

Artificial Intelligence

Eng.Eman Elgallad

Lecture 2- Python Basics For AI



1. Introduction to Python
2. Lists and Tuples
3. Strings
4. Dictionaries
5. Conditional and Looping Statements
6. Functions

Introduction



What Is Python?

Python is one of the achievements of free software.

Founder	Guido van Rossum
When and Where	Created in Amsterdam during Christmas in 1989.
Meaning of Name	A big fan of Monty Python's Flying Circus

IT is a general-purpose and advanced programming language.

Python applies to programming in many fields:

- ☐ Data science
- ☐ Writing system tools
- ☐ Developing applications with graphical UIs
- ☐ Writing network-based software
- ☐ Interacting with databases



Python VS Other Languages

Python VS C:

- ☐ Python is dynamic while C is static.
- ☐ Memory is managed by the developer in C but by the interpreter in Python.
- ☐ Python has many libraries, but C has no standard library for a hybrid tuple (Python list) and a hash table (Python dictionary).
- ☐ Python cannot be used to write a kernel, but C can.
- ☐ C or C++ extends Python functions based on the Python APIs.

Python VS Other Languages

Python VS Java:

- ☐ Python is dynamic while Java is static.
- ☐ Python supports object-oriented and function-based programming while Java supports object-oriented programming only.
- ☐ Python is simpler than Java, and typically applies to quick prototyping.
- ☐ Python and Java enable multiple programmers to develop a large project together step by step.

Advantages of Python

- **Simple:** Python is an ideal programming language of simplicity.
- **Easy-to-learn:** few key words, simple structure.
- **Open-source:** Python is one of Free/Libre and Open-Source Software (FLOSS).
- **Interpretability:** Python program does not need to be compiled into binary code and can be directly run from source code.
- **Scalability:** If you want to speed up running of some key code or keep some algorithms not open, you can write some programs using C or C++ and use them in Python.
- **Embedded:** Python can be embedded in a C or C++ program to provide scripting to program users.
- **Object-oriented:** Python supports both process-oriented programming and Functional programming. In a process-oriented language, a program is built from a process or a function that is merely reusable code. In an object-oriented language, programs are built from the combination of data and functionality.
- **Rich libraries:** The Python standard library is huge. It can help you with all kinds of work, including regular expressions, document generation, unit tests, threads, databases, Web browsers, CGI, Numpy, Pandas, HTML, WAV files, password systems, GUI, TK, and other system-related operations.

Features of Python Statements

- **Dynamic:** Objects (like attributes and methods) can be changed during execution.

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

    def speak(self):
        return f"{self.name} says woof!"

dog1 = Dog("Buddy")
print(dog1.name) # Output: "Buddy"

# Adding a new attribute dynamically
dog1.age = 3
print(dog1.age) # Output: 3
```

Features of Python Statements Cont.

- Python uses **indentation** instead of a pair of curly braces {} to divide a block of statements.

```
def print_numbers():  
    for i in range(1, 6):  
        # This block is indented  
        print(i)  
        print(i * i)  
    # The block ends here, no closing curly brace needed
```

- Multiple statements on one line are separated by “;”.

But it is not recommended and goes against the Pythonic style guidelines. Python emphasizes readability and clarity, so it's better to have one statement per line.

```
x = 10; y = 20; z = x + y  
print(z) # Output: 30
```

Features of Python Statements Cont.

- the **hash (#)** symbol is used to comment out a single line, while **triple quotes** ("""...""") are used for docstrings, which can also be used to comment out multiple lines.

```
# This is a comment
print("Hello, World!")

"""
This is a docstring.
It can span multiple lines.
"""

print("This won't be executed.")
```

- Variables **do not** need type definitions, variables are dynamically typed, meaning you don't need to explicitly declare their data types. It is determined during runtime based on the value assigned

```
x = 10
print(type(x)) # Output: <class 'int'>

x = "Hello, World!"
print(type(x)) # Output: <class 'str'>
```

Features of Python Statements Cont.

- the Functional Programming (FP) is available.

```
# Functional approach using map function
def square(x):
    return x * x

numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(square, numbers))
print(squared_numbers) # Output: [1, 4, 9, 16, 25]
```

Functional programming allows you to write more expressive code, and Python's support for it gives you the flexibility to choose between different programming styles based on the problem at hand.

Lists and Tuples

Lists:

- A list is an ordered set of elements that you can add and delete at any time.
- The element types in the list can be different. You can access each element in the list by index. The first digit of the index starts from 0, and the reverse index starts from -1.

Common Operations on Lists

Create and Access List:

Creating a list

```
fruits = ["apple", "banana", "orange", "grape"]
```

Accessing elements by index

```
print(fruits[0])
```

Output: "apple"

```
print(fruits[-1])
```

Output: "grape"

Methods of Python Listing

list.append(obj)

list.count(obj)

list.extend(seq)

list.index(obj)

list.insert(index, obj)

list.pop(obj=list[-1])

list.remove(obj)

list.sort([func])

list.reverse()

Append an element to the end of the list

```
fruits.append("pear")
```

```
print(fruits)
```

Output: ["apple", "banana", "orange", "grape", "pear"]

Insert an element at a specific index

```
fruits.insert(1, "kiwi")
```

```
print(fruits)
```

Output: ["apple", "kiwi", "banana", "orange", "grape", "pear"]

Remove an element by index

```
del fruits[0]
```

```
print(fruits)
```

Output: ["kiwi", "banana", "grape", "pear"]

List method examples

```
fruits = ["apple", "banana", "orange", "apple"]
```

```
# Count occurrences of an element in the list
```

```
print(fruits.count("apple"))
```

```
# Output: 2
```

```
# Extend the list by appending elements from another list
```

```
fruits_to_add = ["kiwi", "grape"]
```

```
fruits.extend(fruits_to_add)
```

```
print(fruits)
```

```
# Output: ["apple", "banana", "orange", "apple", "pear", "kiwi", "grape"]
```

```
# Find the index of the first occurrence of an element in the list
```

```
print(fruits.index("banana"))
```

```
# Output: 1
```

List method examples Cont.

Remove the last element from the list and return it

```
popped_fruit = fruits.pop()
```

```
print(popped_fruit)
```

Output: "grape"

```
print(fruits)
```

Output: ["apple", "banana", "mango", "orange", "apple", "pear", "kiwi"]

Remove the first occurrence of an element from the list

```
fruits.remove("apple")
```

```
print(fruits)
```

Output: ["banana", "mango", "orange", "apple", "pear", "kiwi"]

List method examples Cont.

Sort the list in ascending order

```
fruits.sort()
```

```
print(fruits)
```

Output:

```
["apple", "banana", "kiwi", "mango", "orange", "pear"]
```

Reverse the elements in the list

```
fruits.reverse()
```

```
print(fruits)
```

Output:

```
["pear", "orange", "mango", "kiwi", "banana", "apple"]
```

Tuple

- Tuple is expressed using ().
- Like a list, a tuple cannot be modified once initialized, and the elements need to be identified when a tuple is defined.
- A tuple does not have the append () or insert () method, nor can it be assigned to another element. It has the same fetching methods as a list.
- Because a tuple is unchangeable, the code is more secure. Therefore, if possible, use a tuple instead of a list.
- A tuple is simple to create. You only need to add elements to
- parentheses and separate them with commas.

Common Tuple Operations

1. Accessing Elements in a Tuple:

Accessing elements by index

```
fruits_tuple = ("apple", "banana", "orange", "grape")
```

```
print(fruits_tuple[0])
```

Output: "apple"

```
print(fruits_tuple[-1])
```

Output: "grape"

Common Tuple Operations

2. Modifying Tuples (Tuple Calculation):

Tuples are immutable in Python, meaning their elements cannot be changed once they are created. So, you cannot directly modify a tuple. However, you can perform tuple operations that create new tuples based on existing tuples.

Tuple Concatenation

```
tuple1 = (1, 2, 3)
```

```
tuple2 = (4, 5, 6)
```

```
concatenated_tuple = tuple1 + tuple2
```

```
print(concatenated_tuple)
```

```
# Output: (1, 2, 3, 4, 5, 6)
```

Tuple Repetition

```
repeated_tuple = tuple1 * 3
```

```
print(repeated_tuple)
```

```
# Output: (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Common Tuple Operations

3. Deleting a Tuple (using del):

Since tuples are immutable, **you cannot delete elements within a tuple**. However, you can delete the entire tuple using the `del` keyword.

Creating a tuple

```
fruits_tuple = ("apple", "banana", "orange", "grape")
```

Deleting the entire tuple

```
del fruits_tuple
```

After this point, the tuple 'fruits_tuple' is no longer accessible or exists in memory

4. Tuple Operators (+ and *):

As shown in the previous example, you can use the `+` operator to concatenate two tuples, and the `\*` operator to repeat a tuple.

Common Tuple Operations

5. Tuple Indexing and Slicing (Interception):

Tuples support indexing and slicing, just like lists.

```
numbers_tuple = (1, 2, 3, 4, 5)
```

```
# Indexing
```

```
print(numbers_tuple[2])
```

```
# Output: 3
```

```
# Slicing
```

```
print(numbers_tuple[1:4])
```

```
# Output: (2, 3, 4)
```

Embedded Functions of tuple

-len(tuple) -max(tuple)
-min(tuple) -tuple(seq)

1. `len(tuple)`: Returns the number of elements in the tuple.

```
numbers_tuple = (1, 2, 3, 4, 5)  
print(len(numbers_tuple))
```

Output: 5

2. `max(tuple)`: Returns the largest element in the tuple.

```
numbers_tuple = (1, 2, 3, 4, 5)  
print(max(numbers_tuple))
```

Output: 5

3. `min(tuple)`: Returns the smallest element in the tuple.

```
numbers_tuple = (1, 2, 3, 4, 5)  
print(min(numbers_tuple))
```

Output: 1

4. `tuple(seq)`: Converts a sequence (e.g., a list or string) into a tuple.

```
my_list = [1, 2, 3, 4, 5]  
converted_tuple = tuple(my_list)  
print(converted_tuple)
```

Output: (1, 2, 3, 4, 5)

Strings

Strings

In Python, a string is a sequence of 0 or more characters, and a string is one of several sequences built in Python. Strings are unchangeable and are like string constants in the C and C++ languages.

1. Creating Strings:

Single quotes

```
single_quote_str = 'Hello, I am a string.'
```

Double quotes

```
double_quote_str = "I am also a string."
```

Triple quotes for multiline strings

```
multiline_str = '''This is a multiline  
string in Python.'''
```

2. String Indexing and Slicing:

```
my_string = "Hello, World!"
```

```
# Accessing characters by index
```

```
print(my_string[0])
```

```
# Output: "H"
```

```
print(my_string[-1])
```

```
# Output: "!"
```

```
# Slicing to get substrings
```

```
print(my_string[7:12])
```

```
# Output: "World"
```

3. String Concatenation:

```
first_name = "John"
```

```
last_name = "Doe"
```

```
full_name = first_name + " " +
```

```
last_name
```

```
print(full_name)
```

```
# Output: "John Doe"
```

4. String Length:

```
text = "Hello, Python!"
```

```
print(len(text))
```

```
# Output: 14
```

5- Strings Methods

Strings come with various built-in methods for various operations. Some common methods include:

```
my_string = "Hello, Python!"
```

```
# Convert to lowercase
```

```
print(my_string.lower())
```

```
# Output: "hello, python!"
```

```
# Convert to uppercase
```

```
print(my_string.upper())
```

```
# Output: "HELLO, PYTHON!"
```

```
# Replace characters
```

```
print(my_string.replace("Python", "World"))
```

```
# Output: "Hello, World!"
```

```
# Split the string into a list of words
```

```
print(my_string.split())
```

```
# Output: ['Hello,', 'Python!']
```

6. String Formatting:

```
name = "Alice"
```

```
age = 30
```

```
message = "Hello, my name is {} and I am {} years old.".format(name, age)
```

```
print(message)
```

```
# Output: "Hello, my name is Alice and I am 30 years old."
```

7. Escape Sequences:

```
# Using escape sequences
```

```
print("This is a new line.\nThis is a tab:\t\tTab is used for indentation.")
```

Dictionaries



Dictionaries:

- A dictionary is a collection of key-value pairs.
- Each key-value pair in the dictionary is separated by a colon (":").
- Key-value pairs are separated by a comma (",").
- The entire dictionary is enclosed in curly braces "{}".
- Keys in a dictionary are generally unique, and their type is unchangeable (immutable).
- If a key repeats, the last key-value pair replaces the previous one.

Dictionaries do not allow duplicate keys.

- The values in a dictionary can be of any data type (e.g., int, float, string, list, tuple, another dictionary, etc.).
- Dictionaries are mutable, meaning you can add, modify, or delete elements after creation.

A dictionary is represented in the following format:

```
d = {key1: value1,  
      key2: value2,  
      key3: value3}
```

Dictionaries:

Here's code example of creating and accessing elements from a dictionary:

```
person = {  
    "name": "John Doe",  
    "age": 30,  
    "occupation": "Engineer",  
    "is_employed": True}  
# Accessing elements using keys  
print(person["name"])  
# Output: "John Doe"  
print(person["age"])  
# Output: 30  
print(person["is_employed"])  
# Output: True
```

Dictionaries provide an efficient way to store and retrieve data using keys, making them an essential data structure in Python for many programming tasks.

Conditional and Looping Statements

If Statement

In Python, there are three main control structures for flow control: `if`, `for`, and `while`. However, unlike some other programming languages like C, Python does not have a built-in `switch` statement.

Control Structures in Python:

1. `if` Statement:

The `if` statement is used for conditional execution. It allows you to control the flow of your program based on specific conditions. The basic form of an `if` statement in Python is as follows:

```
if Judging condition 1:
```

```
    Statement 1...
```

```
elif Judging condition 2:
```

```
    Statement 2...
```

```
elif Judging condition 3:
```

```
    Statement 3...
```

```
else:
```

```
    Statement 4...
```

Code Example:

```
x = 10
```

```
if x > 5:
```

```
    print("x is greater than 5")
```

```
else:
```

```
    print("x is not greater than 5")
```

For Loop:

2. `for` Loop:

The `for` loop is used to iterate over a sequence (such as lists, tuples, or strings) or other iterable objects.

It allows you to perform a set of actions repeatedly for each item in the sequence.

The basic form of a `for` loop in Python is as follows:

for item in sequence:

Code block to execute for each item in the sequence

Code Example:

```
fruits = ["apple", "banana", "orange"]
```

```
for fruit in fruits:
```

```
    print(fruit)
```

While Loop:

3. `while` Loop:

The `while` loop is used for repeated execution of a code block **as long as a given condition is True**. It is useful when the number of iterations is not known in advance. The basic form of a `while` loop in Python is as follows:

When the condition of a while statement is never a Boolean false, the loop will never end, forming an infinite loop, **also known as a dead loop**. You can use the break statement in a loop to force a dead loop to end.

while condition:

Code block to execute as long as the condition is True

Code Example:

```
count = 1
```

```
while count <= 5:
```

```
    print("Current count:", count)
```

```
    count += 1
```

Nesting Loop :

Python allows one loop to be nested inside another loop. This means you can have loops inside other loops, creating nested structures for more complex iterations.

Syntax of for loop nesting:

```
for iterating_var_outer in sequence_outer:  
    for iterating_var_inner in sequence_inner:  
        # Statements for inner loop  
    # Statements for outer loop
```

Syntax of while loop nesting:

```
while expression_outer:  
    while expression_inner:  
        # Statements for inner loop  
    # Statements for outer loop
```

1. `break` statement:

- The `break` statement is used to terminate the entire loop prematurely.
- When a `break` statement is within the loop, the loop execution is immediately stopped, and the control moves to the next line of code after the loop, skipping any remaining iterations.
- If the loop is nested (i.e., one loop inside another), the `break` statement will only terminate the innermost loop where it is encountered.

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

Output:

```
0  
1  
2
```

2-continue statement:

- The continue statement is used to end the current iteration of the loop prematurely and start the next iteration.
- When a continue statement is encountered within the loop, the current iteration's remaining statements are skipped, and the loop proceeds to the next iteration.
- If the loop is nested, the continue statement will only affect the innermost loop where it is encountered.
- Both the break and continue statements can be used with both for and while loops.

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

Output:

```
0  
1  
2  
4
```

Functions

- A function is a code segment that is organized, reusable, and used to implement a single function or associated functions.
- Functions can improve the modularity of applications and reuse of code.
- Python provides a number of built-in functions, such as `print()`. You can also create your own functions, which are called **user-defined functions**.

Defining a Function

- The function code block begins with a **def** keyword, followed by the **function name and parentheses ()**.
- Any incoming arguments and independent variables **must be placed in the middle of the parentheses**. Parentheses can be used to define arguments.
- The first line of the function can selectively use the document string to hold the description of the function.
- The function content starts with a colon and indents.
- `return[expression]` ends a function, and selectively returns a value to the caller. Returning without an expression is equivalent to returning `none`.

Example of a simple function:

```
def add_numbers(a, b):  
    """This function adds two numbers."""  
    result = a + b  
    return result
```

In this example, we define a function named **add_numbers** that takes two parameters a and b. It adds the two numbers and returns the result.

Calling Function:

After the basic structure of this function is complete, you can execute it through another function call, or you can execute it directly from the Python prompt.

```
# Define a function
def test(str):
    "print any incoming string"
    return str;

# Call a function
test("I want to call a user-defined function!");
test("call the same function again");
```

Variables Scope

1. A variable defined within a function has a local scope and is called a **local variable**. It means that the variable is only accessible within the specific function where it is defined. Local variables are created when the function is called and destroyed when the function exits. Attempting to access a local variable outside of its function will result in an error.
2. A variable defined outside of any function has a global scope and is called a **global variable**. Global variables are accessible throughout the entire program, including all functions. They are created when the program starts and exist until the program ends.

Variables Scope Example

```
global_var = 10

def example_function():
    local_var = 5
    print("Local variable:", local_var)
    print("Global variable:", global_var)

example_function()
print("Global variable (outside function):", global_var)
```

In this example, `global\_var` is a global variable, and `local\_var` is a local variable defined within the `example\_function()`.

The `example\_function()` can access the global variable `global\_var`, but it cannot access `local\_var` outside of the function.

3. When a function is called, all variable names stated within the function are added to the local scope of that function. This means that variables with the same name as global variables will create a new local variable with the same name within the function, and **any modifications made to the local variable will not affect the global variable.**

Variables Scope Cont.

4. If you want to modify a global variable from within a function, you need to use the `'global'` keyword to declare the variable as global. This way, Python will know that you are referring to the global variable and not creating a new local variable with the same name.

```
global_var = 10

def modify_global_variable():
    global global_var # Declare global_var as global within the function
    global_var = 20   # Modifying the global variable

modify_global_variable()
print("Global variable (modified):", global_var)
```

```
Global variable (modified): 20
```



THANK
YOU!