ILP 2024 : Computing

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

W2S1
20 May 2024

Objectives

- Data and data structure
- List data structure
- Loops with list
- Function
- Recursion

Data and Data Structure

- What is data?
  - Facts that can be recorded or stored
  - It can be in various forms such as numbers, text, images, audio, video, etc.
  - Data is the basic building block of information and is used as input for processing to produce meaningful output.
  - Eg. Age, Name, Address, Photographs, Music files, etc.

- What is data structure?
  - It refers to a particular way of organizing and storing data
  - Facilitating efficient retrieval, insertion, and manipulation of data

Data Structure

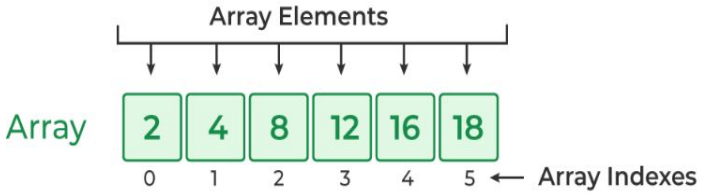
Examples of Data Structures

- Arrays (not in Python)
- List
- Linked Lists
- Stack
- Queue
- Binary Tree
- B Tree
- B^+ Tree
- Heaps

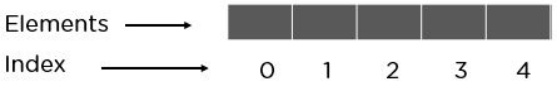
- List
- Stacks
- Queues
- Binary Tree

Array

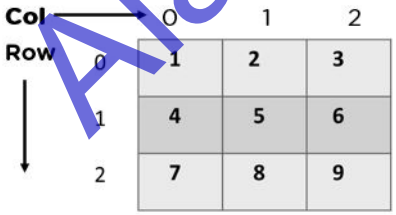
- Collection of similar (same type) data
- Arrays consist of contiguous memory locations.



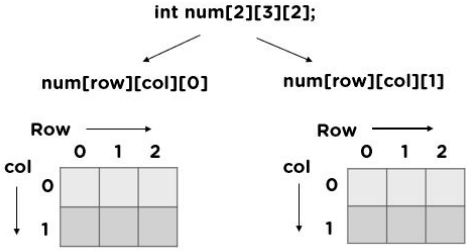
- Types



1D

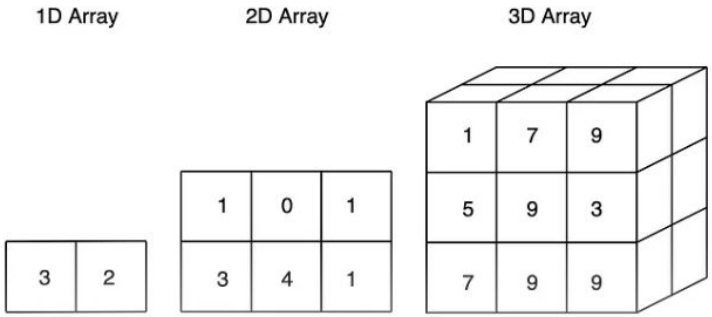


2D



3D

Contd …




```
int a[6] = {2, 3, 5, 7, 11, 13}; in C
```

Advantages of Arrays:

- **Efficient Access**: Direct access to elements by index.
- **Simplicity**: Simple and straightforward data structure.

Limitations of Arrays:

- **Fixed Size:** Size is fixed and cannot be changed dynamically.
- **Homogeneous Elements:** Elements must be of the same type.
- **Static Structure:** Cannot easily accommodate insertions and deletions.

Note: In Python, we can not use **array** directly, instead we can use **list.** We will see later how to use array in Python

List

- Lists are a fundamental data structure in Python.
  - They are used to store an collection of items (may not be same type)

- Each element is identified by its **index**
  - Indexing starts from 0 for the first element.

- **Dynamic:** Lists can grow or shrink in size as needed.

List Vs Array

Aspect	List	Array
Dynamic vs. Fixed Size	Lists are dynamic data structures that can grow or shrink in size dynamically.	Arrays have a fixed size determined at the time of declaration.
Memory Allocation	Lists may not store elements in contiguous memory locations.	Arrays store elements in contiguous memory locations.
Flexibility	Lists offer more flexibility in terms of adding, removing, and modifying elements.	Arrays are less flexible since they have a fixed size.
Performance	Lists may have slightly lower performance due to dynamic memory allocation.	Arrays generally offer better performance for direct element access.

Contd …

Syntax for Creating Lists:

- Lists are defined using square brackets `[ ]`.
- Elements are separated by commas `,`.
- Example: `my_list = [1, 2, 'hello', True]`


```python
# Creating a list object
my_list = [1, 2, 'hello', True]

print(my_list)

print(my_list[3])

print(type(my_list))

print(type(my_list[3]))
```

```
[1, 2, 'hello', True]
True
<class 'list'>
<class 'bool'>
```

```python
var1 = 101
var2 = 99
var3 = 97
var4 = 95

another_list = [var1, var2, var3, var4]
print(another_list)
```

```
[101, 99, 97, 95]
```

List Operations

Append: Add an element to the end of the list.

Insert: Insert an element at a specific index.

Remove: Remove the first occurrence of a specified value.

Pop: Remove and return the element at a specified index.

Length: Get the number of elements in the list.


```python
#List Operations

# Creating a list
my_list = [1, 2, 3, 4, 5]
print("Original List:", my_list)

# Append operation
my_list.append(6)
print("After Append(6):", my_list)

# Insert operation
my_list.insert(2, 10)
print("After Insert(2, 10):", my_list)

# Remove operation
my_list.remove(3)
print("After Remove(3):", my_list)

# Pop operation
popped_element = my_list.pop(4)
print("Popped Element:", popped_element)
print("After Pop(4):", my_list)

# Length operation
list_length = len(my_list)
print("Length of List:", list_length)
```

```
Original List: [1, 2, 3, 4, 5]
After Append(6): [1, 2, 3, 4, 5, 6]
After Insert(2, 10): [1, 2, 10, 3, 4, 5, 6]
After Remove(3): [1, 2, 10, 4, 5, 6]
Popped Element: 5
After Pop(4): [1, 2, 10, 4, 6]
Length of List: 5
```

Contd …

Definition (**length of a list**): The length of a list consists of the **number of elements** in the list.

It can be simply computed with the `len()` function, which returns a positive integer corresponding to the number of elements in the list.

```python
#How to create a list

my_empty_list = [] # declare an empty list

for i in range(10,20):
    my_empty_list.append(i)

print("Now my list contains: ", my_empty_list)

print("Length of my list is:", len(my_empty_list))

for index in range(len(my_empty_list)):
    print(index, "-->", my_empty_list[index])
```

```
Now my list contains:  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
Length of my list is: 10
0 --> 10
1 --> 11
2 --> 12
3 --> 13
4 --> 14
5 --> 15
6 --> 16
7 --> 17
8 --> 18
9 --> 19
```

Concatenation

Concatenation: you can merge two lists together, by using the **+** operator.

```
#Concatenation

first_list = [1, 2, 3, 4]

second_list = ['a', 'b']

third_list = first_list + second_list

print(third_list)
```

```
[1, 2, 3, 4, 'a', 'b']
```

Contd …

List Slicing:

- List slicing allows extracting a portion of a list.
- Syntax: `my_list[start_index:end_index:steps]`
- Example: `print(my_list[1:3])`

```
#List Slicing

my_list = [1, 2, 'hello', True, 4, 5, 8, 0, 'a']

print(my_list[1:3]) #end_index is excluded

print(my_list[:])

print(my_list[5:])

print(my_list[:4])

print(my_list[1:8:2])
```

```
[2, 'hello']
[1, 2, 'hello', True, 4, 5, 8, 0, 'a']
[5, 8, 0, 'a']
[1, 2, 'hello', True]
[2, True, 5, 0]
```

Contd …

List Slicing: (in reverse order)

- `my_list[-start_index: -end_index: -steps]`

```
my_list = [1, 2, 'hello', True, 4, 5, 8, 0, 'a']

print(my_list[-1 : -len(my_list) : -1])

print("\n")
print(my_list[-3 : -len(my_list) : -1])
```

```
['a', 0, 8, 5, 4, True, 'hello', 2]


[8, 5, 4, True, 'hello', 2]
```

Common methods

sort(): Sort the elements of the list in ascending order.

reverse(): Reverse the order of elements in the list.

count(): Count the occurrences of a specified value.

index(): Return the index of the first occurrence of a value.

```python
# common methods use with list

# Creating a list
my_list = [4, 2, 1, 3, 2, 4, 3]
print("Original List:", my_list)

# sort() method - Sort the elements of the list in ascending order
my_list.sort()
print("After Sorting:", my_list)

# reverse() method - Reverse the order of elements in the list
my_list.reverse()
print("After Reversing:", my_list)

# count() method - Count the occurrences of a specified value
count_2 = my_list.count(2)
print("Count of 2:", count_2)

# index() method - Return the index of the first occurrence of a value
index_3 = my_list.index(3)
print("Index of 3:", index_3)
```

```
Original List: [4, 2, 1, 3, 2, 4, 3]
After Sorting: [1, 2, 2, 3, 3, 4, 4]
After Reversing: [4, 4, 3, 3, 2, 2, 1]
Count of 2: 2
Index of 3: 2
```

Practise test 1

Create a list with 10 random numbers. Print smallest number of the list. Print the index of smallest number.

```
Random Numbers List: [6, 5, 3, 4, 8, 1, 9, 2, 12, 11]
```

```
smallest_number = min(random_numbers)
print("Smallest Number:", smallest_number)
```

Biggest number?

```
'''
Create a list with 10 random numbers.
Print smallest number of the list. Print the
index of smallest number.
'''

# Create a list with 10 random numbers
random_numbers = [6,5,3,4,8,1,9,2,12,11]
print("Random Numbers List:", random_numbers)

# sort the list
random_numbers.sort()
print("Sorted list --> ", random_numbers)


print("Smallest Number:",random_numbers[0])

small_number_index = random_numbers.index(random_numbers[0])

print("Index of Smallest Number:", small_number_index)
```

```
Random Numbers List: [6, 5, 3, 4, 8, 1, 9, 2, 12, 11]
Sorted list -->  [1, 2, 3, 4, 5, 6, 8, 9, 11, 12]
Smallest Number: 1
Index of Smallest Number: 0
```

list with for loop

```
my_list = [1, 2, 'hello', True]

print(my_list)

print(my_list[0])
print(my_list[1])
print(my_list[2])
print(my_list[3])
```

```
[1, 2, 'hello', True]
1
2
hello
True
```

Sounds boring !!!

```
for i in my_list:
    print(i)
```

```
for i in my_list:
    print(i)
```

```
1
2
hello
True
```

The `in` keyword indicates that the iteration variable (`i`) will take values in the given `list`

Contd …

```python
# Predefined grades for 5 subjects
grade1 = 85
grade2 = 90
grade3 = 75
grade4 = 80
grade5 = 95
number_of_subjects = 5
# Calculate average
average = (grade1 + grade2 + grade3 + grade4 + grade5) / number_of_subjects

# Print average
print("Average grade:", average)
```

```
Average grade: 85.0
```

```python
grades = [85, 90, 75, 80, 95]

total_grade = 0

for g in grades:
    total_grade = total_grade + g

print("Average grade:", total_grade/len(???))
```

```
Average grade: 85.0
```

Sounds boring !!!

For 100 students :(

Can be done easily !!!

2D list (lists of lists)

```python
grades = [85, 90, 75, 80, 95]

for g in grades:
    print(g)
```

```
85
90
75
80
95
```

This one for one student

```python
grades = [
    [85, 90, 75, 80, 95],  # Grades for student 1
    [70, 85, 90, 65, 75],  # Grades for student 2
    [90, 95, 85, 80, 85]   # Grades for student 3
]

print("Student: 1, Subject: 1  -->", grades[0][0])
print("Student: 1, Subject: 2  -->", grades[0][1])
print("Student: 1, Subject: 3  -->", grades[0][2])

print("----------------------------")

print("Student: 3, Subject: 1  -->", grades[2][0])
print("Student: 3, Subject: 2  -->", grades[2][1])

print("\n-----------All studendes details----------\n")

for line in grades:
    print(line)
    # print("------------")
    # for element in line:
    #     print(element)
```

```
Student: 1, Subject: 1  --> 85
Student: 1, Subject: 2  --> 90
Student: 1, Subject: 3  --> 75
----------------------------
Student: 3, Subject: 1  --> 90
Student: 3, Subject: 2  --> 95

-----------All studendes details----------

[85, 90, 75, 80, 95]
[70, 85, 90, 65, 75]
[90, 95, 85, 80, 85]
```

This one for three students

```python
grades = [
    [85, 90, 75, 80, 95],  # Grades for student 1
    [70, 85, 90, 65, 75],  # Grades for student 2
    [90, 95, 85, 80, 85]   # Grades for student 3
]

for student in grades:
    print(student)
    print("------------")
    for subjects in student:
        print(subjects)

    print("\n")
```

```
[85, 90, 75, 80, 95]
------------
85
90
75
80
95


[70, 85, 90, 65, 75]
------------
70
85
90
65
75


[90, 95, 85, 80, 85]
------------
90
95
85
80
85
```

enumerate() function

iterate over a sequence while keeping track of the **index** position and the **corresponding value** at each iteration

```python
#use of enumerate function

my_list = ['apple', 'banana', 'cherry']

for index, value in enumerate(my_list):
    print(index, value)
```

```
0 apple
1 banana
2 cherry
```

```python
grades = [
    [85, 90, 75, 80, 95],  # Grades for student 1
    [70, 85, 90, 65, 75],  # Grades for student 2
    [90, 95, 85, 80, 85]   # Grades for student 3
]

for s in grades:
     print(s)

print("Printing the grades of each student")

for s in grades:
    print("Student")
    for i in s:
        print("Grade", i)
```

```
[85, 90, 75, 80, 95]
[70, 85, 90, 65, 75]
[90, 95, 85, 80, 85]
Printing the grades of each student
Student
Grade 85
Grade 90
Grade 75
Grade 80
Grade 95
Student
Grade 70
Grade 85
Grade 90
Grade 65
Grade 75
Student
Grade 90
Grade 95
Grade 85
Grade 80
Grade 85
```

```python
grades = [
    [85, 90, 75, 80, 95],  # Grades for student 1
    [70, 85, 90, 65, 75],  # Grades for student 2
    [90, 95, 85, 80, 85]   # Grades for student 3
]


print("Printing the grades of each student")

for s_index, s_grades in enumerate(grades):
    print("Student ", s_index+1, "has obtained: ", s_grades)
```

```
Printing the grades of each student
Student  1 has obtained:  [85, 90, 75, 80, 95]
Student  2 has obtained:  [70, 85, 90, 65, 75]
Student  3 has obtained:  [90, 95, 85, 80, 85]
```

```python
grades = [
    [85, 90, 75, 80, 95],  # Grades for student 1
    [70, 85, 90, 65, 75],  # Grades for student 2
    [90, 95, 85, 80, 85]   # Grades for student 3
]

for s_index, s_grades in enumerate(grades):

    print("\nStudent ", s_index+1, "Details:")

    avg_grade = sum(s_grades)/len(s_grades)

    print("He/She Grades: ", s_grades)
    print("His/Her Avg. Grades", avg_grade)
```

Try without using `sum()`

```
Student  1 Details:
He/She Grades:  [85, 90, 75, 80, 95]
His/Her Avg. Grades 85.0

Student  2 Details:
He/She Grades:  [70, 85, 90, 65, 75]
His/Her Avg. Grades 77.0

Student  3 Details:
He/She Grades:  [90, 95, 85, 80, 85]
His/Her Avg. Grades 87.0
```

Do it now in your notebook

Practise Test

1. Write a Python program that takes a list of integers as input and returns the sum of all even numbers in the list.

2. Write a Python program that takes two lists of integers as input and returns a new list where each element is the product of the corresponding elements in the input lists. Give error message if the length of the lists are not same. **Hint:** `append()`

3. Write a Python program that takes a list as input and returns a new list with duplicate elements removed, while preserving the original order of elements. **Hint:** `count()` or `if` item `not` in my_list:

4. Write a Python program that takes an integer ***n*** as input and returns a list of all prime numbers up to ***n***.

5. Write a Python program that iterates the items of a list using `while` loop if the length of the list is less than 5. Otherwise, use `for` loop to iterate.

6. Create a two equal length lists using `range()`, where one list contains even numbers (0, 2, 4…) and the other list contains odd(1, 2, 3, …) numbers. Create a third list by merging these two lists.

Sorting algorithm

Can you find the smallest number ?

53, 21, 38

Can you find the smallest number ?

89, 42, 17, 76, 8, 93, 58, 23, 66, 4, 32, 87, 50, 65, 2, 19, 88, 14, 61, 29, 95, 31, 72, 3, 40, 96, 48, 69, 37, 51, 20, 94, 70, 26, 82, 12, 97, 73, 21, 45, 83, 55, 10, 60, 35, 74, 13, 54, 7, 85, 39, 67, 15, 80, 41, 77, 25, 79, 47, 9, 75, 91, 64, 1, 86, 57, 24, 5, 38, 92, 22, 84, 46, 78, 28, 68, 6, 59, 30, 81, 43, 63, 0, 99, 33, 98, 27, 11, 56, 16, 49, 71, 90, 18, 34, 53, 62, 36, 44, 52.

Unorganized data

Sorting

Organized data

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99.

Application of sorting

- Flight Booking Systems
  - Sorting algorithms help to organize flight options based on factors like price, departure time, duration, or number of stops to find suitable flights
- E-commerce Websites
  - Sorting algorithms are utilized to arrange products in search results based on relevance, price, ratings, or popularity.
- Finding the trending Youtube Videos

??? Sort

- ??? sort is one of the simplest sorting algorithm
- It forms the foundation for understanding other sorting algorithms.

??? Sort: How it works

Input: 19, 18, 14, 61, 9

19, 18, 14, 61, 9

18, 19, 14, 61, 9

18, 19, 14, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 9, 61

18, 14, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 9, 19, 61

14, 18, 9, 19, 61

14, 18, 9, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9,14, 18, 19, 61

9,14, 18, 19, 61

??? Sort: Implementation

i = 0 *j = 1*

19, 18, 14, 61, 9

18, 19, 14, 61, 9

i *j*

18, 19, 14, 61, 9

18, 14, 19, 61, 9

i *j*

18, 14, 19, 61, 9

i=3 *j=4*

18, 14, 19, 61, 9

18, 14, 19, 9, 61

- Start from the first and second elements
- If the first element is greater than the next one, swap them
- Move to the next pair of adjacent elements
- Do comparison and swap process
- Repeat until the end of array
- Repeat the above steps until the entire array is sorted.

Outer loop: k=1 to n-1

Inner loop: j=1 to n-1

i = 0, j = 1

if A[i]> A[j]:

temp =A[i]
A[i] =A[j]
A[j] =temp

i +=1
j +=1

N-1
Iterations

N-1
Comparison

4 x 4 = 16 comparisons !!!

Practise Test

Can you implement the ??? sort?

Optimised Bubble Sort

19, 18, 14, 61, 9

18, 19, 14, 61, 9

18, 19, 14, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 61, 9

18, 14, 19, 9, 61

18, 14, 19, 61, 9

18, 14, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 19, 9, 61

14, 18, 9, 19, 61

14, 18, 9, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 18, 9, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

14, 9, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

9, 14, 18, 19, 61

Outer loop: k=1 to n-1

Inner loop j=1 to n-1

i = j-1

if A[i]> A[j]:

temp =A[i]

A[i] =A[j]

A[j] =temp

n=n-1

Unordered

4 + 3 + 2 +1 =10 comparisons !!!

Optimised Bubble Sort

Input: 99, 1, 2, 3,4

99, 1, 2, 3,4

1, 99, 2, 3,4

1, 99, 2, 3,4

1, 2, 99, 3,4

1, 2, 99, 3,4

1, 2, 3, 99,4

1, 2, 3, 4,99

Outer loop: k=1 to n-1

Inner loop: j=1 to n-1

if A[j-1]> A[j]:

temp =A[j-1]
A[j-1] =A[j]
A[j] =temp

n =n-1

4 + 3 + 2 +1 =10 comparisons !!!

Let's stop when no swapping is done in an iteration.

if A[j-1]> A[j]:

temp =A[j-1]
A[j-1] =A[j]
A[j] =temp

n =n-1
swap = true

4 + 3 =7 comparisons !!!

Optimised Bubble Sort

Input: 3, 2, 1, 7, 8

Unordered

m = i

3, 2, 1, 7, 8

2, 3, 1, 7, 8

2, 1, 3, 7, 8

swapping

1, 2, 3, 7, 8

2, 3, 1, 7, 8

2, 1, 3, 7, 8

2, 1, 3, 7, 8

2, 1, 3, 7, 8

4 + 1 =5 comparisons !!!

What if we remember the point where the last swapping is done.

```
while (swap):
    swap = false
    Inner loop:  j=1 to m
        if   A[j-1]> A[j]:
            temp =A[j-1]
            A[j-1]  =A[j]
            A[j]  =temp
            m = j-1
            swap = true
```

Q. How many comparisons will be done by the optimized Bubble sort for the following input?

9, 8, 7, 6, 5

Ans: 10 comparisons. ((n-1) x n)/2

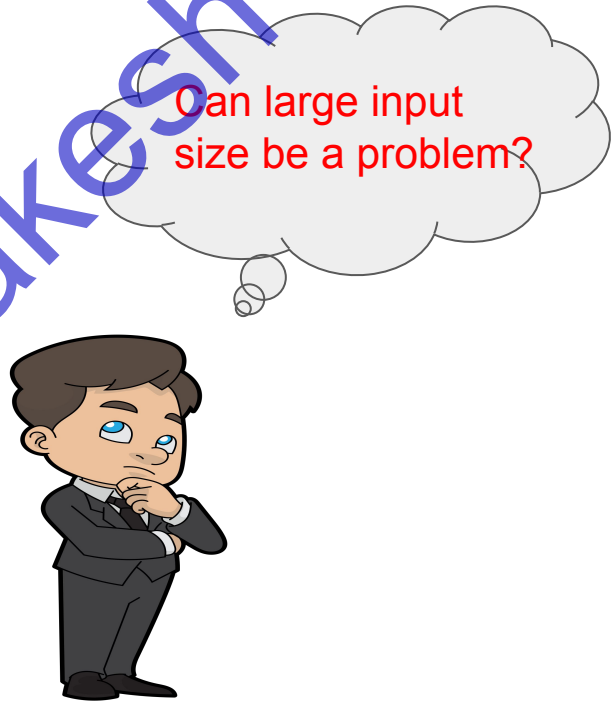
Q. What do we need to change in the code to do sorting in descending order?

Ans:

```
if arr[j-1] < arr[j]:
        swap()
```

But, a computer can do the comparison easily. <u>In very less amount of time.</u>

Why do we need worry?



Function

```python
#-------------------Introduction to Function-

# Calculate the average of a list of numbers
numbers = [85, 90, 75, 80, 95]

# Sum all numbers
total = 0
for num in numbers:
    total += num

# Calculate average
average = total / len(numbers)

print("Average:", average)
```

```
Average: 85.0
```

Calculates the average of the given list

```python
list2 = [70, 85, 90, 65, 75]
```

Now, imagine you need to calculate the average of **multiple lists** in your program

Thanks to **Function** ! We don't need to write/copy-paste the same code every time

Contd …

```python
# Define a function to calculate average
def calculate_average(numbers):
    total = sum(numbers)
    return total / len(numbers)


# Calculate average of the first list
list1 = [85, 90, 75, 80, 95]
average1 = calculate_average(list1)
print("Average of list 1:", average1)


# Calculate average of the second list
list2 = [70, 85, 90, 65, 75]
average2 = calculate_average(list2)
print("Average of list 2:", average2)
```

```
Average of list 1: 85.0
Average of list 2: 77.0
```

Function

- A function is a **block of code** which only runs **when it is called**.
- It is a block of **reusable** code that perform **a specific task**.
- You can also **create your own custom** functions built-in functions like `print()` and `len()`
- Every function should **return** something

```python
return total / len(numbers)
```

Contd …

```python
# Define a function to calculate average
def calculate_average(numbers):
    total = sum(numbers)
    return total / len(numbers)


# Calculate average of the first list
list1 = [85, 90, 75, 80, 95]
average1 = calculate_average(list1)
print("Average of list 1:", average1)


# Calculate average of the second list
list2 = [70, 85, 90, 65, 75]
average2 = calculate_average(list2)
print("Average of list 2:", average2)
```

```
Average of list 1: 85.0
Average of list 2: 77.0
```

Defining Functions

```python
def function_name(parameter/argument):

    print "Hello, " + parameter + "!"
```

Calling Functions

```python
function_name(parameter)
```

Function

```python
def add(a, b):
    """Add two numbers."""
    return a + b

result = add(3, 5)
print (result)
```

```
8
```

```python
def divide(a, b):

    if b == 0:
        return "Error: Division by zero!"
    else:
        return a / b

    print("A Function stop excecuting after 'returning' a value ")

print(divide(3,5))
print(divide(3,0))
```

```
0.6
Error: Division by zero!
```

Quickly write a function to multiply two numbers and return only for odd results.

Lifetime and scope of a variable

```python
#scope and lifetime of a variable
def multiply (a, b):

    print("Variable1: ", var1)

    local_result = a*b

    if (result%2 != 0):
        return result
    else:
        print("Result is an even number")
        return None # by default a function returns 'None'


var1 = 6 #global variable

product = multiply(2, 10)

print("Product is: ", product)

print("Result: ", local_result)
```

```
Variable1:  6
Result is an even number
Product is:  None
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most r
Cell In[138], line 21
     17 product = multiply(2, 10)
     19 print("Product is: ", product)
---> 21 print("Result: ", local_result)

NameError: name 'local_result' is not defined
```

- **Global variables** are defined **outside of any function** and can be **accessed from anywhere** in the program.
- Global variables can be accessed, modified, and reassigned from within any function or block of code.
- **Local variables** are defined **within a function** or block of code and are only **accessible within** that **function or block**.

Contd…

- Local variables **shadow** (or hide) global variables with the same name. Inside a function, a **local variable** with the same name as a global variable **takes precedence**.

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

```
Python is fantastic
Python is awesome
```

- To change the value of a global variable inside a function, we need to use `global` keyword:

```
x = "awesome"

def myfunc():
  global x
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

```
Python is fantastic
Python is fantastic
```

Function(s) inside a function

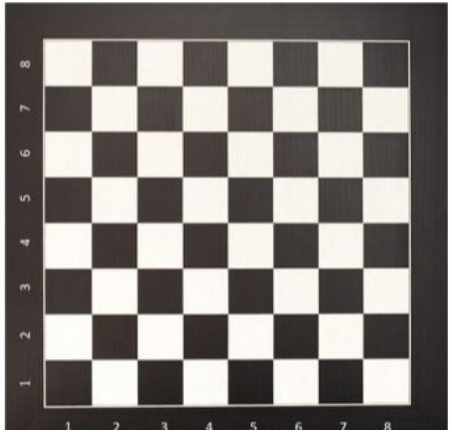
```python
#Function inside a function

def grading(g1, g2):

    total = add_grades(g1, g2)
    average = total/2

    if average >= 50:
        return 'A'
    else:
        return 'F'


def add_grades(m1, m2):
    return m1+m2

stud1 = grading(100,45)

print(stud1)
```

```
A
```

Practise Test (function + list)

- Write a function `sum_list(lst)` that takes a list of numbers as input and **returns** the sum of all the elements in the list.
- Write a function `count_occurrences(lst, target)` that takes a list and a target value as input and **returns** the number of times the target value appears in the list.
- Write a function `reverse_list(lst)` that takes a list as input and **returns** a new list with the elements reversed.
- Write a function `is_palindrome(lst)` that takes a list as input and **returns** `True` if the list is a palindrome (reads the same forwards and backwards), otherwise returns `False`.

Practise test



You will have to write two functions:

- To collect user's inputs on rows and columns indexes
- To check and print if the square is black or white.

Lambda function

- **Lambda functions:** Also known as **anonymous functions**, are **small**
- Can have any number of **arguments** but only **one expression**.

```
lambda arguments: expression
```



```python
# Examples of  lambda functions
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8


x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
8
13
```

Recursion

- **Recursion** is a powerful programming technique where **a function calls itself** in order to solve a problem.

- It is based on the idea of breaking down a problem into smaller, simpler instances of the same problem.
  - Eg: n! = n * (n-1) * (n-2) * ... * 1

- A recursive function typically has two parts:
  - **Base Case**: A condition that specifies when the recursion should stop.
  - **Recursive Case**: A call to the function itself with modified arguments to move closer to the base case.

```python
#recursion

def factorial(n):
    if n == 0:                # Base case
        return 1
    else:                     # Recursive case
        return n * factorial(n - 1)

print(factorial(5))
```

```
120
```


Note: Recursion is well-suited for problems that can be broken down into smaller instances of the same problem. It is particularly useful for problems with a clear base case and a repetitive structure.

Practice test

The Fibonacci sequence is a series of numbers where each number is the sum of the two preceding ones.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Can you write a recursive function to find n^{th} term of the fibonacci series?

Hint: **fib(n) = fib(n-1) + fib(n-2) for n > 1**