Prepared By - PRIYANKA NEOGI

Topics Completed :

1. Python as Calculator
2. Python's Popularity and Disadvantages.
3. Python Variable
4. Functions Definition
5. Data Types
6. Variable Declaration
7. Boolean operation
8. String: Error in quotes
9. Comment
10. String Concatenation
11. Take Input from User
12. Extract one character from String
13. Extract part of strings [SLICING]
14. Extract Data Non Sequential Manner
15. Reverse the String
16. Replace Values in String - Reassignment
17. Immutability of Strings

1. Python as Calculator:

- Python can be used as a calculator for performing various mathematical calculations.
- You can use it to perform basic arithmetic operations like addition, subtraction, multiplication, and division, as well as more complex mathematical operations.
- You can use Python's interactive shell or create a script to perform more complex calculations and manipulate data as needed.
- Python's extensive standard library provides many additional mathematical functions and modules for specialized calculations

```
1+2

3

3453+345

3798
```

API :

```
API in simple terms using a common real-world analogy:


You're at a restaurant. You -->(the customer) want to order food from
the
kitchen --> (the chef).
```

```
However, you can't just walk into the kitchen and start cooking
yourself.
There's a waiter --> (the API) who takes your order and delivers it to
the kitchen.
The waiter knows how to communicate your request to the chef and how
to bring the cooked food back to you.

You (the customer): You represent a software program or app that wants
to do something, like
requesting information or performing a task.


The Kitchen (the chef): This is like a computer system or service that
has the information or functionality
you need. It could be a weather service, a social media platform, or
any other piece of software.

The Waiter (the API): The waiter is like an API, which stands for
"Application Programming Interface."
Just as the waiter takes your order and conveys it to the chef, an API
takes your request and sends it to
the computer system. It also brings back the system's response to you.
The waiter doesn't have to know the cooking.


Your Order (the request): This is the specific thing you want the
kitchen to do, like "I'd like a
cheeseburger" or "Tell me the weather in New York." In the digital
world, your order is a set of
instructions or data sent to the API.

The Food (the response): This is what you get back from the kitchen
after your order has been processed. In the digital realm, it could be
information, like the weather forecast, or it could be an action
performed, like posting a tweet on social media.
```

An API is like a waiter that helps different software programs communicate and work together. It makes it possible for your app to request services or data from other apps or systems without having to know how those other apps are built or work internally. It's a way for different pieces of software to "talk" to each other and share information, just like a waiter helps you get the food you want from the kitchen.

API in Technical Terms :

- An API, or `Application Programming Interface`, is a set of rules and protocols that allows different software applications to communicate with each other.
- It defines the methods and data formats that developers can use to request and exchange information between different software systems, whether they are running on the same computer or distributed across a network.

- APIs serve as intermediaries that enable different software components to interact and share data without requiring the developers to understand the internal workings of the systems they are communicating with.
- This abstraction makes it easier to build complex applications by leveraging the functionality provided by other software services or platforms.

Use Cases: APIs are used for a wide range of purposes, including accessing web services (e.g., weather data, social media platforms), integrating with third-party software (e.g., payment gateways, social logins), and building modular, scalable applications.

Examples of well-known APIs include:

1. Google Maps API: Allows developers to integrate maps and location-related services into their applications.
2. Twitter API: Provides access to Twitter's data and functionalities, allowing developers to post tweets, retrieve user timelines, and more.
3. Facebook Graph API: Enables interactions with Facebook's social graph and user data.
4. GitHub API: Allows developers to interact with GitHub repositories, issues, and other platform features programmatically.

Developers often use APIs to enhance their applications, add new features, or integrate with third-party services, saving time and effort by leveraging existing functionality.

IDE (Integrated Development Environment) :

- There are many interpreters available freely to run Python scripts like IDE (Integrated Development Environment) that comes bundled with the Python software available for developers to choose from.

For Example - PyCharm, Visual Studio Code (VS Code), Spyder, IDLE(Integrated Development and Learning Environment)

2. The rising popularity of Python :

1. **Enhanced Code Readability and Conciseness**: Python places a strong emphasis on code readability and allows programmers to write shorter, more concise code, making it easier to understand and maintain. Less verbosity.

2. **Efficient Expression of Logical Concepts**: Python enables developers to express complex logical concepts using fewer lines of code when compared to languages such as C++ or Java.

3. **Support for Multiple Programming Paradigms**: Python is versatile, accommodating multiple programming paradigms, including object-oriented, imperative, functional, and procedural programming. This flexibility appeals to a wide range of developers.

4. **Rich Set of Built-in Functions**: Python provides an extensive collection of built-in functions that cover a broad spectrum of frequently used programming concepts. This simplifies development by reducing the need for custom code for common tasks.

   - Known as the “batteries included” (to describe Python's philosophy) ;It can help do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, email, XML, HTML, WAV files, cryptography, GUI and many more.

5. **Philosophy of Simplicity**: Python's core philosophy is rooted in the principle that "simplicity is the best." This philosophy permeates the language's design, making it accessible and intuitive for both beginners and experienced programmers alike.

6. **Robust Handling**: Robustness is evident through its exceptional

   - **exception handling capabilities**(it can handle them gracefully without crashing or causing major disruptions. This is important for ensuring the stability and reliability of the system) and

   - **built-in memory management techniques**(Memory management is crucial in computing to allocate and deallocate memory efficiently, preventing issues like memory leaks or excessive memory usage. Having built-in memory management techniques means that the system can efficiently utilize memory resources, further contributing to its robustness).

7. **Presence of third-party modules**: In Python it refers to the extensive ecosystem of external libraries and packages that are created by developers and organizations outside of the Python core development team. These third-party modules can be seamlessly integrated into Python programs to extend its functionality and provide solutions to a wide range of tasks and challenges

8. **(IoT)Internet of Things Opportunities**: It refers to the interconnectedness of everyday objects and devices through the internet, enabling them to collect and exchange data, communicate with each other, and be remotely controlled or monitored.

DISADVANTAGES OF PYTHON :

1. **Performance Concerns**: Python, being an interpreted language, may exhibit slower execution compared to compiled languages like C or Java, posing challenges for performance-critical tasks.

2. **Global Interpreter Lock (GIL)**: Python's Global Interpreter Lock (GIL) restricts concurrent execution of Python code by multiple threads, potentially limiting the parallelism and concurrency capabilities of certain applications, which can limit its ability to take advantage of multiple CPU cores.

3. **High Memory Usage**: Python can be memory-intensive, particularly when handling extensive datasets or executing intricate algorithms, leading to increased memory consumption.

4. **Dynamic Typing**: Python's dynamic typing allows variable types to change during runtime, which can complicate error detection and potentially introduce bugs.

5. **Package and Version Management**: Python's extensive library ecosystem may occasionally result in version conflicts and package-related challenges when integrating multiple libraries into projects.
6. **Flexibility vs. Rigidity**: Python's flexibility, while advantageous for rapid development and prototyping, can also make code less structured and harder to maintain, presenting a trade-off between adaptability and maintainability.
7. **Learning Curve**: Python, often considered beginner-friendly, can still pose a steep learning curve for newcomers, especially those with no prior programming experience.
8. **Exclusion from Mobile and Browser Environments**: Python may not be well-suited for mobile computing and browser-based applications, as it may lack comprehensive support and performance optimization in these contexts.

3. Python Variable

```
1. Question - When writing code in any programming language, what is
the purpose of writing that code?

We write code to implement logic.

2. Question - Why do we write this logic?

We write logic because, ultimately, we need to manipulate and work
with data.

Consider a simple website as an example. Even on such websites, when
we interact with buttons or call APIs, it involves processing data.

In every programming language and in every process, the logic we write
ultimately deals with processing
data. However, for the system to process data, it first needs to know
where to store that data.

The system must have an understanding of the data's location before it
can begin processing it. Without this understanding, the system will
not initiate the processing of the data.

This is where variables come into play.
```

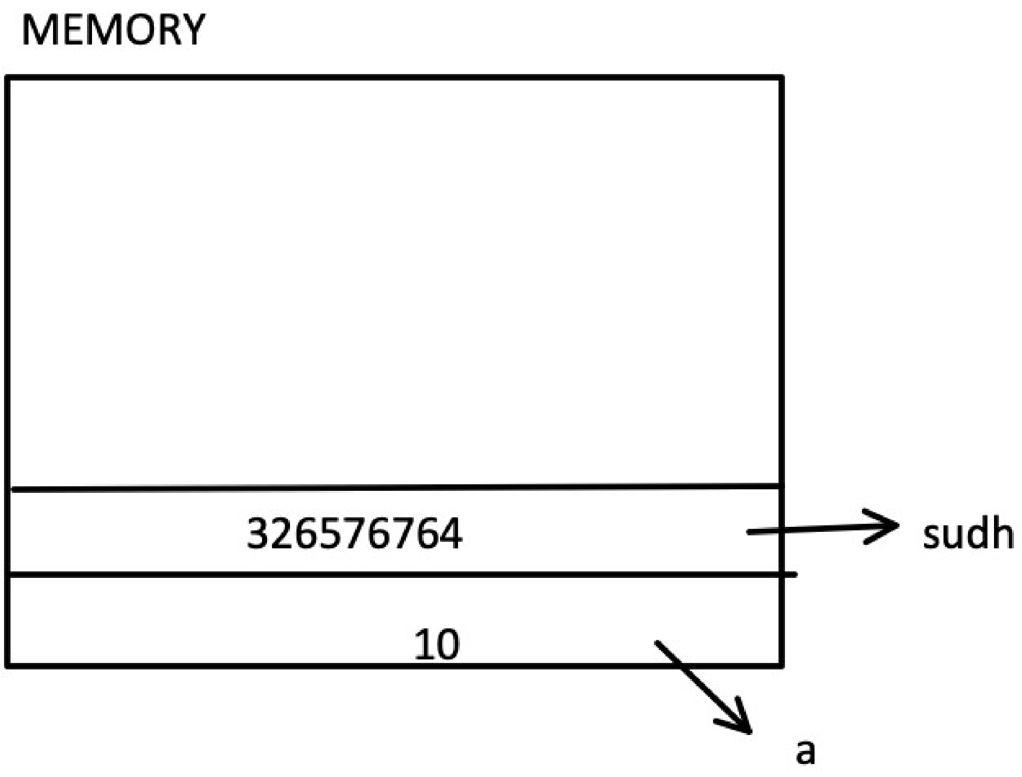
In Python, when we're working on solving a problem or implementing a particular logic, for processing or to manipulate and work with data we have to store the data. To do this effectively, we use variables to store and manage that data.

A variable is essentially a placeholder that allows us to store various types of data, which we can then manipulate using the logic we've written. That's what a variable is all about.

```
a = 10
```

```
a
```

```
10
```



When 'a' equals 10, it signifies that the system is attempting to establish a space for data, with 10 representing my specific data. 'a' serves as the container for storing the value 10.

Now, what occurs internally?

- A computer comprises components like the CPU, RAM, and hard disk for data storage. RAM is used for temporary data storage, while the CPU handles various operations. When we assign a value of 10 to a variable, the system attempts to access memory, which could be either RAM or CPU memory, depending on how the variable is processed.
- Since we are dealing with a temporary variable, it is stored in the main memory (RAM). The system allocates space for the value 10 within RAM, associating it with variable 'a.' Thus, the value 10 is stored in this reserved space.

The value 10 serves as a reference for what?

- It acts as a reference for the variable 'a.' I am associating 'a' with the data 10, so whenever you request the value of 'a,' it will provide you with the value 10.
- For instance, consider my name, 'my name.' As a person, my name is a placeholder that represents me, and it reflects my characteristics and attributes when talking or

working. Similarly, when we use a variable to store data, like variable 'a,' it serves as a container where the value 10 is stored. When you access variable 'a,' it returns the property, which is 10 in this case.

In essence, a variable is a placeholder that can hold various types of data.

You have the flexibility to assign any name you like your name as a variable name.

- The '=' symbol is indeed the assignment operator in many programming languages. Its primary purpose is to assign a value to a variable, effectively storing that value in the variable.

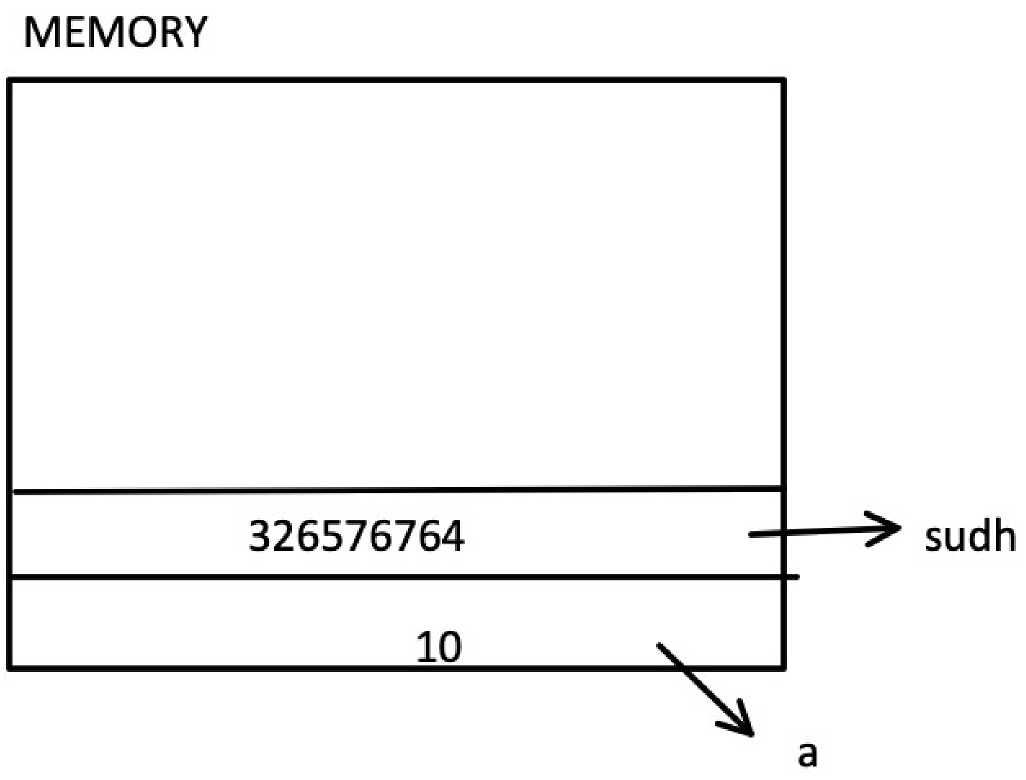
```
type(a)
```

```
int
```

type( ):

- To determine the type of a variable in many programming languages, you can use the `type` function or a similar built-in function. This function typically returns the data type or class of the variable, helping you understand the kind of data it contains.
- I'm attempting to store data within a variable called 'a,' which resides in memory. In contrast to some other programming languages where you must explicitly specify the data type before assigning a value to a variable, data in Python is more flexible.
- In Python, there's no requirement for explicit data type declarations when defining variables. You don't need to specify the data type for a particular variable beforehand.
- Despite not explicitly declaring the data type, Python has the ability to automatically deduce the data type based on the characteristics of the data itself.
- Python is considered a dynamically typed language, meaning that it can decide the data type without prior specification. This self-detecting property allows Python to adapt to the nature of the data, even if you've never explicitly mentioned the type you intend to store.
- With these different types of stored variables having different data types in memory we can further process the data as per requirement.

```
sudh = 326576764
```



When I attempt to call 'sudh,' it provides me with the exact data contained within it.

- It is attempting to allocate a dedicated space in memory for the data associated with the variable named 'sudh.' * Within this allocated memory space, it stores the value '326576764.'
- This value serves as a reference to the variable 'sudh.'
- So, whenever I make a call to 'sudh,' it retrieves and provides me with the data '326576764.'

```
sudh
```

```
326576764
```

Keyword - None:

This is a unique constant employed to represent a null value or absence of a value.

It's crucial to keep in mind that neither 0 nor any empty container (such as an empty list) equates to None.

None is an instance of its own data type, known as NoneType.

It is not feasible to generate multiple instances of None and assign them to variables.

```
print(None == 0)
```

```
False
```

4. Functions:

Functions are a category of built-in logic or predefined syntax that we can utilize to accomplish specific tasks or objectives. Python offers a wide array of functions designed to address various programming needs.

PYTHON SCOPES:

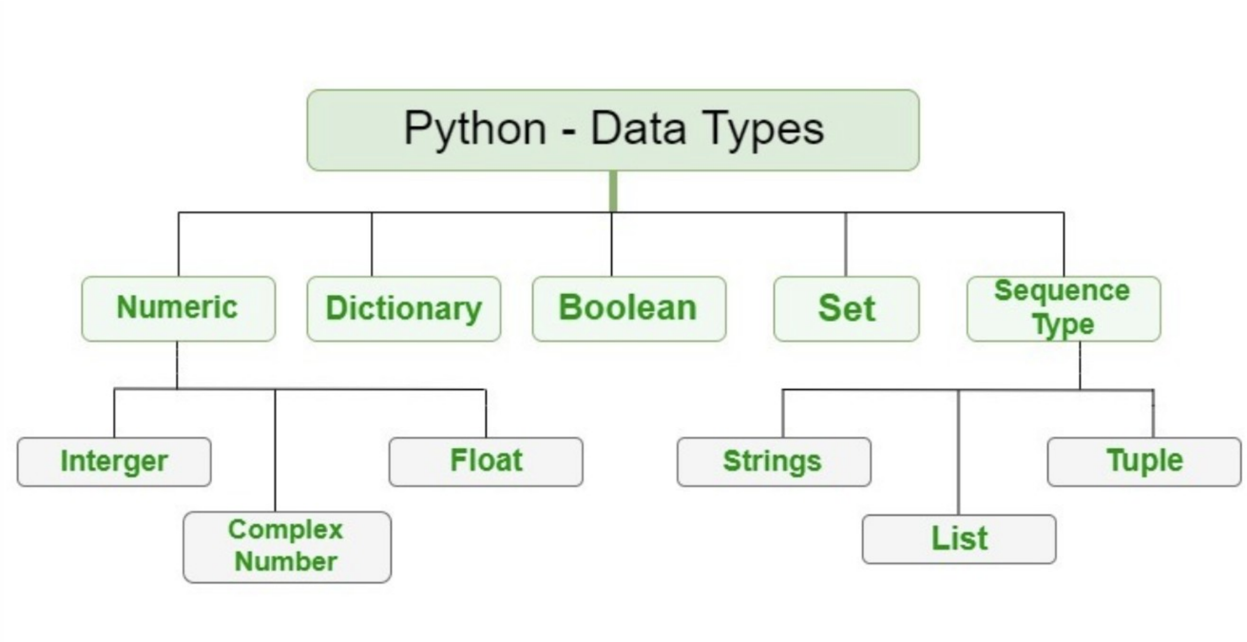
It's important to understand that Python doesn't rely on braces ({}); instead,

It uses indentation to define the scope of code blocks.

```
if True:
    print("This code block is indented, so it's part of the 'if'
statement.")
    print("Python uses indentation to determine code scope.")
else:
    print("This block is not executed because the 'if' condition is
True.")
```

5. DATA TYPES :

Data Type represent the type of data present inside a variable.



Python have 4 types of built in Data Structures namely :

- List
- Dictionary
- Tuple
- Set

1. INTEGER

- The integer data type, represented by `int` class , is a fundamental data type in many programming languages.
- It is used to represent whole numbers, both positive and negative, without any fractional or decimal parts.
- Integers can be used for a wide range of purposes, including counting, indexing, and performing arithmetic operations.
- You can perform various mathematical operations with integer variables, such as addition, subtraction, multiplication, and division.
- In Python, there is no limit to how long an integer value can be.

```python
my_integer = 42
type(my_integer)
```

```
int
```

2. STRINGS

1. **Strings in Python are arrays of bytes representing Unicode characters**:
   - A string is a data type used to store text.
   - These strings are essentially sequences or arrays of bytes, and they are capable of representing characters from various languages and symbol sets thanks to Unicode support.
   - This ensures that Python can handle text data in a versatile manner.
2. **A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote**:
   - You can enclose your text or characters within single quotes (' '), double quotes (" "), or triple quotes (''' ''').
   - Python provides this flexibility to make it convenient for developers to define strings in various situations.
3. **In Python, there is no character data type, a character is a string of length one**:
   - Unlike some other programming languages, Python does not have a separate data type for individual characters.
   - Instead, a single character is treated as a string with a length of one.
   - So, if you want to work with a single character, you can use a string containing just that character.
4. **It is represented by the str class**:
   - Strings in Python are represented using the `str` class.
   - The `str` class provides a range of methods and functionality for working with strings, making it a powerful tool for manipulating and processing text data in Python.

In summary, Python's strings are versatile data structures used to store text, which can contain characters from multiple languages and symbol sets due to Unicode support. You can create strings using different types of quotes, and even a single character is considered a string with a length of one. Python's `str` class is used to work with and manipulate these strings effectively.

```
Here 'Pwskills,' which consists of a combination of characters.

In programming, we refer to such combinations as "strings."

In Python, a string is essentially a sequence of characters.

So, when we talk about 'Pwskills' as a data type in Python, we're
referring to it as a string.

s = 'Pwskills'

type(s)

str

s

'Pwskills'

s1 = "Python"

s1

'Python'
```

Python is flexible, and regardless of which type of quotation marks you use, it will always represent the string with `single quotes` when it returns the data.

- When reading a PDF document in Python, it's not always necessary to keep every paragraph, word, or line as a single string. The choice of how to represent and store the text content from a PDF depends on your specific use case and what you intend to do with the data.
- The choice depends on the specific requirements of your project. Storing text at different levels of granularity (whole document, paragraphs, lines, words) allows you to perform various types of analysis and processing.

3. FLOAT

A float is a data type used to represent decimal or floating-point numbers.

This value is represented by the `float class`.

Floats are used to store real numbers, which can have a fractional part.

They are part of Python's numeric data types.

Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.

Floats can be used in mathematical operations just like integers. Python automatically performs type conversions when necessary.

Python also supports special floating-point values like positive and negative infinity (float('inf'), float('-inf')) and "not a number" (NaN) (float('nan')).

```
f = 456.76

type(f)

float
```

4. COMPLEX NUMBERS

In Python, complex numbers are a built-in data type used to represent numbers in the form `a + bj`, where a and b are real numbers, and `j` represents the imaginary unit, which is the square root of `-1`.

Complex numbers are often used in scientific and engineering applications, such as electrical engineering and signal processing.

```
# 'j' is the notation of iota in python
c = 5 + 6j
```

'5' represents the real part and '6' represents the imaginary part.

```
type(c)

complex

c

(5+6j)
```

You can create complex numbers using the

- complex() constructor or
- by specifying them directly with the j suffix.

```
z1 = complex(2, 3)  # Creates a complex number 2 + 3j
z2 = 1 + 2j  # Another way to create a complex number
print(z1)
print(z2)

(2+3j)
(1+2j)
```

Extarcting Real an Imaginary part from Complex Numbers

```
c.real

5.0
```

`When the system represents the integer part, it always appears with a0appended at the end as decimal.`

```
c.imag
6.0
type(c.imag)
float
```

- type(c.imag) -> only extracts the numerical value of the iota part. Hence its data type is float.
- This complete combination of real and imaginary part will be a 'Complex' datatype.

Python's `cmath` module provides functions for complex number operations, including trigonometric functions and exponentiation.

5. BOOLEAN

A boolean variable in Python is a data type that can hold one of two possible values: True or False.

Booleans are often used to represent binary logic,

- True corresponds to "yes" or "1"
- False corresponds to "no" or "0"

```
b = True
type(b)
bool
b1 = False
b1
False
type(b)
bool
```

Initialize a variable 'b' with a boolean value.

If the variable is set it to `True`, the system will automatically recognize it as a boolean variable when we inspect its data type i.e. `bool`.

Mathematical Operation

```
a1 = 45
a2 = 32

a1 + a2
```

```
77

a1 * a2

1440

a1 - a2

13

a1/a2

1.40625
```

Question: What is the rationale behind first creating variables, storing data in them, and then conducting mathematical operations on those variables? What is the purpose of introducing variables in this process?

- Creating a variable enhances its reusability.
- This eliminates the need to repeatedly write the value, promoting reusability.
- Additionally, it serves as a reference point for anyone reviewing my code, providing clarity on the purpose and content of the datatype in the variable.

Creating Variables in a single line

- Instead of writing a code in two lines
- We can create and declare any number of variables in a single line

Example -

Here the system will automatically understand that `a1 = 34` and `a2 = 45`.

```
a1 , a2 = 34 , 45

a1

34

a2

45

a,b,c,d = 354,'sudh',34.56,True

a

354

b

'sudh'

c
```

```
34.56

d

True
```

6. VARIABLE DECLARATION RULE:

1. Variable Names:

```
Variable names can consist of letters (both uppercase and lowercase),
digits, and underscores.

They must start with a letter (a-z, A-Z) or an underscore (_) followed
by letters, digits, or underscores.

Valid variable names: my_var, _value, temp123, data_file_2
```

1. Invalid variable names:

```
123abc (starts with a digit),

my-var (contains a hyphen),

@name, %name (contains special characters)
```

1. Case Sensitivity:

```
Python is case-sensitive, which means that uppercase and lowercase
letters are considered distinct.
So, myVar and myvar would be treated as two different variables.
```

1. Reserved Keywords:

```
You cannot use reserved keywords (also known as keywords or reserved
words) as variable names because they have special meanings in Python.
Examples of reserved keywords include if, else, while, class, def, and
many others. Attempting to use a
reserved keyword as a variable name will result in a syntax error.
```

1. Convention:

```
While not a strict rule, it's a common convention in Python to use
lowercase letters and underscores for
variable names (e.g., my_variable, count_of_items). This is known as
"snake_case" and is widely adopted
in Python code to enhance readability.
```

1. Descriptive Names:

```
Choose meaningful and descriptive variable names that convey the
purpose of the variable. This makes your
```

```
code more understandable to others (and to yourself) and is considered
good programming practice.
Here are some variable declaration examples that adhere to these
rules:

my_variable = 42

count_of_items = 10

user_name = "John"
```

Remember that following these rules and conventions not only helps avoid errors but also makes your code more maintainable and easier to understand for both you and others who may read your code.

```
%a = 234

UsageError: Line magic function `%a` not found.
```

Can't put a numeric value before .

```
1a = 3553

  Cell In[100], line 1
    1a = 3553
    ^
SyntaxError: invalid decimal literal
```

One of the wonderful aspects of Python is that it provides helpful hints by throwing errors when you make mistakes.

```
b1 = 65423
```

Can put a numeric value after.

```
@h = 64

  Cell In[102], line 1
    @h = 64
     ^
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of
'='?
```

'_' underscore can be used for a variable name.

```
_a = 34

-b = 543

  Cell In[104], line 1
    -b = 543
    ^
SyntaxError: cannot assign to expression here. Maybe you meant '=='
```

```
instead of '='?
```

When using reserved words 'keywords'

```
True = 45

  Cell In[105], line 1
    True = 45
    ^
SyntaxError: cannot assign to True
```

- Using reserved keywords as variable names, such as naming a variable "int" and assigning it a value, may initially seem acceptable. However, it can lead to confusion when you repeatedly use such variable names. The compiler will struggle to comprehend your code because "int" is a reserved keyword. Currently, no error is raised during assignment, but it will result in an error if you attempt to reassign a value to it. Therefore, it's advisable to avoid using reserved keywords as variable names.

```
int = 43

int

43
```

Coding Standardization is followed while declaring variable names

7. Boolean operation:

True -> 1

False -> 0

```
b1 = True

b2 = False
```

Internally depicted by system: when mathematical operation is performed on boolean data type , answer will be `0 , 1 or inf`

True = 1

False = 0

```
b1-b2

1

b1*b1

1
```

```
b1/b2

---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[112], line 1
----> 1 b1/b2

ZeroDivisionError: division by zero
```

Numpy Library will give output as `inf`. In core python it is `ZeroDivisionError`, error thrown by python.

8. String: Error in quotes:

```
s2 = "My name is Priyanka Neogi"

s2

'My name is Priyanka Neogi'

type(s2)

str

s3 = 'This is my first python class for Data Science Master's'

  Cell In[116], line 1
    s3 = 'This is my first python class for Data Science Master's'
                                                                  ^
SyntaxError: unterminated string literal (detected at line 1)
```

From - This.....Master -> It's detecting it as string because of apostrophe before letter `s`. Three quotes are used here, Sytem is not able to understand from where the string starts and where it ends. that's why its telling `unterminated string literal`. It is giving `^` to help you identify the mistake in that section. It's finding an extra quote here at `^`.

```
s4 = "This is my first python class for Data Science Master's"

s4

"This is my first python class for Data Science Master's"

s5 = 'This is my first python class for Data Science Master"s'

s5

'This is my first python class for Data Science Master"s'
```

If you are using Single quotes inside the string ... then always use - Double quotes outside for string enclosure.

If you are using Double quotes inside the string ... then always use - Single quotes outside for string enclosure.

9. Comments

To treat that line as comment in Python :

- Use hastags - # to create single line comments
- Use triple quote - (''') or (""") to create multi line comments
- Comments can provide explanations for complex or non-intuitive code. They help clarify the reasoning behind certain design choices.
- It's a best practice to write a comment in the code.

Then python will not interprete, it will not compile these lines

```
This is my python code to declare variable

n = 10
```

```
  Cell In[121], line 1
    This is my python code to declare variable
               ^
SyntaxError: invalid syntax
```

The initial line being written cannot undergo compilation as code, which is the reason behind the error. If there is an error in first line of code it will not execute the next line even if it is correct.

```
#This is my python code to declare variable

n = 10

'''this is my first python class
I am trying to learn variable declaration
so far I have learnt different variable types.'''
```

```
'this is my first python class\nI am trying to learn variable declaration\nso far I have learnt different variable types.'
```

```
"""this is my first python class
I am trying to learn variable declaration
so far I have learnt different variable types."""
```

```
'this is my first python class\nI am trying to learn variable declaration\nso far I have learnt different variable types.'
```

10. String Concatenation :

Regarding the '+' operator:

i. When applied to numeric values, it performs addition.

ii. When applied to string values, it performs concatenation.

iii. For successful concatenation, both values being concatenated must have the same data type.

```
s1 = 'sudh'

a = 10

s1 + a
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[127], line 1
----> 1 s1 + a

TypeError: can only concatenate str (not "int") to str
```

The variable 'a' holds an integer data type, while 's1' contains a string value. An error is occurring because the '+' operator can only concatenate strings to strings, not integers. In this context, attempting to use the '+' operator with a string and an integer results in an error.

```
str(a)
```

```
'10'
```

```
 '10'
```

```
'10'
```

```
s1+ " " + str(10)
```

```
'sudh 10'
```

To resolve this issue, you can convert the 'a' variable's data type to a string before performing the concatenation operation. You can achieve this by using the built-in function str() for type casting. Alternatively, you can enclose the integer value 10 within single quotes to represent it as a string.

```
'sudh' + 'kumar'
```

```
'sudhkumar'
```

```
'sudh' + " " + 'kumar'
```

```
'sudh kumar'
```

Concatenation using space

```
a

10

# integer to string
b = str(a)

b

'10'

# string to integer
int(b)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[136], line 2
      1 # string to integer
----> 2 int(b)

TypeError: 'int' object is not callable
```

Typecasting : Type casting involves converting the data type from one type to another.

Q - 'int' object is not callable?

Can't use reserve keyword as a variable name.

`int` is used as a variable name before that's why it will throw the error when used later because the value of the `int` reserved keyword itself has changed.

It will show callable error.

Solution - It's required to restart the kernel.

```
s = 'sudh'

int(s)

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[138], line 1
----> 1 int(s)

TypeError: 'int' object is not callable
```

Previously, when considering type conversion, the value was numeric, but 'sudh' is not a numeric variable; it is a combination of characters. It's important to note that you can only convert a string variable into an `int` data type if the string contains only numeric values. The string value shoud not include special characters as well.

11. Take Input from User :

This function allows us to obtain user input, enabling the flexibility to change the variable's value as needed.

The value of the variable will be determined by the input provided. It will accept and store whatever input is given. Instead of hardcoding a fixed value, we use the `input()` function, allowing for a wide range of input possibilities.

You can use the `input()` function to accept various inputs, but by default, it stores them as strings. If you want a different data type, you can perform type casting to achieve the desired result.

```
b = input()

123

b

'123'

type(b)

str
```

Initially, the variable 'b' is set to take user input, and it defaults to storing the input as a string. For instance, you can input an integer, and 'b' will hold it as a string. If you try to check the type of 'b' at this point, it will indicate that it's a string.

```
b = input()

34.45

b

'34.45'

type(b)

str
```

You can input a floating-point number, and 'b' will still store it as a string. If you check the type of 'b' again, it will still be a string.

```
c = input()

True

c
```

```
'True'

type(c)

str
```

Even if you input "TRUE" or "False," 'b' will store it as a string. Checking the type of 'b' will confirm that it remains a string.

```
d = input()

123

# using 'int()' function to typecast
type(int(d))

int

d = int(input())

34

d

34
```

Now, can we address this with type casting?

Yes, we can. If your goal is to obtain an integer, you can execute type casting by converting 'b' to an integer using `int(b)`. This will change the type of 'b' to an integer. You can then check its type, and it will indeed be an integer.

```
d = bool(input())

True

d

True

type(d)

bool

e = bool(input())

False

e

True

h = bool(input())
```

```
h

False
```

bool( ) function returns `True` if some parameter is passed and `False` otherwise if input is empty, when input function is used with Boolean Typecasting.

```
g = False

g

False
```

12. Extract one character from String

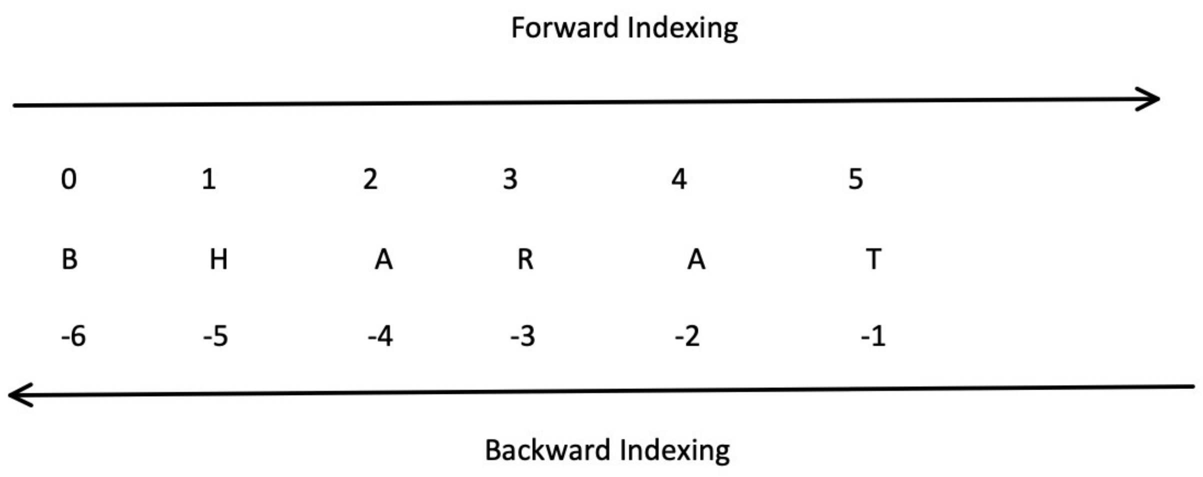
String is a combination of characters including space.

INDEXING

- Indexing by default by system - Can extract the 'character' using Indexing.

  1. Forward Indexing -> 0,1,2,3....

  2. Backward Indexing -> ......-4,-3,-2,-1



Let's consider a scenario where I declare a string 's' and assign it the value 'SUDH'.

Now, suppose someone asks if it's possible to extract individual characters or a range of characters from this string.

The answer is yes, and this is where the concepts of indexing and slicing come into play.

When a string like 'SUDH' is stored internally inside the memory, it's indexed both forwards and backwards.

```
s = 'sudh'

# Accessing 'u'
# 1. Forward Indexing
s[1]

'u'

s[3]

'h'
```

Forward indexing starts from 0, so 'S' is at index 0, 'U' at index 1, 'D' at index 2, and 'H' at index 3.

You can use these indexes to access individual characters. For example, to access 'U' from 'SUDH', you can use `s[1]` because 'U' is at index 1. Similarly, to access 'H', you can use `s[3]` because 'H' is at index 3.

```
s[100]

----------------------------------------------------------------------
-----
IndexError                                Traceback (most recent call
last)
Cell In[19], line 1
----> 1 s[100]

IndexError: string index out of range
```

If you try to access an index that is out of range, such as `s[100]`, you'll receive an "index error" because there is no character at index 100 in the string. Don't have data at that range.

```
s[-1]

'h'

s[-4]

's'
```

Backward indexing starts from -1, so 'H' is at index -1, 'D' at index -2, 'U' at index -3, and 'S' at index -4.

you can use negative indexing to access characters from the end of the string. For example, `s[-1]` will give you 'H', and `s[-4]` will give you 'S'.

```
s[-300]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[22], line 1
----> 1 s[-300]

IndexError: string index out of range
```

If you attempt to access an index that doesn't exist in the string, like `s[-300]`, you'll still receive an "index out of range" error because there's no character at index -300 in the string.

In summary, you can access characters in a string using both

forward and

backward indexing.

Valid indexes fall within the range of `-len(s)` to `len(s) - 1`.

13. Extract part of strings [SLICING]

```
s1 = 'pwskills'
```

Let's consider a situation where I have a variable called 's1' with the value 'PWSKILLS.'

The system internally stores 'PWSKILLS' with indexes. So, let's look at the indexing forward from 0 to 7 and backward from -1 to -8:

1. Forward indexing: 0, 1, 2, 3, 4, 5, 6, 7

2. Backward indexing: -1, -2, -3, -4, -5, -6, -7, -8

```
# Extract 'pw'
s1[0:2]

'pw'
```

To extract a portion of this string, for example, 'PW.'

This is where slicing comes into play. We need to extract characters starting from index 0 ('P') and go up to index 1 ('W').

Python slicing: the endpoint is exclusive. So, if we want to include 'W,' we have to use an endpoint of 2.

data given in output will be -> Upper bound-1.

It will not be able to extract data from upper bound .

To extract data till index 1, you have to take one value more i.e. `2`.

This means we're starting at index 0 and going up to index 2 (exclusive), which gives us 'PW.'

```python
# Extract 'skill'
s1[2:7]
```

```
'skill'
```

If I intend to access data up to the 6th position, I should specify the upper bound as the data range + 1. This way, I can effectively access data up to the 6th position. To put it simply, if I'm starting from index 2 and want to go up to index 7.

```python
# Extract 'skills'
s1[2:8]
```

```
'skills'
```

Suppose I need to extract the word 'skills,' not just 'skill,' from the string. We understand that our starting point is 2, but what should be our endpoint?

- The important thing to remember is that we only have indexes up to 7. We don't have an index of 8. when performing slicing operations, Python is more forgiving. Even if you specify an endpoint that exceeds the string's length, it still works. So, when extracting 'skills,' we can indeed specify an endpoint of 8, even though there's no index 8 in the string. It will successfully extract 'skills' from the string.

```python
s1[2:800]
```

```
'skills'
```

Here's the key point: Python slicing is flexible, and even if you specify an endpoint that is out of range (like one[0:8]), it will work and give you the maximum available slice. In this case, it would provide 'PWSKILL' without raising an error.

So, in summary, slicing allows you to extract specific portions of a string by specifying the starting and ending indexes (where the ending index is exclusive), and Python is forgiving if you provide an endpoint beyond the string's length.

When it comes to `indexing operations`, if you attempt to access indexes that are not available, Python will raise an error.

However, the situation is different when it comes to `slicing operations`. In slicing, you have more flexibility. You can specify indexes that exceed the actual length of the string, and Python will handle it gracefully.

For example, you can write 800 as an endpoint in slicing, and it will work, even though there's no index 800 in the string.

```
s1[2:]
```

```
'skills'
```

You can keep empty at the endpoint in slicing, and it will work. It will go till the end of the string.

14. Extract Data Non Sequential Manner

```
s1[0:7:2]        # 2 is step size here
```

```
'psil'
```

```
0 -> p

0+2 -> s

0+2+2 -> i

0+2+2+2 -> l
```

To extract specific characters 'P,' 'S,' 'I,' and 'L' from a string. Let's dive into this new question.

Initially, I could extract continuous data, which was straightforward. However, now the **task is to isolate individual characters 'P,' 'S,' 'I,' and 'L' from the string.** The challenge is to do this without getting the entire string.

To achieve this, I'll utilize slicing, just as before. Starting with the string 'S1,' my goal is to extract 'P' first, then 'S,' followed by 'I,' and finally 'L.' So, I'll set the starting point to 0 because 'P' is located at index 0. Now, for the endpoint, I need to consider how far I want to go. Since I want 'P,' 'S,' 'I,' and 'L,' I'll set the endpoint to 7, encompassing all these characters.

However, there's a crucial factor to introduce: the step parameter. In this scenario, I want to grab these characters sequentially, skipping 1 data. Therefore, I'll set the step to 2. This means I'll start at index 0, then proceed to index 2, then 4, and so on.

```
s1[0:7:1]      #  1 is step size here
```

```
'pwskill'
```

Default step size = 1.

In this context, you can specify three essential parameters:

- the first one sets the starting point,

- the second defines the endpoint,

- and the third determines the step or jump value you want to use.

Question : There's a noteworthy point to consider. Some may wonder if setting the jump value to zero would work.?

- The reasoning behind this is that when the jump is zero, it implies that you're essentially not moving or making any progress within the data. It's akin to starting at a position and staying there without transitioning to the next character.

Question :

So, I want to find this data in reverse order 'SLLI'. To achieve this, let's determine our starting point, which is "SLII." We can start from the 7th index, denoted as 7. Now, what's our endpoint?

Using "upper bound -1" for reverse direction. Therefore, moving backward, we can set our endpoint to 3.

```
s1[7:3]
```

```
''
```

I expected to retrieve the data successfully, but it turned out to be blank. So, what went wrong?

There is a conflict as step size is by default '1'

- Imagine a scale starting from 0 and going positively: 0, 1, 2, 3, 4, 5, and so on.
- Going in the opposite direction, it's -1, -2, -3, -4, -5, and so on.
- You are trying to move from +7 to +3.
- This means you are moving from a positive position to another positive position.
- By default, the jump or step size is +1.
- So, your data is moving in a positive direction, while your jump is also in a positive direction. This creates a conflict, which is why you are not getting an error but a blank result.

```
s1[7:3:-1]
```

```
'slli'
```

To fix this, you need to change the jump or step size to -1, indicating a move in the negative direction. Now, if you execute it with a jump of -1, you will be able to retrieve the data successfully. The reason is that your data is moving from +7 to +3, but this time your jump is in the negative direction, avoiding the conflict. This is why you are getting the desired data.

Repeating the above step with negative indexing.

```
s1[-1:-5]
```

```
''
```

By going from the -1 column to what point?

- I can go up to -5, but strangely, I'm encountering the same issue as before. Illustrate this using a scale again. - Here's my scale: zero, the positive side, and the negative side. I'm attempting to transition from -1 to -5, so I'm moving from -1 to -5.

Now, let's consider the step or jump direction. By default, it's +1, right?

- So, when I try to take a step with a value of +1, I'm moving in the positive direction. Can I say that this causes a conflict? Yes, it does, and as a result, I'm getting a blank result. We won't retrieve any information in this case.

There is a conflict as step size is by default '1', that's why output is blank.

```
s1[-1:-5:-1]
```

```
'slli'
```

What if I write -1 here? Would it work?

- Yes, it would because now my step direction has changed. I'm taking a step in the negative direction, matching the data movement direction. In this situation, I'll be able to obtain data.

So, to sum it up, if the step direction matches the data movement direction, you can retrieve data. Otherwise, it won't work. It's that simple. Whether you reference the negative or positive scale doesn't matter; the key is matching the directions.

15. Reverse the String

Reversing a string means it should display "SLIIKSWP," which is what I'm aiming for. So, what should I set as the starting point for reversing a string?

1. The starting point could be 7 in the positive direction or

2. maybe -1 in the negative direction.

-- **So, I can try starting with 7, where 7 is the last index, and go until what point? Should I go to 0? Will this work when I execute it? Can I retrieve the data this way?**

```
# Positive Indexing
s1[7:0]
```

```
''
```

Let's revisit the concept of the positive and negative scales. Imagine a scale with zero in the middle, the positive side, and the negative side. I'm trying to move from 7 to 0, and my default jump direction is positive. So, I'm jumping in a positive direction, which creates a conflict and results in a blank output.

```
# Positive Indexing
s1[7:0:-1]
```

```
'slliksw'
```

Can change the jump direction to -1, and when I execute it, I can obtain some data. However, there's an issue; it seems that "P" is missing from what I wanted. I specified to go from 7 to 0 in reverse, and I am getting data, but "P" is missing. Why is this happening?

- Whatever upper bound you give, it will take data up to just before that bound. So, if I'm moving in this direction and my upper bound is 0, it will give me data up to 1 index before 0. This is why "P" is missing.

```
s1[7:-1:-1]
```

```
''
```

I tried -1 as the upper bound, but it didn't work. Output is empty string. So, what should I write?

```
s1[7:-9:-1]
```

```
'sllikswp'
```

This works; starting from 7, going to -9 in the negative direction, and then jumping again with a negative step. Its should be more than upper bound -8 (negative indexing)

But if I try -8 or -7, it doesn't work. Any idea why this is happening?

```
s1[7:-8:-1]
```

```
'slliksw'
```

Not working. Its should be more than upper bound -8. (negative indexing)

```
s1[7:-700:-1]
```

```
'sllikswp'
```

Working. Its should be more than upper bound -8. (negative indexing). Here it is -700 which is greather than -8.

Just set the jump to -9, or any value beyond it will work.

The reason is we have data up to -8 (negative indexing) . Beyond -8, we don't have any more data.

So, when you specify -8, it will try to extract data until the end.

But if you specify a value beyond -8, it's treated as an upper bound because we don't have data beyond -8.

This is similar to what we did with +700 (in positive indexing) ; even if it's not available, it will go to the last available value.

```
s1[7::-1]

'sllikswp'
```

Even if I leave `upperbound` blank, I've specified the starting point but not the endpoint.

By default, it will continue until infinity, meaning it will extract all available data.

When I execute it, I will get the reversed string. It's perfectly fine to leave `upperbound` blank; it will gather all the available data. So, it starts from 7, moves in the specified direction, and extracts data as long as it's available. In this case, I am able to retrieve some data, and I've successfully reversed it.

```
s1[::-1]

'sllikswp'
```

I can even write it like this: `[:   :  -1]` and execute it.

The system will automatically recognize that I'm moving in a negative direction and extract all the data in reverse.

It's simple; I don't need an inbuilt reverse function for a string. Just use ": : -1," and you'll get the reverse of the data.

16. Replace Values in String - Reassignment

Reassignment operation:

```
a = 10

a = 20
```

Initially, we have `a` variable assigned the value of 10.

I can update this variable to 20.

So, which value will `a` be left with, 10 or 20? What is the final value of 'a'?

It's 20 because it will take the updated value.

```
s1[0]

'p'
```

Let's consider a scenario where I'm attempting to extract data from the zeroth index, which contains the letter "P."

```
s1[0] = 's'

----------------------------------------------------------------------
-----
TypeError                                 Traceback (most recent call
last)
Cell In[47], line 1
----> 1 s1[0] = 's'

TypeError: 'str' object does not support item assignment
```

Want to modify this value at the zeroth index. Specifically, I'd like to replace this "P" with another character, for example, "S."

In the context of string items, reassignment is not possible.

17. Immutability of Strings

```
s1

'pwskills'

s1 = 'swskills'

s1

'swskills'
```

Suppose you initially wrote "P," and then you tried to write "SWSKLLS" Now, I'm able to achieve the desired outcome. So, you suggested that instead of "P," you want to write "S". I can do this by reassigning the value.

However, this is not considered assignment in the context of a string.

This is not equivalent to assigning a value to a particular index.

```
s1[0] = 'p'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 1
----> 1 s1[0] = 'p'

TypeError: 'str' object does not support item assignment
```

With strings, this kind of assignment does not work.

It works with other data types, like lists, but not with strings.

This is because strings are considered immutable.

What does immutability mean?

- It means that if you cannot change the data at a specific index without altering the entire string, the data type is considered immutable.

- In this case, I'm discussing reassignment within the same string, not replacing the entire string.

So, can I state that strings are immutable?

- Yes, I can. Strings are an example of an immutable data type.

- To clarify, strings do not allow direct modification at a particular index while preserving the overall string.

- This concept of mutability and immutability distinguishes data types that can be changed at specific points from those that cannot.

- Strings fall into the latter category as they are immutable.