# W5.2 Applied

## Re-implement Built-in Functions

In this activity, you will re-implement some of the built-in functions of Python. When implementing each function, please use the appropriate type annotation. You cannot use the Python built-in function you are implementing within its own implementation. In the absence of additional restrictions, you can use other Python built-in functions and methods.

# Re-implement abs()

Write a function `my_abs(num)` that models a limited version of the behaviour of the built-in function (it is not necessary to deal with complex numbers).

**Input:** a float `num`.

**Output:** a float that is the absolute value of `num`.

## Re-implement sum()

Write a function `my_sum(lst)` that models a limited version of the behaviour of the built-in function.

**Input:** a list of numbers `lst`.

**Output:** the sum (of type float) of all numbers in `lst`.

## Re-implement is_in()

Write a function `my_is_in(char, string)` that models a limited version of the behaviour of the `in` keyword (for more details, refer to the Python docs). You **cannot** use `char in string` in your function, but you **can use a for loop** which is a different use of the `in` keyword.

**Input:** a string `char` of a single character (you may limit to alphanumeric), and a string `string` comprising both upper and lower case alphanumeric and non-alphanumeric characters.

**Output:** a Boolean, `True` if `char` is found in `string`; otherwise `False`.

ℹ Note that Python’s `in` keyword is more powerful than the function `my_is_in()` in that it is able to test whether a string appears as a substring of another string. You are asked here only to implement the simpler function that tests if a single character appears in a string.

## Check if a list is sorted

Write a function `my_is_sorted(lst)` that checks whether a given list of sortable elements is sorted from smallest to largest.

**Input:** a list `lst` of comparable elements.

**Output:** a Boolean, `True` if for all items in the list `lst`, the item at index `i` is less than or equal to the item at index `i+1`; otherwise `False`.

## Re-implement any()

Write a function `my_any(lst)` that models a limited version of the behaviour of the built-in function `any()`. You may use the built-in function `bool()`.

**Input:** a list of objects `lst`.

**Output:** a Boolean: `True` if at least one object in `lst` has a truth value of `True`; otherwise `False`.

Think about the behaviour of the function for the empty list as input.

## Re-implement enumerate()

Write a function `my_enumerate(lst)` that models a simplified version of the behaviour of the built-in function. In Python, `enumerate()` returns an enumerate object – enumerate is a specific type that is different to the list type.

**Input:** a list `lst`.

**Output:** a list of tuples, where each tuple is of the form `(i, element)`; `i` is the index of the corresponding element within `lst`.

# Magic Square

A magic square is a table with $n$ rows and $n$ columns such that each square is filled with distinct positive integers $i$ where $1 \leq i \leq n^2$, and the sum of the integers in each row, in each column, and the two main diagonals are equal.

Here is a $5 \times 5$ magic square where each row, column and main diagonal add to 65.

$$\begin{array}{ccccccccc}
 & & 17 & 24 & 1 & 8 & 15 & \cdots & 65 \\
 & & 23 & 5 & 7 & 14 & 16 & \cdots & 65 \\
 & & 4 & 6 & 13 & 20 & 22 & \cdots & 65 \\
 & & 10 & 12 & 19 & 21 & 3 & \cdots & 65 \\
 & & 11 & 18 & 25 & 2 & 9 & \cdots & 65 \\
 & \iddots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\
65 & & 65 & 65 & 65 & 65 & 65 & & 65
\end{array}$$

Write a function named `is_magic` which takes as input a magic square in the form of a list of lists (where each row is a single list, and the table is a list of rows) and returns `True` if the table is a magic square and `False` otherwise. Assume the input is already a square table (i.e., the number of rows and columns are equal) and the table is filled with distinct positive integers $i$ where $1 \leq i \leq n^2$.

# You are the Function

For the last activity, you will be interacting in your groups/tables trying to figure out what is returned through the `main` function. In your groups you will be required to conduct a code trace of the functions below.

You have the following set of functions:

```
def total(lst):
    total_sum = 0
    for elem in lst:
        total_sum += elem

def list_names():
    lst = []
    for person in group: # this is NOT Python code, and this is where you ask each member for the a
        x = person.middle_name()[:8]
        if person.middle_name() == None:
            x = person.favourite_animal()[:8]
        lst += [x]

def get_letter(string):
    count = 0
    for letter in string:
        count += 1
    count //= 2
    i = 0
    while count >= 0:
        i += 1
        count -= 1
    return get_letter_position(string[i])

def get_letter_position(letter):
    count = 1
    alphabet = '!abcdefghijklmnopqrstuvwxyz'
    while count <= 20:
        if alphabet[count] == letter:
            return count
        count += 1
    return 0

def main():
    num_lst = []
    names = list_names()
    for name in list_names():
        num_lst += [get_letter(name)]
    return total(num_lst)
```

Start off by writing all the functions on the whiteboard. Each member of your group should take a function. Your tutor will call the main function. If your tutor is unavailable, whoever takes main should decide when to begin executing. While you are playing at being a function, you may only interact with the rest of your group by taking input and giving output (This includes the `list_names` function where the owner of that function must ask each person in the group for the necessary information once they are called).

# Feedback

**Question 1**

## Feedback

What worked best in this lesson?

*No response*

**Question 2**

## Feedback

What needs improvement most?

*No response*