Python Programming for Everybody – Coursera

Course 1: Getting Started with Python

Chapter 1: Why we program?

1.1 – Programming Overview

- A program is a sequence of stored instructions that tells a computer exactly what to do.
- Computers aren't intelligent — they only execute the commands given to them.
- Well-written instructions let computers perform complex tasks such as data analysis or voice recognition.

Role of the Programmer

- Programmers act as intermediaries between hardware and users.
- Hardware provides capability; software gives it purpose.
- Programming means imagining user needs and building reliable software to meet them.

Precision and Errors

- Computers are exact and can't fix mistakes.
- A single syntax or logic error can cause failure.
- Humans overlook small errors — computers require perfect accuracy.

Key Takeaways

- Programming lets you build tools, automate tasks, and analyze data.
- It teaches logical, step-by-step thinking.
- It changes your view — from using technology to creating it.

1.2 – Hardware Overview

Computers consist of key parts that work together to run programs. Understanding them shows how Python executes inside the machine.

Main Components

- CPU (Central Processing Unit): The "brain." Executes instructions using the fetch–execute cycle billions of times per second.
- Main Memory (RAM): Fast, temporary workspace for active programs. Data is lost when power is off.
- Secondary Memory: Permanent storage — hard drives, SSDs, or USBs — keeps data after shutdown.
- Input/Output (I/O): Communication with users.
  - *Input:* keyboard, mouse, camera
  - *Output:* monitor, printer, speakers
- Motherboard: The main board connecting CPU, memory, storage, and I/O.

How It Works

1. Save program to secondary storage.
2. Load into main memory when run.
3. CPU fetches and executes instructions.
4. After shutdown, RAM clears but files remain stored.

Key Takeaways

- CPU: Executes instructions
- RAM: Temporary workspace
- Storage: Permanent data
- I/O: User interaction
- Motherboard: Connects everything

1.3 - Elements of Python

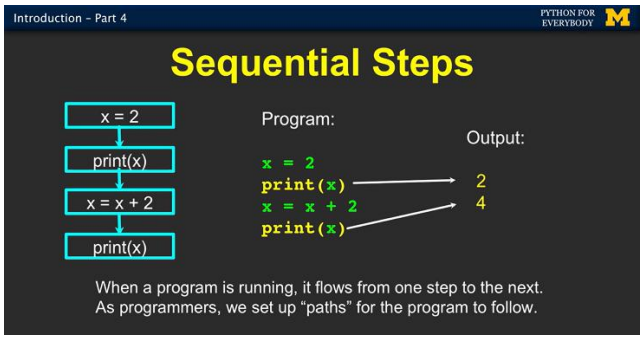
- **Assignment & Variables:** = assigns a value to a variable, not tests equality — e.g. x = x + 1 updates x.
- **Expressions & Operators:** Operators (+, =, -, *) combine values; functions like print() Process inputs.
- **Syntax & Errors:** Python enforces strict syntax. Invalid structure / misspelled statements raise SyntaxError.
- **Reserved Words:** Certain keywords (for, if, import, etc) have predefined meanings & can't be used as vars.
- **Program Structure:** Code builds up from statements (lines) to scripts (files), analogous to alphabets → words → sentences → paragraphs → stories in natural language.

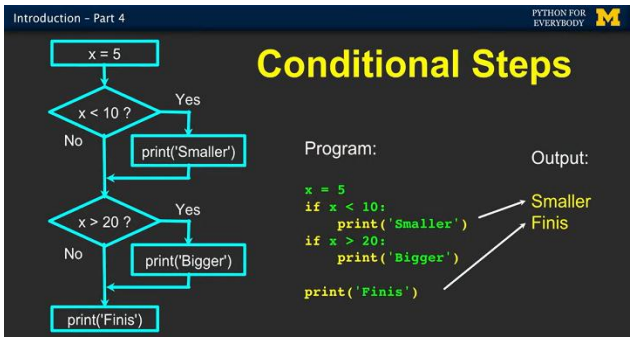
1.4 – Program Overflow

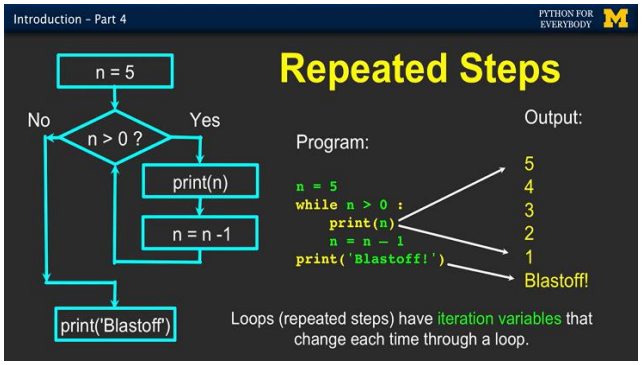
A program consists of a sequence of ordered steps, similar to a recipe or instructions, where some steps may be skipped if certain conditions apply, and others may need to be repeated.

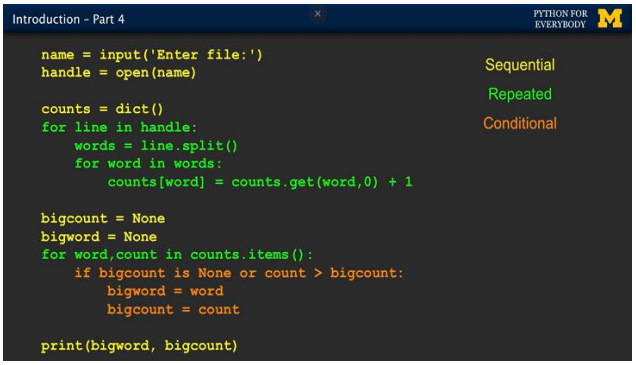
Four Basic Programming Patterns

1. Sequential: Do one thing after another.
2. Conditional: Make decisions using if statements (do something only if a condition is true).
3. Repetition (Loops): Repeat steps using while or for. Loops use iteration variables to avoid running forever.
4. Store and Retrieve: Using variables and data structures (covered later)









Flow of Code

- Code runs sequentially unless redirected by conditionals or loops.
- Indentation defines what code belongs inside loops or conditionals.
- Loops and conditionals can be **nested** inside each other to create more complex behavior.

Key Takeaway:

Programs are built by combining small, basic patterns—like doing steps in order, making decisions, repeating tasks, and storing data—to create more complex behavior.

Chapter 2: Variables and Expressions

This chapter explains how a program uses computer memory to store, retrieve, and process data.

2.1 - Constants, Reserved Words & Variables

- **Constants**

  Fixed values that don't change:

  Example: x = 123 # Step 1: Python stores 123 in memory and makes the label x point to it.

  x = 98.6 # Step 2: Python creates a *new* value 98.6 and makes x point to that instead.

  Python sees that the old value is **no longer referenced**, so its automatic system called the **garbage collector** eventually frees that memory. So the old constant doesn't "move" or "change" — it just sits unused until Python safely deletes it from memory.

- **Reserved Words:**

  Special Python words (cannot be used as variable names): if, else, class, for, return, while … etc

- **Variables & Assignment**

  Variables store data in memory:

  x = 12.2

  y = 14

  x = 100

  🧠 **Python does:** [x: 12.2] → [y: 14] → [x: 100]

  Rules:

  - Start with a **letter** or **underscore** (_name). May contain **letters, numbers, underscores.**
  - Are **case-sensitive** (Spam, spam, and SPAM are different)

- **Mnemonic (Meaningful) Names**

  Use names that make sense for humans:

Poor	Better	Best
x, y	a, b	hours, rate, pay

  Python doesn't care — but humans do.

- **Assignment Direction**

  = means "store result," not mathematical equality:

  x = x + 1 → x becomes x + 1

  If x = 6, it becomes 7.

2.2 - Numerical Expressions in Python

- Python uses operators: +, -, *, /, ** (power), % (remainder).
- % gives the remainder; useful for even/odd checks, dice rolls, or cards.
- ** calculates powers, e.g., 4 ** 3 = 64.
- Order of operations: **parentheses → power → multiplication/division/remainder → addition/subtraction**.
- Use parentheses to make expressions clear and avoid mistakes.

2.3 – Variable Types

Python tracks both the **value** and **type** of each variable, which determines what operations can be done with it.
The main types are:

- int – whole numbers (e.g., 5, 8)
- float – decimals (e.g., 3.14, 4.0)
- str – text (e.g., "hello", '123')

➢ **Operators Work Differently by Type (How Types work?)**

- 1 + 4 → **adds numbers** → result is 5
- "hello" + " there" → **joins strings** → result is "hello there"
- "hello" + 1 → ❌ **error!** Can't combine string and number

When Python doesn't understand how to combine 2 different types, it raises a **TypeError** (shown as a **traceback**)

➢ **Checking and Converting Types**

- Use type(x) → tells you what type a variable or constant is
- Convert types using functions:
  - int(98.6) → 98 truncates the decimal — it doesn't round.
  - int('123') → 123
  - float('98.6') → 98.6
  - str(42) → '42'

If you try to convert something invalid (like int('abc')), Python gives an error.

➢ **Division Rules**

- In **Python 3**, division (/) always gives a **float**:
  - 9 / 2 → 4.5
  - 99 / 100 → 0.99
- In **Python 2**, 9 / 2 would give 4 (it cut off decimals).

Note About Floating Points

- Floating point numbers can represent very large/small values but may be slightly imprecise.
- Because of rounding issues, **don't use floats for money**.

➢ **Input from the User**

- input("Who are you? ") → displays prompt and waits for typing.
- Whatever the user types is stored as a **string**.

Example:

```python
name = input("Who are you? ")
print("Welcome", name)
```

  - If you type Chuck, output → Welcome Chuck
    Even if you type numbers, Python still treats them as strings unless you convert them.

➢ **How to Write a Comment**

```python
# This is a comment
print("Hello")  # This prints a message
```

Everything after # on that line is ignored by Python.
Why use comments?: Help others (and your future self) understand the program

Chapter 3: Conditional Code

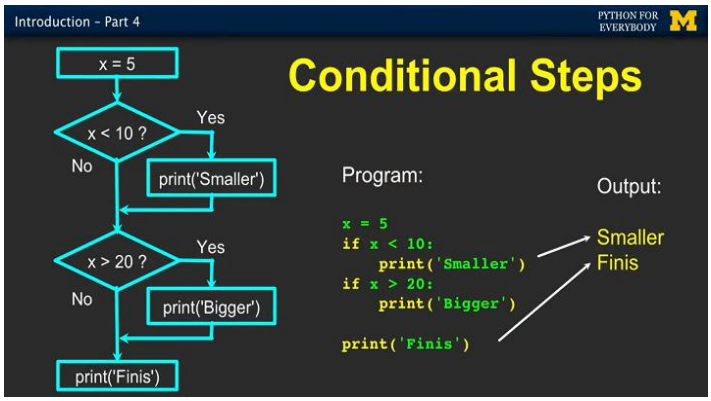
In this section we move from sequential code that simply runs one line of code after another to conditional code where some steps are skipped. It is a very simple concept - but it is how computer software makes "choices".

➢ **Part I –** if statement

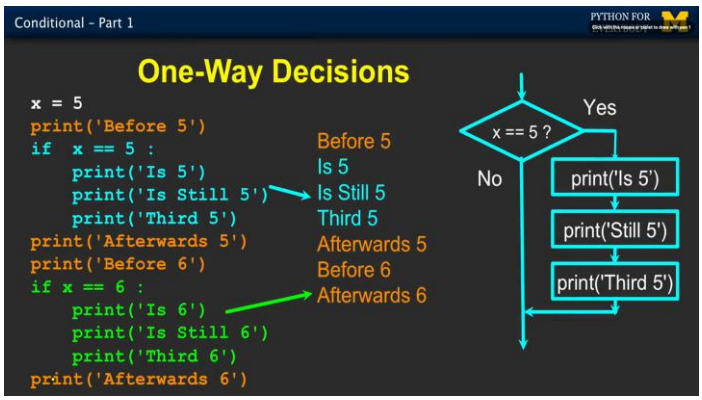
Conditional statements let Python make decisions — they allow code to run *only if* certain conditions are true. The main tool for this is the **if statement**.

Syntax -> if condition:
indented block runs if condition is true

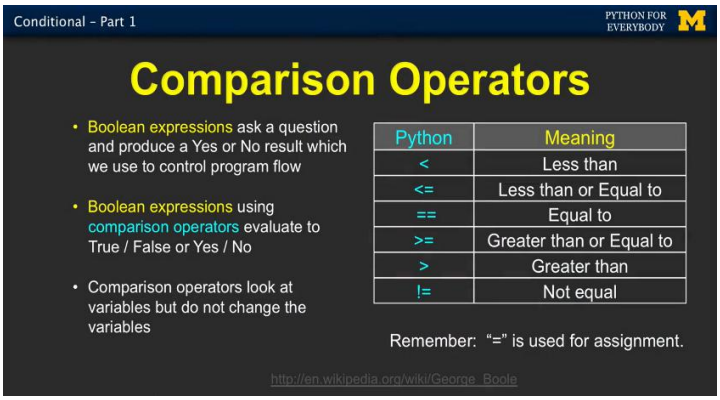
Example



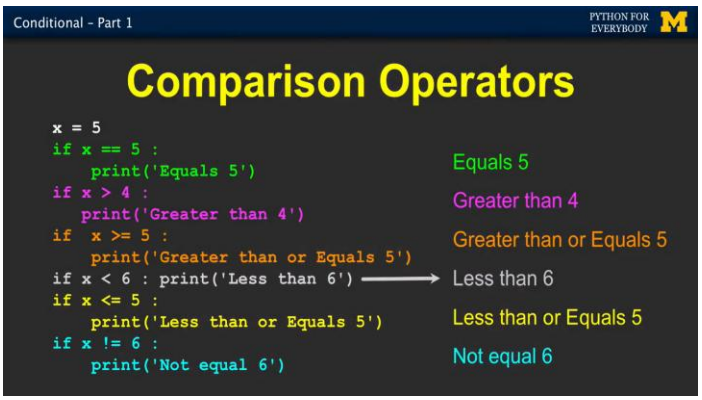
In Python, indentation matters.



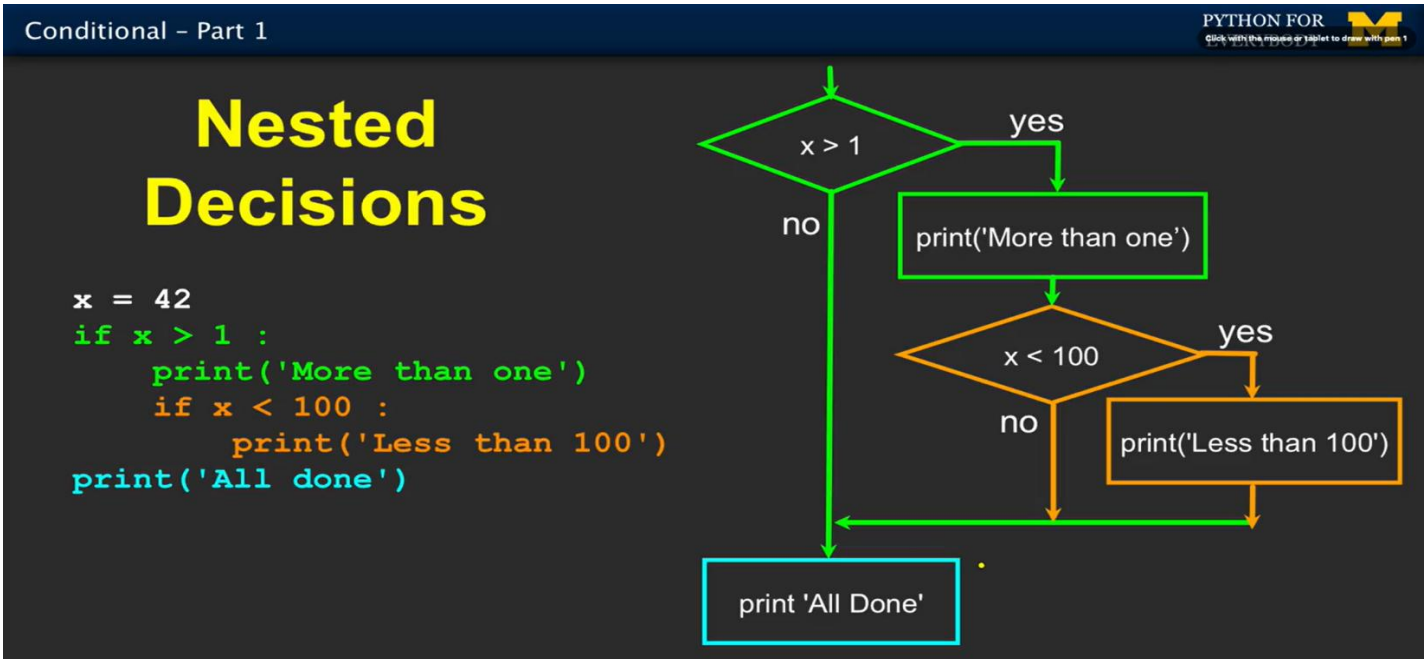
Comparison Operators



Example

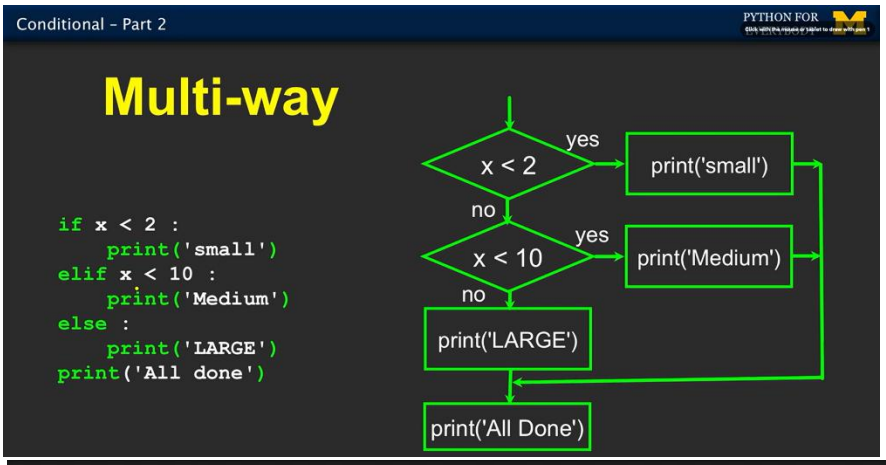


Nested Blocks - You can put one if or loop inside another. Each deeper block is indented further.



➢ **Part II –** elif & else statements

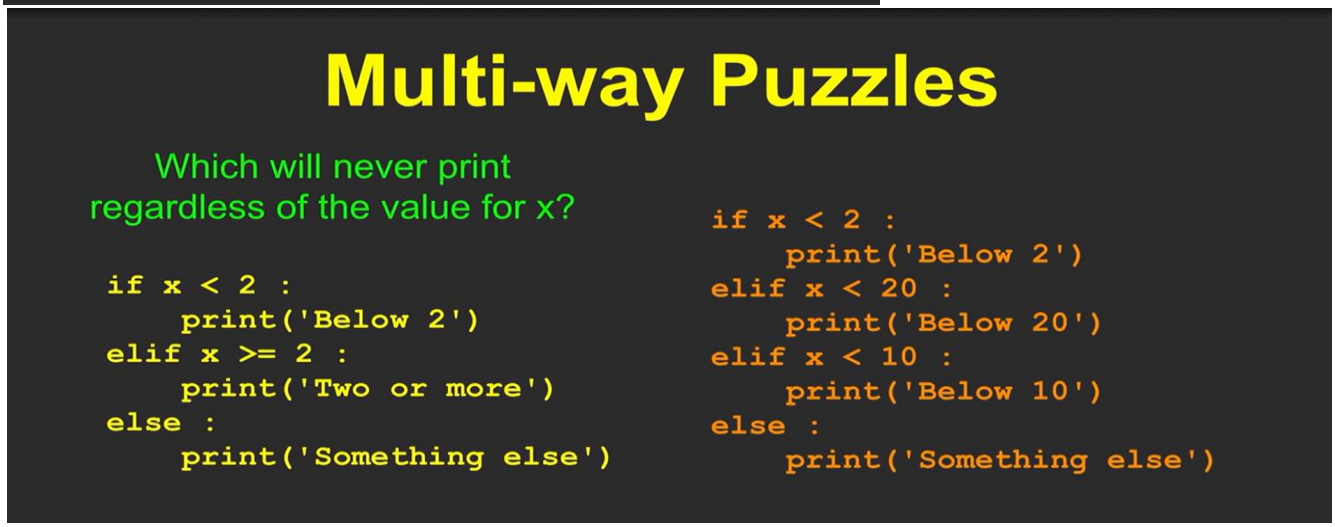
elif means **"else if" — used for** multi-way decisions**.**



1st: Python checks each condition in order, and only one block runs (the first true one).

2nd: You can have **many elifs**, but be careful — earlier conditions can make later ones unreachable.

3rd: **else** is optional; if it's missing and no condition is true, nothing happens.



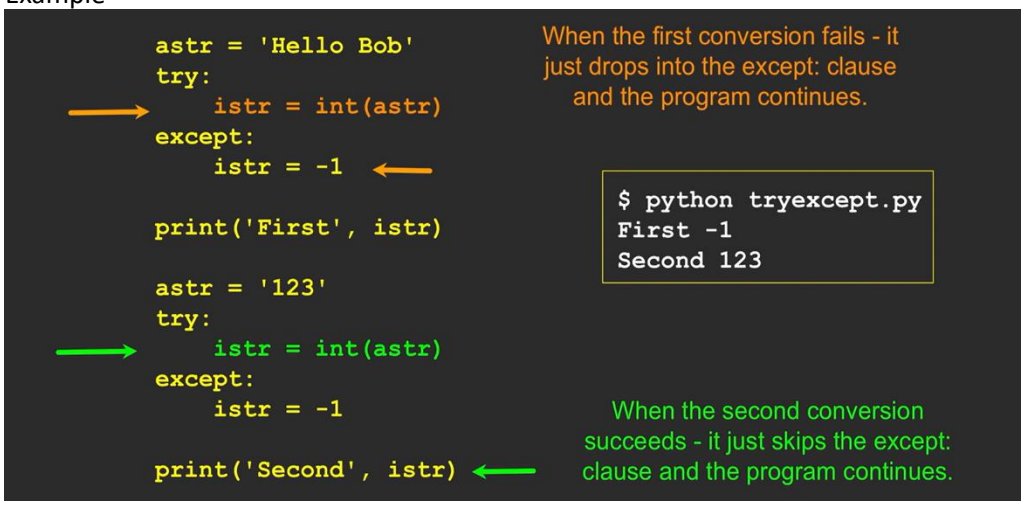
➢ **Part III –** try and except

A way to handle errors in your code without making your program crash. Act like an **insurance policy** — they let your program **keep running** even if something goes wrong. It's a way to eliminate the **tracebacks**.

🧩 Structure

```
try:
    # code that might fail
except:
    # code to run if there's an error
```

Example



💡 Key Points

🔹 try/except **prevents tracebacks** (error messages).

🔹 Use it only around **dangerous lines** — not your entire program.

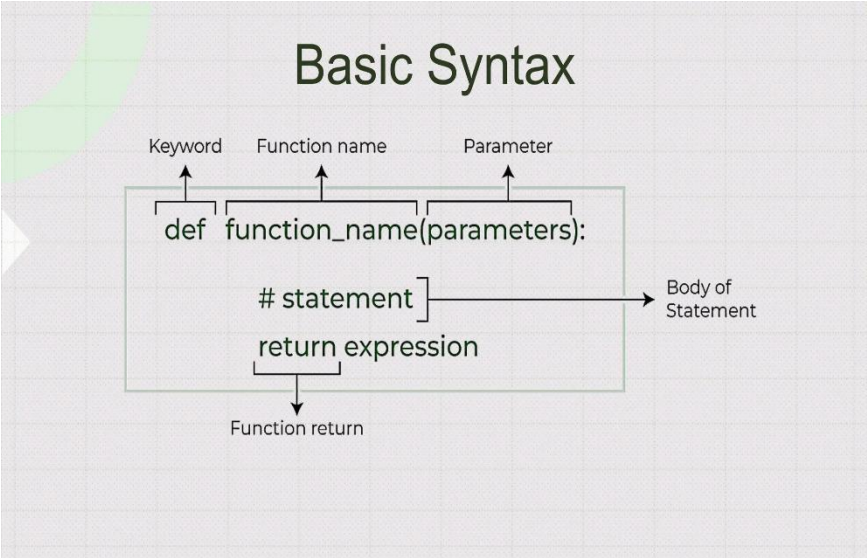
🔹 If nothing goes wrong, the except part is **skipped**.

🔹 A good program **handles user mistakes** gracefully.

Chapter 4: Functions

4.1 – Using Functions (Store and Reuse Pattern)

Functions allow you to **store code once** and **reuse it** without repeating yourself (DRY principle).



Defining a Function

🔹 **def** tells Python "I'm defining a function." but it **does not run it**. Func body is Indented

Parameters vs Arguments

🔹 Parameters are placeholders for the data we can pass to functions.

🔹 Arguments are the actual values we pass.

Return statement does two things **:**

1. **Stops the function** and exits back to where it was called.
2. **Gives back a value** (the "residual value") to that caller.

We have two types of arguments:

b. **Positional arguments**: their position (order) matters

c. **Keyword arguments**: position doesn't matter - we prefix them with the parameter name.

Examples:

```
greet_user("John", "Smith")                      # Positional arguments
calculate_total(order=50, shipping=5, tax=0.1)   #  Keyword arguments
```

Flow Graph (Function Execution)

```
Main Program
     |
     v
 Define function  -----> Stored, not executed
     |
     v
  Call function
     |
     v
  Execute function code
     |
     v
 Return to main program
```

Takeaway:

- Functions store reusable code
- They can take inputs (arguments)
- They can return outputs (return)
- Reduce repetition and make code cleaner

Chapter 4: Loops and iteration

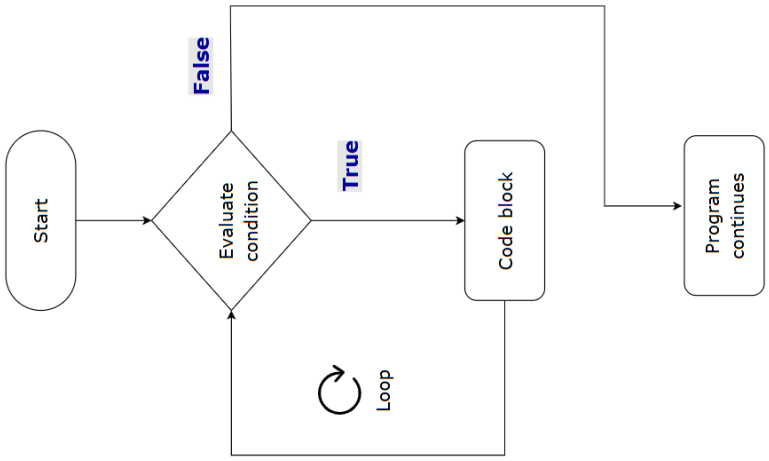
Loops and iteration complete our four basic programming patterns. Loops are the way we tell Python to do something over and over. Loops are the way we build programs that stay with a problem until the problem is solved.

Types of Loops in Python

1. **while loops** → *Indefinite loops*
   Used when you **don't know in advance** how many times you'll need to repeat something.
   Keeps looping **while** a condition is true.

2. **for loops** → *Definite loops*
   Used when you **know** how many times to repeat (like looping over a list or range).

Example: while Loop



A condition gives a **true** or **false** result—if **true**, the code **runs**; if **false**, it's **skipped**—just like an **if statement**. But unlike **if**, the **loop** goes back and **checks again**, repeating until the **condition** becomes **false**

Rotate for a clear view.

Infinite Loop

Each **loop** uses an **iteration variable** — something that **changes** each time (e.g, **n = n - 1**). If that **variable** doesn't **change**, the **condition** stays **true** forever — creating an **infinite loop**, which can **freeze the computer** and **drain the battery**.

Python also provides two special commands for controlling loops:

- **break** → immediately **exits** the loop.
- **continue** → **skips** the rest of the current iteration and goes back to the top of the loop.

These tools help you **control when and how a loop ends**.
A well-designed loop always has a clear **exit condition** so it doesn't run forever.

Solving Loops

The trick is "**knowing**" about the whole loop when you're stuck writing code that only sees one **entry** at a time

```
+--------------------------------+
| Set initial variables          |
| to starting values             |
+--------------------------------+
for item in collection:
   +-----------------------------------------+
   | Do something to each entry              |
   | separately, updating a variable         |
   +-----------------------------------------+
+-----------------------------------------+
| After loop, examine variables           |
| for final answer                        |
+-----------------------------------------+
```

Gurmad Harir
November 2025