# User sign-up and login system using a text file

def save\_credentials(username, password):

with open("user\_credentials.txt", "a") as file:

file.write(f"{username}:{password}\n")

def check\_credentials(username, password):

with open("user\_credentials.txt", "r") as file:

for line in file:

stored\_username, stored\_password = line.strip().split(':')

if username == stored\_username and password == stored\_password:

return True

return False

while True:

print("Bank App:")

print("1. Sign Up")

print("2. Log In")

print("3. Exit")

choice = input("Enter your choice: ")

if choice == "1":

username = input("Enter your username: ")

password = input("Enter your password: ")

save\_credentials(username, password)

print("Account created successfully!")

elif choice == "2":

username = input("Enter your username: ")

password = input("Enter your password: ")

if check\_credentials(username, password):

print("Logged in successfully!")

else:

print("Invalid username or password.")

elif choice == "3":

break

print("Thank you for using the Bank App!")

**WEEK ONE PYTHON**

**Introduction to variables**

**Va**riables are atemporal storage space in a computer’s memory.

* In Python, variables are used to store and manage data.
* A variable is a symbolic name that refers to a value or an object.
* You can think of a variable as a container for holding data.
* Variables allow you to work with and manipulate data in your Python program.

Every function is created with the initial value. A variable can be in three states.

Declaration: In this state, the variable is defined but has not yet been assigned a value. It exists in the program's scope but doesn't hold any meaningful data. For example, in many programming languages, if you declare a variable without assigning a value, it might contain null, undefined, or some similar default value.

Initialization: This is the state where the variable is assigned an initial value. It now holds a meaningful data or object reference. The variable is "initialized" with a value, and this value can change over time.

Variable changed (Execution): this the process of running a program so when the grogram is executed the instructions will be processed by the CPU in a sequence form so during this process of execution many operations changes including variable changes.

In Python, variables are defined in a standard way, by using the assignment character (=)

1. Variables Store Data: Variables are used to store and manage data in Python. They act as containers for values, making it easy to access and manipulate data within a program.

2. Dynamic Typing: Python is dynamically typed, which means that variable types are determined automatically based on the assigned value. You don't need to specify the data type explicitly.

3. Variable Naming Rules

- Variable names must start with a letter or an underscore (\_).

- They can contain letters, numbers, and underscores.

- Variable names are case-sensitive (e.g., `myVar` and `myvar` are different).

- Avoid using Python reserved words (e.g., `if`, `while`, `for`) as variable names.

4. Assignment Variables are assigned values using the assignment operator `=`. For example, `x = 10` assigns the value 10 to the variable `x`.

5. Reassignment: You can change the value of a variable by reassigning it. For example, `x = 20` would change the value of `x` to 20.

6. Variable Scope

- Variables have a scope, which defines where in the code they can be accessed.

- Local variables are defined within a function and are only accessible within that function.

- Global variables are defined outside of any function and can be accessed from anywhere in the code.

7. Printing Variables: You can print the value of a variable using the `print()` function. For example, `print(x)` will print the value of the variable `x`.

8. Data Types: Variables in Python can hold various data types, such as integers, floating-point numbers, strings, lists, dictionaries, and more. The data type is determined by the assigned value.

9. Variable Naming Conventions: It's common to use lowercase variable names with words separated by underscores, following the snake\_case naming convention (e.g., `my\_variable\_name`).

10. Constants In Python, there are no true constants (variables that cannot be changed). By convention, constants are represented in UPPERCASE with underscores (e.g., `MAX\_VALUE = 100`), but their values can still be changed. It's a convention rather than an enforced rule.

11. Variable Deletion: You can delete a variable using the `del` statement (e.g., `del x`). After deletion, the variable is no longer accessible.

12. Null or None: Python uses `None` to represent a null value or the absence of a value.

13. Type Casting: You can change the data type of a variable using type casting functions, such as `int()`, `str()`, and `float()`.

**INTRODUCTION TO DATA TYPES**

In Python, data types are classifications that specify which type of value a variable can hold. Understanding data types is crucial because it dictates how variables can be used and what operations can be performed on them. Python supports several built-in data types, and you can also create custom data types using classes.

**Integers**

-C

**Floating-Point Numbers**

- Floating-point numbers represent real numbers with a decimal point.

`pi = 3.14`, `price = 19.99`

- Common operations: all arithmetic operations, including exponentiation.

**Strings**

- Strings represent sequences of characters enclosed in single or double quotes.

`name = "Alice"`, `message = 'Hello, world!'`

- Common operations: concatenation, slicing, indexing, and string methods.

**Booleans bool**

- Booleans represent binary values, `True` or `False`.

`is happy = True`, `is raining = False`

- Common operations: logical operations (and, or, not), comparisons.

**Floating point numbers**

Floating-Point Numbers

- Floating-point numbers in Python are used to represent real numbers with decimal points.

- They are versatile and can handle a wide range of values, from very small to very large.

Creating Floating-Point Numbers

- You can create a floating-point number by simply writing a number with a decimal point, like `3.14`.

- Python also supports scientific notation for floating-point numbers, e.g., `1.5e3` represents 1500.0.

Arithmetic Operations with Floating-Point Numbers

- You can perform arithmetic operations (addition, subtraction, multiplication, division) on floating-point numbers just like with integers.

- However, be aware that floating-point arithmetic may introduce small rounding errors due to the binary representation of decimal numbers.

Mixing Integers and Floating-Point Numbers

- Python allows you to mix integers and floating-point numbers in arithmetic operations. The result is a floating-point number.

- For example, `2 + 3.5` results in `5.5`.

Rounding Errors in Floating-Point Arithmetic

- Due to the binary nature of computers, some decimal numbers cannot be represented exactly as binary fractions, leading to rounding errors.

- It's essential to be aware of these issues, especially in financial or precise scientific calculations.

Formatting Floating-Point Numbers

- You can format floating-point numbers to control the number of decimal places displayed using the `format ()` function or f-strings in Python 3.6+.

- For example, `"{:.2f}”. format(3.14159)` formats the number as "3.14."

Casting Floating-Point Numbers to Integers

- You can cast a floating-point number to an integer using the `int()` function, which truncates the decimal part.

- For example, `int(5.8)` results in `5`.

**STRINGS**

Single (' '), double (" "), or triple (''' 'or''' ") quotations are used to encapsulate them.

- You can build strings by using quotation marks around text, such as {name = "Alice"}.

- Strings can contain special characters, such as escape sequences (e.g., {\n} for a newline).

- The {+} operator can be used to concatenate or combine strings.

{greeting = "Hello" + "" + "World"} is one example.

- Characters are arranged in strings, and each character has an index.

- Indexing allows you to retrieve specific characters within a string; for example, `name[0]} returns the first character.

**Lambda expressions**

Lambda expressions, often referred to as "anonymous functions" or "lambda functions," are a concise way to create small, unnamed functions in Python. They are useful for creating simple, one-line functions without the need to define a formal function using the `def` keyword. Lambda functions are typically used when you need a quick function for a short, specific purpose.

Introduction to Lambda Expressions

- A lambda expression is a small, anonymous function that can have any number of arguments but can only have one expression.

- The syntax for a lambda function is: `lambda arguments: expression`.

Creating Lambda Functions

- You can create lambda functions for various purposes, such as mathematical operations, filtering, sorting, and more.

- For example, `add = lambda x, y: x + y` creates a lambda function that adds two numbers.

Use Cases for Lambda Functions

- Lambda functions are often used when you need a simple function that will be used only once, such as for sorting a list of objects based on a specific attribute or filtering a list.

- They can be used in combination with functions like `map()`, `filter()`, and `sorted()`.

Sorting with Lambda Functions

- Lambda functions can be used as the key function for sorting data structures.

- For example, `sorted(names, key=lambda name: len(name))` sorts a list of names by their length.

Filtering with Lambda Functions

- Lambda functions can be used to filter data based on a condition.

- For example, `even\_numbers = filter(lambda x: x % 2 == 0, numbers)` filters out even numbers from a list.

**Casting**

1. Dynamic Typing: Python is a dynamically-typed language, which means you don't need to declare the data type of a variable explicitly. Python interprets the type based on the value assigned to it. This allows for flexibility in casting.

2. Implicit Casting (Coercion): Python automatically performs some type conversions when it makes sense. For example, when you perform operations between different data types, Python may implicitly cast them to a common data type to complete the operation. For instance, when you add an integer and a float, the result is a float.

3. Explicit Casting: In cases where you need to perform specific type conversions, you can use explicit casting functions to ensure the conversion. Common casting functions in Python include:

- `int()`: Converts a value to an integer.

- `float()`: Converts a value to a floating-point number.

- `str()`: Converts a value to a string.

- `bool()`: Converts a value to a boolean.

**data types**

Data types are categories that classify values or variables in a programming language based on the type of data they can hold.

Dynamic Typing Python is a dynamically typed language, meaning you don't need to declare the data type explicitly. The type is determined at runtime.

3. Common Data Types Python supports various data types, including integers, floating-point numbers, strings, lists, tuples, dictionaries, and booleans.

4. Integers (`int`) Used to represent whole numbers, positive or negative, without a fractional part.

5. Floating-Point Numbers (`float`) Used to represent real numbers with decimal points or in scientific notation.

6. Strings (`str`): Used to store text or characters. Enclosed in single, double, or triple quotes.

7. Lists (`list`): Ordered collections of elements that can be of different data types. Lists are mutable.

8. Tuples (`tuple`): Ordered collections of elements like lists, but they are immutable, meaning their elements cannot be changed after creation.

9. Dictionaries (`dict`): Unordered collections of key-value pairs, where keys are unique and used to access values.

10. Booleans (`bool`): Represent either `True` or `False`. Used for logical operations and conditional statements.

11. Type Conversion (Casting): You can change the data type of a value using casting functions like `int()`, `float()`, `str()`, and `bool()`.