Introduction to Computers and Programming.

What is a Computer?

A computer is an electronic device that helps us do many things, such as writing documents, playing games, browsing the internet, and solving problems. It works by taking input, processing it, and giving output. For example, when you type on a keyboard, the computer processes your words and shows them on the screen.

Computers have different parts:

- **Monitor** – Displays what you see on the screen.
- **Keyboard** – Used to type and give instructions.
- **Mouse** – Helps to click and select things on the screen.
- **CPU (Central Processing Unit)** – The brain of the computer that processes information.
- **Storage (Hard Drive/SSD)** – Saves files, documents, and programs.

What is Programming?

Programming is the process of giving instructions to a computer so it can do specific tasks. These instructions are written in a language that computers understand, called a **programming language**.

Examples of programming languages:

- **Scratch** – A fun and easy way to learn coding using blocks.
- **Python** – A simple language used for many applications.
- **JavaScript** – Used to make websites interactive.

Why Learn Programming?

- **Creativity** – You can create your own games and apps.
- **Problem-Solving** – Helps you think logically to solve challenges.
- **Future Skills** – Coding is used in many jobs like software development and robotics.

In summary, computers help us with many tasks, and programming allows us to tell computers what to do. Learning to code is fun and opens many opportunities!

🔧 Step 1: Install Python

> ⛔ **Skip this step if Python is already installed**

1. Visithttps://docs.google.com/document/d/16P-7LH8ebYbVz_Vu3VkLuVKZDDY4iQ0I4Ypqj9TWZzA/edit?usp=sharing
2. Download and install **Python** for your system.

After installing, open **Command Prompt or Terminal** and check:

```
python --version
```

🔧 Step 2: Install PyCharm

> ⛔ **Skip this step if Pycharm is already installed**

1. Go tohttps://docs.google.com/document/d/16P-7LH8ebYbVz_Vu3VkLuVKZDDY4iQ0I4Ypqj9TWZzA/edit?usp=sharing
2. Choose **Community Edition** (Free)
3. Download and install it.

🔧 Step 3: Create Your API Project in PyCharm

1. Open **PyCharm**
2. Click **New Project**
3. Name your project `firstapi`
4. Make sure the **Python interpreter** is selected
5. Click **Create**

🐍 Introduction to Python

What is Python?

Python is a computer language that helps us tell the computer what to do. Just like we use English or Swahili to talk to each other, we use Python to "talk" to the computer.

🧠 What is Syntax?

Syntax is a set of rules, like grammar in English.
If you write a sentence like *"I school go to"*, it sounds wrong. The same way, in Python, if you don't follow the rules, it gives an error.

🔡 Case Sensitivity

Python cares about **capital** and **small** letters.
For example:

- `name` and `Name` are not the same in Python.

So, be careful how you write!

📄 What is a Statement?

A **statement** is one complete instruction in Python.

Like saying: "Turn off the lights." That's one instruction.

In Python, we use a new line to show where one statement ends and the next begins.

⬅️ What is Indentation?

Indentation means leaving a space at the beginning of a line.
Python uses this to know which code belongs together.

It's like a story:

if it is raining:

    take an umbrella

The second line is indented (moved to the right) to show it belongs to the “if” part.

📦 Python Data Types

Data types are like different types of boxes we use to keep different things. You can't keep water in a shoe box, and you can't keep clothes in a bottle!

Python also has different "boxes" (data types) to keep different kinds of information.

---

1. 🔢 Integers (`int`)

These are **whole numbers**.
Examples: `5`, `-10`, `100`
🧒 *Think of counting apples: 1, 2, 3…*

---

2. 💧 Floating-Point Numbers (`float`)

These are **numbers with decimals**.
Examples: `3.14`, `0.5`, `-1.0`
🥛 *Like measuring milk: 1.5 litres*

---

3. 📝 Strings (`str`)

These are **words, sentences, or any characters in quotes**.
Examples: `"Hello"` or `'12345'`
📦 *Like a box full of letters and symbols*

---

4. ✅ Booleans (`bool`)

These are only two answers: **True** or **False**.
Examples: `True`, `False`
💡 *Like a switch: ON or OFF*

---

5. 📋 Lists (`list`)

A **list** is a group of items.
Examples: `[1, 2, 3]`, `["apple", "banana", "cherry"]`
🛒 *Like your shopping list!*

---

6. 🏷️ Dictionaries (`dict`)

A **dictionary** has a **key** (label) and a **value** (meaning).
Example: `{"name": "Amina", "age": 13}`
👩‍🏫 *Like your school ID: your name and age are written with labels.*

---

7. ✍️ Tuples (`tuple`)

A **tuple** is like a list, but you **can't change it**.
Example: `(1, 2, 3)`
✒️ *Like writing with a pen—you can't erase it.*

---

8. 🎒 Sets (`set`)

A **set** is a group of items, but it removes **duplicates**.
Example: `{1, 2, 2, 3}` becomes `{1, 2, 3}`
🧺 *Like a basket that only allows one of each item.*

By learning these data types, you'll understand how to organize your ideas in Python—just like sorting different things into the right boxes at home!

🧱 What is a Variable?

A **variable** is like a **container or a label** that stores information.
You can **name** a variable and **put something inside** it—like a number, a word, or even a list.

🧃 Example in real life:
Imagine a cup with your name on it. You can pour juice into the cup and later change it to water. The cup is like a **variable**, and the juice or water is the **value**.

📘 In Python:

```
name = "Amina"       # name is a variable that stores the string "Amina"

age = 13             # age is a variable storing the number 13
```

🧪 Python Practice Lab – Beginner Concepts

🟩 Lab 1: Playing with Variables and Data Types

```
# 1. Create some variables and print them

name = "Brian"

age = 12

height = 1.55

is_student = True


print("Name:", name)

print("Age:", age)

print("Height in meters:", height)

print("Is a student?", is_student)
```

✏️ **Your Turn**:
Change the name, age, and height to match yours.

🟨 Lab 2: Try All Data Types

```
# 1. Integer

number_of_apples = 5


# 2. Float

price_of_juice = 1.75


# 3. String

favourite_color = "Blue"


# 4. Boolean

has_homework = False


# 5. List

fruits = ["apple", "banana", "mango"]


# 6. Dictionary

student = {"name": "Amina", "age": 13}


# 7. Tuple

days = ("Monday", "Tuesday", "Wednesday")


# 8. Set

unique_numbers = {1, 2, 3, 2, 1}
```

🔍 **Try this**:
Print each variable using `print()`.
Example:

print("My favorite color is", favourite_color)

print("Fruits:", fruits)

print("Student name:", student["name"])

Let's break down the main **data structures—List**, **Set**, **Tuple**, and **Dictionary**

🧺 LISTS – A Row of Boxes

🧠 What is a List?

A **list** in Python is like a **row of labeled boxes** where you can keep many things—like numbers, words, or both.

Imagine you have a **pencil case** with 5 compartments. Each one can hold a different item: a pencil, an eraser, a sharpener, etc.

items = ["pencil", "eraser", "sharpener", "ruler", "pen"]

📍 What is an Index?

Each item in a list has a **position number**, starting from 0.

Index	**0**	**1**	**2**	**3**	**4**
Item	pencil	eraser	sharpener	ruler	pen

print(items[0]) # prints "pencil"

print(items[3]) # prints "ruler"

🎒 SETS – A Bag That Removes Duplicates

🧠 What is a Set?

A **set** is like a bag that **doesn't allow duplicates**. You can throw in anything, but if you throw in the same thing twice, Python only keeps it once.

unique_numbers = {2, 4, 6, 2, 4, 8}

print(unique_numbers)

🗒️ **Output**:

{8, 2, 4, 6}

The repeated 2s and 4s are removed!

💡 **Also**: Sets are **unordered**, so they don't have indexes.

📦 TUPLES – Boxes You Can't Change

🧠 What is a Tuple?

A **tuple** is just like a list, but **you cannot change** what's inside after creating it.

🧃 Real-life example: Imagine a juice box that is sealed. You can look at what flavor it is, but you can't pour new juice inside.

fruits = ("apple", "banana", "cherry")

print(fruits[1]) # prints "banana"

Trying to change it like this: `fruits[1] = "mango"` will give an error.

🗂️ DICTIONARIES – Labels with Info

🧠 What is a Dictionary?

A **dictionary** in Python is like a **school ID card**. Each piece of information has a label (called a **key**) and the actual info (called a **value**).

```
student = {
    "name": "Linet",
    "age": 13,
    "grade": "7B"
}
```

💬 We can get information using the keys:

```
print(student["name"])   # Linet
print(student["grade"])  # 7B
```

> Think of `"name"`, `"age"`, and `"grade"` as labels on the ID, and their values are what's written next to them.

🧺 DATA STRUCTURES EXPLAINED

🧃 LIST – A shelf with things in order

Imagine you have a **shelf** where you keep your toys. You place them in order:

toys = ["car", "robot", "ball"]

You can say:

print(toys[0]) # shows "car"

✅ A **list** stores many things, like numbers or words
✅ You can pick items by their **position** (called an index)
📍 Indexing starts from **0**, not 1. So:

- `toys[0]` is the **first** toy
- `toys[1]` is the **second** toy

🎩 TUPLE – A locked list

Now imagine your teacher gives you a **sealed box** with 3 awards:

prizes = ("medal", "cup", "badge")

You **cannot add, remove, or change** anything inside.

✅ Use when you don't want your list to be changed
✅ Still has positions like a list: `prizes[1]` is "cup"

🔁 SET – A magic bag that removes duplicates

Say you're collecting marbles:

marbles = {"red", "blue", "red", "green"}

You look in your bag:

```
print(marbles)  # Only one "red"
```

✅ A **set** keeps each item **only once**
✅ No order — you can't say "give me the first one"

🧠 **DICTIONARY – A list that uses names instead of numbers**

You have a small card about a student:

```
student = {
  "name": "Amina",
  "age": 10,
  "best_subject": "Math"
}
```

You can ask:

```
print(student["name"])  # shows "Amina"
```

✅ A dictionary stores **key and value** (name and info)
✅ Keys like "name" or "age" tell you **what** the value means

---

🤔 WHAT IS AN INDEX?

An **index** is a number showing **where** something is in a list or tuple.

```
colors = ["red", "green", "blue"]
# red = index 0
# green = index 1
# blue = index 2
```

Think of it like seats in a movie row:

- Seat 0 = first
- Seat 1 = second
- Seat 2 = third

📊 COMPARISON OPERATORS = “Is this bigger? Smaller? Same?”

They ask **questions** and answer `True` or `False` (Yes or No)

Operator	Means	Example	Answer
`==`	is equal	`5 == 5`	True
`!=`	not equal	`5 != 4`	True
`>`	greater than	`7 > 6`	True
`<`	less than	`3 < 5`	True
`>=`	greater or equal	`6 >= 6`	True
`<=`	less or equal	`4 <= 5`	True

🧠 Think like this:

- “Is 6 the same as 6?” → Yes = `True`
- “Is 4 bigger than 10?” → No = `False`

🔁 LOOPS = “Do this again and again”

🧍 FOR loop → “Go through each item”

```
colors = ["red", "blue", "green"]

for color in colors:
    print(color)
```

✅ It looks at each color **one by one** and says it

🔄 WHILE loop → “Keep going as long as it’s True”

```
count = 1

while count <= 3:
    print("Count:", count)
    count = count + 1
```

✅ It counts until it gets to 4, then stops

🧰 FUNCTIONS = "A magic button you press to do something"

You write a **job** once, then press the button to do it anytime.

```
def greet():
    print("Hi there!")


greet()

greet()
```

Function with input:

```
def greet(name):
    print("Hi", name + "!")


greet("Amina")
```

📦 MODULES = “Extra tools you can borrow”

You can use tools others built, like a **calculator**.

```
import math

print(math.sqrt(25))  # square root = 5
```

Another:

```
import random

print(random.randint(1, 10))  # random number between 1 and 10
```

🧪 MINI LAB

🎮 "Grade Checker Game"

```
def check_grade(score):
    if score >= 90:
        print("🏅 You got an A!")
    elif score >= 70:
        print("✅ You passed!")
```

```
    else:
        print("❌ You need to study more.")

# Try it with different scores
check_grade(95)
check_grade(75)
check_grade(60)
```

🧁 "Snack Shelf Example (Lists and Loops)"

```
snacks = ["cookie", "apple", "juice"]
for snack in snacks:
    print("I will eat:", snack)
```

🕹️ "Guessing Game using loops and comparison"

```
secret = 7
guess = int(input("Guess a number between 1 and 10: "))

while guess != secret:
    print("Wrong! Try again.")
    guess = int(input("Guess again: "))

print("Yay! You guessed it!")
```

🏎️ What is Object-Oriented Programming?

Object-Oriented Programming (we call it **OOP**) is like **building real-world things in code**.

You make something like a **blueprint** (called a **class**) and then you can build **real things** (called **objects**) from it.

🚗 Let's Build a Car – In Code!

Imagine you want to make many cars in a game.

All cars have:

- A name (like "Toyota")
- A year (like 2022)
- A price (like 1 million)

Instead of typing each one from scratch, we build a **blueprint** that tells the computer:

> "Here's what every car has and what it can do."

🔨 THE BLUEPRINT (A Class)

```
class Car():

    def __init__(self, modelName, yearm, price):

        self.modelName = modelName

        self.yearm = yearm

        self.price = price
```

🧠 Let's break this down:

- `class Car`: This is the **blueprint** for a car.
- `def __init__`: This is called when you **create a new car**. It's like setting the car's name, year, and price.

- `self`: This means “this car”.
- `self.modelName = modelName`: We give the car its **name**
- `self.yearm = yearm`: We give the car its **year**
- `self.price = price`: We give the car its **price**

💸 A Function Inside the Car (price_inc)

```
def price_inc(self):
    self.price = self.price * 1.15
```

This says:

> “If this car gets more expensive (like adding tax), increase its price by 15%.”

🏁 LET’S MAKE A REAL CAR

```
car1 = Car("Toyota", 2020, 1000000)
print(car1.modelName)  # Toyota
print(car1.price)      # 1000000

car1.price_inc()       # Price goes up
print(car1.price)      # 1150000
```

🎓 OOP WORDS – Kid Friendly

Word	What It Means	Easy Example
Class	A **blueprint** to make something	Like a recipe for cake
Object	A **real thing** made from the class	The cake you bake from that recipe
`__init__`	The setup when we make a new object	Mixing ingredients for a new cake
`self`	Means "**this one**"	Talking about *this car*
Method	A **thing the object can do**	Like "drive" or "honk" or "raise price"

📦 Real-Life Comparison: Toy Factory

- `class Toy`: Blueprint for a toy
- `toy1 = Toy("Teddy", 2023)`: We made a teddy bear toy!
- `toy2 = Toy("Robot", 2024)`: We made a robot toy!

Each toy has its own name and year, but **comes from the same plan**.

🧪 Mini Practice – Create Your Own Animal

```
class Animal:

  def __init__(self, name, sound):
```

```
        self.name = name
        self.sound = sound

    def speak(self):
        print(self.name + " says " + self.sound)

cat = Animal("Kitty", "Meow")
dog = Animal("Doggo", "Woof")

cat.speak()  # Kitty says Meow
dog.speak()  # Doggo says Woof
```

🧠 1. Class – The Blueprint

Think of a **class** like a **recipe** or **design** for something you want to build in code.

```
class Dog:
    def __init__(self, name, color):
        self.name = name
        self.color = color
```

💬 *"We made a Dog plan! Every dog will have a name and a color."*

🧸 2. Object – The Real Thing

An **object** is something **you make** using the class. It's like baking a cake using the recipe.

```
my_dog = Dog("Buddy", "Brown")
```

💬 *"Buddy is a real dog made from the Dog class!"*

⚙️ 3. Attributes – The Things It Has

These are like **facts or features** about the object.

```
print(my_dog.name)   # Buddy
print(my_dog.color)  # Brown
```

💬 *"Name and color are the dog's attributes!"*

🛠️ 4. Methods – The Things It Can Do

Methods are **actions** or **behaviors** the object can do.

```
class Dog:
    def __init__(self, name, color):
        self.name = name
        self.color = color

    def bark(self):
        print(self.name + " says Woof!")

my_dog = Dog("Buddy", "Brown")
my_dog.bark()  # Buddy says Woof!
```

💬 *"bark() is something the dog can DO!"*

📦 5. Encapsulation – Keep Data Safe Inside

Encapsulation means **hiding the stuff inside** an object so we don't break it by accident.

Example: Use underscore _ to show it's meant to be private.

```
class BankAccount:
    def __init__(self, balance):
        self._balance = balance  # don't touch directly

    def show_balance(self):
        print("Your balance is", self._balance)

    def deposit(self, amount):
        self._balance += amount
```

💬 *"The bank keeps your money safe. You can only change it using deposit!"*

👪 6. Inheritance – Kids Get Traits from Parents

Just like **you get traits from your parents**, classes can get code from other classes.

```
class Animal:
    def speak(self):
        print("This animal makes a sound.")


class Dog(Animal):
    def speak(self):
        print("Woof!")


d = Dog()
d.speak()  # Woof!
```

💬 *"Dog gets stuff from Animal, but can change how it speaks!"*

🎭 7. Polymorphism – Many Shapes, One Idea

This means **many things can do the same action in their own way**.

```
class Bird:
    def speak(self):
        print("Tweet!")


class Cat:
    def speak(self):
```

```
    print("Meow!")

for animal in [Bird(), Cat()]:
    animal.speak()
```

💬 *"Both speak(), but one says Tweet and one says Meow!"*

🧬 8. Abstraction – Show Only What's Needed

This means **we hide the hard parts** and only show **what the user needs to know**.

Imagine a remote control:

- You press buttons
- You don't see the wires inside

Example:

```
class TV:
    def turn_on(self):
        print("TV is now ON")

my_tv = TV()
my_tv.turn_on()  # We don't need to know how!
```

💬 *"Just press the button – don't worry about how it works inside!"*

🧪 Mini Practice: Student Class

```
class Student:
    def __init__(self, name, grade):
```

```
        self.name = name
        self.grade = grade

    def show_grade(self):
        print(self.name, "is in Grade", self.grade)

s1 = Student("Liam", 4)
s1.show_grade()  # Liam is in Grade 4
```

🧠 Summary Table (Kid Friendly):

OOP Word	What It Means Like	Easy Example
Class	A recipe/blueprint	Plan for building cars
Object	A real thing	A real car, like “Toyota”
Attribute	Something it has	Name, color, price
Method	Something it does	Drive, speak, bark
Encapsulation	Hide the inside	ATM hides your balance details
Inheritance	Get from parents	Dog inherits from Animal
Polymorphism	Acts same, looks diff	Different animals can `speak()`

Abstraction	Show only what's needed	Remote turns on TV, we don't see wires

🧠 What is Exception Handling?

Sometimes, when a program runs, it makes a **mistake** (called an **error**). Instead of letting the program crash, Python lets you **catch the error** and do something nice, like showing a message.

🧪 Try, Except – Like a Safety Net

Think of `try` as *"try to do something"*, and `except` as *"if it doesn't work, do this instead."*

Example 1 – Dividing Numbers

```
try:
    number = int(input("Type a number: "))
    answer = 10 / number
    print("Answer is", answer)
except:
    print("Oops! Something went wrong.")
```

What it does:

- If you type `2`, it shows: `Answer is 5.0`
- If you type `0`, it says: `Oops! Something went wrong.` (Because dividing by zero is not allowed!)

🎒 Real Life Example

Imagine you're trying to open your school bag.

```
try:
    open("school_bag")
except:
```

```
    print("Oops! The bag is locked.")
```

Even if something goes wrong, the program **won't break**. It will say "Oops!" instead.

🔍 Extra Blocks

`else` – Runs if no error

```
try:
    print("Adding numbers...")
    total = 5 + 5
except:
    print("Something went wrong!")
else:
    print("Great! No problem.")
```

`finally` – Runs always

```
try:
    print("Doing homework...")
except:
    print("Error!")
finally:
    print("Done with work!")  # Always shows this
```

📝 Summary

- `try` – Try something risky
- `except` – What to do if it fails
- `else` – What to do if it works
- `finally` – Do this no matter what

📂 What is File Handling?

Just like you open your **exercise book** to read or write, Python can **open files** on the computer to:

- 📝 Write something
- 📖 Read something
- 🧽 Delete or clean

🗝️ Important Words

- `open()` – To open the file
- `read()` – To read from the file
- `write()` – To write to the file
- `close()` – To close the file (just like closing a book)

✍️ Example 1 – Writing to a File

file = open("my_notes.txt", "w") # "w" = write

file.write("Hello! My name is Hallan.")

file.close()

📄 This makes a file called `my_notes.txt` and writes:
`Hello! My name is Hallan.`

📖 Example 2 – Reading from a File

file = open("my_notes.txt", "r") # "r" = read

```
content = file.read()

print(content)

file.close()
```

🧠 This reads what's inside `my_notes.txt` and shows:

```
Hello! My name is Hallan.
```

➕ Example 3 – Adding More (Append)

```
file = open("my_notes.txt", "a")  # "a" = add (append)

file.write("\nI love coding in Python!")

file.close()
```

📝 Now the file says:

Hello! My name is Hallan.

I love coding in Python!

✅ Using `with` (like magic!)

```
with open("my_notes.txt", "r") as file:
    print(file.read())
```

🪄 Python closes the file **automatically**!

📌 Summary

Word	Meaning
`open()`	Open a file
`read()`	Read from a file
`write()`	Write new stuff in a file
`append()`	Add more to the file
`close()`	Close the file (very important)
`with`	A neat way to open and close

Here's a **super simple file handling example** that uses `try`, `except`, `else`, and `finally`

📁 Example: Reading from a File Safely

```
try:
    file = open("my_story.txt", "r")   # Try to open the file
    content = file.read()              # Try to read from it
except FileNotFoundError:
    print("Oops! The file was not found.")  # If the file doesn't exist
else:
    print("File opened successfully!")
    print("Here's what's inside the file:")
    print(content)                     # Show the content
finally:
    print("This message always shows, even if there's an error.")
```

💡 What Happens?

1. ✅ If the file **exists**, it reads the story and shows it.
2. ❌ If the file is **not there**, it says:
   `"Oops! The file was not found."`
3. 🧹 `finally` always runs, like the bell at the end of a lesson.

📌 Summary of Keywords in This Example:

Word	What it does
`try`	Tries to do something risky
`except`	Catches the error if something goes wrong
`else`	Runs only if there’s no error
`finally`	Runs every time, no matter what