Python Notes for Beginners

By Mohit Soni

Strings

Think of a string as a sequence of characters. It's like a sentence or a word. In Python, you create a string by enclosing characters in quotes. For example, `"Hello, World!"` is a string.

Operations on Strings:
1. Concatenation: Joining two strings together. Like `"Hello" + "World"` becomes `"HelloWorld"`.
2. Indexing: Accessing a specific character. For example, in `"Hello"`, `H` is at index 0, `e` is at index 1.
3. Length: Finding out how many characters are in the string. For `"Hello"`, the length is 5.

Operations on Strings:	Example 1: Strings

1. Concatenation: Joining two strings together. Like `"Hello" + "World"` becomes `"HelloWorld"`.
 	
2. Indexing: Accessing a specific character. For example, in `"Hello"`, `H` is at index 0, `e` is at index 1.
 	
3. Length: Finding out how many characters are in the string. For `"Hello"`, the length is 5.
	# Creating a string
	greeting = "Hello, World!"
	print(full_greeting)
	
	# Concatenating strings
	full_greeting = greeting + " Have a nice day!"
	print("First character:", first_char)
	
	# Indexing
	first_char = greeting[0] # 'H'
	print("First character:", first_char)
	
	# Length of the string
	length = len(greeting) # 13
	print("Length of greeting:", length)

Lists

A list is like a collection of items. Imagine a shopping list where you write down what you need to buy. In Python, a list is created by putting items inside square brackets, like `[1, 2, 3]` or `["apple", "banana", "cherry"]`.

Operations on Lists:
1. Adding Items: You can add an item to the list with `.append(item)`.
2. Removing Items: You can remove an item with `.remove(item)` or `.pop(index)`.
3. Length: Like strings, you can find out the number of items in a list with `len(list)`.
4. Indexing and Slicing: You can access and slice lists just like strings.

	Example 2: Lists

	# Creating a list
	fruits = ["apple", "banana", "cherry"]
	print("Fruits List:", fruits)

1. Adding Items: You can add an item to the list with `.append(item)`.	# Adding an item to the list
	fruits.append("orange")
	print("Second Fruit:", second_fruit)

2. Removing Items: You can remove an item with `.remove(item)` or `.pop(index)`.	# Removing an item from the list
	fruits.remove("banana")

3. Length: Like strings, you can find out the number of items in a list with `len(list)`.	# Length of the list
	length_of_list = len(fruits) # 3

4. Indexing : You can access lists just like strings.	# Indexing
	second_fruit = fruits[1] # 'cherry'

Tuples

A tuple is similar to a list, but it's fixed. Once you create a tuple, you can't change it. It's like a list that's set in stone. You create a tuple using parentheses, like `(1, 2, 3)` or `("a", "b", "c")`.

Operations on Tuples:
1. Indexing and Slicing: You can access and slice tuples like you do with lists and strings.
2. Length: You can find the number of items in a tuple with `len(tuple)`.
3. Immutability: You can't add or remove items from a tuple once it's created.

	Example 3: Tuples

1. Indexing: You can access tuples like you do with lists and strings.	# Creating a tuple
	numbers = (1, 2, 3, 4, 5)
	print("Numbers Tuple:", numbers)
	
	# Creating a tuple
	my_tuple = (10, 20, 30, 40, 50)
	
	# Accessing elements using indexing
	first_element = my_tuple[0] # Accessing the first element (10)
	second_element = my_tuple[1] # Accessing the second element (20)
	last_element = my_tuple[-1] # Accessing the last element (50)
	
	# Print the results
	print("First Element:", first_element)
	print("Second Element:", second_element)
	print("Last Element:", last_element)

2. Length: You can find the number of items in a tuple with `len(tuple)`.	# Length of the tuple
	length_of_tuple = len(numbers) # 5
	print("Length of Tuple:", length_of_tuple)

3. Immutability: You can't add or remove items from a tuple once it's created.	

Summary

- Strings are sequences of characters.
- Lists are collections of items that can be changed.
- Tuples are like lists but can't be changed after they're created.

String Manipulation In Python

Input function ()

1. Creating and Concatenating Strings □
Strings are simply text in Python. You create a string by enclosing text in quotes. You can join two strings together using the `+` operator. This is known as concatenation.

```
#Creating strings
greeting = "Hello"
name = "Alice"

#Concatenating strings
full_greeting = greeting + ", " + name + "!"
print(full_greeting)   Output: Hello, Alice!
```

---

2. Accessing Characters in a String ☐
You can access individual characters in a string using their index. In Python, indices start at 0.

```
#A string
word = "Python"

#Accessing characters
```

```
first_letter = word[0]   First character
third_letter = word[2]   Third character

print(first_letter)   Output: P
print(third_letter)   Output: t
```

---

3. Replacing Part of a String ☐

You can replace part of a string with another string using the `replace` method.

```
#A simple string
sentence = "I love programming in Java."

 Replacing 'Java' with 'Python'
new_sentence = sentence.replace("Java", "Python")
print(new_sentence)   Output: I love programming in Python.
```

---

4. Finding the Length of a String

You can find out how many characters are in a string using the `len()` function.

```
#A string
word = "Hello"

#Find the length of the string
length = len(word)
print(length)   Output: 5
```

---

5. Converting Numbers to Strings

To concatenate a number with a string, you first need to convert the number to a string using the `str()` function.

```
#A number
age = 25

#Convert number to string and concatenate
```

```
message = "I am " + str(age) + " years old."
print(message)   Output: I am 25 years old.
```

Replace Method

➔ The `replace` method in Python is a string method used to replace a specified phrase with another specified phrase. Here's a brief overview of how it works:

Syntax of the `replace` method. The basic syntax of the `replace` method is:

string.replace(old, new, count)

- **`old`**: This is the string you want to search for.
- **`new`**: This is the string that will replace the `old` string.
- **`count`** (optional): This specifies the number of times you want to replace the `old` string.

☐**How it Works**

1. Searching: The method looks through the entire string for the `old` string.
2. Replacing: Every time it finds the `old` string, it replaces it with the `new` string.
3. Resulting String: A new string is returned with the replacements. The original string remains unchanged.

<u>Examples</u>

Let's go through some examples to understand this better.

```
# Basic Replacement

text = "I like apples."
new_text = text.replace("apples", "oranges")
print(new_text)  # Output: I like oranges.
```

➔ In this example, every occurrence of `**apples**` in the string `**text**` is replaced with `**oranges**`.

- **Important Notes**

- Case Sensitivity: The `replace` method is case-sensitive. `"Apples"` and `"apples"` are considered different.

- Immutability of Strings: Strings in Python are immutable, meaning the original string is not modified. Instead, a new string is returned.

The `replace` method is widely used for modifying strings, especially when you need to change specific parts of a string or clean up data.

Practice Question 1: Simple Concatenation

Task: Create two variables, `first_name` and `last_name`. Assign your first name to `first_name` and your last name to `last_name`. Then, concatenate these two variables to form a full name and print it. Add a space between the first name and last name.

Example Code:

```
first_name = "Akash"
last_name = "Amit"
full_name = first_name + " " + last_name
print(full_name)
```

Practice Question 2: Basic Replacement

Task: You have a string variable `sentence` that contains the text "I love coding in C++". Replace "C++" with "Python" in this string and print the new string.

Example Code:

```
sentence = "I love coding in C++"
new_sentence = sentence.replace("C++", "Python")
print(new_sentence)
```

Practice Question 3: Character Access and Counting

Task: Given the string `word = "Programming"`, perform the following:

1. Print the first character of the string.
2. Print the last character of the string.

3. Print the length of the string.

Example Code:

```
word = "Programming"
first_char = word[0]
last_char = word[-1]
length_of_word = len(word)

print("First Character:", first_char)
print("Last Character:", last_char)
print("Length of the Word:", length_of_word)
```

Dictionary in Python

- **What is a Dictionary in Python?**

➔ Imagine you have a real-world address book where you keep your friends' names and their phone numbers. In Python, a dictionary works quite similarly. It stores data in pairs: a **'key'** and a **'value'**. The **'key'** is like your friend's name, and the 'value' is like their phone number.

☐Operations You Can Do with a Dictionary

1. Create: Make a new dictionary.
2. Read: Look up a value using its key.
3. Update: Change the value associated with a key.
4. Delete: Remove a key-value pair.

Real-World Indian Example
Suppose you're keeping track of different spices and their prices in rupees.

```
spices = {
   "Turmeric": 80,
   "Cumin": 60,
   "Cardamom": 100
}
```

Here, "Turmeric", "Cumin", and "Cardamom" are keys, and 80, 60, and 100 are their respective values.

Practice Python Code
Let's write some basic code for operations. Remember, we won't use loops or if-else statements.

1. Create a Dictionary

```
fruits = {"Mango": 50, "Banana": 20, "Apple": 80}
```

2. Read a Value

```
print(fruits["Mango"])  # This will print the price of Mango
```

3. Update a Value

```
fruits["Banana"] = 25  # Changing the price of Banan
```

4. Delete a Key-Value Pair

```
del fruits["Apple"]  # Removing Apple from the dictionary
```

Basic Dictionary Questions for Practice

1. Create a dictionary named `vehicles` where the keys are different types of vehicles like "Car", "Bike", "Auto", and the values are their colors.

2. Update the color of "Bike" in the `vehicles` dictionary to a color of your choice.

3. Delete the "Auto" from the `vehicles` dictionary.

Input Functions

The **input()** function in Python is used to take input from the user. When this function is called, the program stops and waits for the user to type something and press Enter. The function then returns the user's input as a string.

Data Type: The **input()** function always returns the user input as a string, even if the user enters numbers.
Prompt Argument: You can pass a string to **input()** to use as a prompt message.
Type Conversion: If you need the input to be a different type (e.g., an integer or a float), you can convert it using functions like **int()**, **float()**, etc.
Here's an example with type conversion:

```
age = input("Enter your age: ")
age = int(age)
print("You will be " + str(age + 1) + " next year.")
```

Or

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

```
age =int(input("Enter your Age= "))
print("You will be " + str(age+1), " next year")
```

Practice Questions

Question 1: **Personal Greeting**

Write a Python program that asks the user for their name using the `input` function. Store the name in a variable, and then print a greeting message that includes the name. For example, if the user enters "Mohit", the program should print "Hello, Mohit!".

Question 2: **Age in Years**

Create a Python program that prompts the user to enter their age in years using the `input` function. Convert this age from a string to an integer and store it in a variable. Then, calculate and print the age in months (assume 12 months in a year). For example, if the user enters "20", the program should output "You are 240 months old."

Question 3: **Simple Calculator**

Develop a simple calculator using Python. The program should:
- Ask the user to enter two numbers (use `input` function).
- Store these numbers in two separate variables.
- Convert the input strings into integers or floats.
- Calculate and print the sum of these two numbers.
- For example, if the user enters 5 and 3, the program should print "The sum of 5 and 3 is 8."

Operators in Python

Operators in Python are special symbols that perform operations on variables and values. Python supports various types of operators, which can be classified as follows:

1. Arithmetic Operators

These operators are used to perform mathematical operations.

Addition (+)

```
a = 10
b = 5
print(a + b)  # Output: 15
```

-

Subtraction (-)

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

-

Multiplication (*)

```
a = 10
b = 5
print(a * b)  # Output: 50
```

-

Division (/)

```
a = 10
b = 5
print(a / b)  # Output: 2.0
```

-

Modulus (%)

```
a = 10
b = 3
print(a % b)  # Output: 1
```

-

Exponentiation (**)

```
a = 2
b = 3
print(a ** b)  # Output: 8
```

-

Floor Division (//)

```
a = 10
b = 3
print(a // b)  # Output: 3
```

---

2. Comparison Operators

These operators are used to compare two values.

Equal (==)

```
a = 10
b = 10
print(a == b)  # Output: True
```

-

Not Equal (!=)

```
a = 10
b = 5
print(a != b)  # Output: True
```

-

Greater Than (>)

```
a = 10
b = 5
print(a > b)  # Output: True
```

-

Less Than (<)

```
a = 10
b = 5
print(a < b)  # Output: False
```

-

Greater Than or Equal To (>=)

```
a = 10
b = 10
print(a >= b)  # Output: True
```

-

Less Than or Equal To (<=)

```
a = 10
b = 5
print(a <= b)  # Output: False
```

---

3. Assignment Operators

These operators are used to assign values to variables.

Assign (=)

```
a = 10

print(a)  # Output: 10
```

-

Add and Assign (+=)

```
a = 10
a += 5
print(a)  # Output: 15
```

-

Subtract and Assign (-=)

```
a = 10
a -= 5
print(a)  # Output: 5
```

-

Multiply and Assign (*=)

```
a = 10
a *= 5
print(a)  # Output: 50
```

-

Divide and Assign (/=)

```
a = 10
a /= 5
print(a)  # Output: 2.0
```

-

Modulus and Assign (%=)

```
a = 10
a %= 3
print(a)  # Output: 1
```

-

Exponentiation and Assign (=)**

```
a = 2
a **= 3
print(a)  # Output: 8
```

-

Floor Division and Assign (//=)

```
a = 10
a //= 3
print(a)  # Output: 3
```

---

4. Logical Operators

These operators are used to combine conditional statements.

and

```
a = True
b = False
print(a and b)  # Output: False, All functions should be correct or true
```

-

or

```
a = True
b = False
print(a or b)  # Output: True, Only one function should  be correct or true
```

-

not

```
a = True
print(not a)  # Output: False, it is used to get a opposite function,
If the function is" correct" or "true" it gives the opposite output "wrong" or "false" or vice versa.
```

---

5. Bitwise Operators

These operators are used to perform bit-level operations.

AND (&)

```
a = 5  # 0101
b = 3  # 0011
print(a & b)  # Output: 1 (0001)
```

-

OR (|)

```
a = 5  # 0101

b = 3  # 0011
print(a | b)  # Output: 7 (0111)
```

-

XOR (^)

```
a = 5  # 0101

b = 3  # 0011
print(a ^ b)  # Output: 6 (0110)
```

-

NOT (~)

```
a = 5  # 0101
print(~a)  # Output: -6 (inverts all bits)
```

-

Left Shift (<<)

```
a = 5  # 0101
print(a << 1)  # Output: 10 (1010)
```

-

Right Shift (>>)

```
a = 5  # 0101
print(a >> 1)  # Output: 2 (0010)
```

---

6. Membership Operators

These operators are used to test if a sequence is presented in an object.

in

```
a = [1, 2, 3, 4, 5]
print(3 in a)  # Output: True
```

-

not in

```
a = [1, 2, 3, 4, 5]
print(6 not in a)  # Output: True
```

---

7. Identity Operators

These operators are used to compare the memory locations of two objects.

is

```
a = 10
b = 10
print(a is b)  # Output: True
```

-

is not

```
a = 10
b = 5
print(a is not b)  # Output: True
```

-

These operators provide the basic tools to manipulate and interact with data in Python.

Conditional Statements

Conditional statements in Python allow you to execute specific blocks of code based on whether certain conditions are met. These conditions are usually expressions that evaluate to either True or False. The primary conditional statements in Python are if, elif, and else.

1. If Statement

The if statement evaluates a condition and executes the block of code within it if the condition is true.

Syntax:

```
if condition:

    # code to execute if condition is true
```

Example:

```
x = 10

if x > 5:
    print("x is greater than 5")
```

---

2. elif Statement

The elif statement stands for "else if" and follows an if statement. It allows you to check multiple conditions. If the preceding if condition is false, the elif condition is evaluated.

Syntax:

```
if condition1:
    # code to execute if condition1 is true
elif condition2:
    # code to execute if condition2 is true
```

Example:

```
x = 10
if x > 15:
    print("x is greater than 15")
elif x > 5:
    print("x is greater than 5 but not greater than 15")
```

---

3. else Statement

The else statement is used to execute a block of code when none of the preceding conditions are true.

Syntax:

```
if condition1:

   # code to execute if condition1 is true
elif condition2:
   # code to execute if condition2 is true
else:
   # code to execute if none of the above conditions are true
```

Example:

```
x = 3
if x > 15:
   print("x is greater than 15")
elif x > 5:
   print("x is greater than 5 but not greater than 15")
else:
   print("x is 5 or less")
```

---

4. Nested if Statements

You can nest if statements within other if statements to check multiple conditions in a more complex way.

Syntax:

```
if condition1:
   if condition2:
      # code to execute if both condition1 and condition2 are true
```

Example:

```
x = 10
y = 20
if x > 5:
   if y > 15:
      print("x is greater than 5 and y is greater than 15")
```

---

5. Ternary Conditional Operator

Python also supports a shorter version of the if statement known as the ternary conditional operator.

Syntax:

```
value_if_true if condition else value_if_false
```

Example:

```
x = 10
result = "Greater than 5" if x > 5 else "5 or less"
print(result)
```

These conditional statements allow you to control the flow of your program based on different conditions.

Loops in Python

In Python, loops are used to execute a block of code repeatedly as long as a certain condition is met. There are two main types of loops in Python:

1. **for loops**: Used for iterating over a sequence (such as a list, tuple, dictionary, set, or string) or other iterable objects.
2. **while loops**: Repeatedly execute a block of code as long as a given condition is true.

for loop

The for loop in Python is used for iterating over a sequence. Here's a basic example:

python
Copy code

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

This will output:

Copy code

```
apple
banana
cherry
```

while loop

The while loop in Python is used to repeat a block of code as long as the specified condition is true. Here's a basic example:

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

This will output:

```
0
1
2
3
4
```

Uses of Loops in Python

1. **Repetition**: Performing an action repeatedly without writing the code multiple times. For example, printing numbers from 1 to 10.
2. **Iteration**: Iterating over elements of a collection (like a list, tuple, or dictionary).
3. **Automation**: Automating repetitive tasks, such as processing items in a collection or reading lines from a file.
4. **Conditional Execution**: Performing actions based on conditions, such as retrying a task until a certain condition is met.
5. **Data Processing**: Manipulating and processing data in collections, such as filtering, transforming, or aggregating data.

Example of for Loop with a Range

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

This will output:

```
0
1
2
3
4
```

Example of while Loop with a Condition

```
n = 5
while n > 0:
  print(n)
  n -= 1
```

This will output:

```
5
4
3
2
```

Additional Control Statements : in Loops

break: Terminates the loop prematurely.

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

1. **continue**: Skips the rest of the code inside the loop for the current iteration and moves to the next iteration.

```
for i in range(10):

  if i == 5:
    continue
  print(i)
```

2. **else**: Executes a block of code once when the loop terminates naturally (i.e., not by a break statement).

```
for i in range(5):

  print(i)
else:
  print("Loop completed")
```

```
# With while loop
i = 0
while i < 5:
    print(i)
    i += 1
else:
    print("Loop completed")
```

3. Nested Loops

Loops can be nested, meaning you can have a loop inside another loop.

```
# Example of nested loops
for i in range(3):
    for j in range(3):
        print(i, j)
```

Looping through a Dictionary

You can loop through a dictionary using .items(), .keys(), or .values().

```
# Looping through dictionary items
person = {"name": "John", "age": 30, "city": "New York"}

for key, value in person.items():
    print(key, value)
```

Using loops, you can write efficient and concise code for a variety of tasks, from simple iterations to complex data processing.

Functions in Python

In Python, a function is a block of reusable code designed to perform a specific task. Functions help in organizing code, making it more readable, and reducing redundancy. They can take inputs, perform operations, and return outputs.

Defining a Function

You define a function using the def keyword, followed by the function name, parentheses, and a colon. The body of the function is indented.

```
def function_name(parameters):
    # code block
    return value
```

Example of a Simple Function

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

print(greet("Alice"))
```

Output:

```
Hello, Alice!
```

Components of a Function

1. Function Name: A descriptive name that follows Python naming conventions.
2. Parameters: Inputs to the function, specified within parentheses. A function can have zero or more parameters.
3. Docstring: An optional string that describes the function's purpose. It is the first statement in the function body and is enclosed in triple quotes.
4. Body: A block of statements that define what the function does.
5. Return Statement: Specifies the value that the function returns to the caller. It is optional; if not present, the function returns None.

Example with Docstring and Multiple Parameters

```
def add(a, b):
    """Return the sum of two numbers."""
    return a + b

result = add(3, 5)
print(result)
```

Output:

```
8
```

Why Use Functions?

1. Reusability: Functions allow you to write code once and reuse it multiple times. This reduces redundancy and makes your code more efficient.

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

    print(square(2))
    print(square(3))
    print(square(4))
```

2. Modularity: Functions help break down complex problems into smaller, manageable pieces. This modular approach makes it easier to understand, test, and maintain the code.

```
def get_area(radius):

  return 3.14159 * radius * radius

def get_perimeter(radius):
  return 2 * 3.14159 * radius

radius = 5
print("Area:", get_area(radius))
print("Perimeter:", get_perimeter(radius))
```

3. Abstraction: Functions abstract away the implementation details. Users of the function only need to know what the function does, not how it does it.

python

```
def fetch_data_from_database(query):

  # Simulating database access
  return ["data1", "data2", "data3"]

data = fetch_data_from_database("SELECT * FROM table")
print(data)
```

4. Maintainability: Functions make code easier to maintain. If a bug is found or an improvement is needed, changes can be made in one place, and all instances where the function is used will benefit.

5. Testing: Functions are easier to test individually. Unit tests can be written to test each function's behavior in isolation.

Built-in Functions vs. User-defined Functions

- **Built-in Functions:** Python provides many built-in functions like print(), len(), sum(), etc., which are always available for use.
  ```
  numbers = [1, 2, 3, 4, 5]
  print(sum(numbers))  # Output: 15
  ```

- **User-defined Functions:** Functions created by the user to perform specific tasks.
  ```
  def multiply(a, b):

  return a * b

  result = multiply(3, 4)
  print(result)  # Output: 12
  ```

- **Summary**

Functions in Python are essential tools for writing clean, efficient, and maintainable code. They promote code reuse, modularity, abstraction, and ease of testing. Whether using built-in functions or creating your own, functions are a fundamental concept in Python programming.
