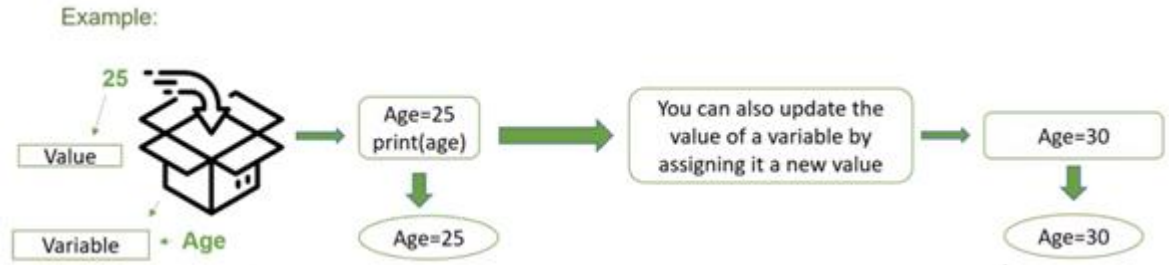
Variables and Data Types

Variables and Data Types are concepts that are essential for understanding how to store and manipulate information in Python.

What are variables?

View them as information containers. These containers allow us to store a variety of data types, including text, integers, and even groups of values.



A variable called "age" has been created, and its value is 25. The value 25 will now be represented by the variable "age" whenever we refer to it in our code.

There are several things we can do with this variable, for example printing it:

Example printing it:

print(age)

Running this code will display the value of the variable "age," which is 25.

Also, we can update the value of a variable by assigning it a new value:

The updated value of age is 30

,

age = 30

By assigning the value 30 to the "age" variable, we have now updated its value. Now if we print age the output will be 30.

The following are the rules for naming variables in Python. Following these rules is essential for writing clean and readable code.

1. **Variable names must start with a letter or an underscore. They cannot begin with a number. For example, ' _count ' and 'name' are valid variable names, but '3total' is not.**

2. **Variable names in Python can contain letters, numbers, and underscores. This flexibility allows us to create informative and versatile variable names. For instance, we can have a variable named 'studentName' to store the name of a student.**

It combines letters and uppercase letters to create a meaningful name.

Additionally, we can include numbers in variable names. For example, imagine we have a variable called 'average2', which stores the average value of a set of numbers.

The use of underscores is also common in variable names. Consider a variable named 'total_cost', which represents the total cost of an item or a transaction.

As you can see, the ability to include letters, numbers, and underscores in variable names provides us with the flexibility to choose descriptive and meaningful names for our variables. By leveraging this flexibility, we can create variable names that accurately reflect the purpose and content of the data they hold. Remember, clear and meaningful variable names make your code more readable and understandable.

3. **It's also important to note that variable names are case-sensitive, meaning that lowercase and uppercase letters are considered different. For example, let's consider two variables: 'age' and 'Age'.**

Although they only differ in capitalization, they are considered distinct variables in Python. They can hold different values and be used in separate parts of your code. Therefore, it's crucial to

be mindful of the case sensitivity when working with variable names. Keeping in mind the case sensitivity of variable names will help you write accurate and error-free code.

4. **Rule number four states that, variable names cannot contain spaces. If you need to represent multiple words, use underscores instead, like 'full_name' or 'my_variable'.**

Variable names cannot contain spaces	Full_name or My_variable

5. **Avoid using Python keywords as variable names. Keywords are reserved words that have special meanings in the Python language, such as 'if',' while', or 'for'. Using them as variable names will cause errors in your code. We shall explore the usage of these keywords ahead.**

Avoid using Python keywords as variable names	if',' while', or 'for

6. **Choose descriptive and meaningful names for your variables. This helps make your code easier to understand and maintain.**
7. **Lastly, keep your variable names concise but informative. Aim for a balance between being descriptive and not making them excessively long. This improves code readability.**

Python is a smart language that figures it out on its own. We don't need to tell the computer what kind of information we'll be storing in a variable.

In Python, we don't need to tell the computer what kind of information we'll be storing in a variable.

Data types are another fundamental concept in Python.

Data Type

Understanding data types is crucial for working with different kinds of information in your programs. A data type is simply a classification for different types of data.

A data type is simply a classification for different types of data

Understanding data types is crucial for working with different kinds of information in your programs

"What is Data?" So data is simply information. It can be numbers, text, images, audio, or any other type of information that can be stored and processed by a computer.

What is Data?

Data is simply information.
It can be numbers, text, images, audio, or any other type of information that can be stored and processed by a computer.

In Python, we use data types to define the kind of information we're working with. So let's take a look at the different types of data in Python.

In Python, we use data types to define the kind of information we're working with. So let's take a look at the different types of data in Python.

The following is an Index of Different Data Types used in Python.

1. **Numeric Data Types**
2. **String Data Types**
3. **Boolean Data Type**
4. **List data type**
5. **Tuple**
6. **Dictionary**
7. **Sets**

1. **Numeric Data Types:**

   Numeric data types represent numbers. There are three main types of numeric data in Python which are Integer, Float, and Complex.

```
#Numeric Data Type

# 3 Main types
# Integer, Float and Complex Numbers
```

The first type we'll discuss is Integers, denoted as 'int'. Integers are whole numbers without decimal points. You can think of them as counting numbers or zero.

For example, let's say we have a variable called 'age', and we assign it the value of 25. In this case, 25 is an integer because it's a whole number without any decimal points.

- 'age = 25'

```
# Integers

Age=25
```

Now, let's move on to Floats, denoted as 'float'. Floats are real numbers that can have decimal points. They are often used to represent values that require more precision or involve fractional parts. To illustrate, let's consider the variable 'weight', to which we assign the value of 55.67. Here, 55.67 is a float because it has a decimal point.

- 'weight'= 55.67

```
# Float
# has decimal points
Weight=55.67
```

Floats are commonly used to represent measurements, such as distances, weights, or values in scientific calculations.

The third main numeric data type in Python is the complex number, denoted as 'complex'. Complex numbers have both a real and an imaginary part and are expressed in the form a + bj, where a is the real part, b is the imaginary part, and j is the imaginary unit (equal to the square root of -1). For example:

- complex_num = 3 + 2j

```
#Complex Numbers
# expressed in a+bj

Complex_num=3+2j
```

In this example, 3 is the real part, 2 is the imaginary part, and j indicates the imaginary unit. Complex numbers are useful in various mathematical and engineering applications.

2. **String Data Types:**

   String data types are used to represent text or a sequence of characters in programming. You can think of strings as a collection of letters, numbers, symbols, or even empty spaces.

   For example, let's say we have a variable called 'name', and we assign it the value of 'John'. In this case, 'John' is a string because it consists of letters.

   - name= 'John'

   Strings are commonly used to store names, addresses, messages, or any textual information.

   Now let’s see another example where we have a variable called "message" and we want to assign it a string value. We can do it like this

   - message = "Hello, world!"

   Here, the string "Hello, world!" is assigned to the variable called "message."

```
# String Data Type
# represent characters, words etc
#it can also have special characters.
name='john'

message='Hello, World!'
```

Strings can also contain special characters and spaces. Like in the previous example where it contains both letters and punctuation marks.

One important thing to remember about strings is that they are enclosed within quotation marks, either single (' ') or double (" ") marks. This tells the computer that we are working with a string.

Strings can also be empty, containing no characters. Take a look at this example:

- empty_string = ""

Where the variable empty string is an empty string.

```
# They are enclosed with quotes

# they can also be empty

Empty_string=" "
```

We've covered the fundamentals of the string data type, let's explore a powerful feature in Python that significantly enhances the way we work with strings:

The f-strings.

While traditional methods of concatenating strings and using placeholders have been useful, Python introduces a more intuitive and concise way to format strings –
f-strings
or "formatted string literal". F-strings let you put expressions right into the string itself inside curly braces {}. This makes it simpler and more direct to create formatted strings.

Let's consider an example to illustrate the power of f-strings. Suppose we have a variable 'name' with the value 'John' and another variable 'age' with the value 25.

Instead of using complex concatenation or placeholders, we can effortlessly create a string representation using an f-string:

- name = 'John'

age = 25

formatted_string = f"My name is '{ name }' and I am '{ age }' years old."

print(formatted_string)

```
# F Strings or Formatted strings

name='John'
Age = 25
formatted_string=f"My name is {name} and my age is {Age} years"
print(formatted_string)
```

Output:

In this example, the f-string syntax, indicated by the 'f' prefix, allows us to embed the variables directly within the string using curly braces {}.

Let's see another example of Calculating the Area of a Rectangle:

- # Variables

length = 10

width = 5

Calculate area

area = length * width (Computes the area of the rectangle using the formula: length * width.)

Now Use the f-string to print the area of a rectangle.

print(f"The area of a rectangle with length ' length ' and width ' width ' is ' area ' square units.")

```
#Example to calculate area of rectangle

length=10
width= 5
area=length*width
print(f"The area of a rectangle with length {length} and width {width} is {area} square units")
```

Output:

```
The area of a rectangle with length 10 and width 5 is 50 square units
```

Inside the f-string, expressions within curly braces {} are used to embed the values of length, width, and area.

Later in the course, we will delve into a detailed discussion about the versatility of strings and the various operations they enable us to perform.

3. Boolean Data Type

What is the Boolean data type?

It's a special data type in programming that can only have two possible values:

True or False.

The Boolean data type is named after the mathematician and logician George Boole, who laid the foundation for modern computer science. It plays a crucial role in decision-making and controlling the flow of programs. And they represent the truth or falsity of a condition.

For example, let’s consider a simple example:

- x = 10

a = x > 5

print(a)

```
#Boolean Data Type
#True or False

#Example
x=10
a=x>5
print(a)
```

Output:

```
True
```

Since x is equal to 10 which is greater than 5, the variable “a” is assigned the value True, and the print function will display True on the screen " because x is indeed greater than 5.

We can then use this boolean value in other parts of our program which we will learn later in the course.

It's a fundamental building block in programming, and you'll encounter it frequently as you dive deeper into coding.

4. **List Data Type**

Lists are a powerful data structure in Python that allows us to store and organize multiple values in a single variable. Think of it as a container that can hold different items, just like a grocery shopping list! Imagine you have a grocery shopping list for the week. Each item on the list represents an element, such as apples, bread, and milk. In Python, you can create a list to store these items in a similar manner:

- grocery_list = ["apples", "bread", "milk"]

```
#List Data Type
#Store and organize multiple values
# Always in Square brackets and are seperated by commas
# can have diff data type
grocery_list=['apples','bread','Milk']
```

A list has been created called a grocery list that contains three elements: "apples", "bread", and "milk". Each item in the list is enclosed in square brackets [ ], the square brackets ([ ]) indicate that we're creating a list, and elements in the list are separated by commas. Lists allow us to store and access multiple items conveniently.

Lists can also contain items of different data types. For instance:

- mixed_list = [42, "Hello, world!", True, 3.14]

```
Mixed_list=[42,"hello, world!",True,3.14]
```

In this case, our list called "mixed_list" contains an integer, a string, a boolean value, and a floating-point number. Lists are flexible and can hold a combination of different data types.

One of the fantastic features of lists is their ability to be modified. We can add or remove items from a list.

```
#lists can be modified also called as muttable
# list can add or remove items
```

We'll practice and explore examples related to list manipulation in later chapters of this course.

5. **Tuple**

In Python, a tuple is similar to a list, but with one important difference: it is immutable data meaning their values cannot be changed once defined, unlike mutable lists. Think of it as a sealed envelope that contains information you can't modify once it's sealed!

Imagine you have a tuple that represents the coordinates of a treasure buried in a secret location. Each coordinate consists of two values: latitude and longitude. In Python, you can create a tuple to store this information like this:

- treasure_location = (40.7128, -74.0060)

```
#Tuple
#similar to list but are immutable
# always enclosed in parentheses

treasure_location=(40.7128,-74.0060)
```

Here, we have created a tuple called treasure_location that contains two elements: the latitude 40.7128 and the longitude -74.0060. The elements are enclosed in parentheses () distinguishing them as a tuple and are separated by commas.

Like lists, tuples can also contain items of different data types. For instance:

- person = ("John Doe", 25, "john.doe@example.com")

```
# can have different data type same as list

person=("john doe",25,"john.doe@example.com")
```

In this case, our tuple called "person" contains a string representing a name, an integer representing age, and another string for an email address. Tuples, just like lists, provide flexibility for storing multiple data types together.

Although tuples are immutable, meaning their contents cannot be changed, we can still access their values. We use indexing to retrieve specific elements from a tuple. Indexing is similar to how we navigate through a numbered list to find a particular item!

Imagine you have a tuple that represents the scores of students in a class. Each element in the tuple corresponds to the score of a specific student. In Python, you can access individual scores by their position using indexing.

- student_scores = (78, 85, 92, 79, 88)

first_score = student_scores[0]

Here, we have a tuple called student_scores that contains the scores of five students. The indexing starts from 0, so the first element, 78, has an index of 0. By using student_scores[0], we retrieve the first score and assign it to the variable first_score.

Similarly, we can access other scores by adjusting the index. For example, to get the third score, 92, we use student_scores[2] because the index for 92 is 2.

third_score = student_scores[2]

In this case, the variable third_score will be assigned the value 92.

It's important to note that indexing in Python starts from 0, not 1. So, the first element is at index 0, the second element is at index 1, and so on.

```
# we can access values using indexing

student_score=(78,85,92,79,88)

first_score=student_score[0]
print(first_score)
# similarly we can access other scores

third_score=student_score[2]
print(third_score)
```

Output:

```
78
92
```

So, although tuples are immutable, we can still access their values using indexing. Indexing helps us retrieve specific elements from a tuple based on their position, just like finding an item in a numbered list. Keep in mind that indexing in Python starts from 0, and the general syntax for indexing is variable_name[index], the variable name followed by square brackets containing the index number. And it remains consistent across various data structures in Python like lists, tuples, and strings.

6. Dictionaries:

Tuples, allow us to store and access multiple values, and dictionaries take it a step further which is our other data type.

In simple terms, a
dictionary
is like a real-life dictionary where you look up a word and find its corresponding definition. Dictionaries are made up of key-value pairs, where each value is associated with a unique key.

Let's see an example where we create a dictionary:

- person = {

"name": "Smriti",

"age": 25,

"country": "India"

}

```
#Dictionary
# key-value pair

#example

Person={
        'name':'smriti',
        'age': 25,
        'country':'india'
        }
print(Person)
```

OutPut:

```
{'name': 'smriti', 'age': 25, 'country': 'india'}
```

Here we have a simple program that stores information about a person: their name, age, and country. The keys are "name", "age" and "country" and their corresponding values are " Smriti" 25, and "India".

To create a dictionary, we use curly braces {} and separate the key-value pairs with colons (:). In this case, we'll call our dictionary "person" and assign values to the keys "name," "age," and "country."

To access a value, we use the key within square brackets []. For example, if we want to access the person's country, we can write:

- country _name = person["country "]

In this case, the variable " country _name " will store the value " India", if we print country _name it will display "India"

print(country _name)

```
country_name=Person["country"]
print(country_name)
```

output:

```
india
```

Furthermore, dictionaries are mutable, meaning we can modify their values once they're created. We can add, update, or remove key-value pairs as needed, making dictionaries dynamic and flexible.

Let's say we want to change the person's age. We can do that by assigning a new value to the "age" key.

Modifying a value in the dictionary

- person["age"] = 30

In this example, we'll update the person's age to 30 by assigning a new value to the "age" key.

Let’s print the dictionary now after updating the age

- print(person)

```
{'name': 'smriti', 'age': 30, 'country': 'india'}
```

We can see that the age has been update from 25 to 30.

Dictionaries also allow us to add new key-value pairs.

We want to include the person's occupation.

We can do that by simply assigning a value to a new key.

Adding a new key-value pair to the dictionary

- person["occupation"] = "Software Engineer"

In this example, we'll add the person's occupation by assigning a value to the "occupation" key.

When we run the program again, we'll see that the output now includes the occupation.

- print(person)

```
# we also add new key value pair

# adding new key value pair
Person['occupation']='Software engineer'
print(Person)
```

Output:

```
{'name': 'smriti', 'age': 30, 'country': 'india', 'occupation':
'Software engineer'}
```

And there you have it! Dictionaries are powerful tools for organizing and retrieving data using meaningful keys.

They're incredibly flexible and widely used in Python.

7. Sets:

A set in Python is like a special container that allows you to hold only unique items.

Sets are defined using curly braces ({ }), and the elements inside the set are separated by commas.

The following example can help you understand:

- numbers = {1, 2, 2, 3, 4, 4, 5}

print(numbers)

```
numbers={1,2,2,3,4,4,5}
print(numbers)
```

Outputs:

```
{1, 2, 3, 4, 5}
```

In this case, we define a set called "numbers" with duplicate elements. However, when we print the set, it automatically removes the duplicates, retaining only the unique elements.

Sets are also mutable, meaning we can add or remove elements from them.

Understanding these data types is crucial for effective programming and solving various problems.

The brackets that we assign to different data types and their subsequent syntax vary from one data type to another.

This helps Python identify the data type and understand the type of information that is stored in the variable.