The following coding activity is about Recursion. To understand recursion, it can be imagined a task that requires for several items to be picked up from a line of houses along a route – each house hosts an item. Furthermore, it can be imagined that the further the house is, the lighter the item is. A sensible approach would be to first walk the whole distance until reaching the final destination and from there, upon return, stop at each house and fetch an item in turn, thus making the task easier. In programming, the reason why we may choose to carry out a task this way is not necessarily because of an equivalent 'weight problem' in software but because of other concerns whose details we will not discuss here. The following three examples are two way of how we might specify a sum function. The first example is a typical, traditional iterative way of carrying out a sum over a list of integers and the other two are recursive examples. The programming language used in this example, is Python (indentation sensitive).

In both recursive cases, an item (integer) is picked up at every entry\index of a list and then added together with the sum calculated so far but whilst in the first example the addition is started only after we arrive at the final entry of the list or in our 'story' after final house is reached, in the second example, although the addition is started, as soon as the first destination is reached, the addition is not yet complete and so therefore a bag is carried along where the intermediate unfinished total is stored until after the last destination upon which the total sum is returned back the entire path. One might then ask, what is the achievement of this? To illustrate this let us consider a scenario where a waiter needs to clear a long table from food leftovers. One thing for sure is obvious that the way to go about it is instead of carrying a stack of many individual plates with bones and leftovers, that as the waiter carries a big bowl with them and as he goes past each seat, he consolidates the contents of all the plates from the table into the big bowl, thereby saving himself to having to carry lots of individual and unevenly sitting plates, perhaps with knives and spoons in between them.

But what may not be so obvious: Let's consider a case where whilst the waiter begins at this end of the table, guests at the other far end of the table have not yet finished eating but by the time he'll reach their places, they will have finished. It is then sensible, unlike before, to begin clearing up in the normal order of things, that is, from the **first** destination that he reaches so that by the time he gets to other end, the guests sitting there also ready for their plates to be removed. Our second example of recursion follows this approach. We carry an accumulator variable with which we consolidate the values into one big total as we arrive to each destination. When we reach the last seat, we send back the full and complete bowl with meat remnants to be put into the rubbish bin.

```
Houses = [1, 2, 3, 4]

def IterativeSum(i):



  Total = 0
  while (i<4):
    Total += Houses[i]
    i = i+1
  return Total;
```

Houses is a list with 4 integer items.

def, short for 'define' is how a function is declared in Python. IterativeSum is the function 'name' and *i* is a variable that will receive an integer value. We will use this value to keep track of where, at which entry, in the list we are – here from 0 – 3. (The first item lies at index 0. Most programming languages start indexing at 0.)

An empty variable 'Total' is initialized.
An ordinary *while* loop in which each integer at every index of the list is picked up in turn, from 0 to 3 and *i* is subsequently incremented. Each time before the while loop is entered, *i* will be checked to make sure it is still

```
print(IterativeSum(0))


def SumRecursive(i):



if (i==0):
   return Houses[i];









else:
#the sum of Houses[0],...,Houses[i]
#is the sum of elements Houses[0],..., Houses[i-1] and
#Houses[i]
    return SumRecursive(i-1)+Houses[i];
#the computation of Houses[0]+...+Houses[n-1]
```

below 4. Otherwise, an 'out of boundary exception' will be thrown and the code will break. When it reaches 4, the loop is terminated, and the *total* is returned.

If is a condition keyword. If *i* is 0, then the first index of the list was reached, and the content of the first item is returned. Here, the content of a given index is extracted by pointing *i* which is the value of a given index, onto that list index from which we would like to get the value. This line is also what is called 'the base condition' because it will get executed at the last recursive call and cause the function to 'unwind' from its nested recursive calls.

(# is the beginning of a 'comment line')
Else, if the end [beginning] of the list has not been reached:

We are calling the function, that is *itself* again, passing it an index number - one less than the current one. This is the equivalent of a person in our first story, making their way to the end of all the houses, stopping at each house and checking if it is the last house or not?) The second half of this line, i.e. after and including the addition sign, will not get executed until the 'base condition is executed and the content of the first index of the list is returned. The first execution of this, second part of this line, will then be by the *'one before the last call*' that was made which will add Houses[0] + Houses[1]. He will then send this *Total* to the caller that called him, i.e. the *third before the last call,* which will add this with Houses[3] and so on…

```
Total = SumRecursive(len(Houses)-1);
print(Total)


def SumAccum(i, acc):

if (i==0):
    return acc + Houses[i];
  else:
```

The function is called the first time, from the outside world. The length of the list (- 1 as indexing starts from 0, as explained above).

In this example, an accumulator is passed along with the index counter.

If i is 0, that is we have reached the first element in the list, acc which will now have stored in it, all elements

```
return SumAccum(i-1, Houses[i]+acc);
```

added but the first element, will be added to the first element also, and returned.
Else, we call the function recursively, and pass it the current index – 1 as well as the content of the current index plus the accumulated total so far.

```
Total = SumAccum(len(Houses)-1,0);
print(Total);
```

We call this function, for the first time, as before but with the *accumulator* first being equal to 0.

```
ispalindrome1 =  "Able was I ere I saw Elba";
ispalindrome2 = "racecar"
isntpalindrome = "raccar"
ispalindrome = ispalindrome1.replace(" ","").lower();
```

This function is an example of a real problem in where employing recursion seems natural more so than perhaps the iterative approach.
Here we format the text\string, so that we get rid of all white spaces and turn everything to the same case so that an *A* and an *a* are the same letter.

```
def CheckPalindrome(scentence):
  if (len(scentence)>1 and scentence[0] == scentence[-
1]):
    return CheckPalindrome(scentence[1:-1]);
  elif len(scentence)==1:
    return 1;
  else:
    return 0
```

The way we are going to solve this problem is by continuously (at every function call) checking if the first and the last character of the entire string (formatted sentence) are the same. If they are, we chop them both off, and pass the shortened string to a newly recursive call and so on.. To deal with the anomaly of when the string is of odd characters but still a palindrome, we will include a check to see if the string 'coming in' contains more than one character and we will short circuit it with an 'and' operator so that an 'out of boundary exception' isn't thrown when the comparison is attempted. If the length of the string is 1, we know it's a palindrome of odd characters. A string of one character is a palindrome by default.

```
print(CheckPalindrome(ispalindrome));

ispalindrome = ispalindrome2.replace(" ","").lower();
print(CheckPalindrome(ispalindrome));
print(CheckPalindrome(isntpalindrome));
```

The CheckPlindrome function is called for the first time.