# Python Morse Code Lab

**Introduction**

This lab is designed to give you further practice with Python programming using functions and lists. It also builds upon your knowledge of fundamental control structures and variables.

In the first part of the lab, you will create a program that continually translates simple messages into Morse Code. You will then recode your solution using a list and a more generalized custom function.

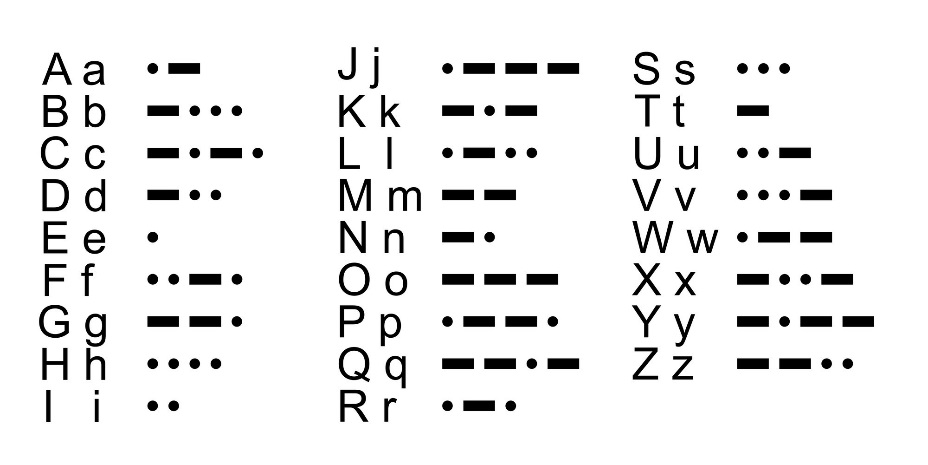
**Preliminaries**

1. Ensure you have PyCharm and Python installed. *This should have been done in previous modules*. If necessary, PyCharm and Anaconda (a convenient Python 3.8 distribution) can be obtained from the following links:
   1. Pycharm: <https://www.jetbrains.com/pycharm/download/#section=windows>
   2. Python (Anaconda): <https://www.anaconda.com/distribution/#windows>
2. In PyCharm, create a new project named PythonMorseCode-Lab-<YOUR NAME>.
3. Download the starter Python file, MorseCodeLab.py, from (canvas link needed)
4. Take a few minutes to examine the starter code:
   1. *How does the loop in the Python file function?*
   2. *How do we stop entering messages to be converted into Morse Code?*

**Challenge 1. Create functions for four Morse Code characters**

Each letter of the Morse Code alphabet can be represented by a pattern of dots and dashes. For example, the letter 'a' can be written as '. –'. Dots and dashes also correspond to short and long sounds. Here is a table showing the patterns of dots and dashes for all the letters of the alphabet.

You will begin by creating functions for four Morse Code letters – a, e, o, and t.



1. In the MorseCodeLab.py starter, define four functions based on the following pseudocode. The functions should be defined *above* the while Loop on line 5.

function **morse\_a**  
create the pattern for ‘a’ in Morse Code  
return that pattern

function **morse\_e**  
create the pattern for ‘a’ in Morse Code  
return that pattern

function **morse\_o**  
create the pattern for ‘a’ in Morse Code  
return that pattern

function **morse\_t**create the pattern for ‘a’ in Morse Code  
return that pattern

2. Test your functions by calling a few of them – at this stage, it does not matter if you call them in the loop or not. You will need to *print* the result of the function calls in order to see the result. At this stage, it won’t be particularly exciting. Below is the result of printing **morse\_a**, **morse\_e**, and **morse\_o** + **morse\_t** – notice anything *wrong*?



3. Define two new functions to create **and print** the following three patterns. Test them in your code, above the while loop.

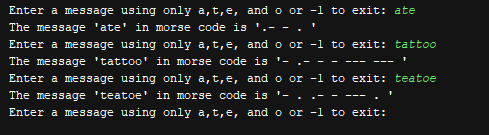
**.-.**

**--.--.**

**-.. --- -. .**

**Challenge 2. Convert a whole message to Morse Code.**

Now that you have custom blocks to generate patterns for these four letters, they can be used to convert a whole message into Morse code.

1. Define a new function, **translate\_to\_mc(message)**. This function will take a string message and return the string converted to Morse Code.
2. Translate the message
   1. Create an empty string, mc\_message
   2. Using a for loop, loop through each letter in letters – you can loop directly through the message like it is a list
   3. For each letter, convert it into Morse Code and add it to mc\_message. You will need a series of four if statements to handle each letter (a, e, o, t)
3. Don’t forget to return the translated message!
4. Test your code by running it and entering the following messages – you will need to add a function call for **translate\_to\_mc** to your code!

ate

tattoo

teatoe

**Challenge 3. Using a list to support easier conversion to Morse code.**

After completing Challenge 3, your code can generate Morse code, but only for messages containing the letters a, e, o and t. *What changes would you need to add another letter, say s? How about the whole alphabet?*

The simplest way to do this would be to add a custom function for every letter, 26 in all, and a commensurate number of if statements to **translate\_to\_mc** to check for each possible letter. This would result in a program with lots of little custom functions and one big function with lots of decision logic. In general, this makes a program hard to manage.

Another approach is to use a list. Actually, we will need two lists, one for the alphabet and one for the Morse code patterns. Let's consider just the four letters we've been using. The lists might look like this:

|  |  |
| --- | --- |
|  |  |

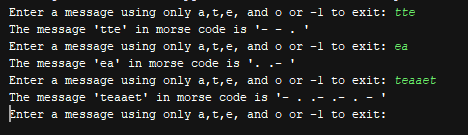
The lists are related in an important way. Notice that the Morse code for the first letter of **alphabet** is found in the first position of **patterns**. The same is true for the second letter and so on. We can formally characterize this relationship as follows: For any position, **pos**, the Morse code for **alphabet[pos]** is found at **patterns[pos]**. Any lists that exhibit this kind of relationship are called **parallel**.

1. Make a list named **alphabet** – you can put this list either as the first line of **translate\_to\_mc** *or* **above** all the functions.
2. Manually add the letters a, e, o and t to alphabet.
3. Make a list named **patterns** – add it just below **alphabet**.
4. Manually add the patterns .-, ., ---, and – to patterns.

**Challenge 4. Update translate\_to\_mc to use the lists**

We can get rid of the **morse** functions we wrote and use the lists **alphabets** and **patterns** instead. This refactoring will maintain the behavior of **translate\_to\_mc** while cutting down on how many functions we need.

1. Inside of the for loop through the letters…
2. Use alphabets.index() in order to get the *position* of the current letter in the alphabet. *What will go in index()*?
3. Using the position you obtain from index, add the appropriate pattern to the morse code string. *How do you get this pattern from patterns?*
4. As last time, do not forget to return the Morse Code string!
5. Add the following blocks to doPattern. These blocks implement the logic needed to examine the string, **pattern**, one character at a time. The variable, **ch**, holds this character.
6. Test your code again with the following messages:



tte

ea

teaaet

**Challenge 5. Update translate\_to\_mc to use handle the whole alphabet**

Now that we have a robust solution to handle a, e, o, and t using lists, why not go further? Let’s support all *26* letters by updating alphabet and patterns. Use the test cases below to test your implementation:

* dog => -.. --- --.
* hippo => .... .. .--. .--. ---
* zoolander => --.. --- --- .-.. .- -. -.. . .-.

After this revision, your program can handle any one word Morse Code translation!

**Spicy Challenges**

1. Revise the program to handle both uppercase and lowercase letters. There are multiple ways to accomplish this
2. When working with long lists that have predefined content, it is sometimes easier to import this content from an external file. One convenient way to store data is in comma separated value (CSV) form. Research how to read CSV files in Python and use a csv file to populate **alphabet** and **patterns** instead of filling the lists out manually.
3. Text to Morse Code translators use the backslash, /, to split up words. Change your program to allow for multiple words to be translated, not just a single word message. Here’s a hint – given a whole sentence in Python stored in the variable **sentence**, *sentence.split()* gives you a list of words in that sentence by splitting the string up using whitespace characters.
4. Revise the program to also translate a Morse code message into a string of letters.