First Steps in Python

A Gentle Introduction to Programming

First Steps in Python: A Gentle Introduction to Programming

Contents

Brought to you by:

NousBase

NousBase is an education-focused initiative by **Cinansys**.

We create learning resources that help people think clearly, build real skills, and grow with confidence in a changing world.

Our work spans areas like programming, artificial intelligence, finance, competitive exams, and employability skills.
We focus on understanding over memorization, clarity over noise, and usefulness over volume.

This book is part of that effort.
It is designed for learners who want to start small, learn honestly, and build step by step.

Author
Chaitanya D. Sangwai

If this book helped you, inspired you, or simply made learning feel easier, we would love to stay connected.

Follow NousBase and explore more at:

Website: https://nousbase.cinansys.com

Instagram: https://www.instagram.com/nous_base/

LinkedIn: https://www.linkedin.com/company/nousbase

Mails us at: books_nousbase@cinansys.com

Learning does not end with one book.
It grows when curiosity continues.

Chapter 0: A Rather Unusual preface

Welcome to your first step into the world of programming.

This book is designed to be your very first book on coding. While we will be using **Python**, our goal goes deeper than just learning a language.

This book builds the **foundations of logic**. It is a **starting point**, a gentle first step into the world of logic, coding, and clear thinking.

What this book aims at:

1. Build your foundations of logic
We'll try to understand the problem, make a plan, try it, and then reflect.
This way, you *think* like a problem solver.

2. Introduce you to Python
Python is simple, friendly, and perfect for beginners.

3. Prepare you for the future
By the end of this book, you won't just know Python syntax.
You'll know *how to learn*, *how to reason*, and *how to build confidence* for everything that comes afterward.

Who is this book for?

Anyone who wants to begin their journey into programming.

Age is just a number; curiosity is the only real requirement.

If you can read, think, and try, you can learn programming.

The "No-Setup" Approach

Most programming books begin with a boring, 20-page chapter on "Environment Setup" and "Installing Python." It is often where beginners give up before they even start.

We are skipping that.

My aim is to teach you coding *quickly*. We don't need to install complex software to learn logic. We are going to use the cloud.

Your First Workspace: Google Colab

To start your journey, we will use a free tool called **Google Colab**. It allows you to write and run Python code directly in your browser.

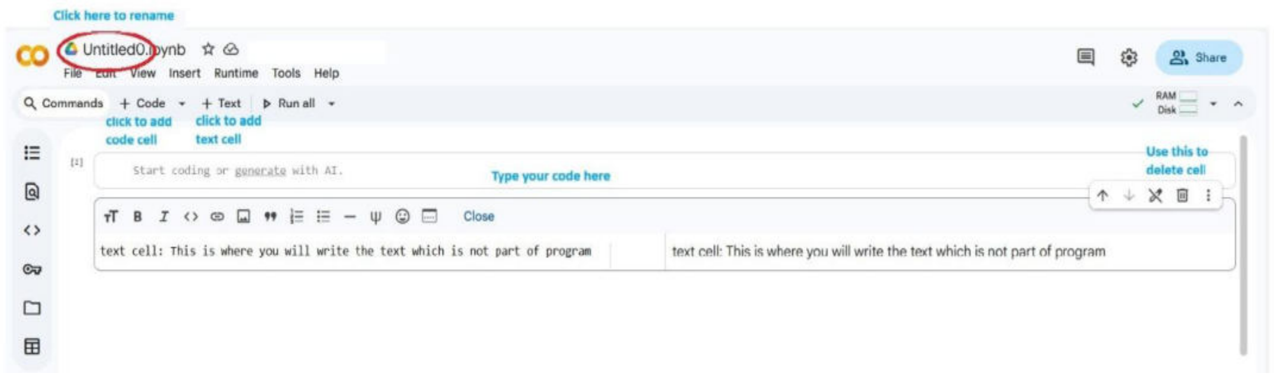
Step 1: Visit the Google Colab website: colab.research.google.com

Step 2: Go to **File -> New Notebook**.

Congratulations, your digital notebook is now open! This is where you will practice programming.

Step 3: You can save this notebook and use it again later. Let's give it a name.

- Click on the file name at the top of the screen (circled in the screenshot below).
- Type a name you like (e.g., *My_First_Code*).



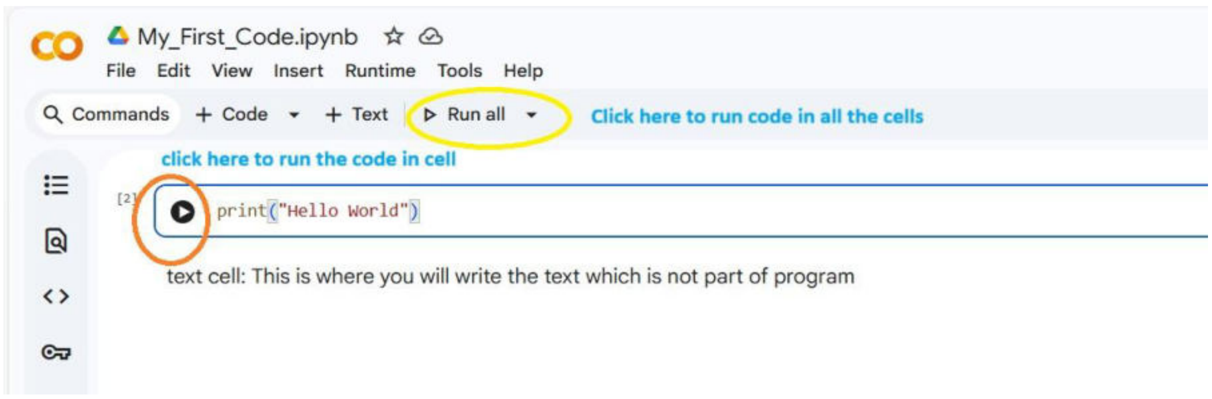
text cell: This is where you will write the text which is not part of program

text cell: This is where you will write the text which is not part of program

Inside the notebook, you will find two kinds of boxes (called *cells*):

- **Code cells** → for writing Python code
- **Text cells** → for notes, explanations, or comments

You can add more cells anytime using the **+ Code** or **+ Text** buttons.


My_First_Code.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Click here to run code in all the cells
click here to run the code in cell

```
print("Hello World")
```

text cell: This is where you will write the text which is not part of program

Your very first program

In a code cell, type: [`we will use this font and box like below for the code lines, as we did below`]: **Type this code. Do not copy. Copy-paste may lead to errors.**

```
print("Hello World")
```

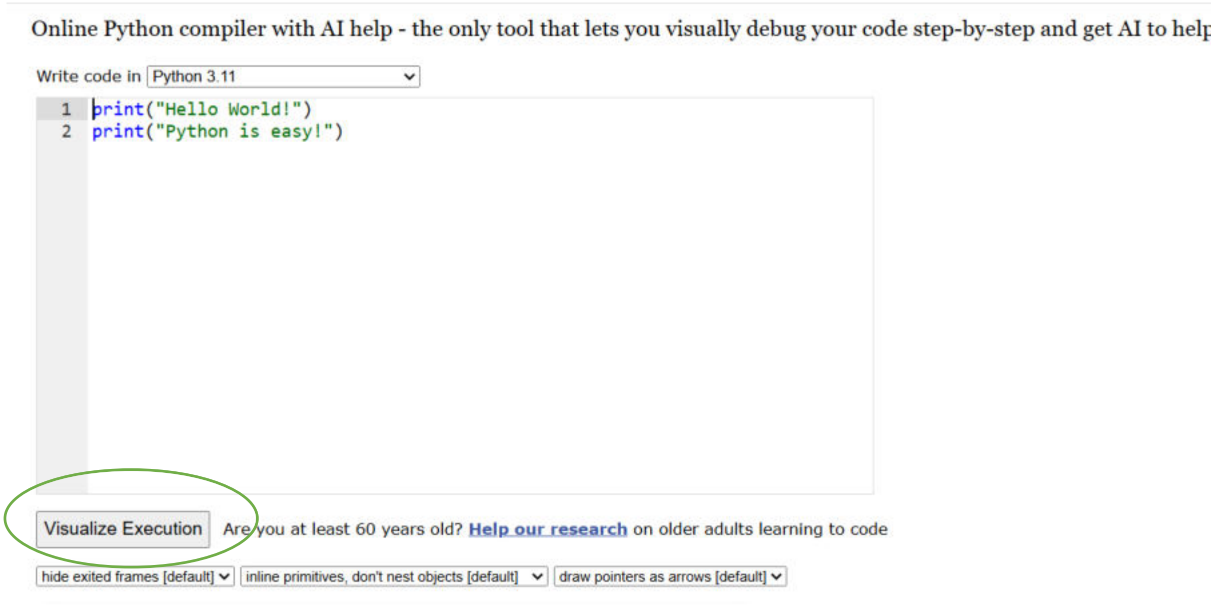
Now press the **play button** on the left, or press **Ctrl + Enter**.

And,
you've written your first program!

It prints a message on the screen, and that simple act marks the beginning of your coding journey.

Another way and very intuitive way to run python code is the site: https://pythontutor.com/python-compiler.html#mode=edit

You can directly write code in left window and click "Visualize Execution", you will get output at right side.



Online Python compiler with AI help - the only tool that lets you visually debug your code step-by-step and get AI to helı

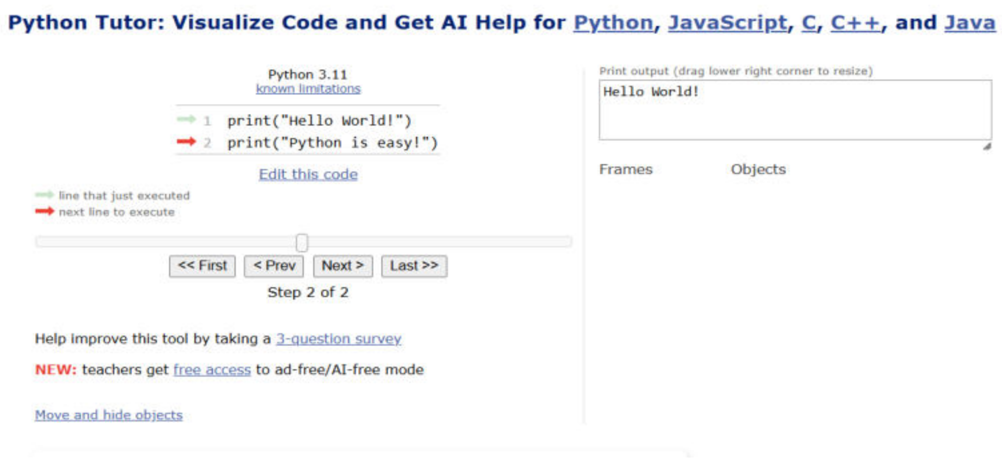
Write code in Python 3.11

```
print("Hello World!")
print("Python is easy!")
```

Are you at least 60 years old? Help our research on older adults learning to code

Visualize Execution
hide exited frames [default]
inline primitives, don't nest objects [default]
draw pointers as arrows [default]

But remember, here code will execute line-by-line. i.e. you will need to click next to execute next line of code



Python Tutor: Visualize Code and Get AI Help for Python, JavaScript, C, C++, and Java

Python 3.11
known limitations

```
1 print("Hello World!")
2 print("Python is easy!")
```

Print output (drag lower right corner to resize)

```
Hello World!
```

Step 2 of 2

Help improve this tool by taking a 3-question survey

NEW: teachers get free access to ad-free/AI-free mode

During the course, we will create a simple but complete application for managing membership of the Science enthusiasts club named **Sci-Mate**.

Next, we will see, why Programming is very easy.

-

Instructions and clarifications

This Book Is For

- Absolute beginners
- Those who want to understand coding, not just syntax
- Those who once learned coding but forgot and want a fresh start

Age is not a bar.

This Book Is NOT

- A reference manual
- A book for intermediate or advanced coders
- A complete guide to Python

This book focuses on foundations and thinking, not coverage.

How to Study This Book

- Code along. Always type the code.
- Type the code, **do not copy-paste.**
- Go slow. Do not rush. Do not worry.
- Read each concept carefully.
- Take help from the internet or Google if you feel stuck.
- Read the TL;DR first, then the detailed explanation.
- Revisit earlier chapters without guilt.
- Build the final project very seriously. Yes, this matters.

When to read the book: Any time. Even right now.

If you like structure, here is a simple suggestion:

- Start on a Monday
- One chapter a day
- Finish theory by Friday
- Work on the project on Saturday
- Enjoy your Sunday

If at any point you feel:

- "This is slow": that is normal
- "I don't get it yet": that is learning
- "I need to read again": that is progress

Programming is not about speed.
It is about **clarity of thought**.

If you finish this book honestly, you will not just know Python.
You will know **how to approach problems**.

That skill stays with you forever.

Chapter 1: Start Programming

TL; DR

- **Programming** = telling a machine exactly what to do, step by step
- **Programming language** = the language we use to talk to machines
- **Variables** = names or labels of boxes that store values
- **Variables** are of different type like int (integer), float, str (string)
- **Expression** = something that produces value
- **Functions** = actions we perform
- **Python** lets us write instructions in a very human-friendly way

Note: Writing a program is just like writing a recipe!

What is Programming?

Programming is **instructing a machine to perform a task**.

In simple words, programming is telling a computer **exactly what to do**, step by step.

At Café**,** you say: "A sandwich, please."

A human understands this instantly.

But notice what is hidden inside that sentence:

1. Take bread
2. Prepare stuffing
3. Put stuffing between bread
4. Place sandwich on plate
5. Serve it

This single sentence **abstracts many steps**.

For a human, a simple sentence "A sandwich, please" is enough.

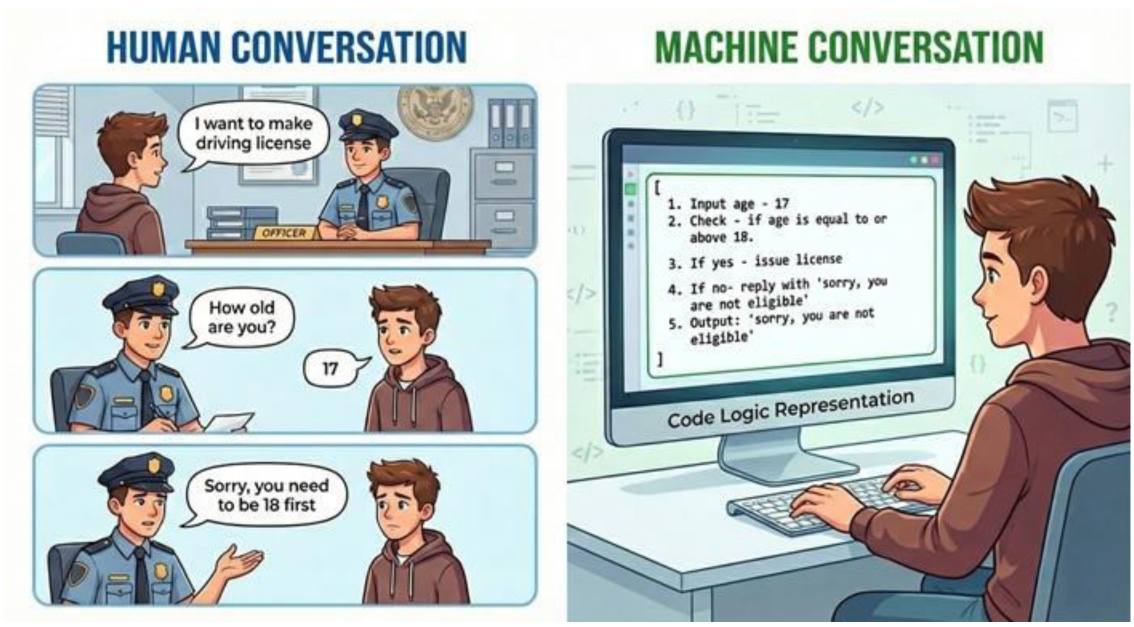
Machines are different. They are incredibly fast, but they are not "smart" in the human sense.

For machine, you will need to instruct it step-by-step, just like above.

And the good news is: once you give them clear steps, they never get tired or bored.

That is programming: Breaking a big idea into small steps the computer definitely understands.

Programming = breaking a task into very small, exact steps that computer understands


HUMAN CONVERSATION
I want to make driving license
OFFICER
How old are you?
17
Sorry, you need to be 18 first
MACHINE CONVERSATION
Code Logic Representation

```
[
  1. Input age - 17
  2. Check - if age is equal to or
     above 18.
  3. If yes - issue license
  4. If no- reply with 'sorry, you
     are not eligible'
  5. Output: 'sorry, you are not
     eligible'
]
```

On the **left side**, a human conversation is happening. In the officer's mind, many steps are **abstracted**: • Ask age • Check eligibility rules • Decide yes or no • Give final response Humans are good at this kind of abstraction.	On the **right side,** this is how computer systems work, guided by step-by-step instructions. The instructions are: • Take age as input • Check if age ≥ 18 • If yes, approve • If no, reject This is programming. We are going to learn how to give these step-by-step instructions i.e. how to program.

What is Programming Language?

If programming is the act of giving instructions, a Programming Language is the tool we use to write them.

Python is such a language. It has specific **keywords** (vocabulary) and **syntax** (grammar rules)

Progress Unlocked

- You now understand what programming is.
- You learned how to convert human thinking into computer steps.
- This skill matters more than any programming language.

How do we instruct computers (how do we program?)

Let's write our first program:

Write A Program To Know About "Functions"

Program 1.1: Display a message "Hi! I am Python"

Visit 'colab' or 'pythontutor' as we have seen in previous chapter.

```
print("Hi! I am Python")
```

Run the program! You can see the output.

Congratulations!! Task 1.1 done!

print ( ) is a function.

Functions are actions we perform.

Processes we do in program like, Print, ask input etc. are functions.

By rules of the language, functions are followed by parenthesis, that is, ( ). Inside those parentheses, we provide some values. Function basically processes those values.

In this case, we provide "Hi! I am Python" as value to function. This value we provide to function is **argument**.

Write A Program To Know About "Variables"

Program 1.2: Say "Hello" to user with his/her name. e.g. "Hello John"

Now, in order to greet the user with his/her name, we need to know the user's name.

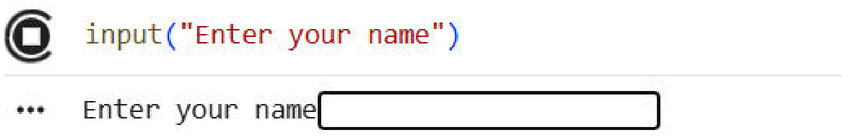
How to know it? Simply, by asking the user!

So, we have 2 steps to solve the problem.

1. Ask user his/her name.
2. Display the message of greeting.

Step 1: Take user's name. for this, we use input( ) function.

It pauses the program and waits for user input.



```
input("Enter your name")
```

```
Enter your name
```

Now, you can enter your name here. But to display the name with greeting, we will need print( ) function.

How to tell the print ( ) function your name?

For this, we use **Variable.**

Variable stores value. It is like a named box storing a value.

Think of it as a **named box** where the computer keeps something for later use.

Example:

age = 10

Step 2: Store name entered by user in variable.

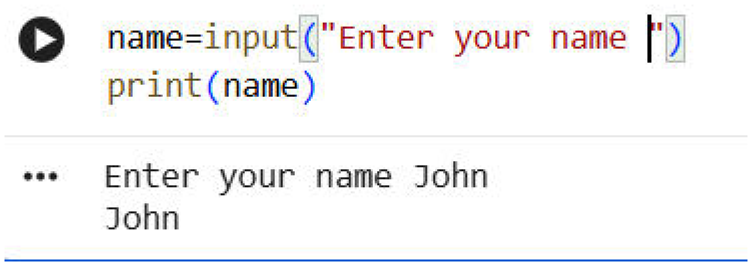
```
name = input("Enter your name")
```

This process assigns value entered by user to the variable "name".

Step 3: Functions accept values stored in variables.

So, we pass variable "name" to print function

```
name = input("Enter your name ")

print(name)
```



```
name=input("Enter your name ")
print(name)
```

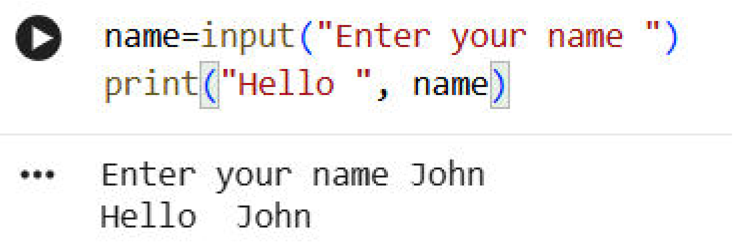
```
Enter your name John
John
```

This is fine, but we want to add greeting too!

print( ) function accepts multiple arguments. So, we can simply add greeting line and name too.

```
name = input("Enter your name ")

print("Hello", name)
```

and you are done!



```
name=input("Enter your name ")
print("Hello ", name)
```

```
Enter your name John
Hello  John
```

Progress Unlocked

- You can now take input, store it, and use it.
- You know what are functions and how to "call" them
- You know what are variables

NOTE:
1. The text we want to display exact -> included in " "
2. Variable name -> we want to display value stored in it -> **NOT** included in " "

We will see more about this further.

Naming the variable.

You can use 'almost' anything as variable name, but beware of two points below.

- Good practice is, name of variable should represent what the value is about. For example, variable name "age" clearly indicates that value it holds is age of someone.
- Also remember, we have said, language has rules.

There are certain rules for naming the variable.

1. Variable name must start with a letter or underscore
2. Cannot start with a number

Write A Program To Know About "Expressions"

Program 1.3: Take 2 numbers from user and display sum of the numbers

Steps:

1. Ask user to input 2 numbers
2. Store values in variables
3. Add the numbers
4. Display the result

Let's start our program.

```
first_number = input("Enter first number")

second_number = input("Enter second number")
```

How to add 2 numbers? How to print?

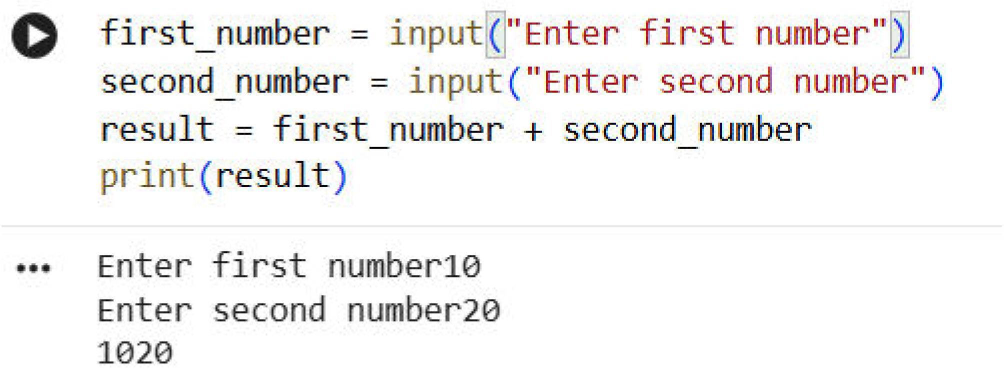
Mathematical operations are simple. We use 'operators' like we use in maths.

Operator	Purpose	Example
+	Addition	a + b
-	Subtraction	a - b
*	Multiplication	a * b
/	Division	a / b
%	Modulus (gives reminder)	a % b [7 % 3 gives 1 as reminder]
//	Floor division	a//b [7 // 3 gives 2]

And to make printing easy, we simply use variable to store the result.

```
first_number = input("Enter first number")

second_number = input("Enter second number")

result = first_number + second_number

print(result)
```

Let's run the program!



```
first_number = input("Enter first number")
second_number = input("Enter second number")
result = first_number + second_number
print(result)
```

```
Enter first number10
Enter second number20
1020
```

Oops!

We wanted 30 as result. Why did we get 1020?

It is because whatever 'input' function returns is treated as text.

We need to understand data types of variables to understand in detail.

Variables and Data Types

Now, we have seen variable stores value.

These values are of different types. Type of variable values is very important concept.

Common basic types are:

1. **int**: short form of integer.
   It is simple numeric value like 35, -10 etc. (numbers without decimal point)
   e.g. a =5
2. **float**: these are numbers with decimal values like 3.5, 10.2 etc. e.g. b= 7.34
3. **str**: short form of string.
   This is made up of characters. It is represented in single or double quotes.
   e. g. “Hello World”, “I like Python” are strings. Remember, strings can also contain numbers, but they will be treated as character.
   c = “Hello there”
   d= “9”
   here, “9” is treated as character and not number.
   This is how input function treats numbers.
4. **bool**: Boolean. This is type which has only 2 values, **True** or **False**.

Type Conversion

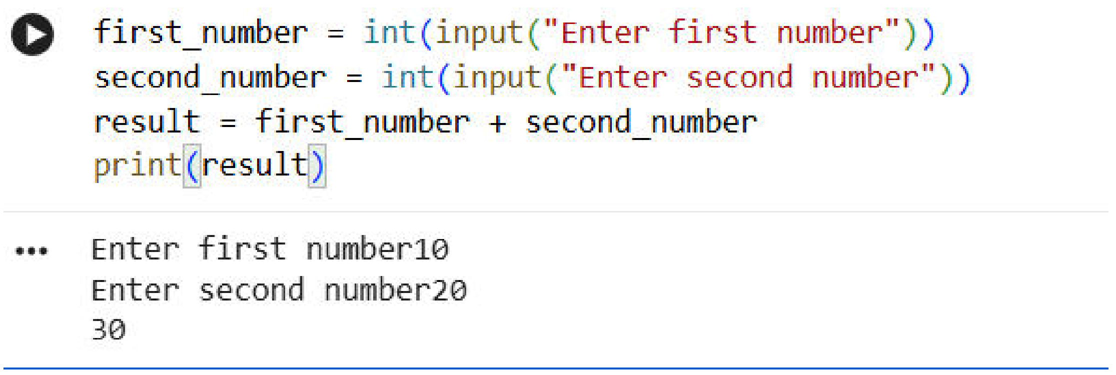
We now know, to perform mathematical operations, whatever input function returns should be converted to numeric type like int or float.

We use type converters for this.

Simply, to convert text/str to int, we use int ( ).

Below is our code:

```
first_number = int(input("Enter first number"))

second_number = int(input("Enter second number"))

result = first_number + second_number

print(result)
```



```
first_number = int(input("Enter first number"))
second_number = int(input("Enter second number"))
result = first_number + second_number
print(result)
```

```
Enter first number10
Enter second number20
30
```

And we are done with the program 1.3!

NOTE: statements that result in certain value are known as **Expressions**.

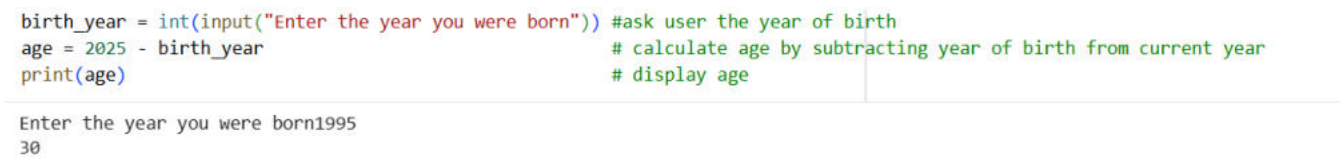
So, (first_number + second_number) is an **Expression**.

Progress Unlocked

- You know about data types
- You can convert data types from one to another
- You can write expressions
- You can perform mathematical operations

Program 1.4: Calculate the age of user

Read the program below and try to understand what's going on



```
birth_year = int(input("Enter the year you were born")) #ask user the year of birth
age = 2025 - birth_year                                 # calculate age by subtracting year of birth from current year
print(age)                                              # display age
```

```
Enter the year you were born1995
30
```

Concepts we learnt revisited:

- **birth_year** and **age**: **Variables** with data type '**int**'
- **input**("Enter the year you were born") and **print**(age) : **functions**
- **"Enter the year you were born"** and **age**: **arguments** for functions input ( ) and print ( )
- **2025- birth_year** : **Expression**
- **int**( ) : **Type conversion**

1-Minute Challenge

Modify the program to print:

```
Good Morning, <your name>
```

No hints. Try it yourself.

Chapter 2: Make Decisions and Repeat

TL; DR

- **Conditions** = Asking "True or False?" to make a decision
- **Blocks** = A group of instructions that belong together
- Python uses indentation to define blocks of code
- **Indentation** = Adding space before a line of code to put it "inside" a block
- **Loops** = Repeating instructions so you don't have to write them multiple times
- **while** loop runs as long as its condition holds True
- Conditions and loops together help solve real problems

Think about your daily life for a moment.

You decide things all the time.

If it is raining, you carry an umbrella.
If your phone battery is low, you charge it.
If you feel sleepy, you go to sleep.

You also repeat many things.

You brush your teeth every day.
You count money note by note.
You practice something again and again until you get better.

Programs work in the same way.

They **take decisions** and they **repeat actions.** If we want to say more technically, programs automate decision making and repetitive tasks.

How decisions are taken

Whenever we take a decision, we first check a condition.

If the condition is true, we do one thing.
If it is false, we do something else.

Programs follow the exact same idea.

Write A Program To Know About "Conditions"

Program 2.1: Check whether a person is eligible to vote.

A person is eligible to vote when he/she is 18 or above. Let's break the solution to steps.

1. You need to know the age of person, so, ask it. (use input function)

2. Check condition, if person’s age is 18 or above
3. If yes, then print “You are eligible to vote”
4. Else, print “You are not eligible to vote”

Python has **keywords** (its vocabulary) like “**if**” and “**else**” to write such decision logic. (**Note**: case has to be same. So “if” is correct keyword and “If” or “IF” isn’t)

```
age = int(input("Enter your age"))

if(age>=18):

      print("Congratulations!")

      print("You are eligible to vote")

else:

      print("Sorry, you are not eligible to vote")
```

We already know, input( ) function.

Next is, “if”, a keyword.

And (age>=18). For regular user, ⩾ symbol is difficult to type. So, python makes our life easy and for “greater than or equal to”, it has provided us with >= sign. Such sign is known as operator.

So, even if you read, intuitively, it reads “if age is greater than or equal to 18”.

If this is true, we proceed to the actions in the “if” block.

Before going further code, we need to understand how Python knows which instructions belong to a decision.

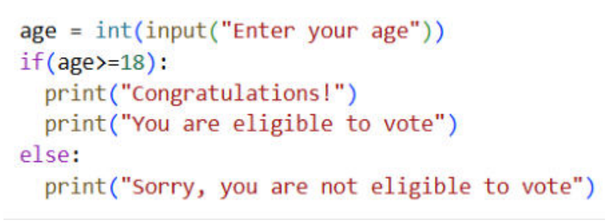
In a human to-do list, we might write:

1. If it rains:
    - Take umbrella
    - Wear raincoat
2. Go to school

Notice how "Take umbrella" is slightly shifted to the right? That tells us it is part of the "If it rains" plan.

Python uses this exact same logic. It is called **Indentation**.

- We use **4 spaces** (or 1 tab) to shift code to the right.
- This creates a **Block** of code.



```python
age = int(input("Enter your age"))
if(age>=18):
  print("Congratulations!")
  print("You are eligible to vote")
else:
  print("Sorry, you are not eligible to vote")
```

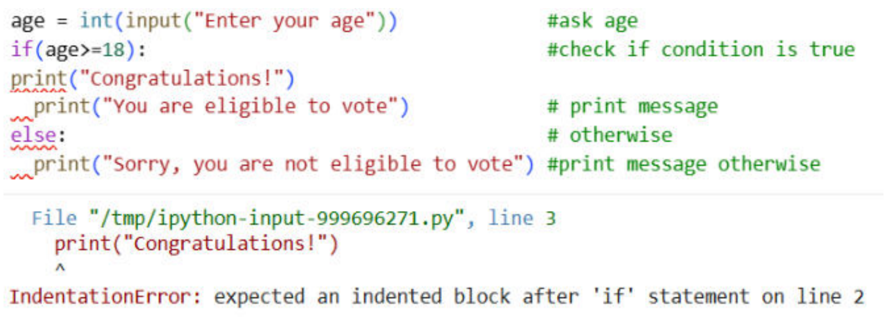
In an adjoining image, we can see code written with correct indentation.

Two lines of print are written below each other and after a gap from the left side

This is indentation.

It shows, two lines belong to same block of "if". Both lines will be executed if the condition of block is true.

Now, let's try to disturb the indentation by shifting "print" to left, just below if.



```python
age = int(input("Enter your age"))                      #ask age
if(age>=18):                                            #check if condition is true
print("Congratulations!")
  print("You are eligible to vote")                     # print message
else:                                                   # otherwise
  print("Sorry, you are not eligible to vote") #print message otherwise
```

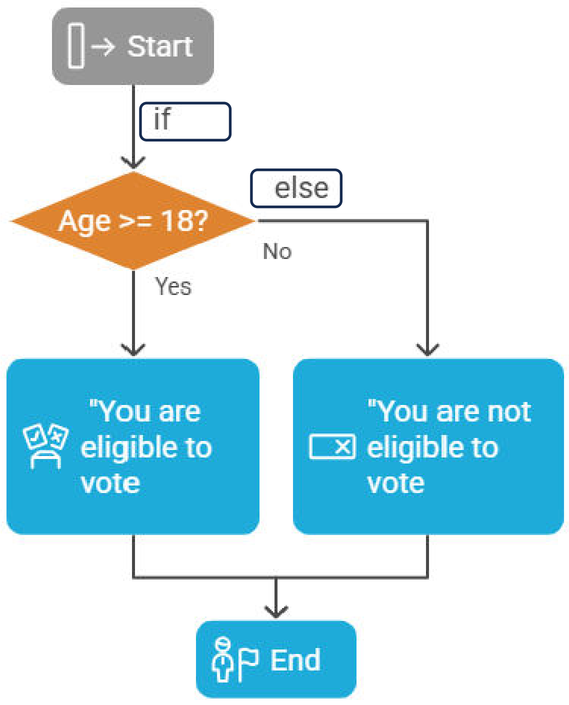
```
  File "/tmp/ipython-input-999696271.py", line 3
    print("Congratulations!")
    ^
IndentationError: expected an indented block after 'if' statement on line 2
```

And....

We get the error.

Error message clearly says, indentation is incorrect.

You should always try to read errors and try to correct those as per error message.



If the age entered by user is greater than or equal to 18, execute the block of "**if**".

Else, execute the block of "**else**". Pretty much intuitive.

This is how we turn the flow of code with if and else blocks.

If the condition is true, only the code in if block is executed.

If the condition is false, code in else block is executed.

Progress Unlocked

- You know how to check conditions
- You can decide what to do if certain condition is satisfied or not satisfied
- You know what are code blocks and indents, why indentation is important

Condition to check (age > = 18) is actually an **expression**.'

It results in bool (Boolean) type of value.

This value can be **True** or **False**.

If the condition is satisfied, expression returns **True** value otherwise, it returns **False** value.

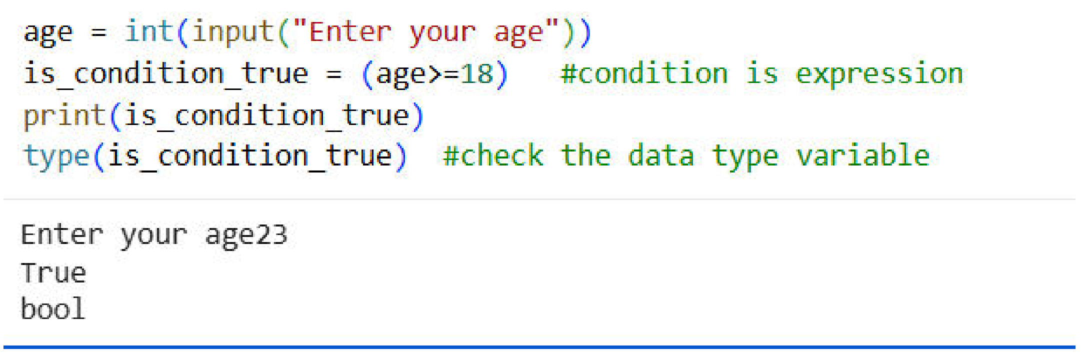
We can actually equate value of this expression to a variable.

See the example below:

```
age = int(input("Enter your age"))
is_condition_true = (age>=18)


if(is_condition_true):

  print("Congratulations!")

  print("You are eligible to vote")

else:

  print("Sorry, you are not eligible to vote")
```

This code prints exactly same result as the previous code.'

See the code below:



```
age = int(input("Enter your age"))
is_condition_true = (age>=18)   #condition is expression
print(is_condition_true)
type(is_condition_true)  #check the data type variable
```

```
Enter your age23
True
bool
```

It prints the value that "is_condition_true" variable holds.

Next is, **type( )** function. This function is used to know the data type of variable.

Notice the statements appearing in green, written after "#"

These are the "**comments**", they are not read or executed by our program. We use comments to explain what's going on in program at particular point. This helps reader of the code to understand the code.

Progress Unlocked

- You know conditions are expressions
- You can check type of variable and write comments in code

Write A Program To Know About "Many Conditions"

Program 2.2: Decide result based on marks

If marks are above 60: Good score, if marks are 40 to 59: Pass, if marks are less than 40: fail

Break the program into steps:

1. Ask the user marks- accept input from user
2. Check, if marks are more than 60, print "Good Score"
3. **Else, if** marks are more than 40, print "Pass"
4. Else, print fail.

Now, python has made our life easier. We have special keyword for "**else, if**" situation like in step 3. The keyword is **"elif"** (short form of 'else if')

And we can simply write the code as before.

```
marks = int(input("Enter your marks"))

if marks >= 60:

    print("Good Score")

elif marks >= 40:

    print("Pass")

else:

    print("Fail")
```

Two points to Remember,

1. one **if** can have many **elif**s, and one and only one else associated with it
2. **else** is optional. Use it when you want to handle all remaining cases
3. **if**s can be nested into one another. We do it when we need to make complex decisions. We will see this in future chapters when we will solve logically deeper problems

Comparison

We have already seen how to express "greater than or equal to". Here is, other symbols to compare.

SYMBOL	MEANING
>	Greater than
<	Smaller than
==	Equal to (note we need here two = signs. "=" is used for assignment
>=	Greater than or equal to
<=	Smaller than or equal to
!=	Not equal to

When these symbols are used in expressions (conditions), **True** or **False** is returned.

We can simultaneously check two conditions also.

For this, we use **and** and **or.**

When action is to be taken if both condition needs to be true, we use **and.**

When action is to be taken if any of the two conditions is true, we use **or.**

For example, a new club in town, **Sci-Mate**, has criteria for membership that a member

1. Must be at least 18 years old
2. Must be graduate in science

We use **and** here.

e.g.

```
if (age>=18) and (is_science_graduate):
      membership = True
else:
      membership = False
```

Later, club modified educational criteria, it started allowing science graduates or post-graduates in any stream.

So, this becomes:

```
if (age>=18) and ((is_science_graduate) or (is_postgraduate)):
      membership = True
else:
      membership = False
```

Notice, how we combined (is_science_graduate) or (is_postgraduate) in one bracket. Such that, value of this expression will be calculated first and then checked together with the other condition of age.

It works just like maths. ( ) first.

Now, two tasks for you:

1. What is the data type of variables is_science_graduate, is_postgraduate and membership
2. Write the complete program which user's age, if he/she is science graduate, if he/she is post graduate and then decide if he/she can become member.

Try to write the program. It is a bit complex at this stage. We will solve together at the end of the chapter.

Progress Unlocked

- You know how to write conditions with comparisons
- You can check multiple conditions simultaneously

Repeat with the Loops

You know how to print.

How will you print numbers from 1 to 10?

print(1)

print(2) and so on.. not a good idea!

What if we want to print numbers up to 1000? Or we want to print numbers up to "n" which will be user provided?

We need to automate the task multiple times.

Python provides us "**for**" keyword. Anything in the block of "**for**" will be repeated as per the condition specified.

Write A Program To Know About "Loops"

Program 2.3: Print numbers from 1 to 10.

Plan the program.

We will use single variable for print function. Say, variable is "i", we will use function **print(i)**.

We will repeat this function 10 times.

During every repetition, we will change value of "i" starting from 1 to 10.

For this purpose, we will use function **range( ).**

range ( ) function takes following inputs: 1. Starting number 2. Stopping number (this will not be included in returned range of numbers) 3. Steps (gap between 2 numbers)

"steps" is optional and has default value of 1. If we don't provide any value, 1 will be assumed.

e.g. range(1,5) gives sequence [1,2,3,4] (5 excluded)

range(1,8,2) gives us [1,3,5,7] (steps =2, each next number is ahead by 2)

now, interesting point, even start number is optional! Default value is 0.

range(6) gives us [0,1,2,3,4,5]

Now, how to use range to assign values to variable?

Simple, **`for i in range(1,11)`** works!

So to print numbers from 1 to 10,

```
for i in range(1,11):
     print(i)
```

Note: Notice the colon(**:**) and indentation to create block

for and **in** are both Python keywords.

It simply translates to "**do following tasks for every 'i' whose value is in sequence of numbers returned by the range**"

We can write above line alternatively as "do following tasks **while** 'i' is greater than or equal to 10, at the end of the tasks, increase value of 'i' by 1."

And Python provides us way to code in this dialect too!

As you might have guessed, we need "**while**" keyword, and condition (like in **if**).

Below is the program:

```
i = 1
while (i<=10):
    print(i)
    i=i+1
```

We assign value 1 to i. Then, we check condition with "while", execute block of "while" and at the end, increase "i" by 1.

i = i +1 seems mathematically incorrect! But in programming, this is just short way of saying,

new value of i = old value of i + 1

See, how Python makes our life so easy, allows us to write in short forms!

But it doesn't stop here. You can even shorten this! Just write **i+=1.**

i+=1 is same as (i= i +1)

Can you guess how to reduce value of i ?

Yes, you are right. Replace + by -.

It even works for multiplication and division too!

So, (i* =2) is same as (i = i * 2). Just makes value of "i" twice.

Write A Program To Know About "using loops and conditions together"

Program 2.4: Print all the even numbers from 1 to "n", where "n" specified by user.

Plan the program!

1. Ask user value of 'n' using input function.
2. Now, "n" needs to be a number bigger than 1 otherwise, there is no point in running the loop!
3. So, check if n is greater than 1, if true, run the loop to print even numbers.
4. Else, display message asking to enter number bigger than 1.

Program so far is:

```
n = int(input("Enter a number"))

if(n>1):

  #run loop to check even numbers

else:

  print("please enter number greater than 1")
```

Now, let's write logic to find even numbers.

1. Loop numbers from 1 to n
2. In every loop, check if number is divisible by 2 (number %2 ==0)
3. If true, print the number

Notice, the expression **number%2** returns the remainder when divided by 2. Condition checks if that remainder is 0.'

Also note that we have converted input into int type using **int( )**.

This can be written as below

```
  for number in range(1,n):

      if((number%2)==0):

           print(number)
```

Now, we replace this **"for"** block inside the if block and we are ready with the whole program.

```
n = int(input("Enter a number"))

if(n>1):

     for number in range(1,n):

          if((number%2)==0):

                print(number)

else:

  print("please enter number greater than 1")
```

Progress Unlocked

- You know how to repeat actions using “for” loop
- You know how to repeat actions using “while” loop
- You know **range( )** and **in**
- You can combine conditions with loops

We have learnt a lot till now.

It’s time to handle now a bit difficult code that I had asked you to do.

Did you succeed? If not, let’s do it now.

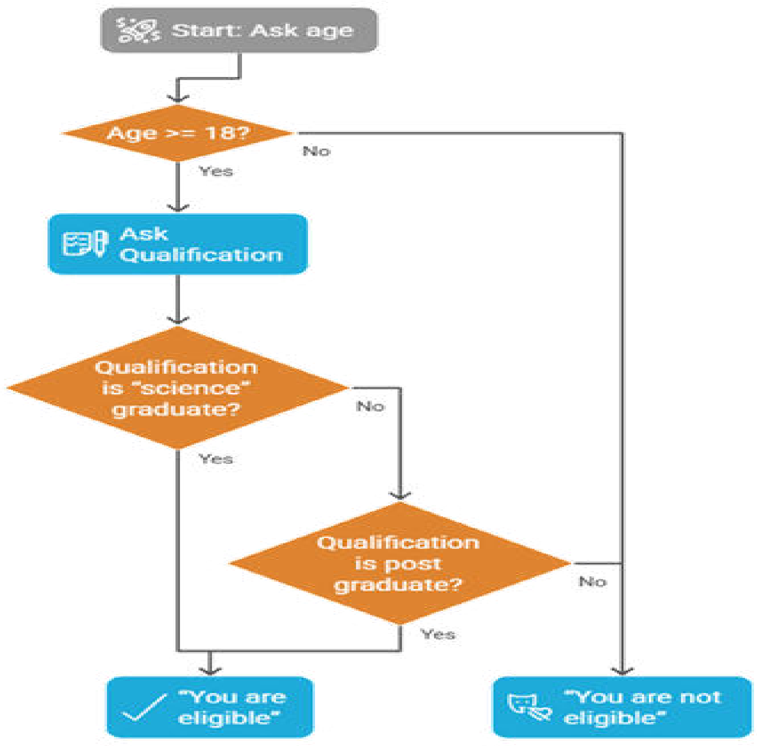
Problem: Write a program to check is user can be the member of a new club in town, **Sci-Mate**

Conditions are:

i) User must be at least 18 years old
ii) User must be science graduate or Post graduate (any stream)

Let’s plan the code step-by-step.

1. Ask user age: By now, it’s simple with input ( ) function. (just don’t forget to convert it to int)
2. Ask user qualifications. Now, as a designer of program, we should ask right questions to user.
   i) It should be easy for user to enter answer
   ii) It should be easy and unambiguous for us(programmer) to process.
3. In the view of this, we can ask user question “Are you a science graduate?”, if user enters positive, we will grant membership. If not, we will ask next question, “Are you a post graduate (in any stream)?”. In case of positive reply, we grant membership, else, we don’t.
4. But what should user reply? User can reply “yes”, “Yes”, “YES”, “Yes, I am” anything and that will make our processing difficult.
5. For this reason, we will direct user to enter only as “y” for “yes” and “n” for “no”. In case of other replies, we will prompt user to enter ‘y’ or ‘n’.



Flow of the code is as shown in an adjoining flow chart.

In future, we will keep improving our program for **Sci-Mate** making it better for user. As of now, we will work as per this plan.

Note that, we have blocks "if" inside one another.

This is called as nested blocks.

Let's start with topmost block of "if"

We will declare **membership** variable before "if" block starts.

The reason for this is, '**scope of variable**'.

To put in simple terms, if we declare "membership" variable out of all if blocks, all if blocks can access it.

If we declare it inside any if block, it will be accessible within that "if" block only.

You will need to think it intuitively.

You can intuitively compare it with the orders issued in office.

Head of the company is outside all blocks, so, can issue commands for all of the company.

Head of the department is part of a department, so, can issue commands to all the teams in his department.

Head of the team is part of team and thus can issue commands within his team.

NOTE: For now, think of it this way. We will learn exact scope rules later.

First stage: check age - Let’s write topmost block (Refer to the flowchart)

```
age = int(input(“Enter your age”)
membership = False
if (age>=18):
    # check qualification conditions
else:
    membership = False
```

Second stage: Check if user is science graduate- We will add qualification logic.

```
age = int(input(“Enter your age”)
membership = False
if (age>=18):
    is_science_graduate = input(“Are you a science
graduate?(y/n)”)
    if(is_science_graduate==”y”):
        membership = True
    elif(is_science_graduate==”n”):
        #check other post graduate
    else:
        print(“please enter either ‘y’ or ‘n’ only”)
else:
    membership = False
```

We check input from user. If input is “y”, user is eligible for membership. Otherwise, input might be “n” or anything else. In case of anything else, which is not valid input, we ask user to stick to “y” or “n”

If response is “n”, we check if user is post graduate.

Final Stage: Finally, we give our verdict.

```
age = int(input("Enter your age")
membership = False
if (age>=18):
    is_science_graduate =input("Are you a science graduate?(y/n)")
    if(is_science_graduate=="y"):
        membership = True
    elif(is_science_graduate=="n"):
        is_post_graduate = input("Are you post graduate?(y/n)")
        if(is_post_graduate =="y"):
            membership = True
        elif(is_post_graduate=="n"):
            membership = False
        else:
            print("please enter either 'y' or 'n' only")
    else:
        print("please enter either 'y' or 'n' only")
else:
    membership = False
if(membership):
    print("You are eligible for membership")
else:
    print("You are not eligible for membership or You entered
invalid input!")
```

We have informed user every time that response should be "y" or "n" only. These are good practices.

You should never take user for granted.

Progress Unlocked

- You have engineered a program that solves some real problem and makes real decisions!! Congratulations!

We have covered a lot in this chapter. Do not worry if you forget syntax. Thinking correctly matters more.

Concepts we learnt revisited:

- **Conditions** are used to change the flow of code.
- Different blocks of code are executed depending on condition is True or False
- We use keywords like **if, elif** and **else** to change the flow of code depending on the condition
- We can combine conditions with **and** and **or**
- We can nest conditions within one inside creating multi-level decision framework
- We can create sequence of numbers with **range( )**
- We use **in** operator to assign a variable different values from the sequence returned by range( )
- We use **for** block to repeat the tasks.
- We can also use **while** to repeat the task till certain condition holds true
- We intuitively learnt about scope of the variable

1-Minute Challenge
Write a program to print squares of even numbers from 1 to 10.

Hint: Use range(1,11), for loop to iterate through range and if condition to check even number.

Chapter 3: Make Programs Better

TL; DR

- **Collections** = Variables that can hold more than one value at a time.
- **Iteration** = Visiting every item in a collection one by one using a for loop.
- **classes** = A blueprint that describes the properties and methods
- **objects** = Items that follow blueprint of classes
- **List []** = An ordered collection of items. (Think: a list of names of students).
- **Index** = The position number of an item in a list (Starts at 0).
- **Tuple ()** = A "locked" list that cannot be changed. (Think: A Sealed Envelope).
- **Dictionary {}** = A collection of data with labels (Keys) instead of numbers. (Think: A Phonebook).

Data with Actions, together

Till now, we have written programs where data and functions mostly lived separately. We created values, stored them in variables, and passed them to functions.

e.g. `print(name)`

But as programs grow, this separation starts feeling limiting.

Some actions naturally belong to specific data.

Consider this statement:

```
marks.append(85)
```

Here, append() is not called on its own. It is called **on** marks.

This tells us something important.

"marks" is not just data. It also knows what actions are allowed on it.

This idea appears again and again in Python (and in Programming)!

Objects and Methods

When data and its related actions live together, we call such a value an **object**. An action that belongs to an object is called a **method**.

Methods are called using a dot after the object name.

```
object.method()
marks.append(85)
```

"marks" is **object** and "append()" is a **method.**

This idea helps us to model real world much easily. Let's see how-

Intuition: Classes and Objects

In the world around, you can find many objects of similar type. we group things by common properties and actions. Such types are an idea of **Class** in programming.

For example, you have 2 pet cats, "*Snowy*" and "*Gingie*". Being from same species 'Cat', You can describe both of them in similar way. So, Cat here is a **Class** from programmer's view.

They have certain properties like, fur color, name, height, weight etc.

They do certain actions like playing, eating or meowing.

These properties and actions are common among cats.

The **idea** of a "cat" defines these properties and actions. Individual cats like "*Snowy*" or "*Gingie*" are actual examples of that idea.

This blueprint of properties and actions is called **Class.** So, "cat" is a class and individual cats like "*Snowy"* or "*Gingie",* which follow this blueprint are called **objects** of class.

Like in real world, almost everything is object; in Python too, almost everything is object.

Classes have methods. Methods are similar to functions, but they belong to a specific object.

That is why they are called using a dot.

```
snowy.eat(food)

gingie.play()
```

Each method call applies only to that object.

Why This Matter Now?

Lists, tuples, and dictionaries that we will study in this chapter are not just collections of data.

They are objects.

They come with their own methods that allow us to:

- add items
- remove items
- inspect data
- transform data

Understanding this makes the upcoming code clearer and more meaningful.

Class: Cat

Properties: Name, Fur_color, Height, Age

Methods: Eat, Play, Meow

Objects:

Properties: Name: Snowy Fur_color: White Height: 15 cms Age: 14 months	**Properties:** Name: Gingie Fur_color: Ginger Height: 13 cms Age: 12 months
Calling Methods	Calling Methods
Snowy.catch(mouse) Note: eat( ) method is called just like function and has "mouse" as argument	**Gingie.eat(food)** Note: eat( ) method is called just like function and has "food" as argument
Snowy.meow( )	**Gingie.play( )**

This is an intuitive introduction to objects and classes.

Progress Unlocked

- You now can understand syntax like obj.method( )
- You intuitively learned the concepts of class and object.
- You are ready to learn about collections

Collections

Remember our club **Sci-Mate**?

Suppose, this Sunday, club has a special event and club wants to display the welcome message to 4 new members of club with their name.

For now, we will just display the message with print( ).

How can we do this? Let's brainstorm the ideas.

Write a print( ) message 4 times with the name of each member “Welcome! John”. → that’s bad idea. We want to make programs efficient

Write a **for** loop → Good idea! But how to have names of members in **for** loop?

Do we need to create 4 variables like member_1, member_2, member_3, member_4? → looks difficult to manage! Won’t work inside loop as every variable is different.

So, what’s the solution?

Can we store many values together and treat them as one unit?

Yes. That is exactly what **collections** do.

Lists: First Collections

A **list** is a collection of values stored in a specific order.

Creating a list is very easy. Just use [ ] and all the values in between them.

In our case,

members = [“John”,“Jane”,“Jack”,“Jill”]

That’s it! List of members is ready!

Here,

- “members” is one variable
- It stores multiple values
- Values are inside square brackets
- Values are separated by commas

Ok! That’s good. But how can we use it in the loop?

1. You can access items of the list with a number. This number is called **index** and it starts with 0. It is based on the position of the item in list.
2. First item of list has index 0. So, index of “John” is 0, index of “Jane” is 1 and so on
3. You just use [ ] to access the item at specific index, e.g. **members[0]** is used for “John”, members[1] is used for “Jane”
4. You can get how many items are in list using function **len( )**. e.g. **len(members)** gives us 4

And that’s clear now.

We will use **in** operator to access items of “members” list. And here is the code!

```
members= ["John","Jane","Jack","Jill"]

for member in members:

  print("Welcome!",member)
```

Processing the lists

We have stored names of the members in the list.

Suppose, during the operation of club, various situations arise and we have to process the list as per requirement. Lists provide bunch of methods for this.

Let’s handle situations one-by-one.

As of now, “members” list is [“John”,”Jane”,”Jack”,”Jill”]

Sr no	Situation	Solution	Output
1	A new member joins the club, named “Johnny”	members.append(“Johnny”)	[“John”,”Jane”,”Jack”,”Jill”,”Johnny”]
2	“Jill” decides to leave the club membership	members.remove(“Jill”)	[“John”,”Jane”,”Jack”,”Johnny”]
3	A new member joins club, also named “John”, he is the second John	members.append(“John”)	['John', 'Jane', 'Jack', 'Johnny', 'John'] **Note:** identical items can be added to the list
4	Club manager wants to know how many “John” are there	members. count(“John”)	2 Counts the occurrences of the specified item
5	To identify the members with same name, club manager decides to add last name initial to newer members of same name	members[4] = “John D” 4 is the index of the newly added John	['John', 'Jane', 'Jack', 'Johnny', 'John D']
6	Manager wants sorted list of names	members.sort()	['Jack', 'Jane', 'John', 'John D', 'Johnny']

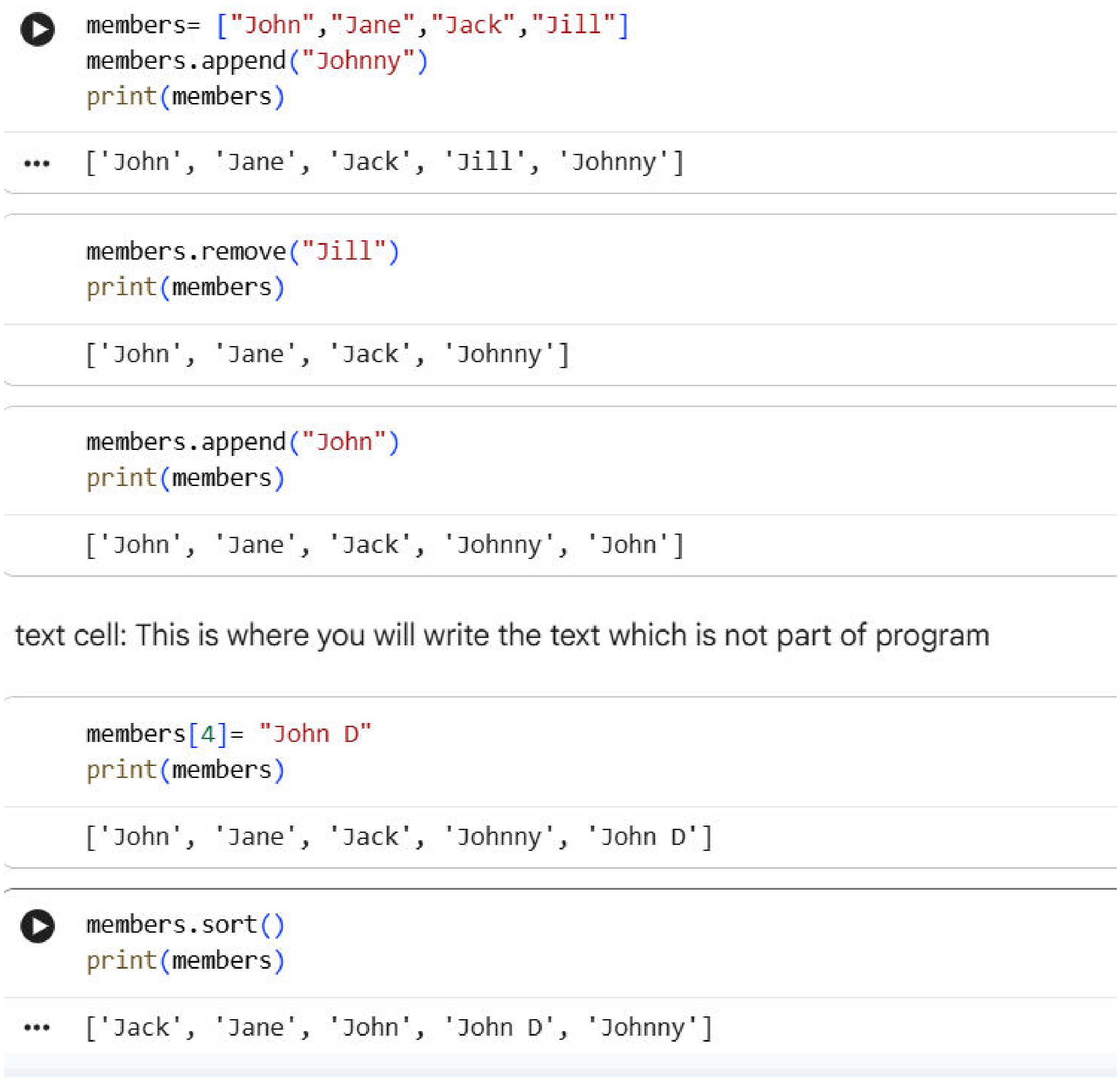
There are some other methods too.

For example, members.clear( ) will clear all the list removing all the members.

If we remove “John” using members.remove(“John”) while having multiple “John”s in the list, the first occurrence of “John” will be removed.

Also, members.index(“John”) will give index of the first occurrence of the “John”

List can contain items of mixed data types. e.g. [“John”, 2025,True] → Allowed and Valid!



```
members= ["John","Jane","Jack","Jill"]
members.append("Johnny")
print(members)
```

```
['John', 'Jane', 'Jack', 'Jill', 'Johnny']
```

```
members.remove("Jill")
print(members)
```

```
['John', 'Jane', 'Jack', 'Johnny']
```

```
members.append("John")
print(members)
```

```
['John', 'Jane', 'Jack', 'Johnny', 'John']
```

text cell: This is where you will write the text which is not part of program

```
members[4]= "John D"
print(members)
```

```
['John', 'Jane', 'Jack', 'Johnny', 'John D']
```

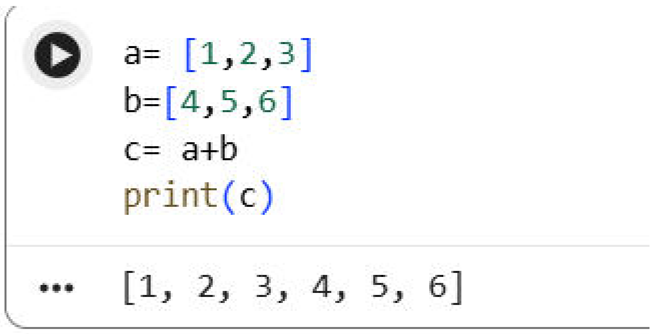
```
members.sort()
print(members)
```

```
['Jack', 'Jane', 'John', 'John D', 'Johnny']
```

We can see the code executed for above situations and the results.

We can also add 2 lists.

Result will be as below:



```
a= [1,2,3]
b=[4,5,6]
c= a+b
print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

Write A Program To Know About “slicing the list”

Program 3.1: Given a list of numbers, sort it, extract the middle one third values, and calculate their average. (List will have items in multiple of 3)

For example, given the list [2, 4, 1, 6, 8, 9], when we sort the list, we get [1,2,4,6,8,9]. Middle one third is [4,6]. Output is average i.e. 5

As our regular strategy, break down the problem.

- Start with a list of numbers.
- Sort the list.
- Find how many values belong to one third of the list.
- Use slicing to extract the middle one third values.
- Calculate the average of the extracted values.
- Print the result.

How to find middle 1/3rd? We know that total items in list will always be in multiples of 3.

If the list has 9 items, middle one third will start at 4th item (index 3) till 6th item(index 5).

If total items in list are ‘n’ then, one-third portion will start at n/3+1 (4th position)

And it will end at 2*(n/3) i.e. at 6th position.

Since list are 0 based, we need to extract the portion of list from index 3 to index 5.

We can extract such part of list using “:”

numbers[3:6] will give us the items from 3rd index to 5th index (6 excluded, like in range).

Now, let’s code:

```
numbers = [2, 4, 1, 6, 8, 9]

numbers.sort()

n = len(numbers)

one_third = n // 3

middle_values = numbers[one_third : 2 * one_third]

average = sum(middle_values) / len(middle_values)

print(average)
```

Note:

numbers[2:] → gives you slice of list from index 2 to last item
numbers[:4] → gives you slice of list from start to index 3
numbers[:-1]→ gives you slice from start to the end except last item! Interesting, isn’t it?

Tuples

A **tuple** looks similar to a list, but it is **fixed**.

```
days = ("Mon", "Tue", "Wed", "Thu", "Fri")
```

Tuples use round brackets.

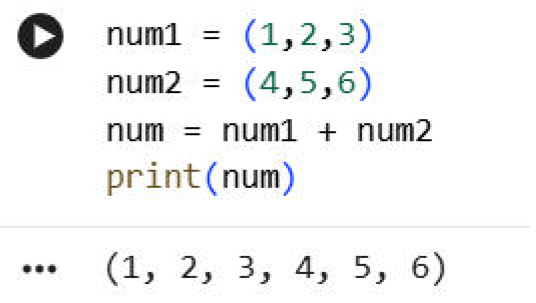
Accessing values of tuple works exactly like list. **days[1]** gives us 2nd value i.e. "Tue".

Important point about tuples is **we cannot change the values in tuple.**

```
days[0] = "Sunday"
```

This gives error!

Only way you can add items to tuples is by joining tuples.



```
num1 = (1,2,3)
num2 = (4,5,6)
num = num1 + num2
print(num)
```

```
(1, 2, 3, 4, 5, 6)
```

Progress Unlocked

- You learnt about purpose of collections
- You know about most important collection "Lists"
- You know about tuples

Dictionary

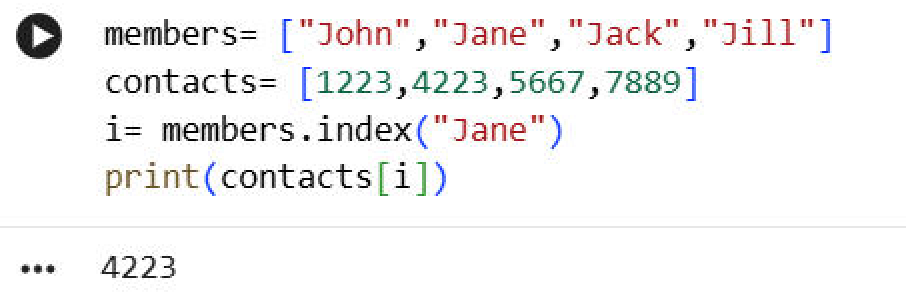
Now, **Sci-Mate** club wants to store the contact numbers of the members.

They tried to store the numbers in list.

```
members= ["John","Jane","Jack","Jill"]
```

```
contacts= [1223,4223,5667,7889]
```

But whenever manager wanted to find the number of a particular member using name, he had to find the index of member from **members** list and use it in **contacts** list.



```
members= ["John","Jane","Jack","Jill"]
contacts= [1223,4223,5667,7889]
i= members.index("Jane")
print(contacts[i])
```

```
4223
```

It was quite a long route!

What if, we can access contact number just by using contacts[“Jane”]?

This is where **dictionary** comes to rescue!

Dictionary is way to store data with labels.

Data is stored in key-value format and accessed using keys. **Key** is for labels and **Value** is for data.

We use curly brackets { } for dictionaries.

```
phone_book = {
   "John":1223
   ,"Jane":4223
   ,"Jack":5667
   ,"Jill":7889
}
```

phone_book[“Jane”] gives us 4223!

What if we want to add “Johnny”’s contact?

It’s simple to add new item to dictionary.

phone_book[“Johnny”] = 1212

It simply adds “Johnny”’s contact to phone_book.

When Jill decides to leave the membership, we simply use **phone_book.pop(“Jill”)**

If John wants to change the contact number?

Easy! Just assign new number to phone_book[“John”].

phone_book[“John”] = 5555

Using dictionary in loops comes in variations.

If we simply use

```
for x in phone_book
```

We x will loop through key values.

It is same as using:

```
for x in phone_book.keys()
```

If we use

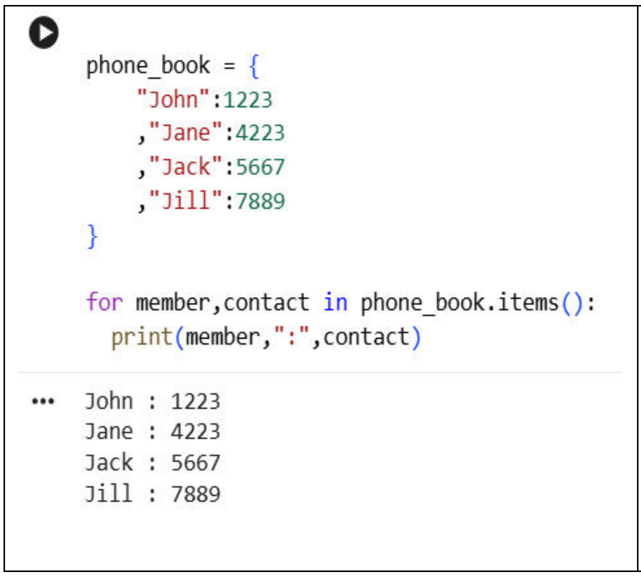
```
for x in phone_book.values()
```

x will loop through values.

And, we can actually loop through both keys and values.

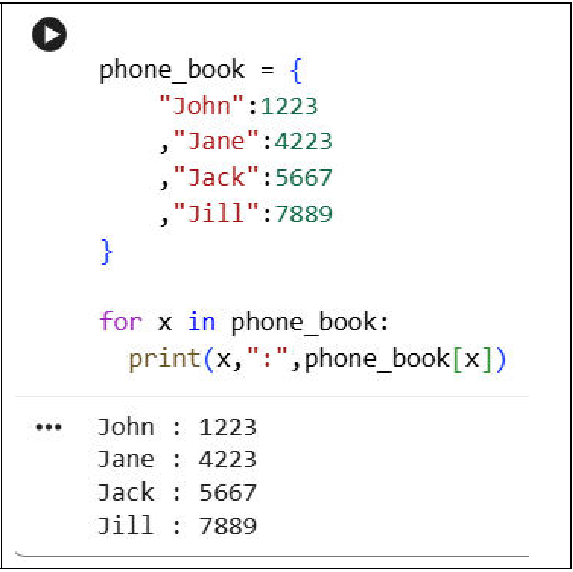
Simply use

```
for member, contact in phone_book.items()
```



```
phone_book = {
    "John":1223
    ,"Jane":4223
    ,"Jack":5667
    ,"Jill":7889
}

for member,contact in phone_book.items():
  print(member,":",contact)
```

```
John : 1223
Jane : 4223
Jack : 5667
Jill : 7889
```



```
phone_book = {
    "John":1223
    ,"Jane":4223
    ,"Jack":5667
    ,"Jill":7889
}

for x in phone_book:
  print(x,":",phone_book[x])
```

```
John : 1223
Jane : 4223
Jack : 5667
Jill : 7889
```

Notice that, both the codes provide same output!

Now, **Sci-Mate** club decides to store more details of each member like name, contact, address, membership id, membership fees paid or not etc.

Dictionary can be really useful in such situation.

We can create dictionary to store values of one member like this-

```
member= {

    "Membership_ID":1,

    "Name":"John",

    "Phone": 1223,

    "Address":"South block, Gotham",

    "Fee_status": "paid"

}
```

Write A Program To Know About "nested dictionaries"

Now, we can store the more details of a member in very structured way.

But, how can we store details of multiple members and yet, able to access the details of with just membership ID?

We use dictionary within the dictionary.

One dictionary will store details of the member.

Other dictionary will dictionary of particular member as value and membership ID as key!

Check the program below:

Dictionaries storing membership details of “John” and “Jane”

{
    "Name":"John",
    "Phone": 1223,
    "Address":"South block, Gotham",
    "Fee_status": "paid"
}
	{
	    "Name":"Jane",
	    "Phone": 4223,
	    "Address":"North block, Gotham",
	    "Fee_status": "Due"
	}

We nest both the dictionaries into one.

members = {

“ID1”: {
   "Name":"John",
   "Phone": 1223,
   "Address":"South block, Gotham",
   "Fee_status": "paid"
},

ID2: {
   "Name":"Jane",
   "Phone": 4223,
   "Address":"North block, Gotham",
   "Fee_status": "Due"
}
}

Keys are ID1 and ID2, values are the details of the members.

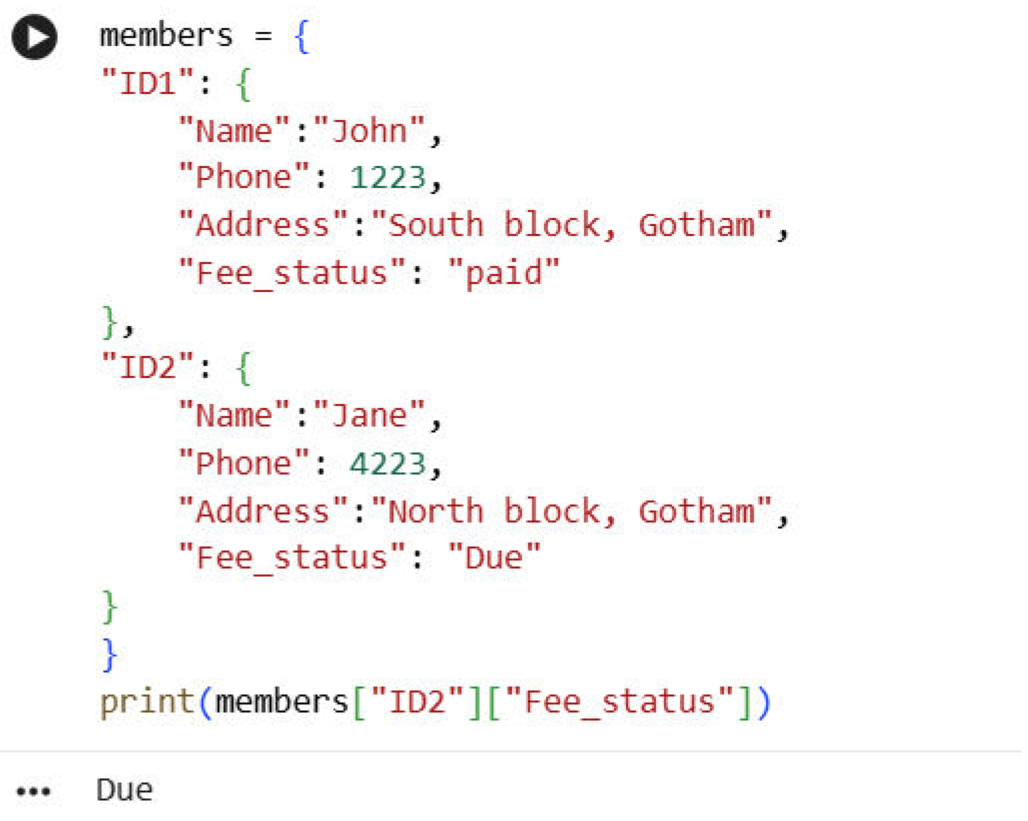
You can read it again, to grasp it.

This is a bit advanced but very useful concept.

And, if we want to know the **Fee_status** of the member with ID2, what we will do?

Simple, access member with ID2 using key “ID2”: members[“ID2”]

And, access Fee_status of the member using key “Fee_status”: members[“ID2”][“Fee_status”].



```python
members = {
"ID1": {
    "Name":"John",
    "Phone": 1223,
    "Address":"South block, Gotham",
    "Fee_status": "paid"
},
"ID2": {
    "Name":"Jane",
    "Phone": 4223,
    "Address":"North block, Gotham",
    "Fee_status": "Due"
}
}
print(members["ID2"]["Fee_status"])
```

```
Due
```

Progress Unlocked

- You know about key, value and dictionary
- You can use dictionary to efficiently store and access data

Concepts we learnt revisited:

- ❖ **Collections** are used to store multiple data values into single variable
- ❖ **Lists, tuples, dictionaries** are some important collections
- ❖ You can manage values in list, sort the list, access subset of list
- ❖ You cannot change the values in tuples
- ❖ **Dictionary** stores value in key-value format. This makes dictionaries super useful to write efficient, quick and clean code.
- ❖ **Lists** are created using [ ], **Tuples** are created using ( ) and **Dictionaries** are created using { } with ":" used for assigning value to key.

1-Minute Challenge

Like nested dictionaries, we can have nested lists too. We primarily use them to store 2-dimensional data.

Write a nested list to store names and contacts of the members of **Sci-Mate.**

Hint: contacts = [["John",1223],["Jane",4223],...] works well.

Contact numbers are at 2nd position i.e. at index 1. What will contacts[3][1] will return? **Try!**

Chapter 4: Manage the Program Better

TL; DR

- **Functions** = A named block of code that performs a specific task.
- Functions are reusable, functions make code easy to manage and reuse
- **Arguments =** Inputs you give to a function.
- **Return =** The output a function gives back.
- Functions teach us how to organize actions.
- Classes teach us how to organize data and actions together.
- **Constructor** = A special method that creates object and sets its initial data

We are already using functions and objects(classes).

It's time to write our functions and classes.

We have seen, how beautifully functions help us cover a lot of code into single line.

A **function** is:

- A named block of code
- That performs one specific task
- And can be used multiple times

Think of a function like a machine.

You give input. - It does work. - It gives output.

Defining Functions and Calling Functions

Until now, we have been calling functions. e.g. `print("Hi")`

When we call function writing function name, parenthesis ( ) and arguments, the code of function actually runs at that point.

But where is that code of the function? It is written in the definition of function.

Function is defined using keyword **def** and function name.

While naming the function, we specify the arguments too.

```
def welcome_member(name):
    print("Welcome to Sci-Mate Club!", name)
```

This is the simple example of a function.

Here, name is an argument.

Some functions like print( ) simply do a certain task and do not return any value as output.

Function like “len( )” which we used to count the number of items in list, return certain value as output.

Write A Program To Know About “functions returning value”

Let’s say, we want to write a function which calculates the age of the applicants for the membership of **Sci-Mate** club based on age.

```
birth_year = int(input(“Enter Year of Your Birth”))

age = get_age(birth_year)
```

Here, we expect age to be returned by the function get_age( )

We will now define the function get_age( ).

```
def get_age(year):

      applicant_age = 2025-year

      return applicant_age
```

We defined the function with keyword “**def**” and argument.

(**NOTE: Arguments** and **Parameters:** Both are slightly different concepts but in-depth explanation is out of scope of this book. We can use them interchangeably)

Inside the function, we used argument value to calculate age of the applicant.

Then we returned the value using keyword **“return”.**

NOTE: Whenever “**return**” line is executed in function code, execution of function stops and we come back to the code where the function was called.

e.g.

```
def get_age(year):

      applicant_age = 2025-year

      return applicant_age

      print(“This is the line after return”)
```

In above code, “This is the line after return” will never be printed. It is after **return** line so, never executed.

Now let’s write a function to check if one is eligible for membership based on age.

```
birth_year = int(input(“Enter Year of Your Birth”))

age = get_age(birth_year)

is_age_eligible = check_age_eligibility(age)
```

define the function.

```
def check_age_eligibility(age):
     if(age>=18):
       return True
    else:
      return False
```

Here, return is inside if or else block. So, return statement will be executed if block condition is satisfied and then only program will also exit.

Challenge: Just like age eligibility, can you write educational qualification checking function too?

Try it!

Plan what you need to ask to user, how you will ask it to user, how you will check the conditions and when you will return True or False.

Just to remind you, eligibility criteria is either science graduate or any post graduate.

Three important concepts related to functions:

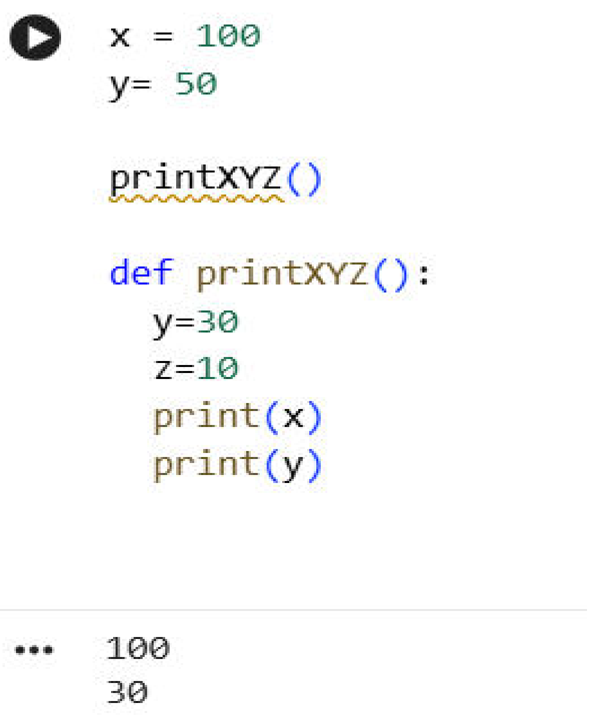
We need to know 3 important concepts related to function:

Scope: Recall, we had intuitively explained scope of variable with office hierarchy example.

To remember simply, a variable inside a function is not available outside.

It is called local variable.

Variable outside the function can be used inside the function and it is called global variable.



```
x = 100
y= 50

printXYZ()

def printXYZ():
  y=30
  z=10
  print(x)
  print(y)
```

```
100
30
```

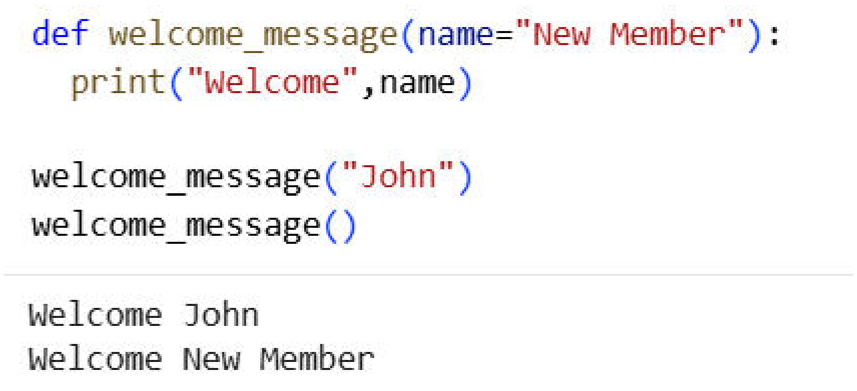
As we can see in the image,
x is outside the function, so function can access it.

y is outside the function (global) and also inside the function (local).
We can clearly see that function prints local y ignoring global.

z is totally local to function. If we try to access it outside the function, we will get the **error**.

Default parameters (arguments): We can assign default value to function parameters(arguments). If we call function without argument, the default value will be used.

We provide default value during the definition of the function.



```
def welcome_message(name="New Member"):
    print("Welcome",name)

welcome_message("John")
welcome_message()
```

```
Welcome John
Welcome New Member
```

We can see in image, when argument “John” is provided, message is printed with the name of John.
When we didn’t provide any argument, message is printed with the default value, which is “New Member”

Keyword arguments: We can call functions with the arguments’ names too. These are known as keyword arguments or named arguments.

For example, get_age(1990) is same as get_age(birth_year=1990).
Remember: birth_year is the parameter/argument name used in definition of the function.

Take an example, if a function definition reads like

```
def calculateBMI(height, weight):
```

Then whenever we call like **calculateBMI(175,75)**; 175 will be assigned height and 75 will be assigned to weight. It follows the sequence in the definition of function.

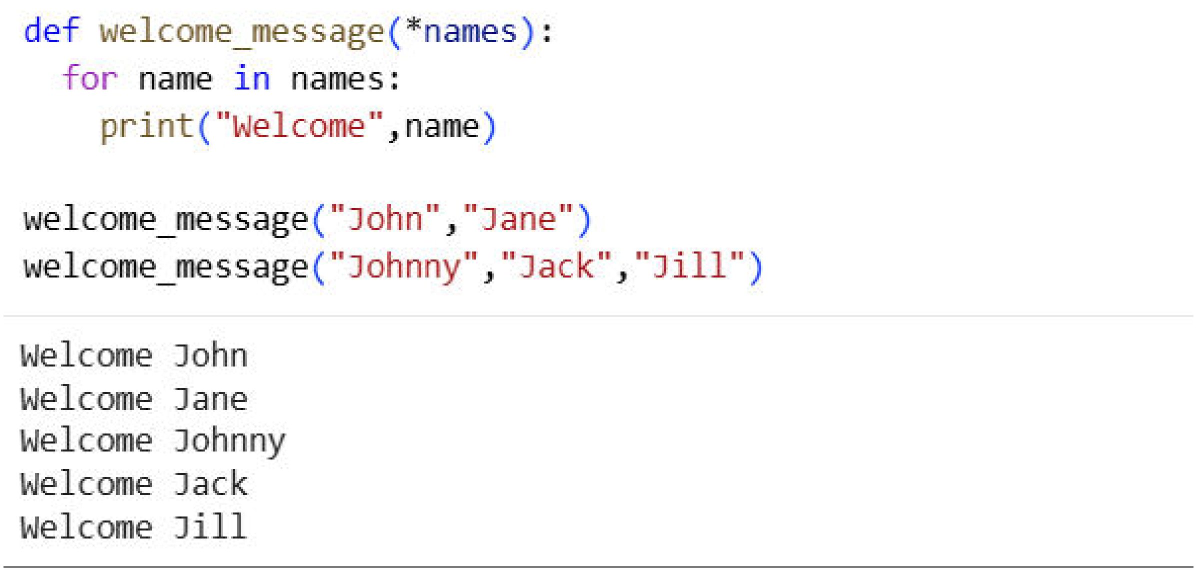
However, we have liberty to change positions if we are using keyword based calls.
So, **calculateBMI(weight=75,height=175)** is correct and valid.

We can also force users to use ‘keyword’ arguments.
For this, we will need to use * in definition of the function.

```
def calculateBMI(*,height, weight):
```

All the arguments after ***** have to be specified with keyword.

Can you have variable number of arguments?
Yes. We use ***argument_name** for this purpose and treat it like collection in definition of the function.



```
def welcome_message(*names):
  for name in names:
    print("Welcome",name)

welcome_message("John","Jane")
welcome_message("Johnny","Jack","Jill")
```

```
Welcome John
Welcome Jane
Welcome Johnny
Welcome Jack
Welcome Jill
```

We can see in image, when we use ***names**, we can use function with any number of arguments.

All these details may seem too much for now, but we will also revisit them while writing our **Sci-Mate** application.

Progress Unlocked

- You know how to write your functions and return value
- You know default arguments, variable number of arguments, named/keyword arguments and scope of variable- you know too much now!

Write A Program To Know About "import"

In our age computation program, we simply calculated age by subtracting birth year from 2025.

Now, program isn't made to run only in 2025. We need to calculate current age using current year.

We can calculate current age using **datetime** class.

But this datetime class isn't by default available in Python. You will need to "import" it, just like we import some goods which are not produced in our country.

What do we need to import? We need to import "Module"

Module is an already written file with python code which has classes, functions, variables etc.

We can use these classes or functions if we import the specific module.

How to use the class in module? Our old familiar syntax of **"."**

Simply, with syntax: **`name_of_module.name_of_class`**

Can you guess how to use method of the class in module now?

Yes, you are right. With **"."** again.

Syntax is: **`name_of_module.name_of_class.method`**

When there are many modules together, we call it "**Library".** Usually, libraries are aimed at specific purpose.

Python has rich set of libraries for various purpose. Our all efforts of re-inventing the wheel are saved if libraries or modules exist for some purpose.

Someone has already written code for us to calculate current date and time.

We just need to import that code (or that module).

We import using the keyword **import**.

So, let's import **datetime** module

```
import datetime
```

Now, typing "datetime" name every time can be tiresome. So we can also assign alias/short name to module while importing.

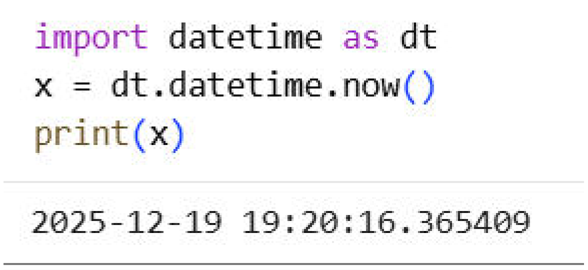
Let's decide short name for "datetime" as dt.

So, our line becomes

```
import datetime as dt
```

This Module has class **datetime** (yes, same name!)

And class has method **now( )** that returns current date time in full format.



```
import datetime as dt
x = dt.datetime.now()
print(x)
```

```
2025-12-19 19:20:16.365409
```

Looks something like this in the image.

We can also format this date returned; I will leave it to you to search and implement that part as an exercise (do it after you finish the book).

As of now, we just need year to calculate date of birth.

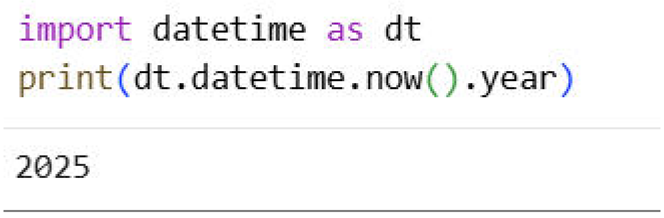
It's simple. Use 'year' property of the object returned by now( ).

So, **x.year** works fine here.

Let's cut some steps.

Instead of assigning value to some variable, we will directly use `dt.datetime.now().year`.

See what do we get



```
import datetime as dt
print(dt.datetime.now().year)
```

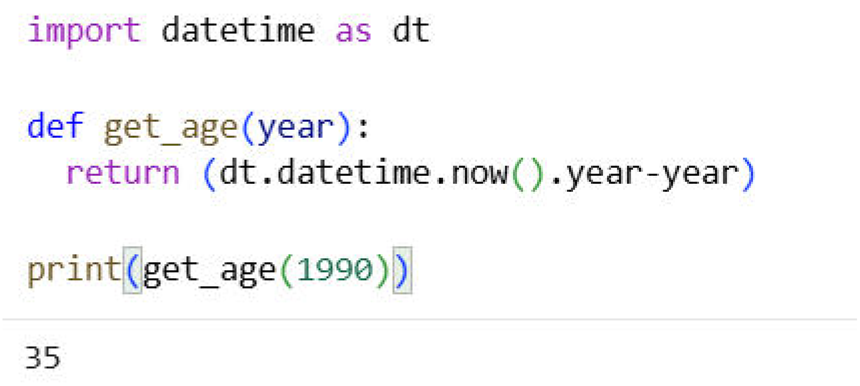
```
2025
```

So, it's fine!

Now even in the function get_age(); let's cut some steps return statement.

```
import datetime as dt

def get_age(year):

  return (dt.datetime.now().year-year)
```

We can see in the image below, this works perfectly fine.



```
import datetime as dt

def get_age(year):
    return (dt.datetime.now().year-year)

print(get_age(1990))
```

```
35
```

So, here we learn how to use 'import' and how to calculate the current age.

Challenge: Learn more about datetime. Use Google or LLM.

We can create a new datetime object with the code below:

```
x = datetime.datetime(2025, 12, 20)
```

Find the methods that we can use on "x" to get month and date.

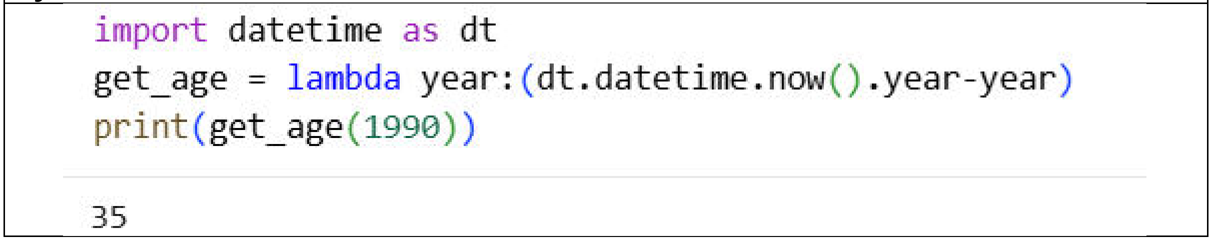
Progress Unlocked

- You know how to import modules and libraries and how to use them
- You can make Sci-Mate application now more realistic and useful.

Optional reading:

We can even shorten get_age( ).
*We can use the keyword **lambda**. It is typically useful for such small methods.*
*We do not use **return** keyword. We just use an expression which calculates value to return.*
See the example below in the image for the syntax.

Syntax for lambda:



```python
import datetime as dt
get_age = lambda year:(dt.datetime.now().year-year)
print(get_age(1990))
```

```
35
```

Challenge: Let's do some exercise:

Below is a code that calculates average of the numbers in list.

Rewrite the above program by splitting it into **two functions**:

```python
numbers = [10, 20, 30, 40, 50]
total = 0
for n in numbers:
    total = total + n                  #calculates total
average = total / len(numbers)         # calculates average
print("Total:", total)
print("Average:", average)
```

*[(**hint**: you can write functions `total()` and `average()`]*

If this feels like a lot, do not worry. You do not need to remember everything now. You just need to recognize these patterns when you see them again.

Classes and Objects

We now know two things about classes:

1. Classes combine data and methods
2. Classes are used to represent a particular type of data

Purpose of using classes and objects is to easily model the real world.

If we want to model real world classroom, **Student** will be the **class**, individual students like **Micky** and **Minnie** are **objects**. Data of objects will be marks, attendance, enrolment number, course etc. Methods will be get_parents_contact( ) etc.

Very intuitively, in case of **Sci-Mate** club, "member" will be the class. name, membership_ID, educational_qualification, address, contact, birth_year will be the data.

Note that, we store birth_year (or date of birth) instead of age, because age will change every year. So, it's wise to store birth_year and compute age whenever required.

Now, this indicates need to have a method to calculate the age.

So, we will have methods like get_age( ) too.

Like functions, there is specific syntax for defining classes too.

Before moving ahead, we will look at the concept of constructor and object creation.

Constructors

We need to "**create**" the objects before using them. We use special type of function/method to create, called **Constructor.**

What does this mean?

As we know objects have data (e.g. name, address etc for member object).

Job of the constructor is to assign some values to these data fields in the object.

If we want to create object member_1 of class Member, we use something similar to-

```
member_1 = Member("John",1223,1990)
```

Yes! That's right. We use name of class followed by parenthesis and arguments inside it.

How many arguments? Well, it depends on the code of the class.

Arguments of constructors can be set to default (like default arguments for methods) thereby allowing us to have constructors accepting different number of arguments.

Let's see now how to create class.

Write A Program To Know About "classes"

We will now write a class that will be used to store the details of the members of class.

Name of the class: of course, **Member** (it is convention to start the name of class with capital letter)

What data object of **Member** will hold? For now, we will store name, birth_year, phone and address.

How the construction of the Member should be?

Well, while creating the new member, we must have his name, birth_year and also at least contact number.

So, we can have constructor with 2 arguments for name and phone.

What methods it will have?

As of now, we can think of 2 methods:

1. get_age( ) method to compute age: note here, as we are storing birth_year as a part of object data, we don't need to provide it as argument. This is how classes make coding more logical
2. set_address() method to set address, which is not part of constructor

classes are defined using keyword "**class".**

So, when we write,

```
class Member:
```

we start the block of codes for the body of the class.

Next step, write constructor.

Constructor has special syntax, it is a method with name **__init__( )**

with 2 underscores around **init** word.

It has one argument as **self.**

Here, self is used to refer the object within its body.

To get intuitive understanding, think of "self" as pronoun.

You refer to yourself as "I", and I refer to myself as "I". We both have different names, identities etc. But for own ourselves, we are just "I".

Same way, object refers to itself as "self" inside the body of class.

Other arguments will be those you want to set, in our case, name, birth_year, phone.

We will set blank text "" as value for address, for now.

So, our class definition looks somewhat like:

```
class Member:
  def __init__(self, name, year, phone):
    self.name = name
    self.birth_year = year
    self.phone = phone
    self.address=""
```

We can now create member like:
`m1 = Member("John",1990,1223)` [we don't write self here!]

And we can access phone number of John using our familiar **"."** notation!

m1.phone gives us m1's (or John's) phone number.

It's time to add methods.

Methods are written just like function inside class body. Methods have access to the data fields of class. Other details, we need to supply using arguments.

Let’s write get_age( ) method.

```
def get_age(self):

  return (dt.datetime.now().year-self.birth_year)
```

We need to import datetime as dt before starting to write the class in order to use the class datetime.

Our code reads-

```
import datetime as dt

class Member:
  def __init__(self, name, year, phone):
    self.name = name
    self.birth_year = year
    self.phone = phone
    self.address=””



  def get_age(self):

     return (dt.datetime.now().year-self.birth_year)
```

It’s time to set now the address. We will define set_address( ) method.

Since we want set new value for address field, this value should be given to set_address method as a part of argument.

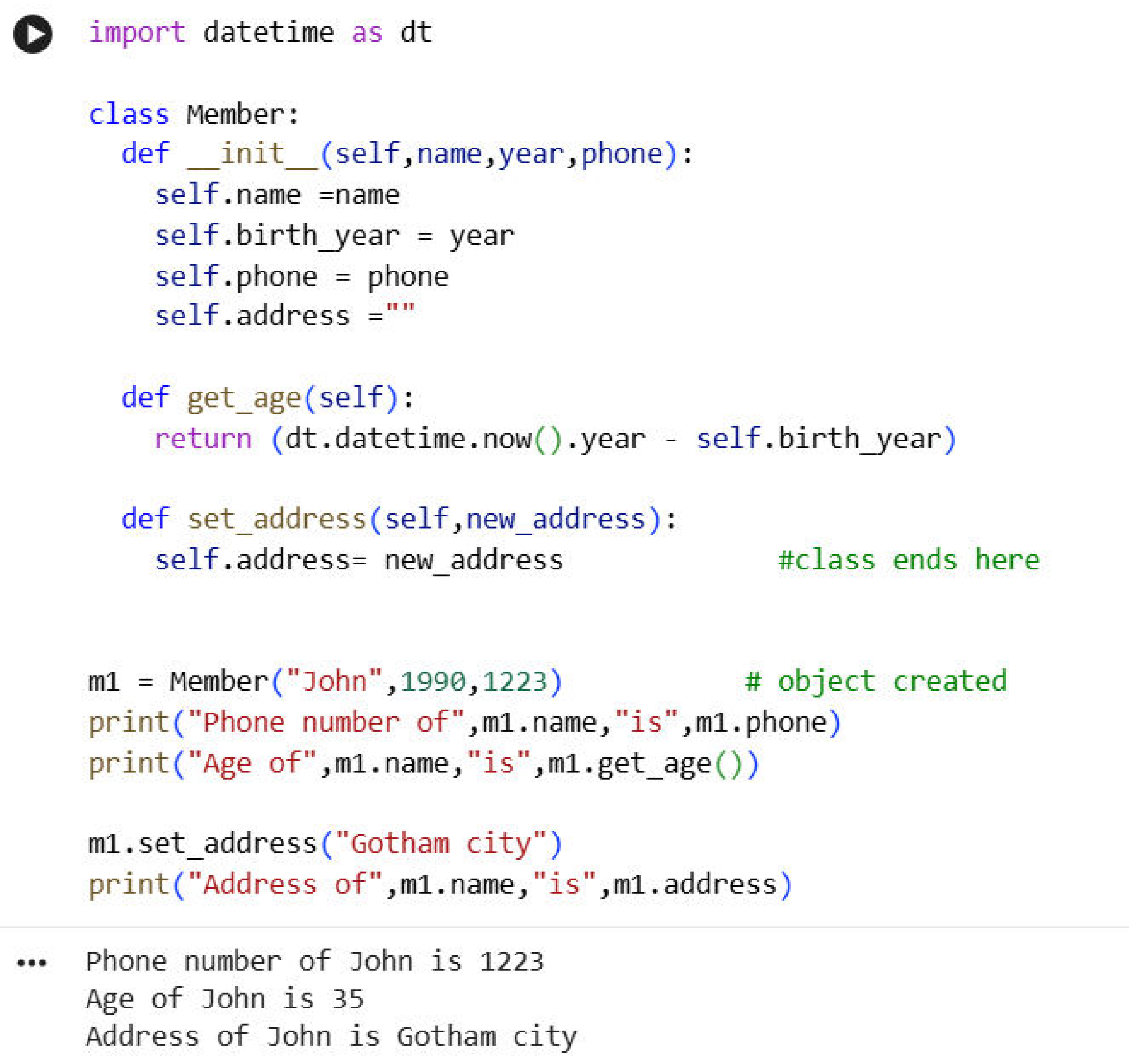
So, our updated class looks like-

```
import datetime as dt

class Member:
  def __init__(self, name, year, phone):
    self.name = name
    self.birth_year = year
    self.phone = phone
    self.address=””



  def get_age(self):

    return (dt.datetime.now().year-self.birth_year)

  def set_address(self,new_address):

    self.address= new_address
```

and our class is ready to use.

Note: Every method has **self** as first argument. This allows us to access the data of the object inside the methods.

Refer to the image below. We have imported datetime, defined class Member with methods, then also created Member object, printed his age, updated address and displayed the updated address. We also accessed the data of the class.



```python
import datetime as dt

class Member:
  def __init__(self,name,year,phone):
    self.name =name
    self.birth_year = year
    self.phone = phone
    self.address =""

  def get_age(self):
    return (dt.datetime.now().year - self.birth_year)

  def set_address(self,new_address):
    self.address= new_address              #class ends here


m1 = Member("John",1990,1223)             # object created
print("Phone number of",m1.name,"is",m1.phone)
print("Age of",m1.name,"is",m1.get_age())

m1.set_address("Gotham city")
print("Address of",m1.name,"is",m1.address)
```

```
Phone number of John is 1223
Age of John is 35
Address of John is Gotham city
```

This paradigm of programming where we model the data and methods with classes and objects is known as **Object-Oriented Programming**.

Progress Unlocked

- You can write your own classes
- You know about Object-Oriented Programming
- You can make Sci-Mate application with Object-Oriented Programming paradigm

We have learnt a lot in this chapter and a lot of the things might be new to you.

If it feels overwhelming, take a break and read chapter again, type the program hands on!

You can create your own class named Person, where you store name, height, weight and age. And a method to calculate BMI. You can use it for yourself and your friends!

Concepts we learnt revisited:

- **Functions** are blocks of code doing specific task
- **Functions** accept arguments as input and **return** output
- Arguments or parameters can be default, can be named/keyword, can be of varying length
- **Classes** combine data and methods together
- We need to create **objects** before using them. We use **Constructors** to create object

1-Minute Challenge

In our **Member** class, create following methods:

1. update_name : to update names of member
2. A new field, fees_status and method to set it "Paid" or "Due"
3. A new field, education and method to set it to "science_graduate" or "post_graduate"

Chapter 5: Dive Deeper in Python

TL; DR

- **Strings** = Text
- **Pandas** helps us store and manage real data in table form
- **Pandas** use dataframes for the management of tabular data
- We can access rows and columns, add or update data, and perform calculations easily.
- Pandas can read and save files easily
- List comprehension basic syntax : [expression **for** item **in** iterator]

Strings

See the program below, it is for checking the eligibility of an applicant for **Sci-Mate** club.

```
age = int(input("Enter your age"))
membership = False
if (age>=18):
  is_science_graduate = input("Are you a science graduate? (y/n)")
  if(is_science_graduate=="y"):
    membership = True
  elif(is_science_graduate=="n"):
    is_post_graduate = input("Are you a post graduate? (y/n) ")
    if(is_post_graduate =="y"):
      membership = True
    elif(is_post_graduate=="n"):
      membership = False
    else:
      print("please enter either ‘y’ or ‘n’ only")
  else:
    print("please enter either 'y' or ‘n’ only")

else:
  membership = False

if(membership):
  print("You are eligible for membership")
else:
  print("You are not eligible for membership or You entered invalid input")
```

Now, when we ask user *"Are you a science graduate? (y/n)"*, we expect user to enter "y" or "n" as directed.

But user may not obey and enter "Y" or "N" or " y" , "n " also.

As a programmer, our aim should be to develop user-friendly software.

So, we won't be strict on user and allow- "y", or "Y" or "y " or " n" etc.

A good solution would be to "standardize" the 'input' by user.

1. Remove the spaces prior and later to the letter
2. Convert it to lower case and compare

All the text that is in double quotes is object of class '**str**', a short form for string.

To create a string, we simply put double quotes " " around the text.

```
a = "This is the text"
print(type(a))
```

Gives us <class 'str'> as output.

`a = ""` creates a blank string.

This is useful when we want to create string and assign it value at later stages.

Now, **str** class has many methods to process strings.

First, we can individually access characters in string just like list.

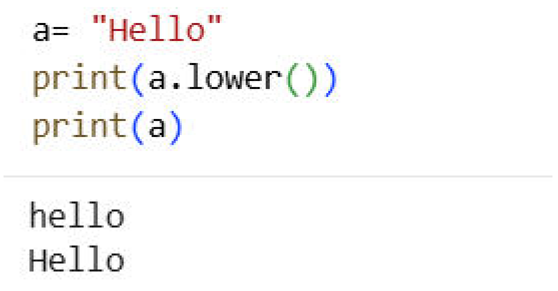
```
a = "Hello"
print(a[1])
```

This code prints "e".

str class has method lower() which returns a string which is all lower case version of the string.

Let's see some of the methods of str class and how they return output.

Note that these methods return new string which is in desired format. Original string remains intact.



```
a= "Hello"
print(a.lower())
print(a)
```

```
hello
Hello
```

Note that, as in image, when we print a.lower() we get lower case "hello", but when print 'a' after that, we get the original string "Hello" only.

Let we declare a string:

a = "I know Python very well!!   "

There are blank spaces at end. We will see what output we get with different methods.

Method	Purpose	Output
a.lower()	Get string with all lower characters	“i know python very well!! ”
a.upper()	Get string with all lower characters	“I KNOW PYTHON VERY WELL!! ”
a.strip()	Get string with former and later spaces removed	“I know Python very well”
“123”.isnumeric()	Returns true if string is all numeric	True

As `a.strip()` returns string, we can use other methods of string with it also.

So, various inputs like “Y”,”y”, “Y ” etc. all will return “y” when we use `t.strip().lower()`.

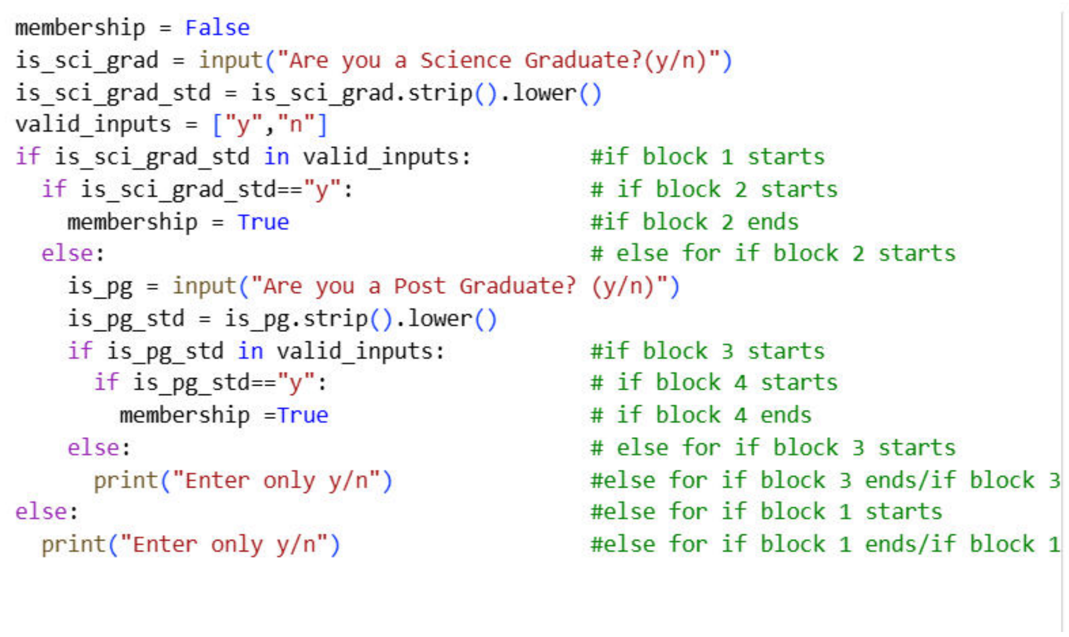
We can easily check it now.

Suppose we want to confirm user has entered only “y” or “n” and not any other text, so that we can warn user. This can be easily achieved using **in** operator as we have seen in previous chapter.

We will create list of valid inputs, like [“y”,”n”] and check if user’s input is **in** the list.

The expression `“y” in [“y”,”n”]` returns **True**.

So, we can re-write qualification check logic now in much better and manageable way.



```
membership = False
is_sci_grad = input("Are you a Science Graduate?(y/n)")
is_sci_grad_std = is_sci_grad.strip().lower()
valid_inputs = ["y","n"]
if is_sci_grad_std in valid_inputs:           #if block 1 starts
  if is_sci_grad_std=="y":                    # if block 2 starts
    membership = True                         #if block 2 ends
  else:                                       # else for if block 2 starts
    is_pg = input("Are you a Post Graduate? (y/n)")
    is_pg_std = is_pg.strip().lower()
    if is_pg_std in valid_inputs:             #if block 3 starts
      if is_pg_std=="y":                      # if block 4 starts
        membership =True                      # if block 4 ends
    else:                                     # else for if block 3 starts
      print("Enter only y/n")                 #else for if block 3 ends/if block 3 ends
else:                                         #else for if block 1 starts
  print("Enter only y/n")                     #else for if block 1 ends/if block 1 ends
```

Carefully read the code. Read multiple times.

Understand the nested structure of “if” blocks. Pay close attention to where one block starts and where it ends. Some “if” blocks don’t have “else”, simply, because it’s unnecessary as membership value is by default set to **False** and in else block, we would set it **False** only.

str class has lots of useful methods. We have just seen introduction to some.

Still, I would like to fetch your attention to 4 more things.

1] Slicing a string: A string “Hello World” can be sliced like we slice lists.

```
a = “Hello World”

b = a[2:5]

print(b)
```

This prints “llo”

2] Joining strings in list: Suppose there is a list of words [“I”,”know”,”the”,”Python”]

We can join all the words in list with any separator like “-“ with **join()** method.

Simply follow:

```
a=["I","know","the","Python"]

b="-".join(a)

print(b)
```

Gives us “I-know-the-Python” as output.

3] Splitting the string: Now, we want to reverse the above operation and want all words separate. We can do it with **split()** method.

```
a="I-know-the-Python"

b=a.split("-")

print(b)
```

This gives us list ['I', 'know', 'the', 'Python']

In above 2 examples, we can very well use space(“ “) as separator or connector.

4] Joining (concatenating) 2 strings: This is very intuitive and simple. Just use “+”.

```
a="Hello"

b="World"

c= a+b

print(c)
```

This gives us output “HelloWorld”! To get “Hello World” output, what would you do?

You guessed it right! Just do c= a + “ “+ b. **Try it!**

Progress Unlocked

- You can now standardize and process strings

Pandas

We have written **Member** class.

What if we can store and manage data in rows and columns style like excel?

A super powerful library **pandas** help us to achieve this.

We will be having a very short introduction to **pandas** library. This library in itself deserves a dedicated book.

In **pandas,** data is stored like a row-column structure. We call it as **dataframe**.

We have seen, how information of members can easily be described in dictionary.

If we have multiple such members, then each member can act as a row of dataframe and keys of dictionary will act as columns.

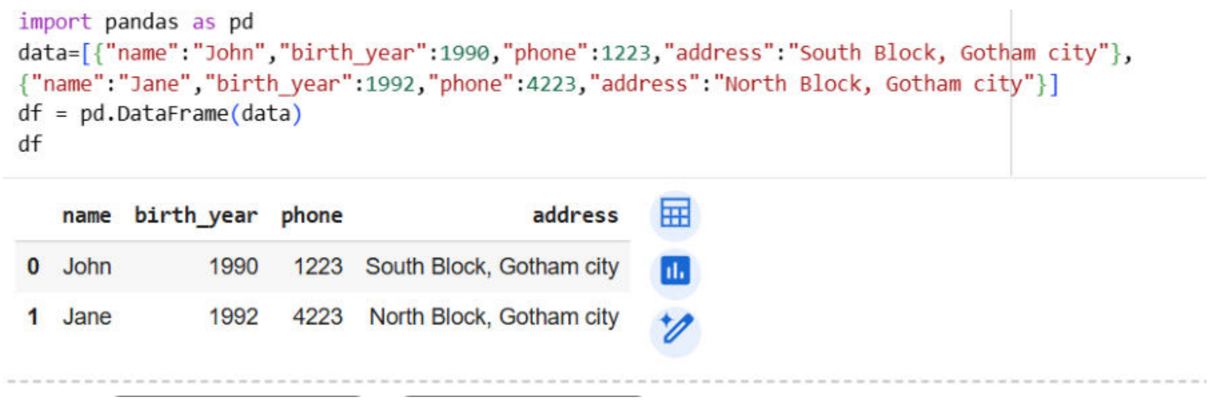
For example,

In following dictionary,

m1 =
{   "name":"John",
    "birth_year":1990,
    "phone": 1223,
   "address": "South Block, Gotham City"

}

name, birth_year, phone, address will serve as columns dataframe and ("John", 1990, 1223, "South Block, Gotham City") as row.
We need to import pandas before using it. We can create dataframe simply as **list of dictionaries**! See the example below:



```
import pandas as pd
data=[{"name":"John","birth_year":1990,"phone":1223,"address":"South Block, Gotham city"},
{"name":"Jane","birth_year":1992,"phone":4223,"address":"North Block, Gotham city"}]
df = pd.DataFrame(data)
df
```

```
   name  birth_year  phone                   address
0  John        1990   1223  South Block, Gotham city
1  Jane        1992   4223  North Block, Gotham city
```

DataFrame constructor is used to create dataframe object.

We feed it **"List of dictionaries"** as data. You can see tabular output above.

NOTE: Data can also be formatted as dictionary of list. key of dictionary becomes column name and value for key become column values.

In the previous example; 0 or 1 in leftmost column is index. Here, index is row-label.

We can directly convert list of objects to dataframe, but we will see it a bit later.

As of now, let's do some operations as per requirement of our club management.

1] Dataframe is ready. Now print the details in 2nd row.

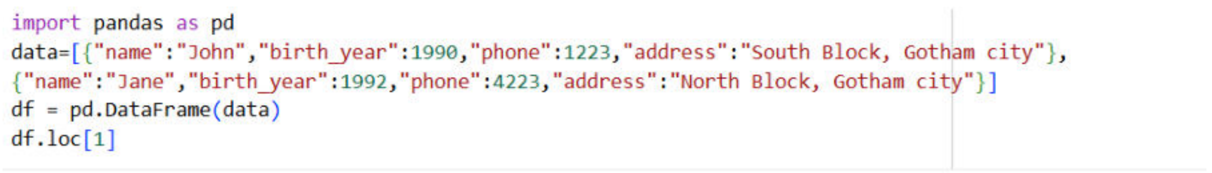
By default, pandas assigns index values as 0, 1, 2, ...

So, the 2nd row has index 1.

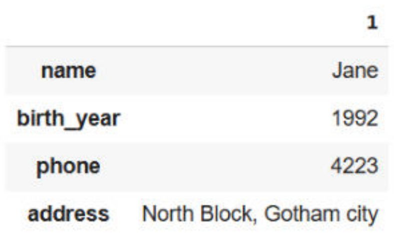
We use .loc to refer to rows using index labels.

Example: df.loc[1]

See the example below:



```
import pandas as pd
data=[{"name":"John","birth_year":1990,"phone":1223,"address":"South Block, Gotham city"},
{"name":"Jane","birth_year":1992,"phone":4223,"address":"North Block, Gotham city"}]
df = pd.DataFrame(data)
df.loc[1]
```



	1
name	Jane
birth_year	1992
phone	4223
address	North Block, Gotham city

2] Count how many members are in dataframe.

Here, our old friend len( ) helps.

`len(df)` gives us number of rows, i.e. 2 here.

3] Add the details of "Jack" to dataframe. (add a row)

We simply go to the index after last row and assign details of "Jack" as list of values. But remember, order of values in list must be same as the order of columns of dataframe.

How to go to the index after last row?

Simple, it's `df[len(df)]`.

If we have 2 members, then occupied indices are 0 and 1.

We have to add "Jack" at next index, i.e. index 2, which is same as len(df).

```
df.loc[len(df)] = ["Jack", 1994,3221,"Gotham city"]
```

This would add "Jack" to the dataframe.

Note: This works because our dataframe is using default numeric indexing. We will continue with this indexing as of now.

4] Check names of all the members [Print column data]

We just have to use `df["name"]` to print all the details in names column.

5] Add the age column (add new column)

Currently, we are not using objects, so we can't use the get_age().

We will need to compute the age.

Here comes the interesting point.

Dataframes are designed such that, you don't need to loop through them. Instead, you write code as if you are writing for a single row and it will applicable to all the rows.

So, assigning **df["age"]** will automatically assign age values to all columns based on computation.

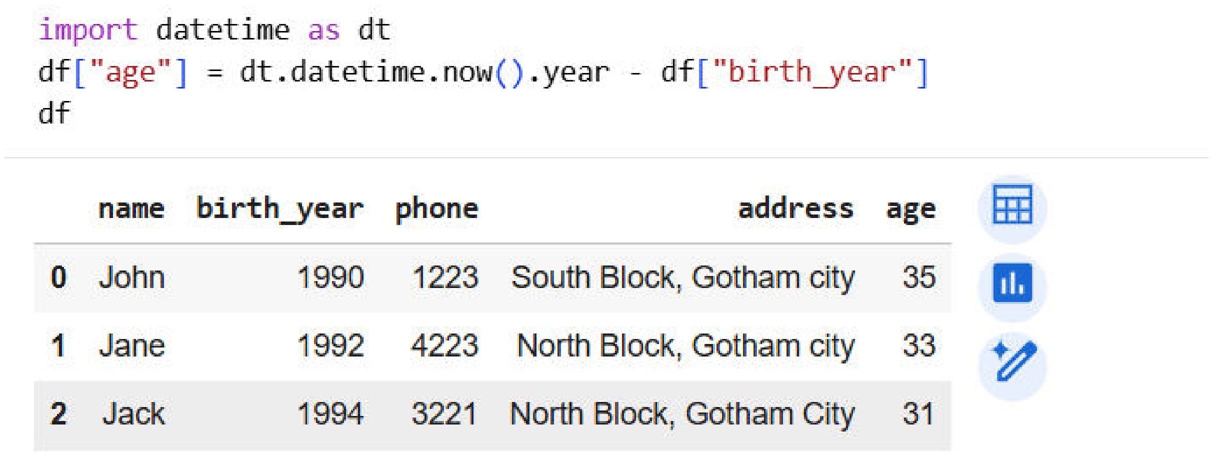
Here, we will need to add computation, for that, we will just use the expression that we used in get_age( ) : `dt.datetime.now().year -year`

So here,

```
df["age"] = dt.datetime.now().year - df["birth_year"]
```

this will run for every year in birth_year column in dataframe.

See the whole code below:



```
import datetime as dt
df["age"] = dt.datetime.now().year - df["birth_year"]
df
```

	name	birth_year	phone	address	age
0	John	1990	1223	South Block, Gotham city	35
1	Jane	1992	4223	North Block, Gotham city	33
2	Jack	1994	3221	North Block, Gotham City	31

NOTE: As a good practice, don't modify `df` itself for addition of new column etc., unless you want to save the modified `df`.

Better option is, copy df to another dataframe using function copy( ) and add column there.

Processing will be done on `df1` keeping `df` intact.

```
import datetime as dt

df1 = df.copy()

df1["age"] = dt.datetime.now().year - df1["birth_year"]
```

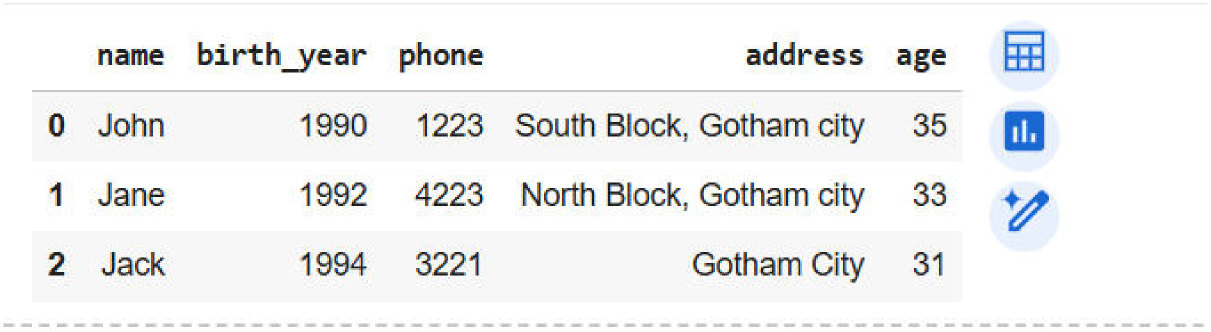
6] Get the average age of members

Simply, use `df["age"].mean()`. mean() gives the average value.

7] Remove the column age now.

We use drop( ) method for this.

```
df.drop(columns="age")
df
```



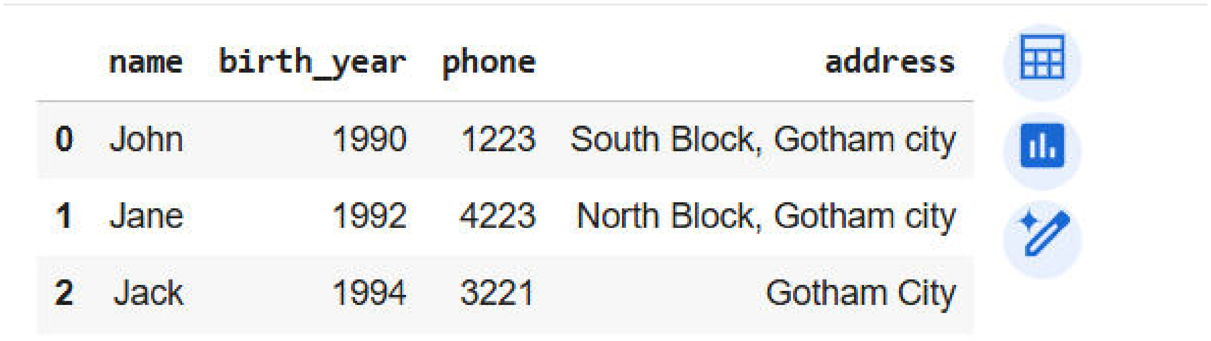
	name	birth_year	phone	address	age
0	John	1990	1223	South Block, Gotham city	35
1	Jane	1992	4223	North Block, Gotham city	33
2	Jack	1994	3221	Gotham City	31

We used drop( ) but still column "age" is there!

What happened?

The drop method doesn't change original dataframe. Instead, it returns new dataframe without age column.

```
df1= df.drop(columns="age")
df1
```



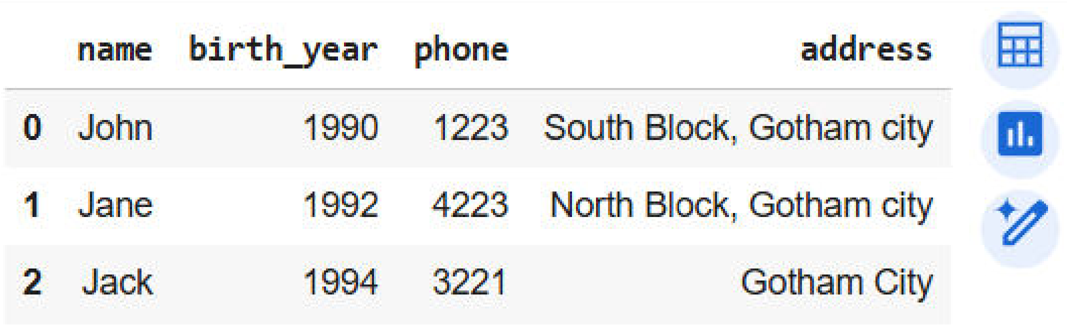
	name	birth_year	phone	address
0	John	1990	1223	South Block, Gotham city
1	Jane	1992	4223	North Block, Gotham city
2	Jack	1994	3221	Gotham City

Now it works!

We can also effect the change in "df" dataframe itself.

For it, we use **inplace=True** in drop method. This gives effect "in place" i.e. to df itself.

```
df.drop(columns="age",inplace=True)
df
```



	name	birth_year	phone	address
0	John	1990	1223	South Block, Gotham city
1	Jane	1992	4223	North Block, Gotham city
2	Jack	1994	3221	Gotham City

8] Find the members born before 1993 [Conditional Filtering]

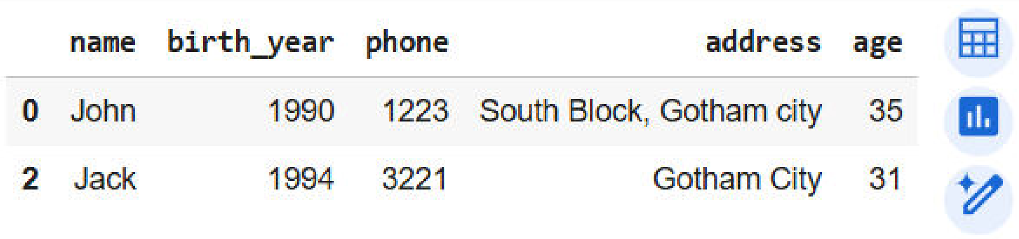
This is very useful feature of **pandas**.

We place a condition inside df[ ].

Only rows where the condition is True are returned (Dataframe will return only those rows where list has "True" at same index)

See the code example below. We created list [True,False,True] and used it to access dataframe. Only those rows with "True" value in list were extracted.

```
list_to_sort_rows = [True,False,True]
df1= df[list_to_sort_rows]
df1
```



	name	birth_year	phone	address	age
0	John	1990	1223	South Block, Gotham city	35
2	Jack	1994	3221	Gotham City	31

We use same feature for conditional filtering.

We put an expression to access the sorted dataframe.

The expression will be our condition.

Here, we want all with birth year prior to 1993.

So, our condition is]

```
df["birth_year"]<1993
```

use same expression to access dataframe and we get conditionally sorted dataframes.

```
e = df["birth_year"]<1993
df1 = df[e]
df1
```

```
   name  birth_year  phone                 address  age
0  John        1990   1223  South Block, Gotham city   35
1  Jane        1992   4223  North Block, Gotham city   33
```

We can see the results in above screenshot.

Now, we can also shorten the steps and exclude defining e and all. Use condition directly inside df[ ].

```
df1 = df[df["birth_year"]<1993]
df1
```

```
   name  birth_year  phone                 address  age
0  John        1990   1223  South Block, Gotham city   35
1  Jane        1992   4223  North Block, Gotham city   33
```

9] Get element from 2nd row and 3rd column (slicing dataframe)

Here, we are trying to excess an element at 2nd row and 3rd column.

Remember, we use 0- based indices.

To access individual element, you can use

`e = df.iloc[row,column]`

So, `e = df.iloc[1,2]` gives us 2nd row, 3rd element

`df1= df.iloc[0:2,0:2]` slice of df: rows 1 to 2 and columns 1 to 2 to df1

`df2= df.iloc[1:3,:]` slice of df in df2, rows 2 to 3, all columns. (last index excluded)

10] Update particular value

"Jack" noticed, his address is just "Gotham City" on record. He wants to update it to "North Block, Gotham City"

For this, we will first access the element using **.loc** and update it.

For row: use the expression that will be true in "Jack"s case.

`df["name"]=="Jack"`

For column: use what we want to replace i.e. “address”

So, code becomes

df.loc[row_expression,column_name_to_update] = new_value

i.e.

```
df.loc[df["name"]=="Jack","address"] = "North Block, Gotham City"
```

Important: Save the dataframe for further use

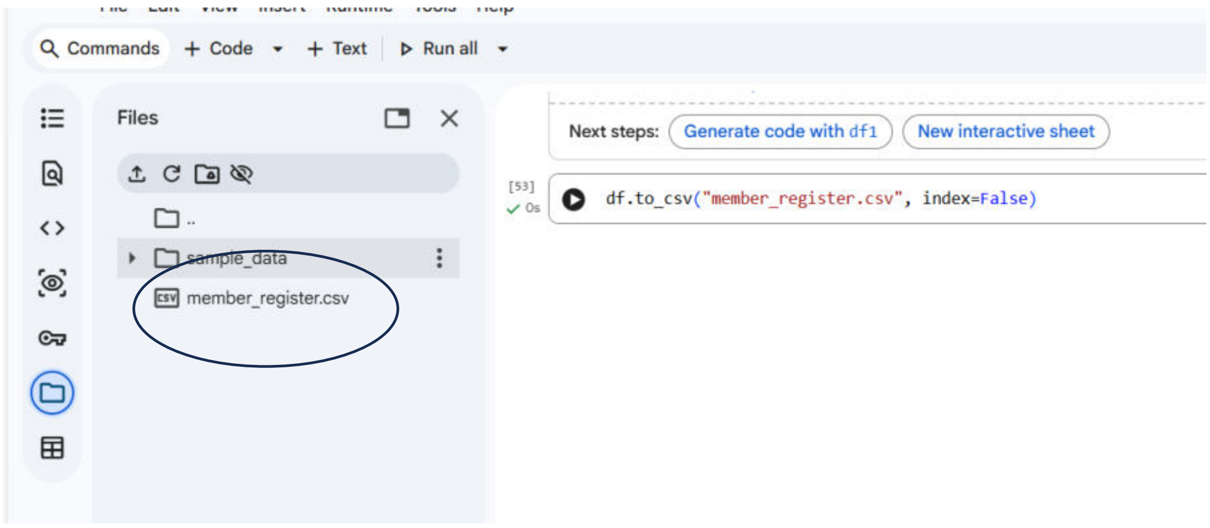
We can save and load dataframe for further use.

As of now, we are using colab. So, this particular part in our book is limited to our project only. You should store data into databases. That discussion is out of scope of this book.

- Use this to save: `df.to_csv("member_register.csv", index=False)`

This will save our dataframe in “folder” in colab.

See below:



```
df.to_csv("member_register.csv", index=False)
```

- Use this to load file again: `df = pd.read_csv("member_register.csv”)`

This file will be saved in **CSV** i.e. **comma separated value** format. In this format, rows are stored in separate lines with columns separated by comma. Pandas can read excel files too.

e.g.
John,1990,1223..
Jane,1992,4223..

We have learnt basics about pandas. Pandas is a big topic in itself.

We just covered the introductory material.

I encourage you to try all of the above operations and also try some of the operations at your own. for example, you can add more members, add more columns , slicing dataframes etc.

Let’s see the summary of all we have learnt about **Pandas:**

Task	Pandas Code / Idea	What it Does
Access a row by index	`df.loc[1]`	Gets row with index label 1
Count number of rows	`len(df)`	Returns total number of members
Add a new row	`df.loc[len(df)] = [...]`	Adds data at the next index
Access a column	`df["name"]`	Prints all values from name column
Add new column	`df1["age"] = current_year - df1["birth_year"]`	Computes age for all rows
Copy dataframe	`df1 = df.copy()`	Keeps original data safe
Average of column	`df["age"].mean()`	Finds average age
Remove a column	`df.drop("age", axis=1)`	Returns dataframe without column
Remove column in-place	`df.drop("age", axis=1, inplace=True)`	Permanently removes column
Filter rows by condition	`df[df["birth_year"] > 1993]`	Returns matching rows only
Access single element	`df.iloc[1, 2]`	2nd row, 3rd column (0-based)
Slice rows & columns	`df.iloc[1:3, 0:2]`	Extracts part of dataframe
Update a value	`df.loc[df["name"]=="Jack","address"] = "..."`	Updates selected cell(s)

Progress Unlocked

- You can use store and process tabular data programmatically

Now, as I had said before, we will see how to add the object of class to dataframe.

Before that, we will study a very interesting topic, **List Comprehension.**

List Comprehension

I have a list of numbers, lst1= [1,5,6,7,12,18]. I want a new list with squares of each number in list.

Of course, you can will create first empty list named lst1_squares, run a for loop on lst1, append the square of each value to lst1_squares.

Perfectly valid.

```
lst1= [1,5,6,7,12,18]
lst2 =[]

for item in lst1:
-    lst2.append(item*item)

print(lst2)
```

But let me tell you, we can convert these 5 lines of code into a single line!

We first declare the new list: `lst2 =  [ ]`

Inside the list, we add expression which is part of **for** block: `lst2= [item * item]`

following to it, we just add **for** block declaration lines:

```
 lst2 = [item*item for item in lst1]
```

How beautiful it is!

And it doesn’t stop here; we can also implement condition.

Suppose, we just want a list of squares of even numbers.

Just add condition after the **for** lines.

We can easily write now the condition to check even numbers: item%2==0

So, `lst2 =[item*item for item in lst1 if item%2==0]`

Gives us [36,144,324] in lst2.

We can also have **else** condition too. Suppose, in lst2, we want squares of even numbers but keep odd numbers as it is:

Only difference is, we now want to clarify condition before looping using for.

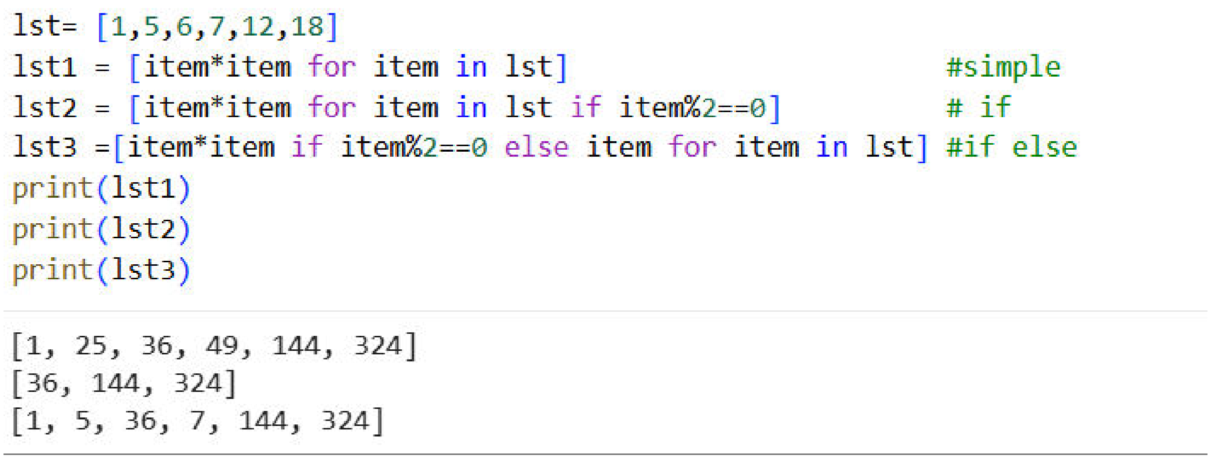
So, `lst2 =[item*item if item%2==0 else item for item in lst1]`

Gives us [1, 5, 36, 7, 144, 324].

We can simply state syntax for list compression as:

- **without if**: [expression **for** item **in** iterator]
- **with if**: [expression **for** item **in** iterator if condition]
- **with if and else**: [expression **if** True condition **else** expression if false condition **for** item **in** iterator]   *(Iterator is basically a collection)*

Here is the example of list comprehension:



```
lst= [1,5,6,7,12,18]
lst1 = [item*item for item in lst]                   #simple
lst2 = [item*item for item in lst if item%2==0]      # if
lst3 =[item*item if item%2==0 else item for item in lst] #if else
print(lst1)
print(lst2)
print(lst3)
```

```
[1, 25, 36, 49, 144, 324]
[36, 144, 324]
[1, 5, 36, 7, 144, 324]
```

List comprehension works for non-numeric data types too.

Challenge: Given a list ["John","Don","Ron","Jane"], can you create a list of names starting with J?

Hint: First letter of string can be checked using my_str[0].

Did you succeed?

If yes, then let's move to our final part.

Progress Unlocked

- You are now ready to work with real data, not just toy examples.

Adding objects to dataframe

For our **Sci-Mate** project, we want to assign object value to new row in dataframe.

1. We know how a list can be equated to row in dataframe when we have default indexing
   df.loc[len(df)] = [list of data field values]
2. We know how to declare list and excess fields of object using **"."**

So, just use

m1 = Member("Ron",1987,"7788,"City Centre, Gotham City"]

```
df.loc[len(df)] = [m1.name, m1.birth_year, m1.phone,m1.address]
```

That's it!

Remember, order of fields in list MUST BE same as order of columns of dataframe.

Here ends our theory sessions. Next, we are going to develop an application for **Sci-Mate** club.

Concepts we learnt revisited:

- ❖ Processing and standardizing user input using string methods like lower() and strip()
- ❖ Understanding strings as objects of the str class and using common string operations
- ❖ Working with tabular data using pandas DataFrames
- ❖ Accessing, adding, updating, and filtering rows and columns in a DataFrame
- ❖ Performing calculations on entire columns without writing loops
- ❖ Saving and loading data using CSV files
- ❖ Writing concise and readable code using list comprehension
- ❖ Mapping object data fields to a DataFrame row

1-Minute Challenge

In our **Member** class, create a new member "Jill", born in 1996, lives in "South block, Gotham City" and you can contact her at "9922"

1. Add Jill's name to dataframe
2. Store dataframe to CSV file
3. Load dataframe from CSV file
4. Display details of dataframe using "df"

Progress Unlocked

- Congratulations!! You have successfully completed your journey of Basic Python
- It's not easy to reach here
- Now, you know something which many people in the world don't know
- You are ready dive in to the ocean of Python and develop your skills!!

Let's Go for Project! 🚀

Quick Revision

(Read this before starting the Sci-Mate project)

Purpose of this revision (Very Important)

This section is **not meant to teach**.
It is meant to **remind**.

While revising, if any point feels:

- confusing
- forgotten
- uncomfortable

That is a **signal to go back to the book**, read that section again, and return here.

If this revision feels mostly familiar, **you are ready for the project**.

Chapter 1: Start Programming

What this chapter actually taught

- Programming means **telling a computer exactly what to do, step by step**
- Computers cannot guess or abstract like humans
- Big tasks must be broken into **small, exact instructions**
- Python is a **programming language** used to write those instructions

Core concepts you learnt

- print() and input() are **functions**
- Variables are **named boxes** that store values
- Data types:
    - int, float, str, bool
- Expressions produce values
- Type conversion using int() and similar functions

You should now be able to

- Take input from user
- Store it in variables
- Perform calculations
- Display output

📌 If this feels weak, revisit:

- "Programming = breaking a task into small steps"

Chapter 2: Make Decisions and Repeat

What this chapter actually taught

- Programs make **decisions** using conditions
- Programs **repeat actions** using loops
- Python uses **indentation** to define blocks of code

Core concepts you learnt

- if, elif, else
- Conditions return True or False
- Comparison operators (>, <, >=, ==, !=)
- Logical operators (and, or)
- for loop with range()
- while loop
- Nested if blocks
- Basic idea of **variable scope** (intuitive)

You should now be able to

- Write decision-based programs
- Combine multiple conditions
- Repeat tasks without rewriting code
- Read and fix indentation errors

📌 If this feels weak, revisit:

- Voting eligibility example
- Sci-Mate membership decision program

Chapter 3: Make Programs Better (Collections, Objects)

What this chapter actually taught

- Data and actions often belong **together**
- Python collections help store **multiple values**
- Objects have **methods** that act on data

Core concepts you learnt

- Lists []
    - append(), remove(), sort(), count()
    - Indexing and slicing
- Tuples () (immutable lists)
- Dictionaries {} with key–value pairs
- Nested dictionaries
- Iterating over collections
- Intuitive introduction to **classes and objects**
- Understanding obj.method() syntax

You should now be able to

- Store and process multiple values
- Choose the right collection for a problem
- Read and write code using lists and dictionaries
- Understand why methods use the . operator

📌 If this feels weak, revisit:

- Members list examples
- Phone book dictionary examples
- Nested dictionary explanation

Chapter 4: Manage the Program Better (Functions and Classes)

What this chapter actually taught

- Functions help **organize actions**
- Classes help **organize data and actions together**
- Object-Oriented Programming models real-world entities

Core concepts you learnt

- Writing functions using def
- Functions that return values
- Arguments:
    - default arguments

    - keyword arguments
    - variable-length arguments
- Scope (local vs global, intuitive)
- Importing modules (import datetime)
- Writing classes
- Constructors __init__
- self keyword
- Creating and using objects
- Methods inside classes

You should now be able to

- Break code into reusable functions
- Write simple classes
- Create objects and access their data
- Understand constructors and methods

📌 If this feels weak, revisit:

- get_age() function
- Member class example
- Constructor explanation

Chapter 5: Dive Deeper in Python (Strings, Pandas, Comprehension)

What this chapter actually taught

Strings

- Strings are objects of class str
- String processing is essential for user-friendly programs
- Important methods:
    - lower(), upper(), strip()
    - split(), join()
    - slicing strings
- Standardizing user input
- Validating input using collections and in

Pandas

- DataFrames store data in **rows and columns**
- Creating DataFrame from list of dictionaries
- Accessing data:
    - loc, iloc
- Adding, updating, deleting rows and columns
- Column-wise calculations (without loops)
- Filtering rows using conditions
- Saving and loading CSV files

List Comprehension

- Writing compact, readable loops
- With and without conditions
- Using list comprehension on strings and numbers

Bridging objects and DataFrames

- Mapping object fields to DataFrame rows
- Maintaining column order

You should now be able to

- Clean and standardize user input
- Work with tabular data programmatically
- Read, update, and store structured data
- Write concise Python using list comprehension

📌 If this feels weak, revisit:

- Input standardization example
- Pandas summary table
- Final object-to-DataFrame example

Big Picture: Are You Ready for the Project?

If you can:

- Read code and understand **what it is trying to do**
- Break problems into steps
- Accept that you may need to revisit sections while coding

Then **yes, you are ready**.

You are **not expected to remember syntax perfectly**.
You are expected to **recognize patterns** and **know where to look back**.

Project

Sci-Mate Club Member Management System

Till now, we learned Python concepts in small pieces.
Now we bring those pieces together and build a **real, usable system**.

This project is not about clever tricks.
It is about **thinking like a software developer**. Read it calmly, do it!

We will write a simple program that helps manage members of the Sci-Mate Club.

The entire code will run on **Google Colab**, so no installation is required.

Step 1: Requirement gathering

Before writing code, we first understand the problem.

Imagine this conversation.

Sci-Mate Club Manager:
"I want a simple system to manage club members."

Instead of jumping into coding, we ask questions.

After discussion, the requirements become clear.

What the system should do

- Store member details
- Add new members
- Show member information when needed
- Update member details
- Remove members safely without losing history

This is enough to begin designing the system.

Step 2: Identifying core operations

Most data-based systems revolve around four basic operations:

- Create
- Read
- Update
- Delete

These are commonly known as **CRUD operations**.

For our club system:

Operation	Meaning in our project
Create	Add a new member
Read	View member details
Update	Change member information
Delete	Mark member as inactive

Important decision:
We will **not delete records permanently**.

Instead, we will add an active field:

- Y means active member
- N means inactive member

This keeps data safe and retrievable in future.

Step 3: Start implementation: Addressing Storage Requirement

We now decide how data will be stored.

We choose:

- **Pandas DataFrame** for working with data
- **CSV file** for saving data permanently

Why this choice:

- Member data is tabular
- CSV works smoothly in Colab
- Pandas makes filtering and updating easy

Let's first import **pandas** and **datetime**

```
import pandas as pd
import datetime as dt
```

We define the file name in one place. This avoids hard-coding the file name everywhere.

```
FILE_NAME = "scimate_members.csv"
```

3.1 Loading data safely

Now we think carefully.

When the program starts:

- Sometimes the CSV file already exists
- Sometimes it does not

Both situations are normal.

If we directly try to read a file that does not exist, Python will stop the program.

So, we tell Python:

Try to read the file.
If it is not found, create an empty data structure instead.

This is done using **try** and **except**.

We have not covered this concept in book.

The code will look like:

```
def load_data():
    try:
        df = pd.read_csv(FILE_NAME)
    except FileNotFoundError:
        df = pd.DataFrame(columns=[
            "member_id",
            "name",
             “birth_year”,
            "phone",
            "address",
            "active"
        ])
    return df
```

Let us read this **line by line in English**.

Line 1: **try**

Python is told:

“I am going to attempt something that may or may not work.”

Line 2: **Load the file**

```
df = pd.read_csv(FILE_NAME)
```

This will work **only if** the file already exists.

If it works:

- df gets the data
- Python skips the except part
- Function continues normally

Line 3: except:

This means:

"If the problem is specifically that the file was not found, do the following."

We are not catching every possible problem.
We are handling **only one known situation**.

And in this case, we create a dataframe using the columns which are fields of class Member.

This becomes our fallback plan.

So instead of stopping the program:

- We create a fresh data structure
- The program continues smoothly

This is a practical way to think about programs, be prepared with **fallback plans.**

3.2 Saving data

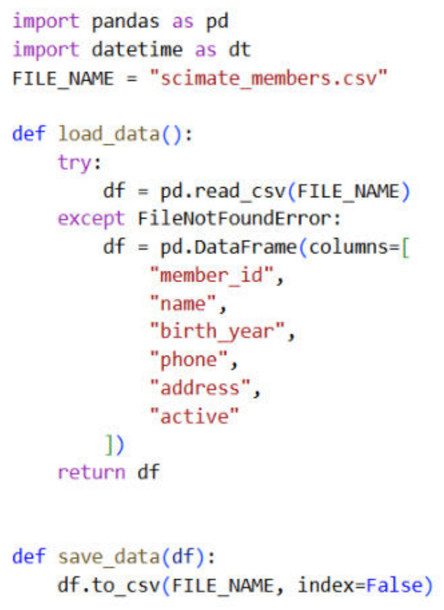
Whenever we change data, we must save it.

```
def save_data(df):

    df.to_csv(FILE_NAME, index=False)
```

when we say, index=False, index of the rows will not be saved to file.

Note that, we are breaking the code into functions and we are passing dataframe as argument to functions.

Achievement Unlocked: Here, we have completed our requirement of storage.



```
import pandas as pd
import datetime as dt
FILE_NAME = "scimate_members.csv"

def load_data():
    try:
        df = pd.read_csv(FILE_NAME)
    except FileNotFoundError:
        df = pd.DataFrame(columns=[
            "member_id",
            "name",
            "birth_year",
            "phone",
            "address",
            "active"
        ])
    return df


def save_data(df):
    df.to_csv(FILE_NAME, index=False)
```

This is how our code looks so far.

We have implemented 2 functions for storage requirements.

Functions try to load the file (where data is stored) into a dataframe, if file doesn't exist, it creates dataframe.

Function returns the dataframe for further processing.

You can take a coffee-break! Then we will move on to the next code.

Step 4: Implementation of CRUD

Let’s now start implementation.

4.1 Create

This is the **Create** operation.

We will:

1. Ask the user for member details
2. Decide the eligibility of the user
3. Generate a unique member ID
4. Add the data to the DataFrame
5. Save the updated data

Now, generating unique ID is something new we are going to do here.

We will simply assign the (index+1) of the row where member’s data will be added.

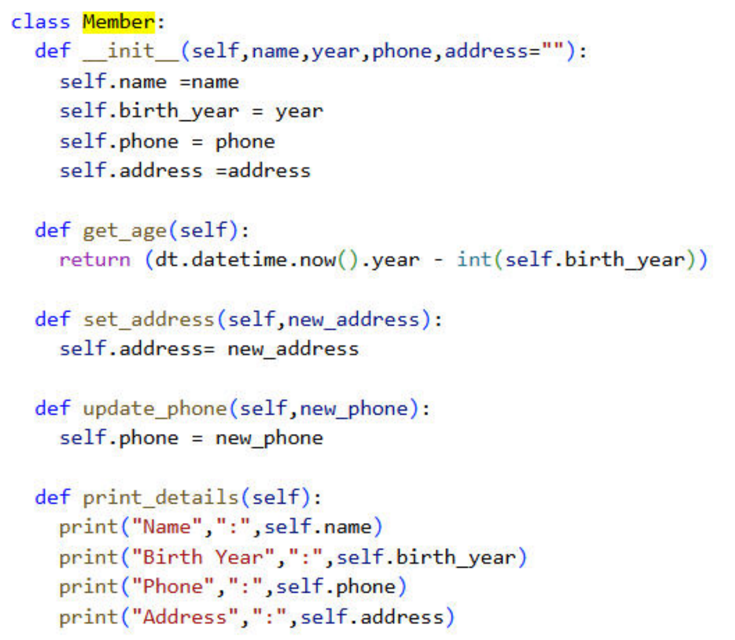
So, first member will have id 1 (index of first row: 0 and 0+1 =1)

If we have 14 members, we will get new id 15.

Before moving further, we will create **Member** class.

It is pretty much similar to what we had studied in the book. We have added 2 methods, to print member details and update phone number.

We also have updated constructor a bit.



```
class Member:
  def __init__(self,name,year,phone,address=""):
    self.name =name
    self.birth_year = year
    self.phone = phone
    self.address =address

  def get_age(self):
    return (dt.datetime.now().year - int(self.birth_year))

  def set_address(self,new_address):
    self.address= new_address

  def update_phone(self,new_phone):
    self.phone = new_phone

  def print_details(self):
    print("Name",":",self.name)
    print("Birth Year",":",self.birth_year)
    print("Phone",":",self.phone)
    print("Address",":",self.address)
```

Code for the Member class looks somewhat like this.

```
class Member:
  def __init__(self,name,year,phone,address):
    self.name =name
    self.birth_year = year
    self.phone = phone
    self.address =address

  def get_age(self):
    return (dt.datetime.now().year - self.birth_year)

  def set_address(self,new_address):
    self.address= new_address

  def update_phone(self,new_phone):
    self.phone = new_phone

  def print_details(self):
    print("Name",":",self.name)
    print("Birth Year",":",self.birth_year)
    print("Phone",":",self.phone)
    print("Address",":",self.address)
```

Now we create a method to ask details of user and check eligibility.

We have made couple of changes to the code.

We combined `strip()` and `lower()` to standardize the input.

We stored message for invalid input into a variable, so that print becomes easy.

We have used `.max()` function on `member_id` field.

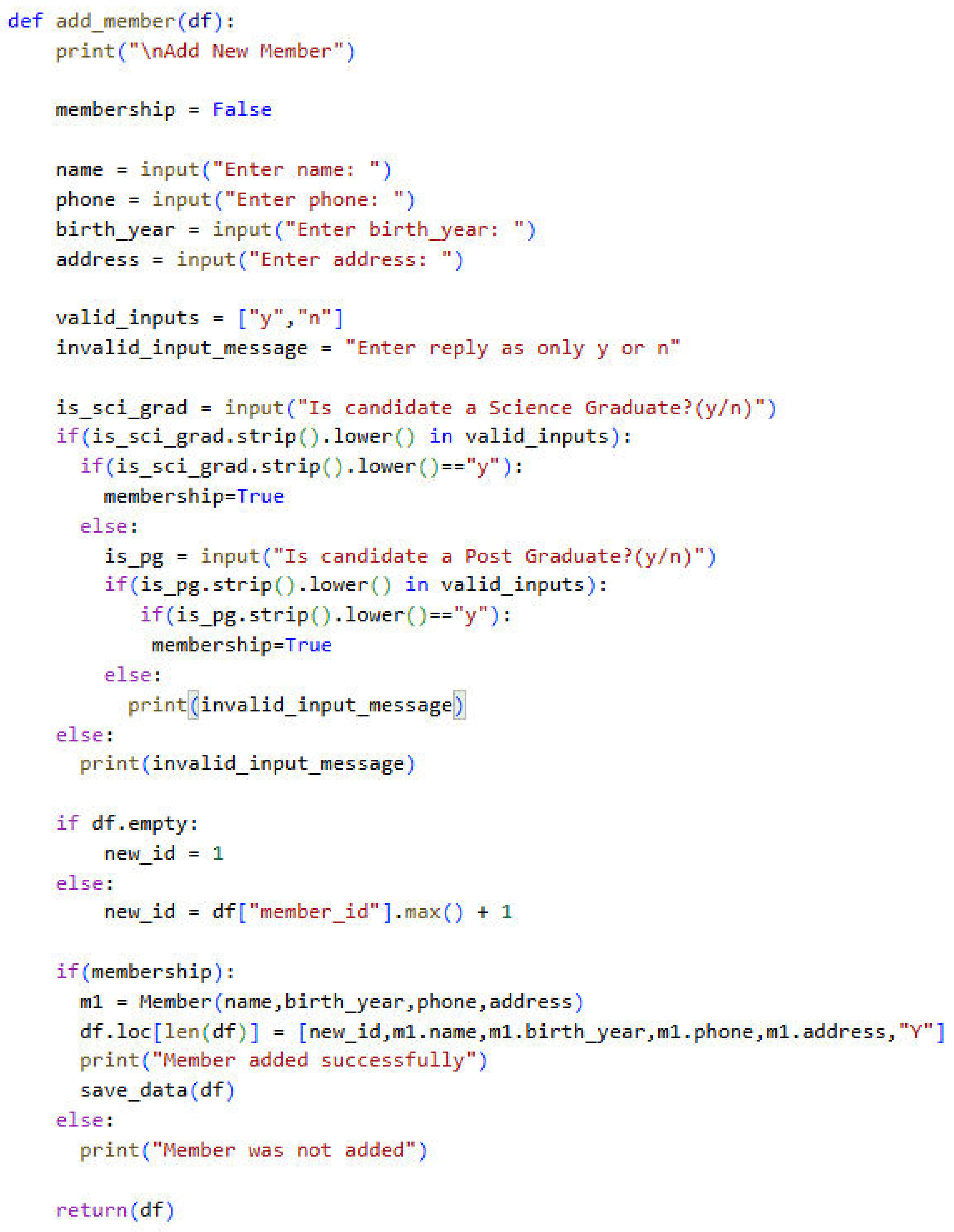
`df["member_id"].max()` gives us the maximum value of in the member_id field.

We are storing "Y" for active member and "N" for those who have left the club.

Since a new member is being added, his active value is "Y".

Following we can see the code of add_member class.

Note: the "\n" before the print line ensures that text to be printed on new line



```python
def add_member(df):
    print("\nAdd New Member")

    membership = False

    name = input("Enter name: ")
    phone = input("Enter phone: ")
    birth_year = input("Enter birth_year: ")
    address = input("Enter address: ")

    valid_inputs = ["y","n"]
    invalid_input_message = "Enter reply as only y or n"

    is_sci_grad = input("Is candidate a Science Graduate?(y/n)")
    if(is_sci_grad.strip().lower() in valid_inputs):
      if(is_sci_grad.strip().lower()=="y"):
        membership=True
      else:
        is_pg = input("Is candidate a Post Graduate?(y/n)")
        if(is_pg.strip().lower() in valid_inputs):
           if(is_pg.strip().lower()=="y"):
            membership=True
        else:
          print(invalid_input_message)
    else:
      print(invalid_input_message)

    if df.empty:
        new_id = 1
    else:
        new_id = df["member_id"].max() + 1

    if(membership):
      m1 = Member(name,birth_year,phone,address)
      df.loc[len(df)] = [new_id,m1.name,m1.birth_year,m1.phone,m1.address,"Y"]
      print("Member added successfully")
      save_data(df)
    else:
      print("Member was not added")

    return(df)
```

Code of the add_member is written as below:

```
def add_member(df):
    print("\nAdd New Member")
    membership = False
    name = input("Enter name: ")
    phone = input("Enter phone: ")
    birth_year = input("Enter birth_year: ")
    address = input("Enter address: ")
    valid_inputs = ["y","n"]
    invalid_input_message = "Enter reply as only y or n"

    is_sci_grad = input("Is candidate a Science Graduate?(y/n)")
    if(is_sci_grad.strip().lower() in valid_inputs):
      if(is_sci_grad.strip().lower()=="y"):
        membership=True
      else:
        is_pg = input("Is candidate a Post Graduate?(y/n)")
        if(is_pg.strip().lower() in valid_inputs):
           if(is_pg.strip().lower()=="y"):
            membership=True
        else:
          print(invalid_input_message)
    else:
      print(invalid_input_message)

    if df.empty:
        new_id = 1
    else:
        new_id = df["member_id"].max() + 1

    if(membership):
      m1 = Member(name,birth_year,phone,address)
      df.loc[len(df)] =
[new_id,m1.name,m1.birth_year,m1.phone,m1.address,"Y"]
```

```
      print("Member added successfully")

      save_data(df)

    else:

      print("Member was not added")


    return(df)
```

4.2 Read

There is a lot of possibility in Read operation.

You can provide options to sort data based on various conditions.

Including many conditions will make our project and book longer.

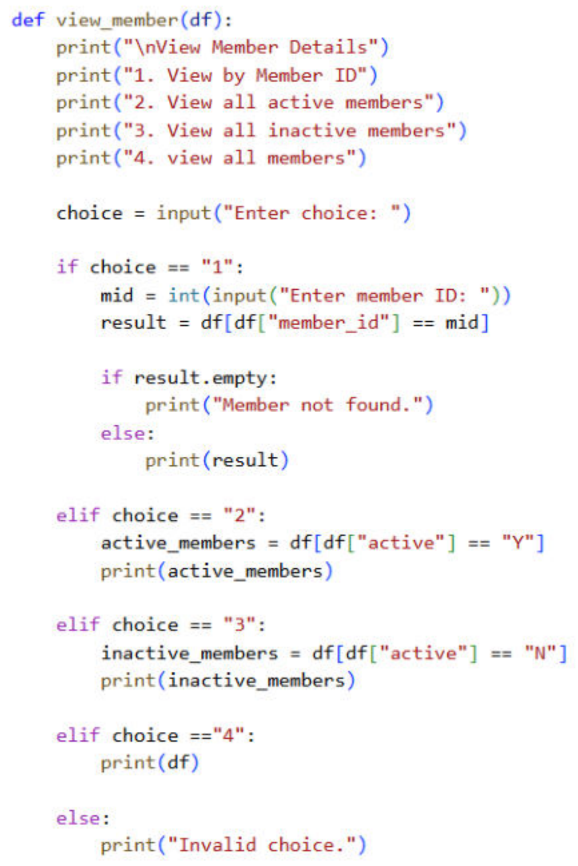
So, we will stick to 3 reads

1. Read all data of all members

2. Read data of all active members

3. Read data of all inactive members

4. Read data of member based on member_id



```
def view_member(df):
    print("\nView Member Details")
    print("1. View by Member ID")
    print("2. View all active members")
    print("3. View all inactive members")
    print("4. view all members")

    choice = input("Enter choice: ")

    if choice == "1":
        mid = int(input("Enter member ID: "))
        result = df[df["member_id"] == mid]

        if result.empty:
            print("Member not found.")
        else:
            print(result)

    elif choice == "2":
        active_members = df[df["active"] == "Y"]
        print(active_members)

    elif choice == "3":
        inactive_members = df[df["active"] == "N"]
        print(inactive_members)

    elif choice =="4":
        print(df)

    else:
        print("Invalid choice.")
```

We run condition to check for which row, entered ID equals to member_id.

Similarly, we use conditions to filter active and inactive members.

As part of exercise, you can introduce more options to filter the data.

The code in text is also shared below along with the adjoining picture.

Note: the "\n" before the print line ensures that text to be printed on new line

```
def view_member(df):
    print("\nView Member Details")
    print("1. View by Member ID")
    print("2. View all active members")
    print("3. View all inactive members")
    print("4. view all members")

    choice = input("Enter choice: ")

    if choice == "1":
        mid = int(input("Enter member ID: "))
        result = df[df["member_id"] == mid]

        if result.empty:
            print("Member not found.")
        else:
            print(result)

    elif choice == "2":
        active_members = df[df["active"] == "Y"]
        print(active_members)

    elif choice == "3":
        inactive_members = df[df["active"] == "N"]
        print(inactive_members)

    elif choice =="4":
        print(df)

    else:
        print("Invalid choice.")
```

4.2 Update

Updating means:

- Data already exists
- We change only one part of it

We must be careful not to update the wrong record.

How update works

1. Ask for member ID
2. Find the corresponding row
3. Ask which field to update (name, phone or address)
4. Apply the change
5. Save the data

In the code below, you might have noticed a line like this:

```
df.at[idx, "phone"] = input("Enter new phone: ")
```

df.at also gives us way to update certain value (like df.loc)

syntax is:

```
df.at[row_index, column_name]
```

- row_index tells Pandas which row
- column_name tells Pandas which column

We have fetched the index using conditional filter and `df.index( )` method, which is also a new implementation.

Now look at this line again:

```
df.at[idx, "phone"] = input("Enter new phone: ")
```

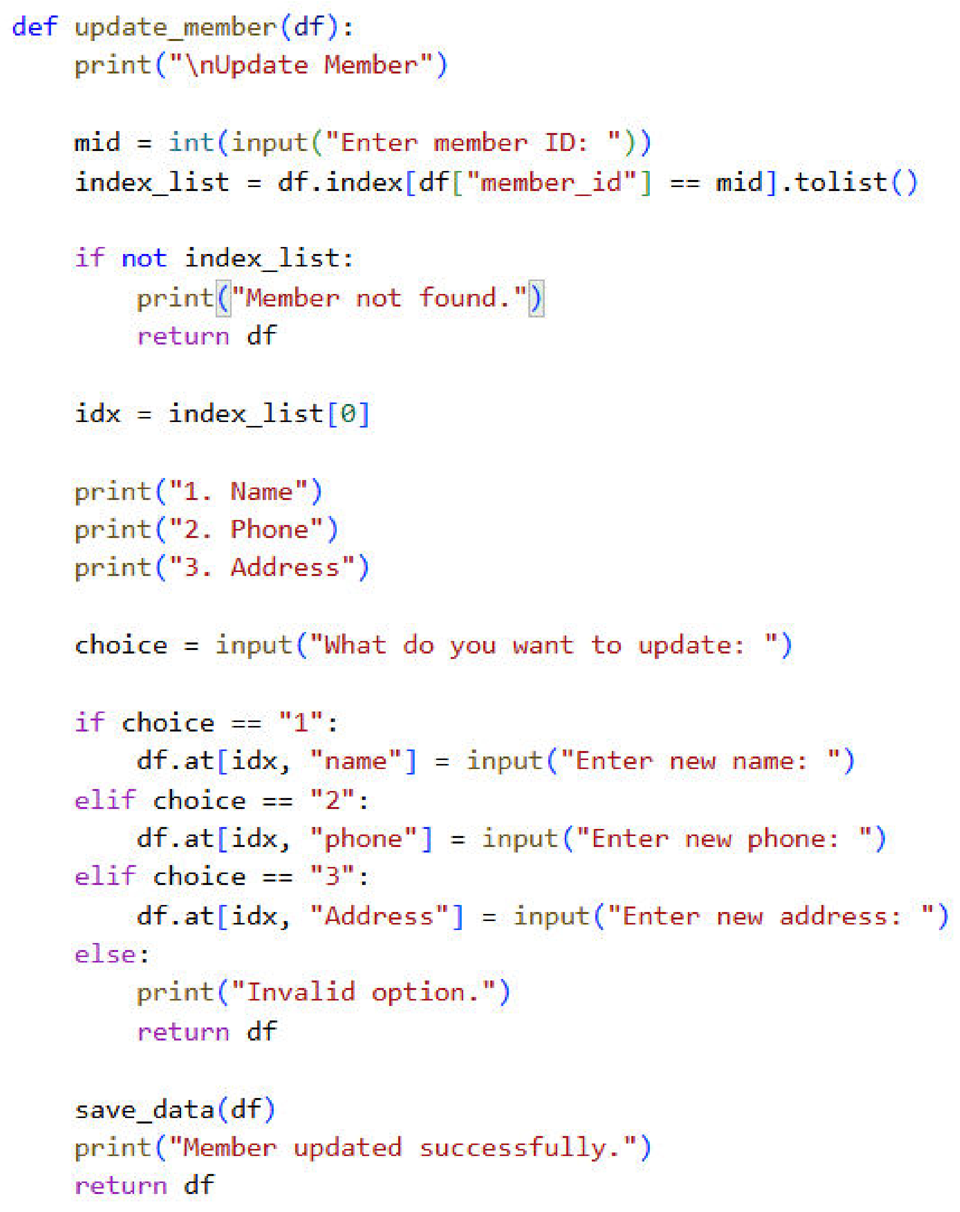
This line does **two things at once**:

1. It takes input from the user
2. It stores that input in the dataframe

This is completely valid and intentional.

Purpose of using these methods and ways is to get you acquainted with the same.

You can also use the methods we discussed in book.



```python
def update_member(df):
    print("\nUpdate Member")

    mid = int(input("Enter member ID: "))
    index_list = df.index[df["member_id"] == mid].tolist()

    if not index_list:
        print("Member not found.")
        return df

    idx = index_list[0]

    print("1. Name")
    print("2. Phone")
    print("3. Address")

    choice = input("What do you want to update: ")

    if choice == "1":
        df.at[idx, "name"] = input("Enter new name: ")
    elif choice == "2":
        df.at[idx, "phone"] = input("Enter new phone: ")
    elif choice == "3":
        df.at[idx, "Address"] = input("Enter new address: ")
    else:
        print("Invalid option.")
        return df

    save_data(df)
    print("Member updated successfully.")
    return df
```

In the code, after receiving member ID, we first check if member exists with this ID or not.

If ID is not found, we simply return to the main menu.

I would once again like to fetch your attention to dataframe “df” passed as an argument to the function and returned as the value by every function.

Imagine it like, when your phone malfunctions, you give to service centre. Service centre processes to repair it and returns it back to you.

Code for the method is as below:

```
def update_member(df):
    print("\nUpdate Member")

    mid = int(input("Enter member ID: "))
    index_list = df.index[df["member_id"] == mid].tolist()

    if not index_list:
        print("Member not found.")
        return df

    idx = index_list[0]

    print("1. Name")
    print("2. Phone")
    print("3. Address")

    choice = input("What do you want to update: ")

    if choice == "1":
        df.at[idx, "name"] = input("Enter new name: ")
    elif choice == "2":
        df.at[idx, "phone"] = input("Enter new phone: ")
    elif choice == "3":
        df.at[idx, "address"] = input("Enter new address: ")
    else:
        print("Invalid option.")
        return df

    save_data(df)
    print("Member updated successfully.")
    return df
```

4.4 Delete

In the code for deleting the member, we just set active status to “N”.

It is not good idea to entirely drop the data. Club may need to preserve the data of inactive members for various reasons too.

Below is the code:

```
def deactivate_member(df):
    print("\nDeactivate Member")

    mid = int(input("Enter member ID: "))
    index_list = df.index[df["member_id"] == mid].tolist()

    if not index_list:
        print("Member not found.")
        return df

    idx = index_list[0]
    df.at[idx, "active"] = "N"

    save_data(df)
    print("Member deactivated.")
    return df
```

This code is pretty much simple.

I expect you to understand this and be able to explain it.

Can you?

Step 5: Co-ordinating all

We have written so many functions! Nearly 6!

But how and when to call them?

We will provide user with a menu driven application.

We will provide a list of menus for various actions. Based on his choice, we will turn the flow of code to appropriate function.

Let’s first create a simple menu.

```
def main_menu():
    print("\nSci-Mate Club Management")
    print("1. Add member")
    print("2. View member")
    print("3. Update member")
    print("4. Remove member")
    print("0. Exit")
```

This gives user an option to enter the menu.

Based on selection, we call appropriate function.

Let's write a controlling function. This function will first run main_menu and then call suitable function and return to the same screen.

```
def run_program():
    df = load_data()
    while True:
        main_menu()
        choice = input("Enter choice: ")
        if choice == "1":
            df = add_member(df)
        elif choice == "2":
            view_member(df)
        elif choice == "3":
            df = update_member(df)
        elif choice == "4":
            df = deactivate_member(df)
        elif choice == "0":
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Try again.")
```

Here, we are using **`while True`**.

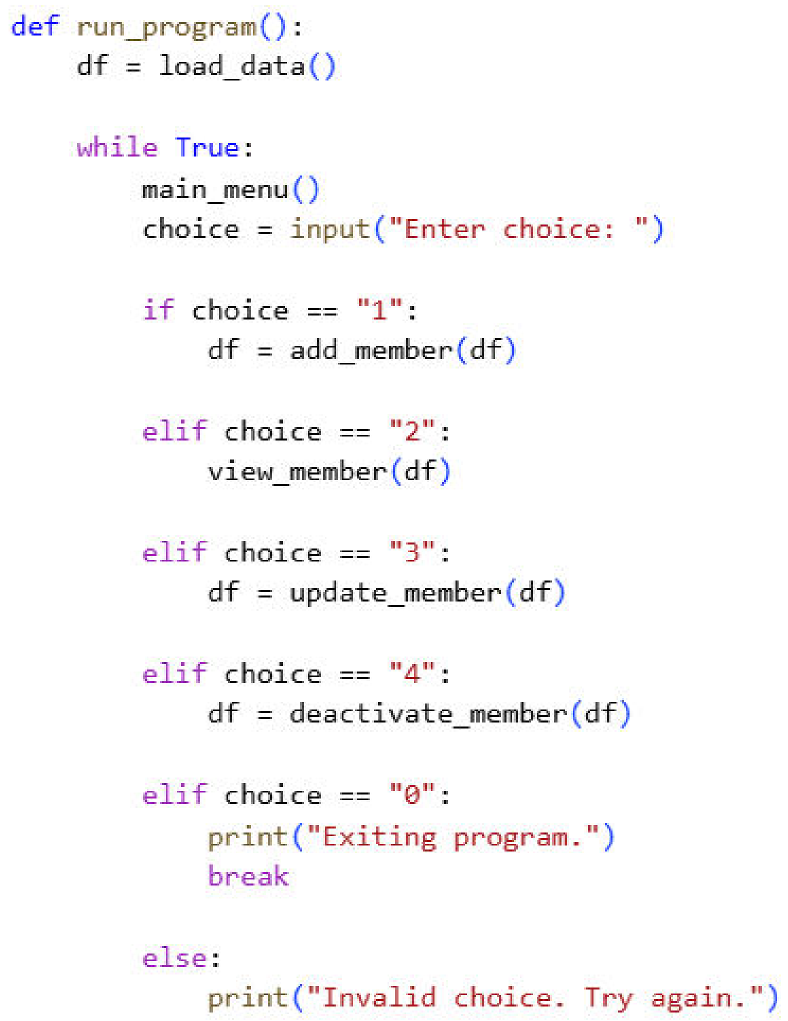
This means, this loop will continue to run forever!

But we care about our machine too!

So, we provide option to an user to exit this **forever** loop.

When user enters “0”, we have used **break** keyword.

This keyword essentially tells that, break this forever loop!



```
def run_program():
    df = load_data()

    while True:
        main_menu()
        choice = input("Enter choice: ")

        if choice == "1":
            df = add_member(df)

        elif choice == "2":
            view_member(df)

        elif choice == "3":
            df = update_member(df)

        elif choice == "4":
            df = deactivate_member(df)

        elif choice == "0":
            print("Exiting program.")
            break

        else:
            print("Invalid choice. Try again.")
```

Here is the screenshot of the code.

And that’s it.

Its time to run the code.

If you have written all code in same cell, no issues. Otherwise, first run all the cells and then in new cell type

```
run_program()
```

Congratulations!!!

You have created a real-life project with Python!!!

Your Next Step

You have now reached an important milestone.

You can think in steps, write Python programs, break problems into parts, and build a small but meaningful project like the **Sci-Mate club system**.

This page is meant to show you **possible next paths**, not to overwhelm you. You do not need to do everything at once. Pick one direction, go slow, and enjoy the process.

1. Moving from Colab to a Professional Setup

Throughout this book, we used **Google Colab**.
That was intentional.

- Colab saved us from installation issues
- You could focus only on **logic and learning**
- Everything worked directly in the browser

Now, if you want to write Python programs regularly or professionally, it is a good idea to install Python on your own system.

You can choose **one** of the following:

- **Visual Studio Code (VS Code)**
  Lightweight, fast, and widely used by professionals
  Excellent for Python, web development, and automation
- **Anaconda**
  Comes with Python and many data-science libraries preinstalled
  Very useful if your future focus is data analysis, ML, or AI

There is no rush. You can continue using Colab even now. Installation is a **next step**, not a requirement.

2. Deep Dive into Data, ML, and AI

If you enjoyed working with data in our project, this path may excite you.

You can now explore Python libraries such as:

- **pandas** for advanced data handling
- **NumPy** for numerical operations
- **Matplotlib** and **Seaborn** for visualizing data
- **SciPy** for scientific and statistical analysis

Practical Sci-Mate Use Case

Imagine this extension of our project:

- Collect **interests of Sci-Mate members**
- Store data like: robotics, astronomy, coding, biology
- Analyze patterns:
    - Which interests are most common?
    - Which members share similar interests?
- Plan activities, workshops, or teams based on patterns

This is how **real ML and data analysis begin**:
small data, clear questions, practical outcomes.

3. From Files to Databases

In this book, we stored data in **CSV files**.
That was the right choice for learning.

Next step:

- Learn how to store data in a **database**

You can start with:

- **SQLite** (simple, file-based, beginner-friendly)
- Then move to **MySQL** or **PostgreSQL**

Benefits:

- Faster data access
- Better structure
- Safer updates
- Used in real applications everywhere

Your Sci-Mate system can become more reliable and scalable with a database.

4. Building a Desktop Application

So far, we ran programs in notebooks and terminals.

Next idea:

- Turn Sci-Mate into a **desktop application**

You can learn:

- **PyQt** or **Tkinter**

With this, you can create:

- Buttons
- Forms
- Windows
- Tables

Imagine a Sci-Mate app where:

- The club manager clicks buttons instead of typing commands
- Members are added through forms
- Reports open in windows

This is where Python starts to feel like **real software**.

5. Building a Web Application

Another powerful direction is web development.

Using Python, you can create:

- Websites
- Dashboards
- Online systems

Popular frameworks:

- **Flask** – simple, flexible, beginner-friendly
- **Django** – powerful, structured, used in large projects

With a web app, Sci-Mate could:

- Have an online member portal
- Allow registrations through a browser
- Show activities and reports online

This combines **Python + logic + real-world impact**.

6. Automation and Scripting

Python is also excellent for automation.

You can learn to:

- Automatically process files
- Generate reports

- Rename, clean, or analyze data
- Send emails or notifications

Many professionals use Python daily just to **save time**.

7. Most Important Advice

Do not try to learn everything together.

- Pick **one direction**
- Build **small projects**
- Make mistakes
- Improve gradually

Programming is not about knowing all tools.
It is about **thinking clearly and solving problems**.

You already have the foundation.
Everything ahead is an extension of what you have learned here.

Take your next step with confidence.

Quiz for Revision and Test

Following are some questions that you can read for revision for testing your understanding.

If you get wrong, don't worry!

Read the concept again.

Sr no	Questions	Answers
1	What does the Python command `print("Hello World")` do?	It displays the text 'Hello World' as output on the screen.
2	What is programming?	It is the process of giving a machine precise, step-by-step instructions to perform a task.
3	In computer science, a language like Python, with its specific keywords and syntax, is known as a _____.	programming language
4	What is a 'function' in Python?	A named block of code that performs a specific action, such as `print()` or `input()`.
5	What is the term for a value passed into a function's parentheses?	An argument.
6	Which Python function pauses a programme to wait for the user to type something and press Enter?	The `input()` function.
7	Concept: Variable	Definition: A named container or label used for storing a value that can be used or changed later in a programme.
8	What is the default data type of any value returned by the `input()` function?	String (`str`).
9	In the command `print("Hello", name)`, why is the variable `name` not enclosed in quotation marks?	To print the value stored inside the variable, rather than the literal word 'name'.
10	According to good practice, what should a variable's name represent?	It should clearly indicate the nature of the value it holds.
11	A variable name in Python must start with either a letter or an _____ character.	underscore
12	A variable name in Python is not allowed to begin with a _____.	number
13	What is the result of using the `+` operator on the two strings '10' and '20'?	The strings are joined together (concatenated) to produce '1020'.
14	What does the `int` data type represent?	Whole numbers without a decimal point, such as 35 or -10.
15	What does the `float` data type represent?	Numbers that have a decimal point, such as 3.5 or 10.2.

16	What is the `str` data type used for?	To represent text, or a sequence of characters, enclosed in quotes.
17	What are the only two possible values for the `bool` data type?	True or False.
18	What is the purpose of the `int()` function when used with `input()`?	To convert the string value received from the user into an integer for mathematical operations.
19	What is an 'expression' in programming?	A combination of values, variables, and operators that produces a new value.
20	To control the flow of a programme based on whether a condition is true, which Python keyword is used?	The `if` keyword.
21	What is the purpose of the `else` keyword in a decision structure?	It specifies a block of code to execute if the preceding `if` and `elif` conditions are all false.
22	How does Python determine which lines of code belong to an `if` block?	Through indentation, where the lines are shifted to the right with spaces or a tab.
23	Failing to indent the code block after an `if` statement will cause an _____.	IndentationError
24	A conditional expression, such as `age >= 18`, evaluates to a value of which data type?	Boolean (`bool`).
25	Which built-in function reveals the data type of a variable?	The `type()` function.
26	What does the `#` symbol signify in a line of Python code?	It indicates the beginning of a comment, which is ignored by the interpreter.
27	What is the role of the `elif` keyword?	It allows for checking additional conditions if the preceding `if` condition was false.
28	What is the correct Python operator to check if two values are equal?	`==` (double equals sign).
29	Which comparison operator is used to test for inequality?	`!=` (exclamation mark equals sign).
30	To execute code only when two separate conditions are both true, which logical operator should be used?	The `and` operator.
31	To execute code when at least one of two conditions is true, which logical operator should be used?	The `or` operator.
32	What is the term for placing one `if` statement inside the block of another `if` statement?	Nesting, or creating nested blocks.
33	Which type of loop is typically used to iterate over a sequence of items, such as a range of numbers?	A `for` loop.
34	What sequence of numbers does `range(1, 6)` generate for a loop?	It generates the numbers 1, 2, 3, 4, 5.

35	What is the purpose of a `while` loop?	To repeat a block of code as long as a specified condition remains true.
36	What is the shorthand operator in Python for `i = i + 1`?	`i += 1`.
37	What does the modulus operator (`%`) do?	It returns the remainder after a division.
38	A number is even if the expression `number % 2` evaluates to _____.	0
39	What is a 'method' in the context of object-oriented programming?	A function that belongs to an object and is called using dot notation (e.g., `my_list.sort()`).
40	A 'class' can be thought of as a _____ for creating objects.	blueprint
41	What is a 'list' in Python?	An ordered, changeable collection of items, enclosed in square brackets (`[]`).
42	What is the index of the first element in any Python list?	0
43	Which function is used to find the total number of items in a list?	The `len()` function.
44	Which list method adds a new element to the very end of the list?	The `.append()` method.
45	To remove the first occurrence of a specific value from a list, which method would you use?	The `.remove()` method.
46	What does the `.sort()` method do when called on a list of strings?	It sorts the items in the list into alphabetical order.
47	How do you access or change the third item in a list named `data`?	Using its index, `data[2]`.
48	The technique of extracting a sub-section of a list, like `my_list[2:5]`, is known as _____.	slicing
49	What is a 'tuple' in Python?	An ordered collection of items, similar to a list, but it is immutable (cannot be changed after creation).
50	Which symbols are used to create a tuple?	Round brackets `()`.
51	Concept: Dictionary	Definition: An unordered collection that stores data in `key:value` pairs, enclosed in curly brackets `{}`.
52	How is a value retrieved from a dictionary?	By specifying its corresponding key in square brackets, e.g., `my_dict['key']`.
53	How can you add a new key-value pair to an existing dictionary?	By assigning a value to a new key, e.g., `phone_book['Johnny'] = 1212`.
54	To remove an item from a dictionary using its key, which method is used?	The `.pop()` method.

55	To iterate through only the keys in a dictionary, one can use the `for key in my_dict.____():` structure.	keys
56	To iterate through only the values in a dictionary, one can use the `for value in my_dict.____():` structure.	values
57	Which dictionary method allows you to loop through both the keys and values at the same time?	The `.items()` method.
58	What is a 'nested dictionary'?	A dictionary that contains another dictionary as one of its values.

You started this book with curiosity, not certainty.
You end it with **clarity**, not completeness.

That is exactly how learning programming should feel.

This book did not try to teach you everything about Python.
Instead, it tried to teach you something far more valuable:

- how to break a problem into steps
- how to think before writing code
- how to stay calm when things do not work
- how to learn by doing, not by memorizing

If you can now read a problem and say,
"Let me think how to solve this step by step,"
then this book has done its job.

From here on, tools will change.
Languages may change.
Even Python itself will evolve.

But the way you think will remain useful.

Do not measure your progress by how much code you know.
Measure it by how confidently you approach a new problem.

This book is not the end of your learning journey.
It is the **first solid step**.

And that is enough to move forward.

✽ ✽ All The Best!! ✽ ✽

About NousBase

NousBase is a space for learning that respects the learner.

We believe education should build capability, not just credentials.
It should help people understand fundamentals, think independently, and grow step by step in a fast-changing world.

This book is **intentionally introductory**.
It is meant to help you begin.
To remove fear.
To make programming feel approachable and logical.

We do not try to cover everything here.
Instead, we focus on strong basics, clear thinking, and the confidence to continue learning on your own.

Our content and apps are designed to be practical, calm, and honest.
We avoid hype.
We avoid shortcuts.
We focus on foundations that last.

If you reached this page, you are ready for the next level.
Revisit chapters without guilt.
Build the project seriously.
Then move forward with curiosity and discipline.

Thank you for learning with us.

-Team NousBase

Follow us and explore more at: