Introduction to Python Programming

Welcome!

- No previous coding experience needed!
- Today you’ll learn:
    - What Python is
    - How to write simple programs
    - Key concepts: variables, data types, control flow, functions, input/output, and collections

What is Python?

- Python is a **popular, easy-to-read** programming language.
- Used in:
    - Web development
    - Data science
    - AI & machine learning
    - Automation
- Python code looks like plain English.
- Current version: **Python 3.12**

Your First Python Program

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

- `print()` shows text on the screen
- Strings are inside quotes `"..."`
- Run your app
    - Open VS Code
    - Create new file `code.py`
    - Save file to some directory
    - Open command prompt and go the directory: `cd <path to your dir>`
    - Run your app `python code.py`

The `print()` Function

```
 print("Hello", "world")
print("Age:", 25)
print("Sum:", 2 + 3)
```

Advanced print options:

```
 print("A", "B", "C", sep="-")  # A-B-C
print("No newline", end="...")
print("Continued")
```

Types

Category	Type	Description	Example
Basic Data Types	`int`	Integer numbers	`x = 42`
	`float`	Floating-point numbers	`x = 3.14`
	`complex`	Complex numbers	`x = 2 + 3j`
	`bool`	Boolean values	`x = True`
	`str`	Text strings	`x = "hello"`
Sequence Types	`list`	Mutable ordered collection	`x = [1, 2, 3]`
	`tuple`	Immutable ordered collection	`x = (1, 2, 3)`
	`range`	Sequence of numbers	`x = range(5)`
Set Types	`set`	Mutable unordered collection of unique items	`x = {1, 2, 3}`
	`frozenset`	Immutable version of a set	`x = frozenset([1, 2, 3])`
Mapping Type	`dict`	Key-value mapping	`x = {"a": 1, "b": 2}`

Variables and `type()`

```
 name = "Alice"
age = 30
is_admin = True

print(type(name))      # <class 'str'>
print(type(age))       # <class 'int'>
print(type(is_admin))  # <class 'bool'>
```

Strings in Python

String literals:

```
 a = 'single quotes'
b = "double quotes"
c = '''triple quotes for
          multi-line
       strings'''
```

Access characters and length:

```
 text = "Python"
print(text[0])       # 'P'
print(len(text))     # 6
```

Formatted strings (f-strings):

```
 name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old.")
```

User Input with `input()`

- The `input()` function is used to read user input from the command line. It always returns the user input as a string, even if the input looks like a number.

```
variable = input("Prompt message: ")
```

- The "Prompt message" is optional and will be shown before waiting for user input.
- Note: `input()` always returns a string:

```
age = input("How old are you? ")
print(type(age))  # Always <class 'str'>, even if you type "25"
```

- Another example:

```
name = input("Enter your name: ")
print(f"Hello, {name}!")
```

- Common error:

```
# This will cause an error if input is not a number
number = int(input("Enter a number: "))
```

- To avoid errors, always validate user input if needed:

```
user_input = input("Enter a number: ")

if user_input.isdigit():
    number = int(user_input)
```

Type Casting

```
 x = "5"
y = int(x) + 10
print(y)  # 15

a = float("3.14")
b = str(42)
c = bool("non-empty")
```

Memory Address: `id()`

```
 x = 100
print(id(x))  # Shows memory address
```

Control Flow: `if`, `elif`, `else`

Control flow allows your program to **make decisions**. In Python, you use `if`, `elif` (else if), and `else` to run different blocks of code based on conditions.

Basic Structure

```
 if condition1:
    # Run this block if condition1 is True
elif condition2:
    # Run this block if condition1 is False AND condition2 is True
else:
    # Run this block if all above conditions are False
```

- Each condition must be an **expression that evaluates to True or False**.
- Use **indentation** (typically 4 spaces) instead of `{}`.
- You can have **zero or many** **`elif`**, but only **one** **`else`** at the end.

Simple Example

```
temperature = 25

if temperature > 30:
    print("It's hot!")
elif temperature > 20:
    print("Nice weather.")
else:
    print("It's a bit chilly.")
```

Output:

```
Nice weather.
```

- Note: `elif` and `else` are here optional!

Comparison Operators

Operator	Meaning	Example
`==`	Equal to	`x == 5`
`!=`	Not equal to	`x != 5`
`>`	Greater than	`x > 5`
`<`	Less than	`x < 5`
`>=`	Greater or equal	`x >= 5`
`<=`	Less or equal	`x <= 5`

Multiple Conditions with Logic Operators

```
x = 10

if x > 5 and x < 20:
    print("x is between 5 and 20")
```

```
y = 15

if y < 10 or y > 12:
    print("y is outside 10-12 range")
```

```
logged_in = True

if not logged_in:
    print("Please log in.")
```

Nested `if` Statements

```
 age = 25
has_ticket = True

if age >= 18:
    if has_ticket:
        print("Entry allowed.")
    else:
        print("Ticket required.")
else:
    print("You must be 18 or older.")
```

Summary of control structures

- Use `if` to check a condition.
- Use `elif` for extra conditions.
- Use `else` as a fallback.
- Combine with `and`, `or`, `not` for more complex logic.
- Indentation is critical!

Python Loops: `while` and `for`

Loops allow you to **repeat blocks of code**. Python supports two main types:

- `while` loop: Repeats while a condition is `True`
- `for` loop: Iterates over items in a sequence like a list or range

`while` Loop – Repeat While Condition is True

Syntax

```
while condition:
    # code block
```

- The condition is checked **before** each iteration.
- If the condition becomes `False`, the loop stops.
- Make sure the loop has an **exit condition**, or it will run forever.

Example 1: Count to 4

```
i = 1
while i <= 4:
    print(i)
    i += 1
```

Output:

```
1
2
3
4
```

Example 2: Infinite Loop ()

```
while True:
    print("This goes on forever!")
```

Press Ctrl+C to stop it in terminal.

Using `break` in `while`

```
 x = 0
while True:
    print(x)
    x += 1
    if x > 5:
        break
```

Using `continue` in `while`

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue
    print(i)
```

Output:

```
1
2
4
5
```

`for` Loop – Iterate Over Items

Syntax

```
for item in iterable:
    # code block
```

- Loops over each item in a **sequence** (like a list, tuple, string, or range).
- You don't need an index unless you want one explicitly.

Example 1: Loop Over List

```python
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Example 2: Loop Over String

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

Example 3: Using `range()`

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

Output:

```
 0
1
2
```

```
for i in range(1, 6):
    print(i)
```

Output:

```
 1
2
3
4
5
```

Using `break` in `for`

```
for i in range(10):
    if i == 4:
        break
    print(i)
```

Using `continue` in `for`

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

Output:

```
 0
1
3
4
```

Summary

Feature	`while` Loop	`for` Loop
Best for	Unknown number of repeats	Known number or iterable
Condition	Checked before each loop	Loops over sequence or range
Common use	Waiting for user input	Iterating lists, strings, ranges

Both loop types support `break` (stop loop) and `continue` (skip current iteration).

`break` and `continue`

```
for i in range(5):
    if i == 3:
        break
    print(i)
```

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

Functions

What is a Function?

- A **function** is a reusable block of code that performs a specific task.
- Functions help you avoid repeating code and make your programs easier to read and maintain.

Why Use Functions?

- **Reusability**: Write once, use many times.
- **Organization**: Break large problems into smaller parts.
- **Readability**: Code becomes easier to understand.
- **Debugging**: Easier to test and fix issues in isolated blocks.

Defining a Function

- Use the `def` keyword followed by the function name and parentheses.
- Example:

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

Calling a Function

- You run (or "call") a function by using its name followed by parentheses:

```
greet()  # Output: Hello, world!
```

Function with Parameters

- Parameters allow you to pass information into a function:

```
def greet(name):
    print(f"Hello, {name}!")
```

```
greet("Alice")  # Output: Hello, Alice!
```

Function with Return Value

- Functions can **return** results using the `return` keyword:

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

```
result = add(2, 3)
print(result)  # Output: 5
```

Default Parameters

- You can give default values to parameters:

```
def greet(name="stranger"):
    print(f"Hello, {name}!")
```

```
greet()          # Output: Hello, stranger!
greet("Alice")  # Output: Hello, Alice!
```

Keyword Arguments

- You can name arguments when calling the function:

```
def subtract(a, b):
    return a - b

print(subtract(b=5, a=10))  # Output: 5
```

Summary

- Functions = named blocks of code.
- Use `def` to define, `()` to call.
- Can have parameters and return values.
- Help make your code DRY (Don't Repeat Yourself).

Collections Overview

Python provides several built-in collection types to store groups of data.

Type	Description	Example
`list`	Ordered, changeable, allows duplicates	`[1, 2, 3]`
`tuple`	Ordered, unchangeable, allows duplicates	`(1, 2, 3)`
`set`	Unordered, no duplicates	`{1, 2, 3}`
`dict`	Key-value pairs, fast lookup	`{"a": 1, "b": 2}`

Lists

- A list is like a dynamic array.
- You can add, remove, and change elements.

```python
 fruits = ["apple", "banana", "cherry"]
fruits.append("orange")      # Add new item
print(fruits[0])             # Access first item
fruits[1] = "blueberry"      # Modify an item
print(fruits)                # ['apple', 'blueberry', 'cherry', 'orange']
```

Tuples

- Tuples are similar to lists, but **immutable** (cannot be changed).
- Useful for fixed data (like coordinates).

```
 point = (10, 20)
x, y = point
print(x, y)                    # 10 20

# point[0] = 100  # ❌ Error: Tuples can't be changed
```

Sets

- Sets are **unordered collections** with **no duplicate items**.
- Great for membership checks and uniqueness.

```
 unique_numbers = {1, 2, 3, 2, 1}
print(unique_numbers)        # {1, 2, 3}
unique_numbers.add(4)
print(3 in unique_numbers)   # True
```

Dictionaries

- Dictionaries store data as **key-value pairs**.
- Fast lookups by key.

```
 person = {"name": "Alice", "age": 30}
print(person["name"])        # Alice
person["age"] = 31           # Update value
person["job"] = "Engineer"   # Add new key-value pair
print(person)
```

Summary Table

Feature	**List**	**Tuple**	**Set**	**Dictionary**
Ordered	✅	✅	❌	✅ (Python 3.7+)
Mutable	✅	❌	✅	✅
Allows Duplicates	✅	✅	❌	Keys: ❌, Values: ✅
Indexed	✅	✅	❌	✅
Use case	General data	Fixed groups	Unique items	Key-value mapping