Foundations of Computing & Programming

Python Course · Module 1

ElnurBDa

What is Programming?

Programming is the process of giving instructions to a computer in a language it understands.

Computers themselves only understand **machine code**, so programming languages allow humans to write instructions more easily.

Programming is used for:

- Automation
- Web development
- **Cybersecurity**
- Data analysis
- System administration

Real-world examples:

- ATMs
- Browsers
- Mobile apps
- Network scanners (Nmap, etc.)

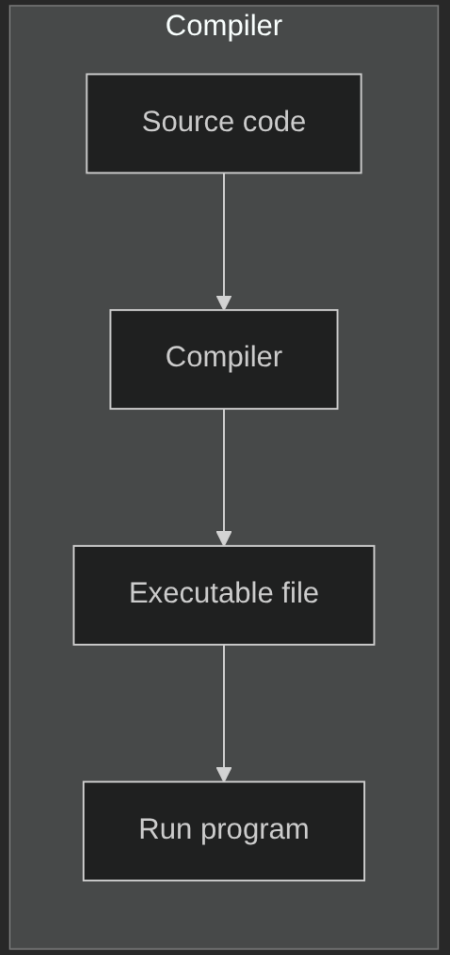
Why Programming Matters

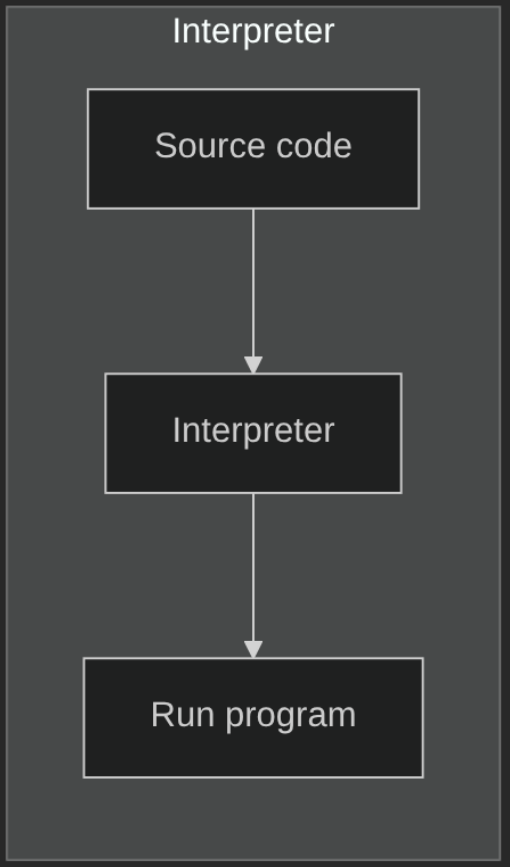
Programming allows you to:

- Solve problems
- Automate repetitive tasks
- Build software
- Understand how computers actually work
- Strengthen cybersecurity skills (scripting, automation)

It is a foundational skill for modern IT and cybersecurity professionals.

Compilers vs Interpreters





Compiler

A **compiler** converts the entire source code into machine code **before** the program runs.

Characteristics:

- Produces a standalone executable
- Faster execution
- Errors found before running

Examples:

- C
- C++
- Go
- Rust

▒▒▒ Example:

hello.c

```c
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

```sh
# compile
gcc hello.c -o hello
# run
chmod +x hello
./hello
# Result: Hello, world!
```

Interpreter

An **interpreter** executes code **line by line**.

Characteristics:

- More flexible
- Easier to test small parts
- Slower execution
- Errors show up while running

Examples:

- Python
- JavaScript
- Ruby

▒▒ Example:

hello.py

```
print("Hello, world!")
```

```
# run
python hello.py
# Result: Hello, world!
```

Algorithms – Introduction

An **algorithm** is a step-by-step procedure to solve a problem.

Characteristics of good algorithms:

- Clarity
- Efficiency
- Consistency
- Finite steps

Common examples in daily life:

- Sorting items
- Making tea
- Following a recipe
- Unlocking your phone

Algorithm Examples

Real-life algorithm example (Making Tea):

1. Boil water
2. Place tea bag in cup
3. Pour water
4. Wait 3–5 minutes
5. Remove tea bag
6. Drink

Pseudocode example:

```
input number
if number > 0:
    print("Positive")
else:
    print("Non-positive")
```

Create a Simple Algorithm – Activity

Write an algorithm for:

1. Logging into a website
2. Buying a product online
3. Starting a computer

Goal: describe steps **clearly** so that even a computer could follow them.

Installing Python

Check whether Python is installed:

```
python3 --version
```

Install on Linux (Debian/Ubuntu):

```
sudo apt update
sudo apt install python3
```

Install on macOS (Homebrew):

```
brew install python
```

Install on Windows:

- Use Microsoft Store
- Or download installer from python.org

Enter interactive mode (REPL):

```
python3
```

Exit REPL with:

```
exit()
# or press Ctrl+D on Linux/macOS
```

Running Python Code

Run a script file:

```
python3 script.py
```

Simple script example (hello.py):

```
print("Hello, world!")
```

Then run:

```
python3 hello.py
```

Python Basics

Python Course · Module 2

ElnurBDa

Variables

A variable stores a **reference** to a value.

Created by assignment:

```
name = "Alice"
age = 20
height = 1.75
```

Rules:

- Start with a letter or _
- Letters, digits, underscores allowed
- Case-sensitive: count ≠ Count

Data Types

▒▒▒ int

Whole numbers, positive or negative.
Example:

```
age = 25
```

▒▒▒ float

Decimal numbers.

```
pi = 3.14
```

▒▒▒ str

Text data in quotes.

```
city = "Baku"
```

▒▒▒ bool

Logical values: True or False.

```
is_student = True
```

▒▒▒ NoneType

Represents “no value”.

```
result = None
```

Input & Output

▒▒▒ Output (stdout)

```
name = "ElnurBDa"
print("Hello")
print("User:", name)
print(f"The F String is used by {name}")
```

```
──────────────── [finished] ────────────────
```

```
Hello
User: ElnurBDa
The F String is used by ElnurBDa
```

▒▒▒ Input (stdin)

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

`input()` **always** returns `str`.

Convert for numeric use:

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

Arithmetic Operators

Operator Summary

```
+   addition
-   subtraction
*   multiplication
/   division (float)
//  floor division (drops decimals)
%   remainder
**  exponent
```

Examples

```
10 + 3   # 13
10 - 3   # 7
10 * 3   # 30
10 / 3   # 3.333...
10 // 3  # 3
10 % 3   # 1
2 ** 3   # 8
```

Comparison Operators

```
x == y   # equal
x != y   # not equal
x > y
x < y
x >= y
x <= y
```

Example:

```
score = 85
score >= 90   # False
score < 100   # True
```

Logical Operators

```
and   # both conditions True
or    # at least one True
not   # reverses boolean
```

Example:

```
age = 20
has_id = True

age >= 18 and has_id   # True
age < 18 or has_id     # True
not has_id             # False
```

Assignment Operators

Shorthand updates:

```
x = 5
x += 2   # 7
x -= 1   # 6
x *= 3   # 18
x /= 2   # 9.0
```

Used often in loops and accumulations.

Type Casting (Conversion)

Convert between types:

```
int("10")      # 10
float("3.14")  # 3.14
str(123)       # "123"
bool("hi")     # True
bool("")       # False
```

Used for:

- numeric input
- formatting
- arithmetic

Example

```python
name = input("Enter your name: ")
age = int(input("Enter your age: "))

next_age = age + 1
is_adult = age >= 18
message = f"{name}, next year you will be {next_age}."

print(message)
print("Adult status:", is_adult)
```

Mini Task

Create a script that:

1. Asks for two numbers
2. Converts them to integers
3. Calculates:
   - sum
   - difference
   - product
4. Prints results

Data Structures

Python Course · Module 3

ElnurBDa

Data Structures – Overview

Python provides multiple built-in structures for organizing data:

- **Lists**
- **Strings**
- **Tuples**
- **Sets**
- **Dictionaries**
- Choosing the right structure

Each structure has unique properties and use-cases.

Lists

Lists store **ordered, changeable** sequences.

```
numbers = [10, 20, 30]

# Operations:
numbers.append(40)
numbers[1] = 99
numbers.remove(10)
print(numbers)

# List indexing:
print(numbers[0])
print(numbers[-1])
print(numbers[1:3])
```

———————— [finished] ————————

```
[99, 30, 40]
99
40
[30, 40]
```

List Characteristics

Advantages:

- Mutable
- Dynamic size
- Supports duplicates

Use-cases:

- Collections
- Queues
- Storing ordered items

Strings

Strings represent **text**.

Properties:

- Ordered
- Immutable
- Indexable
- Iterable

```
s = "Hello"

a = s[0]       # first character
b = s[-1]      # last character
c = s.upper()  # all upper chars
d = len(s)     # Length of the string

print(a, b, c, d)
```

———————————— [finished] ————————————

```
H o HELLO 5
```

Tuples

Tuples store **ordered, immutable** sequences.

```
point = (10, 20)
```

Benefits:

- Fast
- Secure from modification
- Memory-efficient

Tuple unpacking:

```
x, y = point
```

Tuple vs List

- Need modification → **List**
- Need fixed, reliable structure → **Tuple**
- Performance required → **Tuple**

Sets

Sets store **unordered, unique** elements.

```
colors = {"red", "blue", "green"}
```

Operations:

```
colors.add("yellow")
colors.remove("red")
```

Useful for:

- membership checks
- removing duplicates
- mathematical operations

Set Operations

```
a = {1, 2, 3}
b = {3, 4, 5}

x1 = a | b    # union
x2 = a & b    # intersection
x3 = a - b    # difference
x4 = a ^ b    # symmetric difference

print(x1, x2, x3, x4)
```

```
──────────────────── [finished] ────────────────────
```

```
{1, 2, 3, 4, 5} {3} {1, 2} {1, 2, 4, 5}
```

Dictionaries

Dictionaries store **key : value** pairs.

```
user = {
    "name": "Alice",
    "age": 25,
    "city": "Baku"
}

# Access:
n = user["name"]
print(n)
print()

# Modify:
user["age"] = 26
user["country"] = "Azerbaijan"
print(user)
```

———————————— [finished] ————————————

```
Alice

{'name': 'Alice', 'age': 26, 'city': 'Baku', 'country':
'Azerbaijan'}
```

Dictionary Operations

```
user = {
    "name": "Alice",
    "age": 25,
    "city": "Baku"
}

a = user.keys()
b = user.values()
c = user.items()
d = "city" in user  # check key existence
e = "Baku" in user

print(a)
print(b)
print(c)
print(d)
print(e)
```

———————————— [finished] ————————————

```
dict_keys(['name', 'age', 'city'])
dict_values(['Alice', 25, 'Baku'])
dict_items([('name', 'Alice'), ('age', 25), ('city',
'Baku')])
True
False
```

Dictionaries are ideal for structured data.

Example

```python
student = {
    "name": "Elnur",
    "scores": [90, 85, 97],
    "passed": True
}

average = sum(student["scores"]) / len(student["scores"])

print(student["name"], "average:", average)
```

———————————— [finished] ————————————

```
Elnur average: 90.66666666666667
```

Choosing the Right Data Structure

Type	Ordered	Mutable	Unique	Best For
String	Yes	No	N/A	Text
List	Yes	Yes	No	General collections
Tuple	Yes	No	No	Fixed data
Set	No	Yes	Yes	Unique items
Dict	Keys no	Yes	Keys unique	Structured data

Mini Task — Apply Everything

Create a script that:

1. Stores student info in a dictionary
2. Contains: name, age, list of grades
3. Calculates average grade
4. Prints a formatted summary

Control Flow

Python Course · Module 4

ElnurBDa

Control Flow — Overview

Control flow decides **how a program behaves**:

- Conditional statements
- Logical operators
- Loops (`for`, `while`)
- Loop control (`break`, `continue`, `pass`)

This module teaches how Python makes decisions and repeats actions.

Conditional Statements

Basic structure:

```
if condition:
    block
```

A condition must evaluate to **True** or **False**.

Example:

```
age = 20
if age >= 18:
    print("Adult")
```

———————— [finished] ————————

```
Adult
```

---

if / else

```
if condition:
    action_if_true
else:
    action_if_false
```

Example:

```
temperature = 10

if temperature > 15:
    print("Warm")
else:
    print("Cold")
```

———————— [finished] ————————

```
Cold
```

if / elif / else

Used when multiple conditions must be checked in order.

```
grade = "?"
score = 75

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
else:
    grade = "D"

print(grade)
```

———————— [finished] ————————

```
C
```

Logical Operators in Conditions

```
and   # both True
or    # one True
not   # invert
```

Examples:

```
age = 20
has_id = True

age >= 18 and has_id
age < 18 or has_id
not has_id
```

Nested Conditions

```
age = 25
citizen = True

if age >= 18:
    if citizen:
        print("Eligible")
    else:
        print("Not eligible")
```

———————— [finished] ————————

```
Eligible
```

Avoid deep nesting when possible.

---

for Loop — Basics

Iterates over a sequence:

```
for x in [1, 2, 3]:
    print(x)
```

```
──────────────────── [finished] ────────────────────
```

```
1
2
3
```

Strings:

```
for char in "Python":
    print(char)
```

```
──────────────────── [finished] ────────────────────
```

```
P
y
t
h
o
n
```

---

for Loop with range()

```
for i in range(5):
    print(i)
```

```
──────────────────── [finished] ────────────────────
```

```
0
1
2
3
4
```

Custom range:

```
for i in range(2, 10, 2):
    print(i)
```

```
──────────────────── [finished] ────────────────────
```

```
2
4
6
8
```

---

while Loop

Repeats while a condition is **True**.

```
count = 0
while count < 5:
    print(count)
    count += 1
```

———————————— [finished] ————————————

```
0
1
2
3
4
```

Loop Control — break

Stops the loop immediately.

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

———————————— [finished] ————————————

```
0
1
2
3
4
```

Loop Control — continue

Skips current iteration.

```
for i in range(5):
    if i % 2 == 0:
        continue
    print(i)
```

──────────────── [finished] ────────────────

```
1
3
```

Loop Control — pass

`pass` does nothing — placeholder.

```
for i in range(3):
    pass
```

———————————— [finished] ————————————

```
for i in range(3):
```

———————————— [finished with error] ————————————

```
  File "/tmp/nix-shell-25740-0/.presentermIOtbHV/snippet.py", line 1
    for i in range(3):
IndentationError: expected an indented block after 'for' statement on
line 1
```

Example

```python
total = 0

for i in range(1, 6):
    if i % 2 == 0:
        total += i
    else:
        continue

print("Sum of even numbers:", total)
```

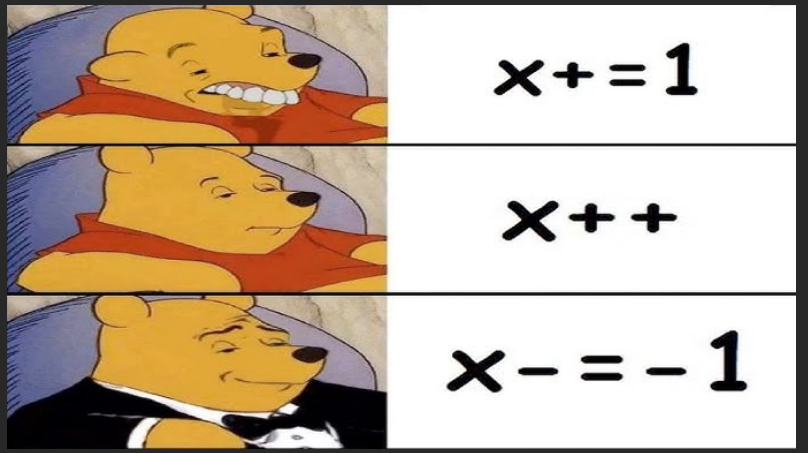
———————— [finished] ————————

```
Sum of even numbers: 6
```

Mini Task

Write a script that:

1. Asks the user for a number
2. Prints all numbers from 1 to that number
3. Prints only **even numbers**
4. Skips odd numbers using `continue`
5. Stops entirely if number exceeds 100 using `break`



Functions & Modular Code

Python Course · Module 5

ElnurBDa

Functions — Introduction

Functions group reusable code into named blocks.

Benefits:

- Avoid repetition
- Improve readability
- Easier debugging
- Modular structure

Basic structure:

```
def name():
    pass
```

Creating Functions

```
def greet():
    print("Hello!")
```

Call:

```
greet()
```

Functions must be defined **before** calling.

Parameters

```
def greet(name):
    print("Hello,", name)
```

Multiple parameters:

```
def add(a, b):
    return a + b
```

Return Values

```
def square(x):
    return x * x
```

Functions without `return` return `None`.

Multiple Returns

```
def values():
    return 10, 20, 30
```

Unpacking:

```
a, b, c = values()
```

Variable Scope

```
x = 10      # global

def func():
    y = 5   # local
```

Local variables override global ones inside functions.

---

global Keyword

```
count = 0

def inc():
    global count
    count += 1
```

Use sparingly; global state makes code harder to manage.

Lambda Functions

```
square = lambda x: x * x
```

Often used inline:

```
add = lambda a, b: a + b
```

Higher-Order Functions

```
nums = [1, 2, 3, 4]

squares = list(map(lambda x: x*x, nums))
```

Using normal function:

```
def sq(x): return x*x
```

Modules — Importing

```
import math
math.sqrt(16)
```

Selective:

```
from math import sqrt
```

Rename:

```
import math as m
```

Built-in Modules

Examples:

- math
- random
- os
- sys
- datetime

Example:

```
import random
x = random.randint(1, 10)
```

Creating Your Own Module

File:

```
mymath.py
```

Content:

```
def add(a, b):
    return a + b
```

Usage:

```
import mymath
mymath.add(2, 3)
```

Organizing Code

```
project/
├── main.py
├── utils.py
└── math_ops.py
```

```python
from utils import greet
from math_ops import add
```

Example

```python
# utils.py
def greet(name):
    return f"Hello {name}"

# math_ops.py
def multiply(a, b):
    return a * b

# main.py
from utils import greet
from math_ops import multiply

print(greet("Elnur"))
print(multiply(4, 5))
```

Mini Task

Create a project with:

1. main.py
2. converter.py with:
   - km_to_miles(km)
   - celsius_to_fahrenheit(c)
3. Import into main.py
4. Ask user input and print converted values
