
    - if <expression>:
    - <indented line of code>
    - ...
  + We evaluate the indented lines of code only if the expression (converted to a bool) is True. You can think of this as "If the expression is True, then evaluate the indented code."
  + **Pro tip: According to PEP 8, lines should be indented using four spaces**.
* The Else Condition
  + Sometimes, we want to evaluate one block of code if the expression is True and another if it's False. For example:
    - if username.lower() == 'coder1': # Line 1
    - print('Welcome back, Coder1!') # Line 2
    - else: # Line 3
    - print('Your username is unrecognized.') # Line 4
    - print('Thank you for using the system.') # Line 5
  + Here, we display a different message based on if the username is 'coder1'.
  + If the expression username.lower() == 'coder1' is True, then the first set of indented lines are evaluated (Line 2). If it's False, then the second set of indented lines are evaluated (Line 4).
  + In either case, Line 5 is at the same indentation level as Line 1. So, it's evaluated immediately after the if block.
* Playing With User Input: input()
  + A fun built-in Python function is input(). This function prompts the user for text input. Then, it returns the user's input as a string. For example:
    - name = input('What is your name? ')
  + Here, the user is prompted What is your name?. If the user enters Petunia, then name is set to the string "Petunia".
  + Here is a helpful way of reading code with function calls: Replace the entire function call (in this case, input('What is your name? ')) with its return value. Thinking about it this way, the statement becomes name = "Petunia".
* The While Statement
  + Here is an example of the while statement in action:
    - print('Type "yes" to continue.') # Line 1
    - # While the user does not enter 'yes', repeat!
    - while input('> ') != 'yes': # Line 2
    - print('Please type "yes" so we can exit this loop!') # Line 3
    - print('Thank you for typing "yes".') # Line 4
  + Let's evaluate the code line by line from top to bottom. Line 1 is evaluated, then we move to Line 2 — the while statement.
  + Think of the while statement as a repeated if statement. Just as with if, we must first evaluate the expression input('> ') != 'yes'.
    - If True, we evaluate the indented code. Afterward, we loop back to Line 2 and re-evaluate it.
    - If False, we jump to the line after the indented code, Line 4.
  + Let's walk through input('> ') != 'yes'. To perform the comparison, we first evaluate input('> '). So, the user is prompted with '> '. Suppose the user enters 'no'. Then, replacing the function call in its entirety with 'no', we get 'no' != 'yes'. This is True, as 'no' is not equal to 'yes'.
  + A while loop is generally used in Python when we don't know when the looping will stop, or when we don't know how many iterations the loop will require. In the last example, we don't know when the user will enter 'yes'. Therefore, using a while loop here is appropriate.
  + The while loop is in many ways the most basic looping construct. All other loops can be rewritten as a while loop.
* Looping Through a List Using While
  + Let's take a look at how we might print each element of a list using a while loop. Generally, in Python, this is done using a for loop (we'll see this next!). Let's walk through an example.
    - primes = [2, 3, 5, 7] # Line 1
    - i = 0 # Line 2
    - while i < len(primes): # Line 3
    - print(primes[i]) # Line 4
    - i = i + 1 # Line 5
    - print('Done!') # Line 6
  + Here, we are using the variable i as an index into the primes list.
  + First, Lines 1 and 2 are evaluated. Note that Lines 3–5 are the while code block. Once the while loop is finished, we'll evaluate Line 6.
    - When you use i as a variable name, it must always refer to an index. This is one of the few times you can use a short variable name, as it's so universally understood.
  + Now, we evaluate Line 3. Is i < len(primes)? Well, i is 0 and len(primes) is 4. So, i < len(primes) becomes 0 < 4, which is True. Hence, we execute Line 4 — the value of primes[0] is displayed (2). Now Line 5 is evaluated and i is incremented to 1.
  + Now, we evaluate Line 3 again. Is i < len(primes)? Now, i is 1 and len(primes) is still 4. So, i < len(primes) becomes 1 < 4, which is True. Hence, we execute Line 4 — the value of primes[1] is displayed (3). Now, Line 5 is evaluated and i is incremented to 2.
  + See if you can continue this line of thought until finally i is 4. At this point, i < len(primes) is evaluated to 4 < 4, which is False. So, we jump to the line after the code block, Line 6, and display Done!.
* Knowledge check
  + What will be printed on the screen after the following code is evaluated? Suppose that list.pop() removes the last (highest-index) element of the list and returns it.
    - summands = [3, 8, 1, 7] # Line 1
    - total = 0 # Line 2
    - while summands: # Line 3
    - total = total + summands.pop() # Line 4
    - total = total - len(summands) # Line 5
    - print(total) # Line 6
  + Answer:
    - 13
    - While summands is non-empty, the while loop continually adds the last element to total, removes the last element from summands, then subtracts the new list length from the total.
    - Let’s evaluate Line 3. Summands has 4 elements so it is non-empty and hence True. So we evaluate the indented lines.
    - To evaluate Line 4, we must first evaluate summands.pop(). This returns 7 and removes 7 from summands. So, Line 4 becomes total = 0 + 7. Because len(summands) is now 3, Line 5 becomes total = 7 – 3.
    - After this first iteration, now total is 4 and summands is [3, 8, 1].
    - Now go back and evaluate Line 3 again.
    - Continuing in this manner, we effectively add all of the elements of summands to total and subtract the lengths 3, 2, 1, and 0.
    - The final value of total is 13 (Make sure you run through each line of code just as the computer does!)
    - If you still find this question to be a bit tricky to understand, try copying the function into repl.it and placing a print total after Line 4 and Line 5. This will allow you to see how total changes through each operation
    - 
* Looping Through a List Using For
  + Let's see how easy it is to loop through the primes list from before:
    - primes = [1, 2, 3, 5] # Line 1
    - for prime\_number in primes: # Line 2
    - print(prime\_number) # Line 3
    - print('Done!') # Line 4
  + In this example, prime\_number is an arbitrarily chosen name — it could just as easily be prime, or even x. Each time we loop, this variable references the next element in primes until no elements are left.
  + Let's see how this works. When Line 2 is evaluated, prime\_number is set to the first element of primes, primes[0]. Effectively, it sets prime\_number = primes[0]. Because Line 3 is indented, it's evaluated next and prime\_number (i.e., primes[0]) is displayed.
  + Now, we loop back to Line 2. Here, prime\_number is set to the next element of primes, primes[1]. On Line 3, we display prime\_number again.
  + This continues until we loop back to Line 2 and we've iterated through each element in primes. At this point, the for loop is done and we jump to the next line after the code block, Line 4.
  + 
* The For Statement
  + The for statement always requires the following structure:
    - for <variable name> in <sequence>:
    - <indented line of code>
    - ...
    - The variable name can be anything. However, don't write over any of your other variable names.
    - The sequence is any object that can be iterated over. For example: lists, strings, tuples, dictionaries (the keys), and sets (though in no particular order).
  + **Reminder: Notice how a colon always indicates that at least the next line of code will be indented.**
  + Let's look at another example:
    - # Sum N = 0,..,5
    - total = 0 # Line 1
    - for num in [0, 1, 2, 3, 4, 5]: # Line 2
    - total += num # Line 3
    - print(total) # Line 4
  + This sums the integers in the list. First, total is 0. In Line 2, num is set to 0, the first element in the list. In Line 3, total = total + num, so total = 0 + 0.
  + At the end of the code block, total is 0. We loop back to Line 2. Now, num is set to the second element, 1. In Line 3, total = total + num, so total = 0 + 1.
  + Finally, we see that total is 15, as expected!
  + **Pro Tip: Python's built-in function sum([0, 1, 2, 3, 4, 5]) accomplishes the same objective as the for loop above.**
* The Range Function
  + It was a bit of a time suck writing out the integers from 1 to 5, wasn't it? What if we wanted to sum the first 100 integers? That would be a hassle.
  + Good news! Python provides us with a built-in function that does this called range().
    - range(N) returns a generator. To create a list from a generator, use list(range(N))
    - For example, list(range(100)) returns a list of integers from 0 up to (but not including) 100. (For a total of 100 - 0 = 100 elements.)
    - list(range(11,100)) returns a list of integers from 11 up to (but not including) 100. (For a total of 100 - 11 = 89 elements.)
  + **In Python 2, range(N) actually generates a list. In Python 3, range(N) returns a generator. To create a list from a generator, just use list() (e.g. list(xrange(N))).**
* Using For With Range
  + Let's rewrite our earlier example to use range().
    - # Sum the first 100 integers
    - total = 0
    - for num in range(101):
    - total += num
    - print(total)
  + Remember, range(N) creates a generator of integers from 0 up to but not including N. So, to sum the first 100 integers, we must set N to 101.