# Week 4 Assignment

Design, build and test the following functions: *{updated 19+22 October 2021}*  
Submit your python script with the filename ‘w4\_yourID.py’.

## signum(a\_num)

If a\_num is greater than zero return 1. If it’s less than zero return integer minus one. Otherwise, return integer zero.  
For example:

| parameter values | return value |
| --- | --- |
| 11 | 1 |
| 0.000702 | 1 |
| -1/2 | -1 |
| 0 | 0 |

**Hints:**  
A few if statements should get the job done.

## middle(p1, p2, p3)

Return the middle value.  
All three parameters are guaranteed to be pairwise comparable using the operators <, <= !=, ==, >=, and >.  
For example:

| parameter values | return value |
| --- | --- |
| 1 , 2 , 3 | 2 |
| 2 , 3 , 1 | 2 |
| 'a' , 'x' , 'c' | 'c' |
| 'abra' , 'cad' , 'abra' | 'abra' |

**Hints:**  
Use if statements with the logical operators and, or and not.  
Alternatively, you can solve this problem with a list and list methods - but I’ll let you discover that method for yourselves.

## isa\_triangle(len1, len2, len3)

The three parameters are guaranteed to be numbers. If they are interpreted as the lengths of the sides of a euclidean triangle, return an integer to indicate which type of triangle they would construct according to this table:

| Return Value | Meaning |
| --- | --- |
| **0** | No triangle possible |
| **1** | Scalene |
| **2** | Isosceles |
| **3** | Equilateral |

**Hints:**  
Triangles have three sides. If all are equal lengths then it’s an equilateral. If only two are equal it’s an isosceles triangle. If all sides are different it’s a scalene triangle.  
As with the first problem, if / elif / else statements combined with the boolean operators and, or and not should do the job.  
Do remember though that there are four possibilities. For example what should isa\_triangle(2,17,2) return? If you can’t see why that is a problematic triangle try sketching it to scale on a piece of paper.

## robberlingo(a\_str)

Translate the text string a\_str text into “rövarspråket” (Swedish for “robber’s language”). That is, double every consonant and place an occurrence of “o” in between. For example, translate(“this is fun”) should return the string “tothohisos isos fofunon”

**Hints:**  
A consonant is any character in the alphabet a..z that is not a vowel. A vowel is any one of 'aeiou'. Create both of those as string variables. Might only need one, we can delete the other later.

The problem asks for a string containing the translation so start by creating an empty string to contain the translation.

The question asks us to read each character of a\_str in turn and act differently if the current\_char is a consonant or not.

* Reading each character in turn is iteration and which means we need a while or a for loop.
* If the current\_character in a\_str is in consonants then add current\_character + 'o' + current\_character to the end of the translation so far
* If not, just add the current\_character.

The keyword operator in can be used to test if a character is in a string. Try 'a' in 'syntax' to see how it works.

Here is some skeleton code you can build on:

a\_string = 'abracadabra' # this will be provided by the function parameter  
vowels = 'aeioi'  
consonants = 'bcdfghjklmnpqrstvwxyz' # check this is correct (I think it is)  
translation = []  
for current\_char in a\_string:  
 # change this to the code that will build the translated string  
 if current\_char in consonants:  
 print('add translated character to translation')  
 else:  
 print('just add character to translation')

When the for loop exits return the translated string.

## pangram(a\_str):

Answer true if a\_str is a pangram and false if not. Ignore capitalisation and punctuation, only consider the characters a-z.  
For example:

| parameter values | return value |
| --- | --- |
| 'the quick brown fox jumps over the lazy dog' | True |
| 'the quick brown fox jumped over the lazy dog' | False |

**Hints:**  
A pangram is a phrase that contains all the letters of the alphabet from ‘a’ to ‘z’ at least once. Other characters in the phrase are effectively ignored. As long as a\_str contains all the characters of the alphabet, it’s a pangram.

The question says to ignore capitalisation so a good first step is to convert str\_1 to lower case with the lower() or casefold() string methods described at [Python.org Text Sequence Type - str](https://docs.python.org/3/library/stdtypes.html#str) and at [w3schools](https://www.w3schools.com/python/python_strings_methods.asp).

As repetitions make no difference to a phrase being a pangram or not, and sets ignore repetitions, they are probably going to be useful in solving this problem. Logically, if all the characters in a\_str are placed in a set and that set contains every letter in the alphabet, then a\_str is a pangram. That sounds like a way to solve the problem so let’s try it.

Start by creating two sets:

* A set containing all the characters in a\_str.
* A set containing all the letters of the alphabet 'abc ... xyz'.

To explore how to create these sets start IDLE and execute the statement set('foobar') and see what you get.

The next step is to remove every character in set of characters in a\_str from the alphabet set and check how many characters are left. If the answer is none then a\_str must contain every letter in the alphabet and hence is a pangram.

Problem solved - provided sets can be ‘subtracted’ and there is a way of determining if a set is empty.

Subtracting one set from another is one of the [set methods and operations](https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset). Scan through the list of methods and operations available and you should soon spot how to do it. Alternatively, just try subtracting one set from another in IDLE and see what the result is.

The built-in function len() returns the number of elements in a collection. It works on sets just as well as it does strings.

## merge2(str1, str2)

Interleave the successive characters of **two** strings of the same length into a single string and return the result.  
For example:

| parameter values | return value |
| --- | --- |
| '' , '' | '' |
| 'a' , 'x' | 'ax' |
| 'abc' , 'def' | 'adbecf' |

**Hints:**  
Strings can be indexed: s[0] is the first character, s[1] the second, s[2] the third and so on. Extending that logic to two lists is easy: str\_1[0] and str\_2[0], then str\_1[1] and str\_2[1], then str\_1[2] and str\_2[2], and so on.

The key to this problem is instead of iterating through the successive characters of the strings, iterate through the index numbers that can be used to access the characters of the two strings. As indexing applies to all strings these integer index values will enable synchronised access to BOTH strings at the same time.

How best to generate that sequence of index numbers? We could use a while loop with a counter but far easier is to use a range() object and combine that with a for loop. Here is the beginnings of some skeleton code. Execute it in IDLE to understand what it is doing:

for index in range(11):  
 print(index, index)

We are now in a position to solve the problem:

* Replace the 11 with the length of the strings by using len(str\_1) (either will do as they are both the same length)
* Create a variable to contain the result string result\_str = ''
* On every iteration of the for loop add the indexed characters from both strings to the end of result\_str

Here my outline for you to build upon.

str\_1 = 'abc' # dummy value. This will be parameter value  
str\_2 = 'xyz' # dummy value. This will be parameter value  
length\_of\_str = len(str\_1) # either will do as they are guaranteed to be same length  
result\_str = ''  
for index in range(length\_of\_str):  
 print(str\_1[index] + str\_2[index]) # change this to code to modify result\_str

Replace the print() statement with your code to add the two characters from str\_1 and str\_2 to result\_str, wrap it in a function, remove my two dummy strings and use parameters instead, test the final code, and done.

## merge3(str1, str2, str3)

Interleave the successive characters of **three** strings of the same length into a single string and return the result.  
For example:

| parameter values | return value |
| --- | --- |
| '', '', '' | '' |
| 'a', 'b', 'c' | 'abc' |
| 'abc', 'def', 'ghj' | 'adgbehcfj' |

**Hints:**  
Same logic and design as for merge2(). The only difference is that this time there are now three strings instead of two.

## letter\_count(a\_str)

Return a dictionary with every character (including spaces and punctuation) in a\_str as a key in the dictionary with a value equal to the number of occurrences of that character in a\_str. Only characters in a\_str are keys in the returned dictionary.

For example:

| parameter values | return value |
| --- | --- |
| '' | {} |
| 'a' | {'a' : 1} |
| 'aaa' | {'a' : 3} |
| 'abbabab' | {'a' : 3, 'b': 4} |
| 'abracadabra' | {'a' : 5, 'b': 2, 'c': c, 'd': c} |

**Hints:**  
First create a dictionary to contain the result. There are several ways to do this but the easiest options are probably d = dict() which creates an empty dictionary and d = dict.fromkeys('abcd', 0) which creates a dictionary with a *key* for every character in 'abcd' associated with a *value* of 0. Try both in IDLE to explore what they do and decide which is best for this problem.

Now that the dictionary is created, we need to iterate through a\_str and update the values associated with each letter. A for loop is ideal for this. For each of the characters look up its entry in the dictionary and increment its value by 1. Something like the code I have outlined below should do the job:

a\_str = 'abracadabra' # parameter value  
char\_count\_dict = dict.fromkeys(a\_str)  
for char in a\_str:  
 # get current value in char\_count\_dict for char  
 # char\_count\_dict[char]  
 # and increment its value by 1  
 # char\_count\_dict[char] = previous value + 1  
 pass # dummy statement just to make my syntax correct  
# and that's it so just return the dictionary char\_count

Find the dictionary methods you need at [Python.org dictionary](https://docs.python.org/3/library/stdtypes.html#mapping-types-dict) or on [W3Schools.com dictionaries](https://www.w3schools.com/python/python_dictionaries_change.asp).

# Too difficult for now

These last questions are not assessed. If you attempt them I will mark your work, but you suffer no penalty for not doing them.

## runup(a\_str)

The parameter a\_str is guaranteed to be a string made entirely of alphanumeric [a…z][0…9] characters. Answer the starting position and length of the longest non-decreasing substring within a\_str. If there are multiple such substrings, report the first. If a\_str is empty return ( -1, 0).  
For example:

| parameter values | return value | substring |
| --- | --- | --- |
| '' | -1, 0 | '' |
| 'z' | 0, 1 | 'z' |
| 'ababcb' | 2, 3 | 'abc' |
| 'bus27herenow' | 1, 4 | 'us27' |
| 'bus21herenow' | 8, 4 | 'enow' |
| '27991' | 0, 4 | '2799' |

**Hints:**  
*There are many ways to solve this problem. This is just one.*

It is often very helpful when trying to solve these more difficult problems to first try and think how it might be solved with ‘paper & pencil’. Doing that ‘paper & pencil’ exercise (yes I actually did do that) shows that just one iteration through the list is all that is needed but there are several things to keep track of at the same time which means it can be tricky keeping everything synchronised.

Starting from the assumption that it would be a for loop, the items to keep track of are:

| Variable | meaning |
| --- | --- |
| current\_char | the character currently being evaluated in this iteration of the for loop |
| previous\_char | the immediately preceding character in a\_str |
| current\_longest | the characters in this current non-decreasing run |
| previous\_longest | the longest sequence of non-decreasing characters found so far |

To start with these are all empty so create them and make them all equal to the empty string ''.

Next, iterate through a\_str with a for loop. Something like this:

a\_str = 'abracadabra' # in the function this will be the parameter value  
current\_char = ''  
previous\_char = ''  
current\_longest = ''  
previous\_longest = ''  
for current\_char in a\_str:  
 ## do things  
 print(current\_char) # just a placeholder

Inside the for loop we need an if statement to cope with two alternative situations:

1. The current non-descending sequence has ended because current\_char < previous\_char.Set the current\_longest to just the current\_char
2. The current non-decreasing sequence is still OK because current\_char >= previous\_char. The current run is still good so add the current\_char to the end of the current\_longest. Compare the current\_longest to the previous\_longest and if it’s longer make it the previous\_longest.

After the if statement assign current\_char to previous\_char because it needs to be available for the next iteration.

When the for loop finishes the longest run will be held in the variable previous\_longest.

The question asks for the starting position of the longest run. We could do that by iterating through a\_str again to look for it, but there is an easier way - use the string method find(). The question also asks for the length of the longest run - but that’s easy, just use len().

result = a\_str.find(previous\_longest), len(previous\_longest)

This code works for all inputs EXCEPT for the empty string. To fix that add an if statement to check:

if a\_str == '': # parameter value is empty string  
 result = (-1, 0)

Finally, add return result statement to function

## mergen\_short()

Interleave the successive characters of **any number** of strings of **any length** into a single string and return the result.  
Interleaving is halted when the shortest string is exhausted.  
For example:

| parameter values | return value |
| --- | --- |
| '', '', '' | '' |
| 'abc', 'd', 'ef' | 'ade' |
| 'abc\_', 'defg', 'hjkmn', 'pqr', 'stuv' | 'adhpsbejqtcfkru' |

**Hints:**  
The problem has many similarities to the previous mergeX() problems but this time there can be any number of parameter strings, and they can all be different lengths. So some immediate problems to solve are:

1. How do functions that take any number of parameters work?
2. How to stop the iteration when the shortest string is exhausted?
3. How to merge an arbitrary number of strings?

Functions that take an arbitrary number of argument parameters are described in the Python tutorial at [4.8.4. Arbitrary Argument Lists](https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists) and at [w3schools](https://www.w3schools.com/python/python_functions.asp) (a little way down the page). The short explanation is that the function definition should look like:

def f\_name(\*args):  
 # code  
 return

The strings will then be available within the function body as the elements of a list called args. So this function will have an outline structure like this:

def merge\_short(\*args): # 'args' is traditional, it can be any variable name  
 # args is now available as a variable within the function body  
 # args is a list of the strings to be interleaved  
 result\_str = [] # destination for result  
 # code goes here . . .   
 return result\_str

For part (2) we need to find the shortest string in the list of strings in args. There are many ways to do this but perhaps the simplest is to create another list and put all the lengths of the strings in that. Then we can use the built-in function min() to evaluate min(list\_of\_lengths). Some code along the lines of this should do the job:

args = ['1st str', 'next str', 'one after that'] # this is just an example, args is automatically created   
list\_of\_lengths = [] # the list of string lengths will go here  
for each\_string in args:  
 list\_of\_lengths.append(len(each\_string))  
# all the lengths of all the strings in args are now in list\_of\_lengths  
shortest\_string\_length = min(list\_of\_lengths)

The answer to part (3) is that as all the strings are in a single list, we can use a for loop with range() to iterate through it, adding one character from each successive string as the list is traversed. Something like:

result = []  
for index in range(shortest\_string\_length):  
 for each\_str in args:  
 result.append(each\_str[index])  
# for loops ended. Assemble list of characters in result into a single string  
result = ''join(result)  
# and return the result

## mergen\_long()

Interleave the successive characters of **any number** of strings of **any length** into a single string and return the result.  
As individual strings become exhausted, continue interleaving with the remaining strings until all are exhausted.  
For example:

| parameter values | return value |
| --- | --- |
| '', '', '' | '' |
| 'abc', 'd', 'ef' | 'adebfc' |
| 'abc\_', 'defg', 'hjkmn', 'pqr', 'stuv' | 'adhpsbejqtcfkru\_gmvn' |

**Hints:**  
Much of the structure of the function mergen\_long()is similar to mergen\_short() but with an additional check before a character is added to the result.  
The value of the variable index used to access elements of the strings will start at zero and successively increment to the last element of the longest string. Because some strings may be shorter than the longest string, we need to check that [index] is not beyond the end of the string before attempting to access any string.