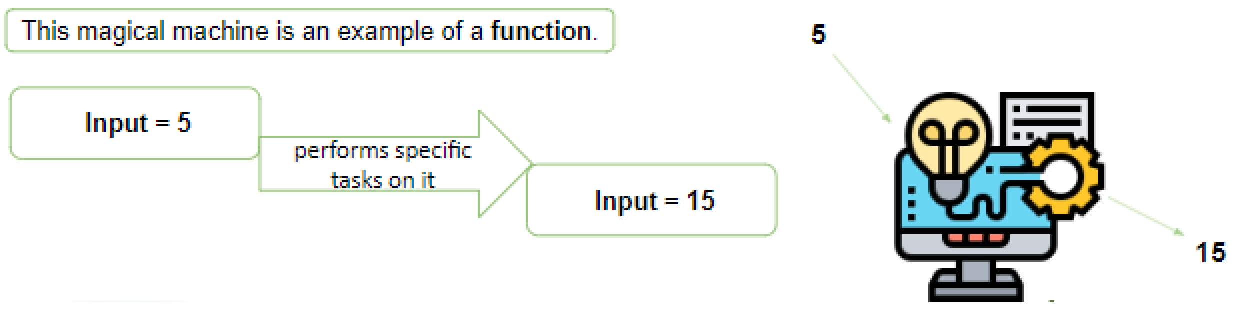
Functions

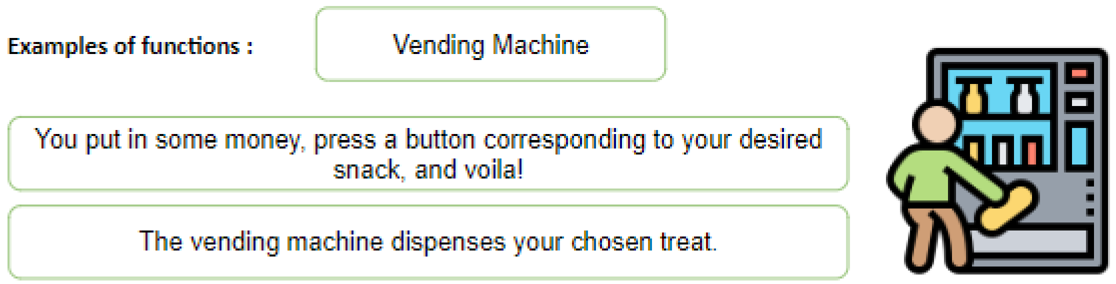
Let's imagine you have a magical machine that can take any number and multiply it by 3. Whenever you input a number, this machine will return the result of that number multiplied by 3. This magical machine is an example of a function.


This magical machine is an example of a function.
Input = 5
performs specific tasks on it
Input = 15
5
15

It takes an input, performs some specific tasks on it, and gives you an output.

Let’s consider another real-world example of functions.

Consider a vending machine. You put in some money, press a button corresponding to your desired snack, and voila! The vending machine dispenses your chosen treat.


Examples of functions :
Vending Machine
You put in some money, press a button corresponding to your desired snack, and voila!
The vending machine dispenses your chosen treat.

The vending machine is like a function: it takes your input (the money and the button you pressed), processes it (checks if the amount is sufficient and dispenses the snack), and gives you the output (the snack you selected).

So, what exactly is a function?

Well, a function in Python is a named block of code that performs a specific task. It takes in input values (if any), performs operations on them, and produces an output value (if specified).

Functions are reusable and help in organizing code by breaking it into smaller, manageable pieces.

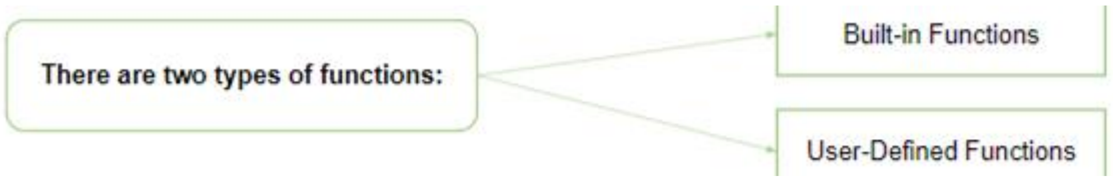
Let's consider an example. Suppose we have a function called calculate_square that takes a number as input and returns the square of that number.

Now, we want to calculate the squares of different numbers at different places in our code. Instead of writing the squaring logic each time, we can simply call the calculate_square function. Here, the calculate_square function is reusable because we can call it with different values whenever we need to calculate a square. We don't have to repeat the squaring logic every time we want to calculate a square.

This reusability saves us from duplicating code, reduces the chances of errors, and makes our code more modular and maintainable.

But functions are not just about code reuse; they also enable us to create more expressive and logical programs. By dividing our code into functions, we can give meaningful names to different parts of our program, making it easier for ourselves and others to understand and maintain the code.

Now, let's talk about the two types of functions:



- built-in functions and
- user-defined functions.

So the first kind of function is Built-in functions.

These are functions that Python already provides for us to use. They come pre-loaded with Python and perform a variety of useful tasks. You can think of them as ready-made tools in a toolbox. For example, the print() function is a built-in function that helps us display messages or values on the screen.

Let's say you want to greet a friend by printing "Hello!" on the screen. You can use the print() function to achieve that. Here's an example:

- print("Hello!")

That's it! With just one line of code, the built-in print() function helps you greet your friend.

In the print("Hello!") example, the argument is the string we want to print.

In summary, built-in functions are pre-defined functions provided by Python for common tasks. We can use them by simply calling the function name followed by round brackets () to enclose the arguments. These functions save us time and effort by providing ready-made solutions to perform specific operations.

During the exercise at the end of the video, we'll have the opportunity to explore and discover even more built-in functions, expanding our knowledge and unlocking new possibilities in Python programming.

Apart from built-in functions, you can also create your own functions. These are called user-defined functions because you define them yourself. User-defined functions allow you to customize and extend Python's capabilities to suit your specific needs.

The basic syntax for declaring a function is as follows:

- def function_name(parameters):

# function body

return result # optional

Step 1: Define the function:

We start by using the**'def'**keyword, followed by the name we want to give to our function. The name should be descriptive and represent the task the function will perform.

Step 2: Parameters:

Inside the parentheses (), we can specify any parameters or inputs that our function needs to perform its task. These parameters act as placeholders for the values that we'll pass when we call the function. Parameters are separated by commas if there are multiple.

: (colon) signifies the beginning of the function body.

Step 3: Function Body:

The function body consists of the block of code that performs the specific task. It is indented beneath the function definition. This block of code can include any valid Python statements, such as calculations, conditionals, loops, or other function calls.

Step 4: Return Statement:

If our function needs to produce a result or an output, we use the return statement. It specifies the value or values that the function will return when it is called. The return statement ends the execution of the function.

Step 5: Calling the Function:

To execute or use the function, we call it by its name followed by parentheses (). If the function has parameters, we pass the corresponding values as arguments inside the parentheses. These arguments can be of any value and should match the order and number of the function's parameters.

Parameters and Arguments:

Think of parameters as variables that a function expects. They're like empty boxes that the function wants to use, but they need you to put something in them. For example, if a function needs two numbers, it might have parameters called num1 and num2.

Now, arguments are the actual values you put into those empty boxes (parameters) when you use the function. So, if the function wants two numbers, you give it real numbers as arguments. If the function parameters are num1 and num2, you might provide the arguments 5 and 10 when you use the function.

By default, a function must be called with the correct number of arguments. Meaning that if your function expects 2 arguments, you have to call the function with 2 arguments, not more, and not less.

Let’s understand these steps with an example where we write the function to add two numbers, a and b.

- def add_numbers(a, b):

  sum = a + b

  return sum

We use the def keyword to define our function called add_numbers. Inside the parentheses, we specify two parameters, ‘a’ and ‘b’, which represent the two numbers we want to add.

Inside the function body, we calculate the sum of ‘a’ and ‘b’ using the + operator and store it in a variable called ‘sum’. Finally, we use the return keyword to specify that we want the sum to be the output of our function.

Now we pass values to the function:

- a = 2

  b = 3

  result = add_numbers(a, b)

In this step, we assign the values 2 and 3 to the variables a and b, respectively. These values will be passed as arguments when we call our function add_numbers. The arguments are the actual values that are provided to the function when it is called. Here, we pass the values of a and b to the add_numbers function using the syntax add_numbers(a, b). The result of the function call, which is the sum of the two numbers, will be stored in a variable called result.

Finally, we use the print() function to display the result. By combining strings and variables using commas, we create a meaningful output message that shows the sum of the two numbers. Here, we print the string "The sum of", followed by the values of a and b, and the string "is:", and then the value stored in the result variable.

- print("The sum of", a, "and", b, "is:", result)

To see the function in action, let's run with the provided example values:

def add_numbers(a, b):

```
sum = a + b
return sum
a = 2
b = 3
result = add_numbers(a, b)
print("The sum of", a, "and", b, "is:", result)
```

When we run this, we should see the following output:

```
The sum of 2 and 3 is: 5
```

In this function, add_numbers, the lines sum = a + b and return sum are indented by four spaces or one tab. This consistent indentation is important because it tells Python that these lines of code belong to the function.

Think of it like organizing items in a drawer. When you open the drawer, you want to see items neatly placed inside.

Similarly, Python expects the lines of code inside a function to be neatly indented, so it can easily identify the function's code block.

As mentioned earlier Without indentation, Python wouldn't know which lines of code are part of the function and which ones are outside. Indentation helps Python understand the structure and hierarchy of your code. It ensures that the lines indented within the function are treated as part of the function's body. Any code outside the function is not indented and is considered separate from the function.

Let's consider another example of a user-defined function without any parameters. This example will highlight how functions can perform specific tasks even without needing any inputs.

- def say_hello():

```
    print("Hello, there!")
say_hello()
```

In this example, we have a user-defined function called say_hello()

In the first line, we define the function using the def keyword, followed by the function name say_hello, and parentheses ().

Inside the function body, we have a single statement, which is the print() function. It prints the message "Hello, there!" to the console.

After defining the function, we call it using say_hello(). This line of code executes the function and triggers the printing of the greeting message.

The various types of Python function arguments. In Python, functions can receive input in different ways, and understanding these types of arguments adds flexibility to how functions can be used.

The main types:

- **Default Argument**: Imagine you're ordering a pizza. By default, it comes with cheese, right? But if you want something special, like pepperoni, you need to tell them. In the pizza world, "cheese" is like a default topping, and if you don't specify, you get cheese automatically. Similarly, in a function, you can set default values for some things. So, if you don't provide a specific value when you use the function, it automatically uses the default.

Let's say we have a function that greets people. By default, it says "Hello," but if you want a different greeting, you can tell it.

```
def greet(name, greeting="Hello"):
print(f"{greeting}, {name}!")
# Using the function without providing a greeting
greet("Bob")  # Output: Hello, Bob!
# Using the function with a custom greeting
greet("Alice", "Hi")  # Output: Hi, Alice!
```

The function greet has a parameter called greeting, and we set its default value to "Hello".

When we call the function without providing a specific greeting (like in the case of greet("Bob")), it uses the default value and says "Hello, Bob!".

When we call the function with a custom greeting like in the case of greet("Alice", "Hi")), it uses the provided value and says "Hi, Alice!". In this, we have changed the greeting from Hello to Hi.

- **Positional Argument**:**As previously discussed a positional argument is a value that you provide to a function or a method based on its position or order in the function's parameter list.**

Imagine you have a simple function called add_numbers that adds two numbers together. Here's the function:

- def add_numbers(x, y):

  result = x + y

  return result

In this function, x and y are parameters that represent the two numbers you want to add. Now, let's use this function with positional arguments:

- result = add_numbers(3, 5)

  print(result)

In this example, 3 is the positional argument for x, and 5 is the positional argument for y. The function will add these two numbers together, and the result will be 8. The order in which you provide the arguments matters – the first argument goes to the first parameter, and the second argument goes to the second parameter.

So, in simple terms, when you use positional arguments, you're passing values to a function based on their position in the function's parameter list.

- **Keyword Argument: In programming, a keyword argument is a value you provide to a function or a method by explicitly**

specifying the parameter name followed by the value. Let’s understand this with an example:

Let's modify the above example for subtraction:

- def subtract_numbers(x, y):

  result = x - y

  return result

Now, let's use this function with both positional and keyword arguments:

- result_pos = subtract_numbers(8, 3)

  print(result_pos)

In this case, 8 is the positional argument for x, and 3 is the positional argument for y. The result will be 5.

Now

- result = subtract_numbers(x=8, y=3)

  print(result)

Here, x=8 and y=3 are keyword arguments. As before, the order doesn't matter, and the result will still be 5 even if you switch the positions of x and y:

- result_ switched = subtract_numbers(y=3, x=8)

  print(result_ switched)

Both calls with positional and keyword arguments result in the same output: 5

- **Arbitrary arguments (*args):arbitrary positional arguments, often denoted as *args allow a function to accept a variable**

number of arguments. This means you can pass any number of values to the function.

Think of a pizza place where customers can choose any number of toppings for their pizza. The chef doesn't know in advance how many toppings each customer will ask for.

- #Easy Python Function

  ```
  def make_pizza(*toppings):
  print("Making a pizza with the following toppings:")
  for topping in toppings:
  print("-", topping)
  ```

- # Ordering Pizzas

  make_pizza("Cheese", "Pepperoni")(Print them separately and show the output according as explained below)

  make_pizza("Mushrooms", "Onions", "Bell Peppers")

  make_pizza("Sausage", "Green Olives", "Tomatoes", "Spinach")

The***toppings**in the function definition collects any number of toppings into a tuple named toppings.

The first make_pizza call has two toppings: "Cheese" and "Pepperoni."

It prints two items: "Cheese" and "Pepperoni."

In the second call, make_pizza("Mushrooms", "Onions", "Bell Peppers"), it prints three items: "Mushrooms," "Onions," and "Bell Peppers."

In the third call, make_pizza("Sausage", "Green Olives", "Tomatoes", "Spinach"), it prints four items: "Sausage," "Green Olives," "Tomatoes," and "Spinach."

This flexibility is what Arbitrary arguments provide—accepting any number of positional arguments and processing them accordingly.

1. **Variable length keyword arguments: Variable length keyword arguments, often denoted by **kwargs in Python function definitions, allow a function to accept an arbitrary number of keyword arguments. Keyword arguments in Python allow you to pass values to a function by specifying the parameter names along with their corresponding values. These arguments are then captured and stored in a dictionary named **kwargs.**

Lets see it with an example for a better understanding:

- def print_info(**kwargs):

```
for key, value in kwargs.items():
print(f"{key}: {value}")
# process each key-value pair as needed
print_info(name='John', age=25, city='New York')
```

The****kwargs**in the function definition indicates that the function can receive any number of keyword arguments. The term kwargs is a convention; you could use any other name, but kwargs (short for "keyword arguments") is commonly used for clarity.

Inside the function, there's a loop that goes through each key-value pair in the kwargs dictionary and prints them. So, it prints something like

Output:

```
name: John

age: 25

city: New York
```

Now let’s add occupation keyword argument to your print_info function call,:

```
print_info(name='John', age=25, city='New York', occupation='Software Engineer')
```

So the output would include the new information:

```
name: John
age: 25
city: New York
occupation: Software Engineer
```

lambda function.

A**lambda function**in Python is a concise way to create anonymous or unnamed functions. These functions are defined using the lambda keyword, followed by the input parameters, a colon, and the expression to be evaluated. Lambda functions are often used for short-lived operations where a full function definition is not necessary.

Here's a basic syntax for a lambda function:

- lambda arguments: expression

Let's take a simple example of a regular function to find the square of a number and then create an equivalent lambda function.

```
# Regular function to find the square of a number
def square_regular(x):
return x ** 2
```

The regular function is called by providing an argument (5 in this case), and the result is assigned to result_regular and printed.

```
result_regular = square_regular(5)
print(result_regular)  # Output: 25
```

Now let’s see an equivalent lambda function to calculate the square:

```
square_lambda = lambda x: x ** 2
```

Here The lambda function is created using the lambda keyword, followed by the parameter list (x), a colon, and the expression (x ** 2).

The lambda function is then assigned to the variable square_lambda.

```
result_lambda = square_lambda(5)
```

```
print(result_lambda)  # Output: 25
```

The lambda function is called similarly to the regular function, by providing an argument (5), and the result is assigned to result_lambda and printed.

To sum it up, learning about Python functions is like discovering a superpower in coding. Functions help us organize our code better, making it easier to understand and reuse. By figuring out how to use functions, we gain the ability to write efficient and neat programs. This knowledge is like a key that opens the door to more advanced Python adventures.