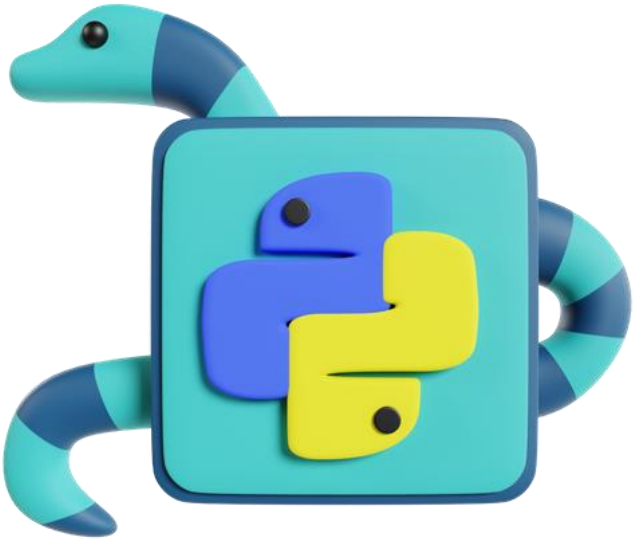
# Python



**FAST / DEEP / SIMPLE**

Edition 1.0

# About the author

Hello 👋

My name is Behnam Khani. You can call me Beh. I'm a software engineer with 10 years of experience in the industry. I have a passion for technology, education, and software development and enjoy combining the three.

This book and my website dejavucode.com are places to share my knowledge and experiences!

📧 **admin@dejavucode.com**

# Copyright Notice and Terms



First published: September 2023

# Table of Contents

# Section 1

# *Important facts about Python*

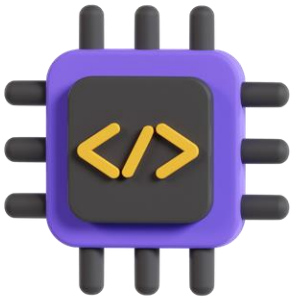
Before we dive into the Python language, let me answer commonly asked questions about Python and programming languages. A proper understanding of what Python is can save us a lot of time and money.

## What is a programming language?

To tell a digital device like a computer what to do, we need programming languages. Suppose that we have some data and need to ask a computer to process the data and prepare a report for us.

At the present time, we cannot use human languages like English to express our wishes to the computers.

Instead, we must use a programming language like Python that computers know about it.



By using a programming language, we're able to write programs that instruct a computer on how, for example, our data must be processed and what kind of reports must be generated.

## What Python offers us?

Python is a simple programming language that anyone can learn at any age. But at the same time, Python is a multi-purpose programming language that has a wide range of usages including:

- Web Development

- Desktop Development
- Data Analysis
- Data Visualization
- Machine Learning and AI (artificial intelligence)
- Game Development
- Task automation and Everyday Tasks
- Much more

In addition, Python is recommended if you're looking for an easy language to learn first. It is quite simple to understand for someone who's new to programming.

## What makes Python powerful?

One of the reasons Python is so powerful and vast is that there are so many libraries and frameworks for it. A programming library is a collection of prewritten codes that save us from needing to reinvent the wheel.

For example, in Data Analysis programs, we display the results with numbers, graphs and charts as data visualization expresses the results better than textual numbers. To visualize the results, we have two options:

1- **Reinvent the wheel:** In this case, we have to write everything from scratch. It means we have to write thousands of lines of code to visualize the data. So, we have to spend a lot of time and money on it.
2- **Use libraries:** Or we can simply use existing libraries written by other companies or individuals. In this case, they have written the ins and outs of drawing charts and graphs. All we need to do is to simply use the

library in our program with a few lines of code without any worries about bugs and errors!

*Therefore, we usually need to use Python along with libraries to write our programs in the most optimal way possible*.

Also, frameworks are similar to libraries in some ways. A framework is a set of codes usually written by other companies or individuals that can be used to define the structure and skeleton of our programs.

*So, both the libraries and frameworks are prewritten codes developed by other developers or companies that we use to save our time and money.*

## The most popular Python libraries and frameworks

There are lots of libraries and frameworks out there. But the most developers and companies usually use:

- Python + Django to develop secure and maintainable websites
- Python + Tkinter or PyQt to develop Graphical User Interfaces or GUI applications
- Python + Pandas for data science
- Python + Matplotlib for data visualization
- Python + NumPy for AI and machine learning
- Python + Pygame to create video games
- Python + Requests or CSV or APScheduler or Selenium and many other libraries for task automation

## Best book |course | bootcamp to learn everything

Even though there are lots of libraries and frameworks for Python, but you don't need to learn all of them!

In other words, it doesn't make sense for one person to be a web developer, a software developer, a game developer, an AI developer or a data science developer at the same time! Instead, we should become a professional in one or two of these.

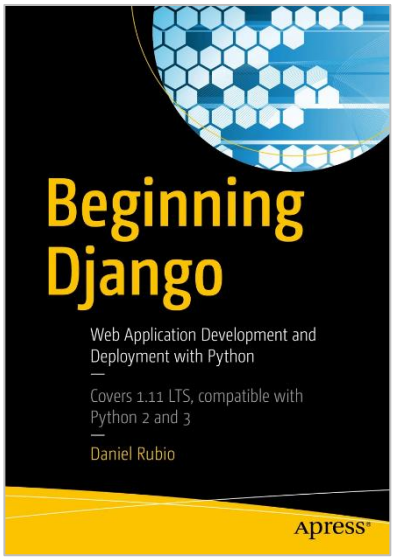
So, there is no book or course or bootcamp that covers everything about Python. Instead, anyone who wants to enter in this field, must do two things:

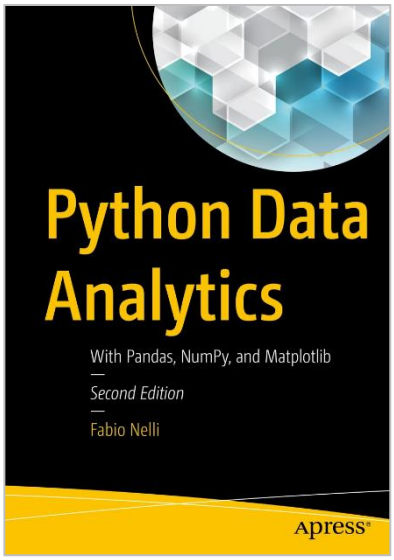
1. Learns the Python programming languages
2. Learns libraries and frameworks that are related to his/her choice

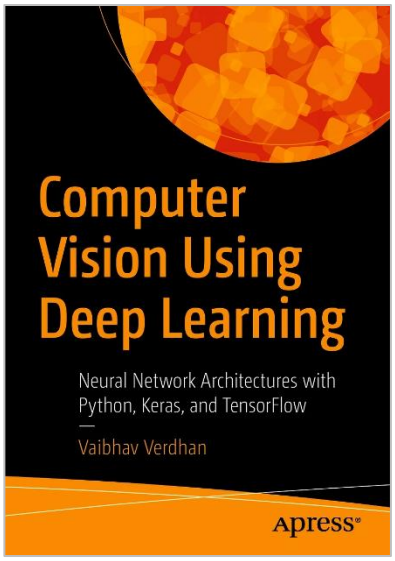
*For example, if you want to be a data scientist developer, you must learn the Python language plus libraries like NumPy*.

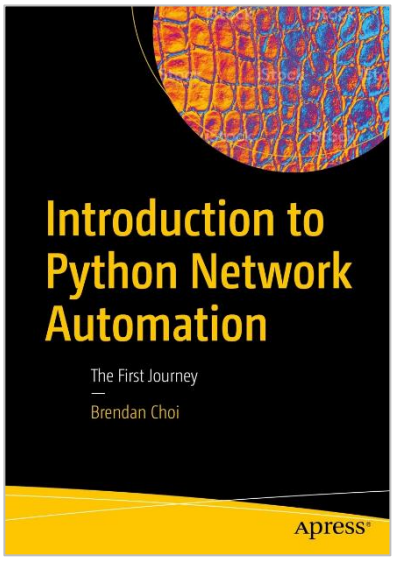
## What this book offers to you?

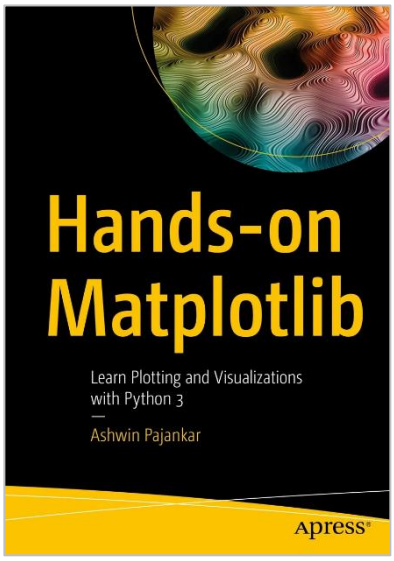
So, in this book, you will get a strong understanding about the basics of Python programming language. After learning the basic concepts about Python, you can choose your desired field and based on your decision, choose another book or course which specifically teaches you a certain library or framework. *That is why we have lots of resources that each of which has addressed a specific field in Python*. Here is a list of books published just by Apress about Python and the libraries and frameworks that work with it:

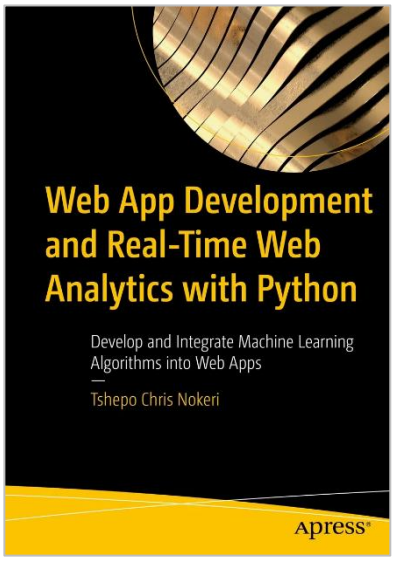

Beginning Django
Web Application Development and Deployment with Python
Covers 1.11 LTS, compatible with Python 2 and 3
Daniel Rubio
Apress

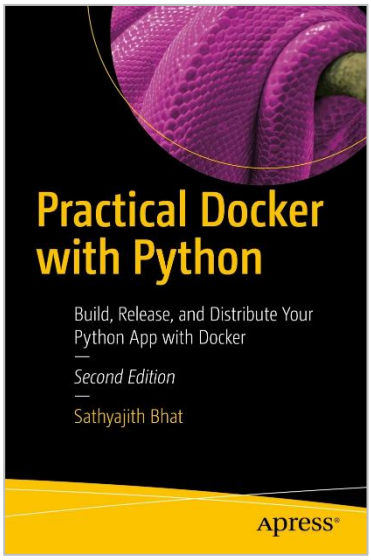

Python Data Analytics
With Pandas, NumPy, and Matplotlib
Second Edition
Fabio Nelli
Apress

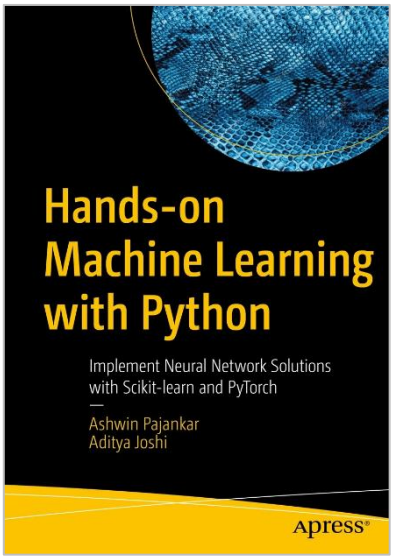

Computer Vision Using Deep Learning
Neural Network Architectures with Python, Keras, and TensorFlow
Vaibhav Verdhan
Apress

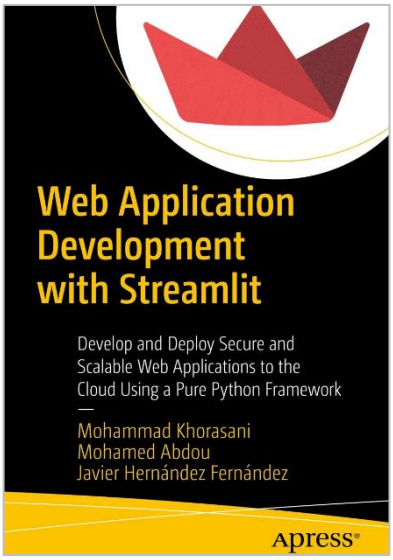

Introduction to Python Network Automation
The First Journey
Brendan Choi
Apress

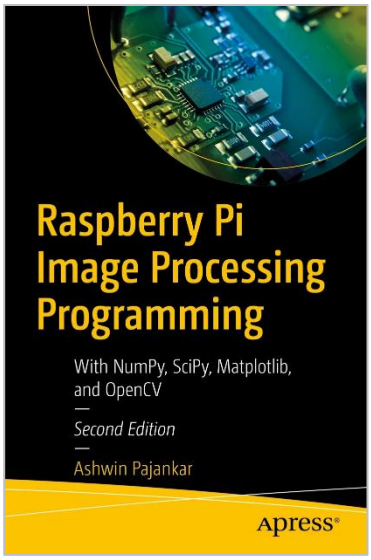

Hands-on Matplotlib
Learn Plotting and Visualizations with Python 3
Ashwin Pajankar
Apress

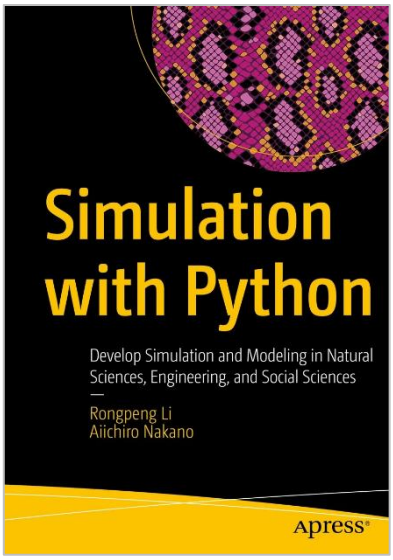

Web App Development and Real-Time Web Analytics with Python
Develop and Integrate Machine Learning Algorithms into Web Apps
Tshepo Chris Nokeri
Apress

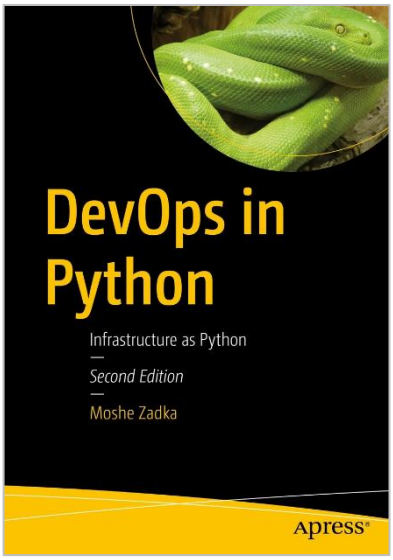

Practical Docker with Python
Build, Release, and Distribute Your Python App with Docker
Second Edition
Sathyajith Bhat
Apress


Hands-on Machine Learning with Python
Implement Neural Network Solutions with Scikit-learn and PyTorch
Ashwin Pajankar
Aditya Joshi
Apress


Web Application Development with Streamlit
Develop and Deploy Secure and Scalable Web Applications to the Cloud Using a Pure Python Framework
Mohammad Khorasani
Mohamed Abdou
Javier Hernández Fernández
Apress




## How to become a professional Python developer?

As a conclusion, the best way to become a professional Python developer is to first learn and understand the foundation concepts about the Python. After that, focus on your favorite field (libraries and frameworks). Don't rush and just focus on learning the Python programming language properly. This is what we're going to do in this book.

# Section 2

# *Software requirements*

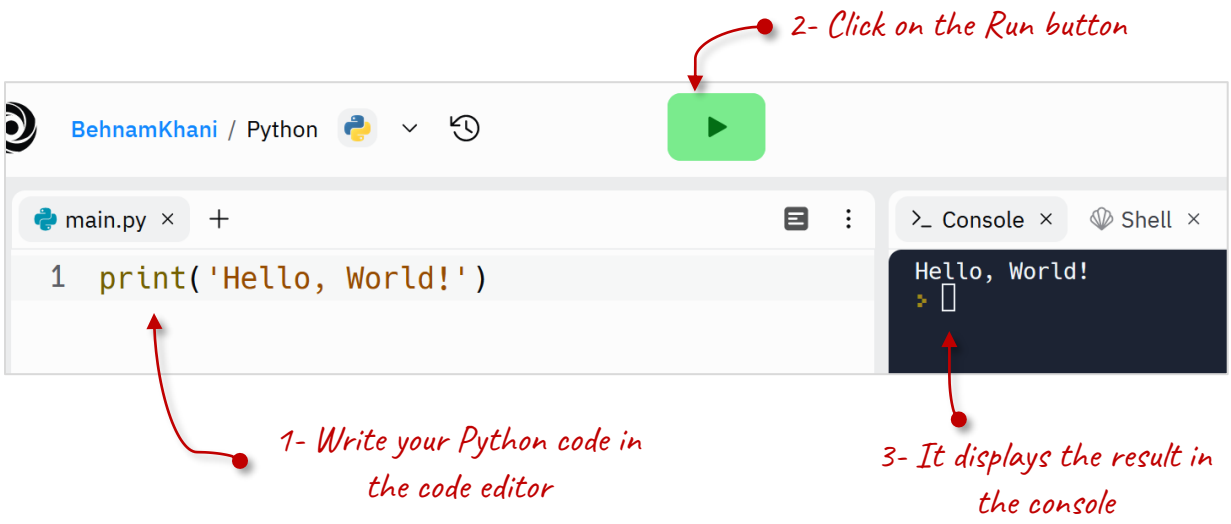
## Does it matter what operating system I use?

Fortunately, we can use Windows, Linux or Mac. However, it is possible to use Android or iOS devices to write Python programs, but I don't recommend it.

## What computer program do people use to write code?

The easiest option for a beginner is to use an online Python IDE (Integrated Development Environment). An IDE is a program that integrates several tools for software development. These tools help us to write, test and debug (find bugs) the programs. And as we want to write Python programs, so we need to have an IDE for this purpose.

There are lots of online IDEs and code editors for Python development like replit. Using these online tools keeps you away from installing tools. All you need to do is to enter your Python code and click on the Run button:




There are other online IDEs and code editors like Online Python.

## PyCharm

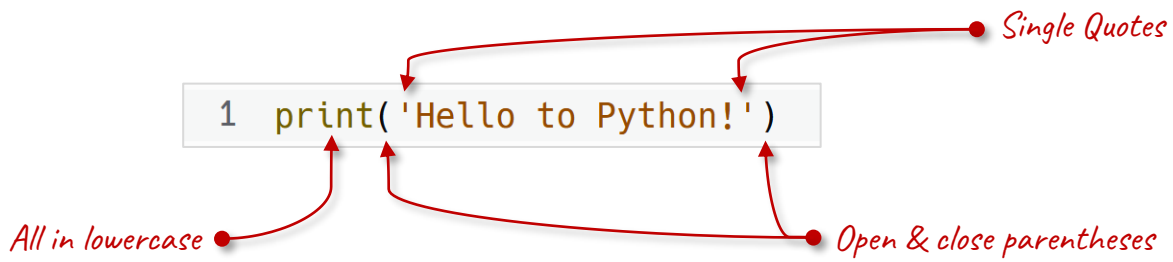
If you need to have a mature and well-configured IDE for Python development, then PyCharm will be your choice. PyCharm is a free code editor that is available on your favorite platform - Windows, macOS, and Linux.

# Section 3

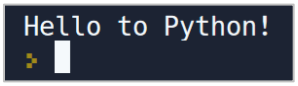
# *Write your first Python program*

## Print your texts in the console

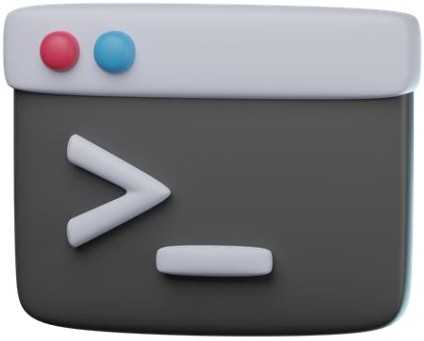
Let's write your first Python program. We want to print a text in the console, like Hello to Python!. To do so, type **print()** all in lowercase, open and close parentheses **()**, and inside these parentheses add two double quotes **""** or two single quotes **''** and then type whatever you want to print:




Now run this program. It prints Hello to Python! in the console:



A console is kind of computer terminal where we can view our program's output in text format. For simplicity's sake, we use console so that we can focus on the core concepts of Python.



However, after learning Python, you can learn libraries like Tkinter to create GUI (Graphical User Interface) programs. Programs that use widgets like Buttons or Menus or List Boxes instead of simple texts.

So, we can print or display a text in the console by using a function named **print()**.

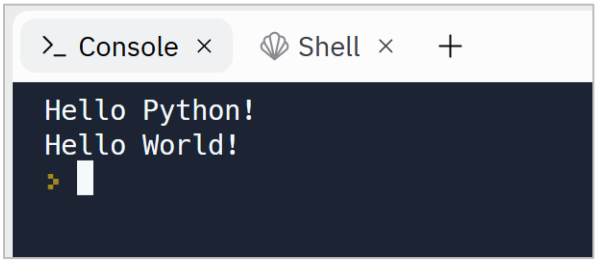
There are many functions in Python that each of which has its own use. I'll talk more about it later. But for now, all you need to know is that Python has its own libraries named Standard Library. The Python Standard Libraries contains libraries that each library contains lots of functions like the **print()**.

In Python, a function is a block of code written by you or other developers that performs a specific task. Functions provide better modularity for our programs.

So, we have a **print()** function that takes a text and prints it in the console. In programming terms, we call text a string. Strings in Python are contained within a pair of single quotes '' or double quotes "". A string can contain any characters including numbers, white space and special character like @:


We can use functions like print() as many times as we need:


The output of this program is:




As you can see, Python executes the code line by line from top to bottom.

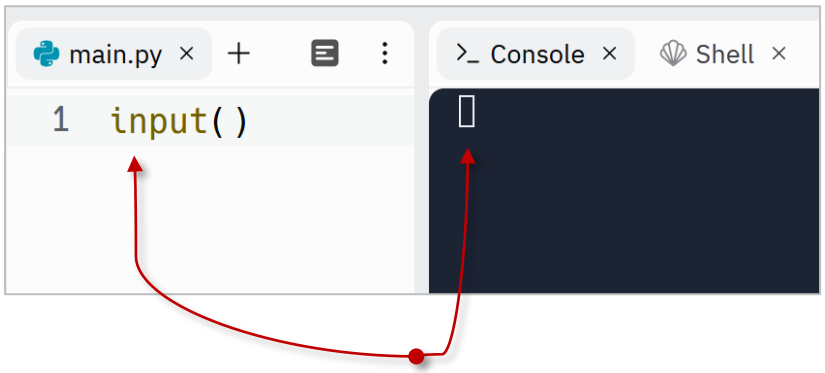
## Get input from our user

In the previous program, you learned about the **print()** function. We can use it to print text messages in the console. But there are times that we need to get values from the user. For example, we need the user enters his name and then print a message based on this name.

To get user input, we need to use a function named **input()**. The input function is a part of Python Standard Library. Let's write a Python program that gets a value from us. Type in **input**, open and close parentheses **()** and run the code:

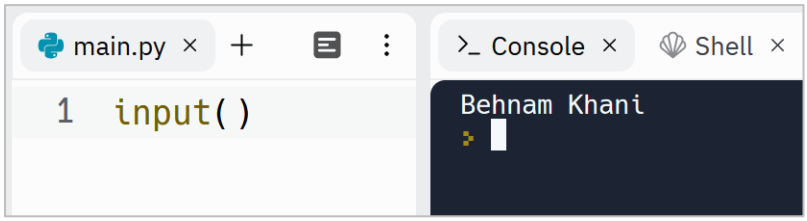
```
1 input()
```

Now when we run this code, the **input()** function waits for the user to type some text. It tells the Python Interpreter to stop and wait for the user to enter text with the keyboard:

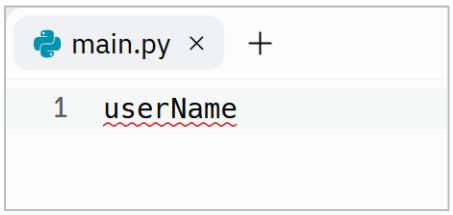



*The input() function waits for us to enter a value*

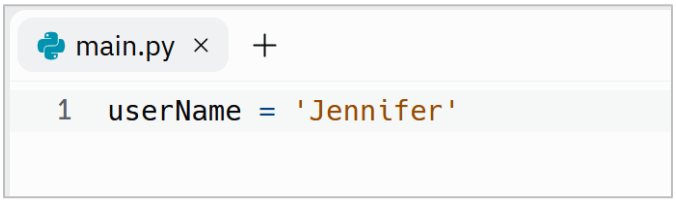
So, type a value in the console, for example enter your name and press Enter:




The result of the **input()** function is the string that we type. But we must save the result of this **input()** function somewhere for later use. To do so, we need to use a variable. We use variables in our programs to remember information. Consider a variable like a container that can holds one value and has a label on it. So, a variable has one value and a name. Whenever we need to assign a value to a variable or get its value, we only need to call its name.
Let's define a variable named **userName**. We can use multiple words to define a variable name, but without spaces:



After defining a variable, we use the = (assignment operator) to hold a value inside it:




Now, the value **'Jennifer'** is stored in the **userName** variable.
We can also hold the result of the **input()** function into the **userName** variable like this:

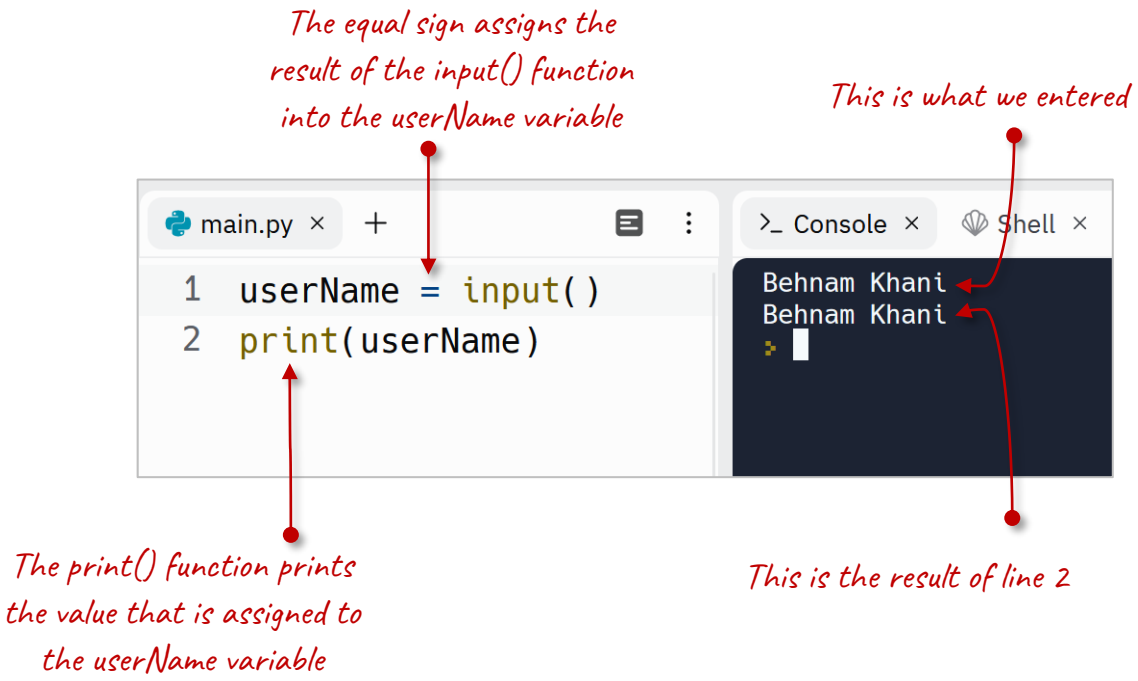
```
1  userName = input()
```

Now when we run the program and enter a value and press the Enter key, Python will assign the entered value into the **userName** variable.

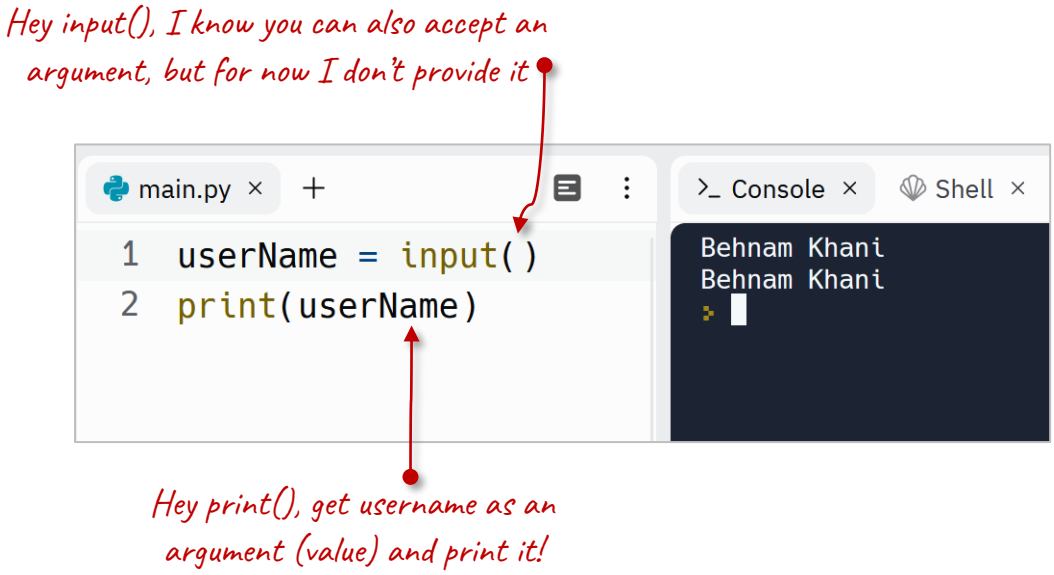
Then we can print the content of this variable by using the **print()** function:

```
1  userName = input()
2  print(userName)
```

When we run it, the **print()** function prints the content of the **userName** variable in the console:



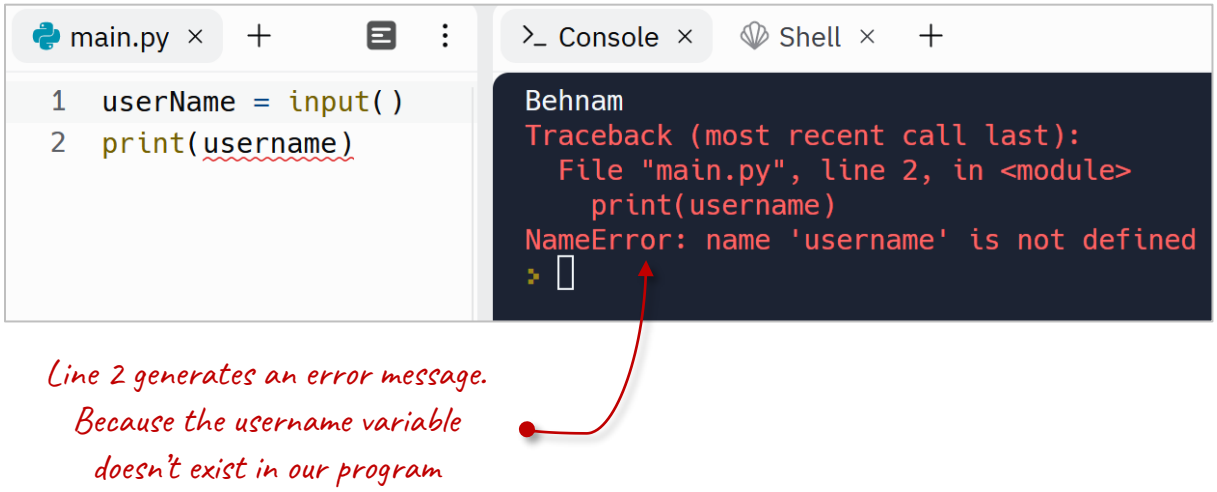

As you know, the **print()** function needs a string to print. In programming terms, we call it an argument. An argument is a way for us to provide more information to a function. We put arguments between the parentheses of a function. Sometimes a function can work without any argument, like the **input()**. However, the input function can get a string as an argument that I'll talk about it later:



In other words, the **print()** function needs a string to display in the console. Sometimes we provide this argument by using single quotes and sometimes we get a string from a function like **input()** and then pass the result of the **input()** function to the **print()** function.

## Python is Case-Sensitive

Python is a Case-Sensitive programming language. It means Python treats uppercase and lowercase letters differently. So, it differentiates the uppercase and lowercase variables:




As you see, it underlines the **username** with red wavy line. It is kind of error message for us. This error message says we don't have a variable named **username** (all in lowercase letters)

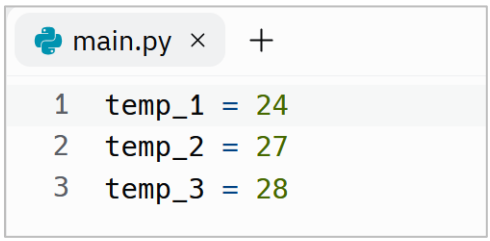
## Important notes to define a variable

Keep these things in mind when choosing a variable name:

- A variable name is arbitrary, for example we can choose t as a variable name that is going to hold a temperature, but variable names should be descriptive and meaningful. Because, when we see the variable name later in the code, rather than having to remember what it contained from the lines before, the name should reveal the purpose of that variable
  Also, when other developers read your code, a meaningful name helps

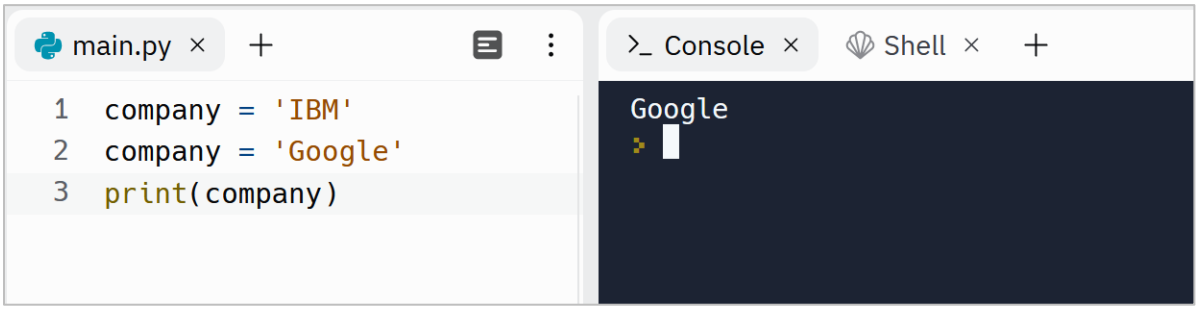
them to guess the purpose of that variable and understand the code in a faster rate.

- Also, if we want to choose multiple words for a variable, we cannot use white spaces. Instead, we can use one of these two conventions:
    - Camel Case: In this convention the first letter of each word is capitalized, except for the first word, like userName.
    - Snake Case: in this convention, all letters of each word is in lowercase and each word is separated by an underscore, like user_name. Snake case is more commonly used in Python.
- And a variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _ ) and starts with a letter or the underscore character. We usually use numbers in a variable names to define variables that are related to each other. For example, if we need to assign three temperatures to three variables, we can define variables like this:

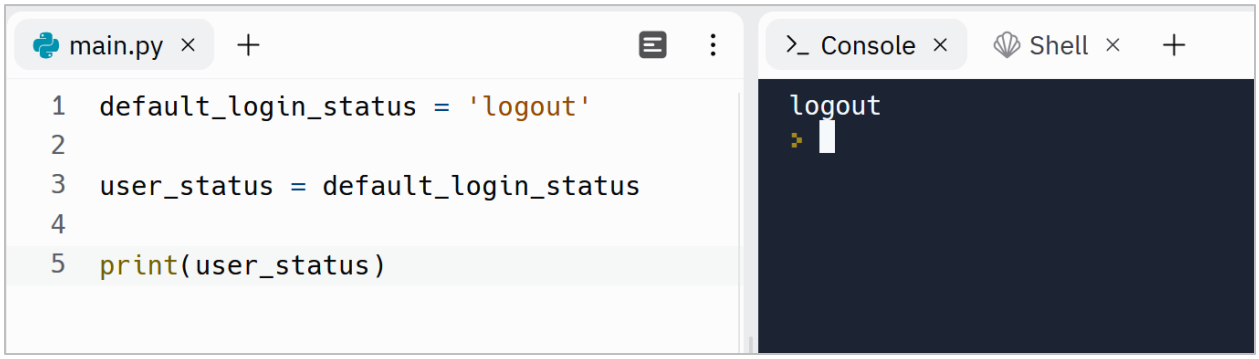



## Updating variables

A variable can hold one value at a time. Sometimes and during the execution of the program, we need to update the value of a variable. To do so, we can use the = (assignment operator):



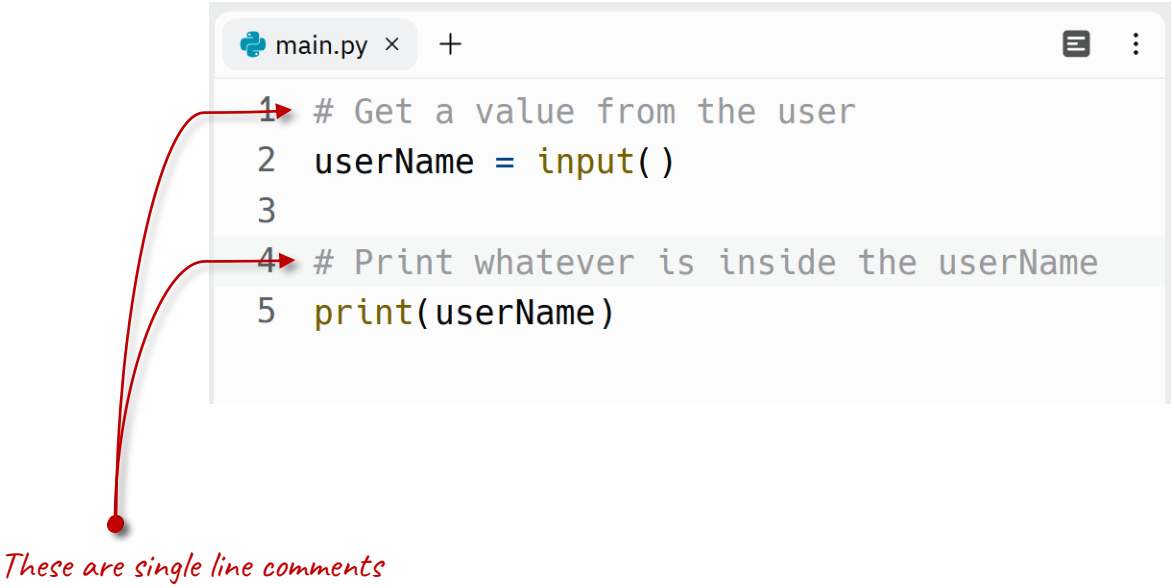
So, we can simply update a variable by reassigning a new value to it. This new value can be a value from another variable. For example, we can assign the value of the **default_login_status** variable to the **user_status** variable:




## Comment our program

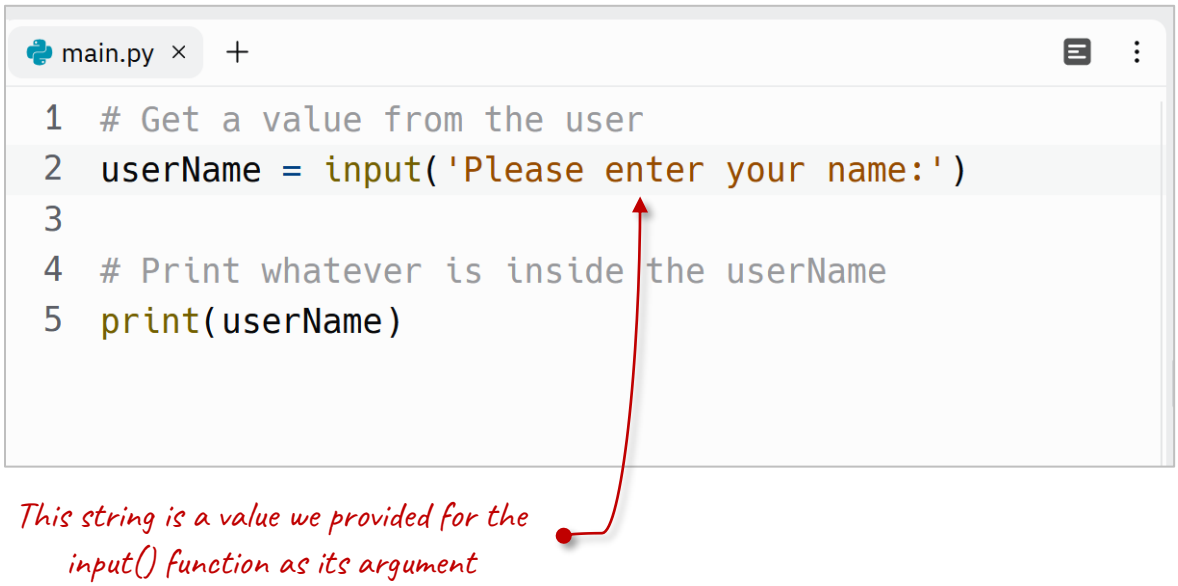
A comment is a text note or description that provides some information about our codes. In other words, by using comments we can note how the code works.

Comments are not executable. In other words, the Python interpreter doesn't care about them and won't execute them. We use a hash character # to comment a single line. For example, we can comment about each line of the previous program like this:



## A more meaningful input() function

As said earlier, we can also provide an argument for the **input()** function. By doing so, our user knows what he should enter. We change the code like this:

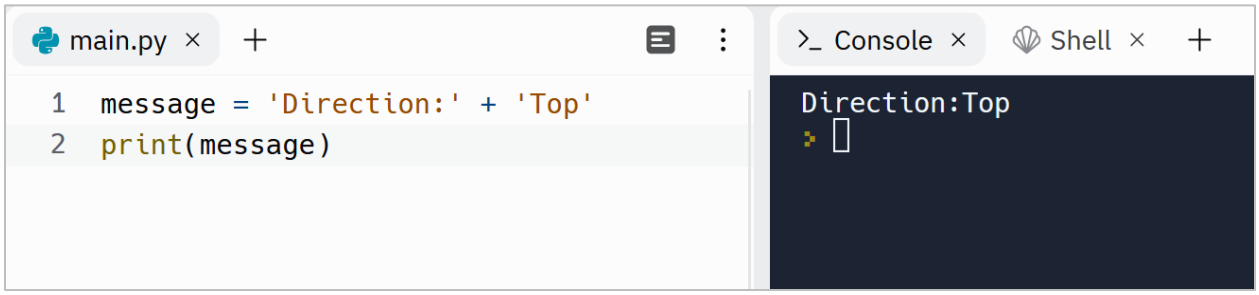



When we run the program, it will display a message and asks us to enter our name:


Now our program is user friendly.

## Expressions in Python

Sometimes we need to combine multiple values to create a new value. We call it an expression. For example, we can concatenate two strings to create a new string. In this example, we concatenated **'Direction:'** with **'Top'** and created a new string **'Direction:Top'**.

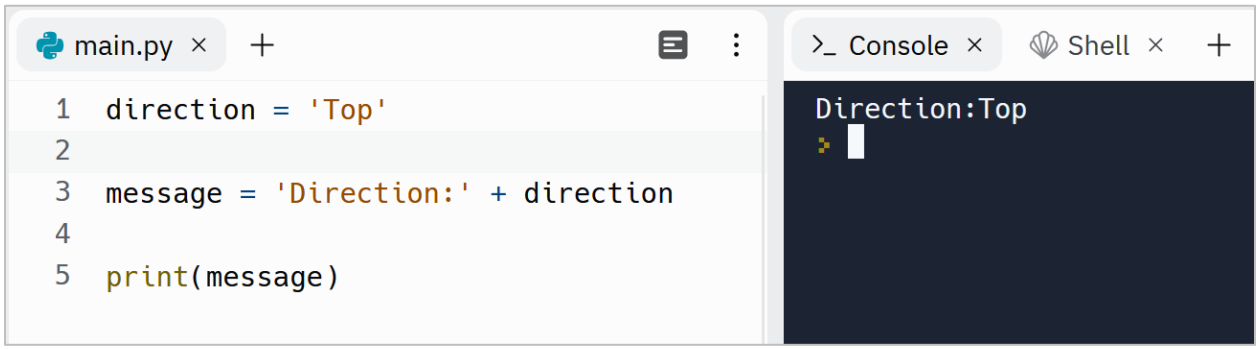



So, this:

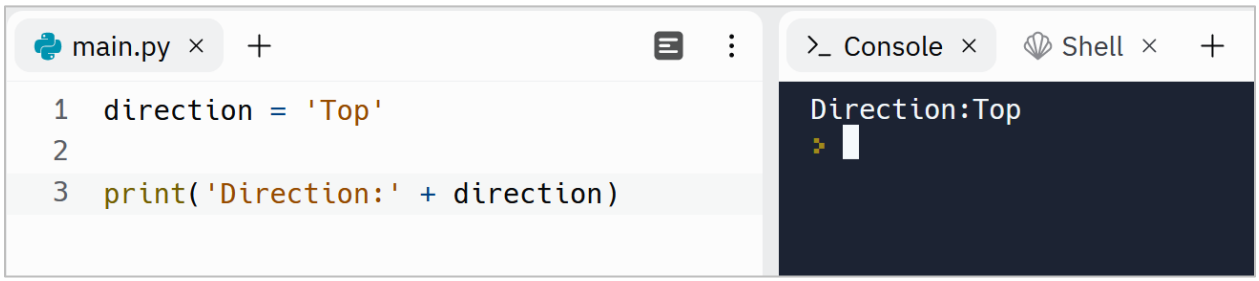
```
'Direction:' + 'Top'
```

is an expression. In this expression, we have used the + (plus) operator to concatenate the two strings.

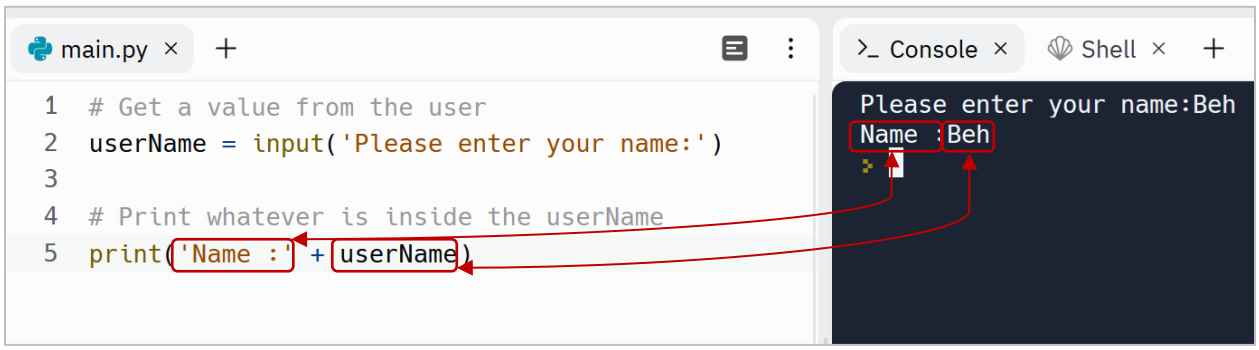
In an expression, we can also use variables. In this example, we have an expression that concatenates the string **'Direction'** with the value that is inside the variable **direction**:




We can even directly pass the expression to the **print()** function:



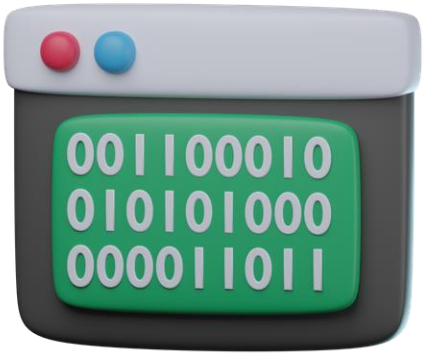

Now let's use an expression to print the **userName** in a more user-friendly manner. To do so, change the code like this:



So, when I enter my name and press the Enter key, it prints the expression inside the parentheses.

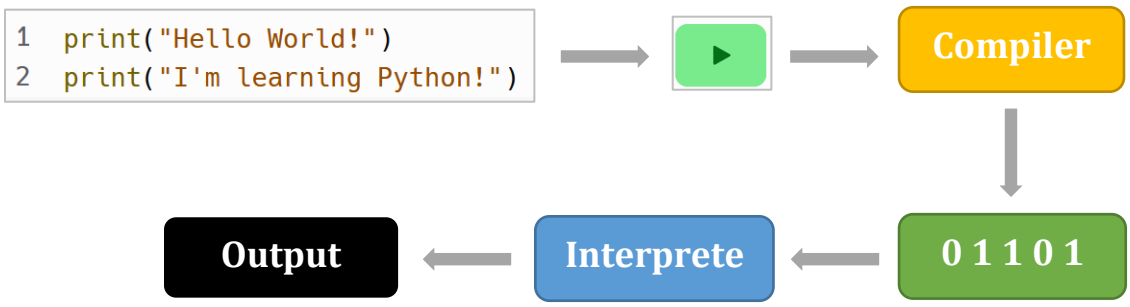
## How Python executes our codes?

So, what happens when we press the Run button? By clicking on the Run button, our code is giving to a compiler. The Python compiler reads the code and converts it into machine code that is called binary. Because a computer (CPU) only understands a binary language.




In fact, the entire code will be converted into zeros and ones. This is what a CPU can understand and execute. In other words, a compiler is like Google Translate for us. It converts the code from Python to Binary.

*When our code is compiled, the computer (CPU) can read, understand and execute it*. This execution process is done by using Python interpreter. Finally, this Interpreter executes the binary codes line by line:



# Python Variable Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

Python Variable Exercises – Dejavu Code Part 1

Python Variable Exercises – Dejavu Code Part 2

Python Variable Exercises – Dejavu Code Part 3

Python Variable Exercises – Dejavu Code Part 4

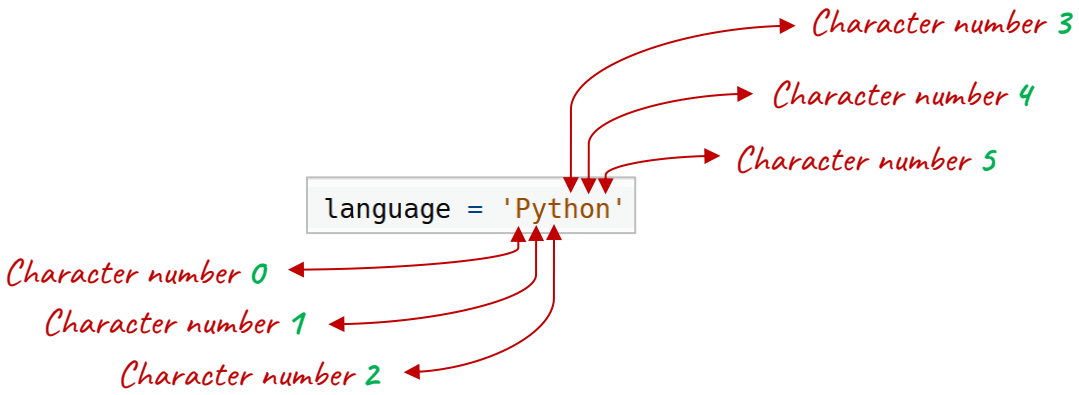
# Section 4

# *Data Types in Python*

In computer programming, a data type determines a set of possible values that a variable can have. Also, a data type determines a set of allowed operations on a variable.

## String data type

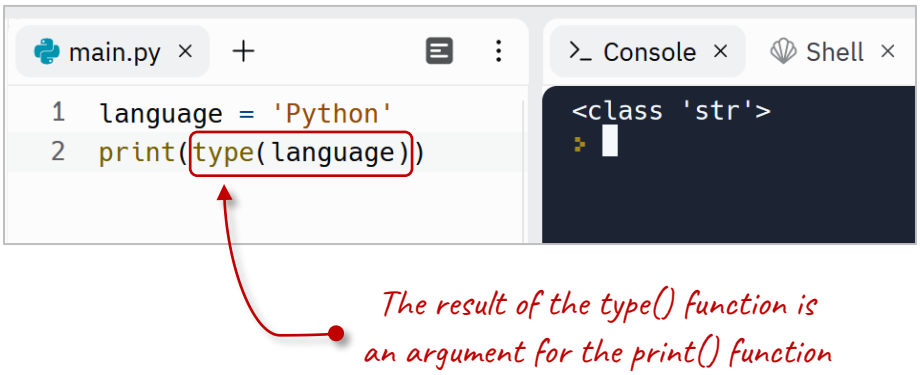
Until now, we've worked with string data type. A string is a sequence of characters enclosed in single quotes or double quotes. In the following example, we have a variable of type string that is named **language** and its value is **Python**:




So, the value of the **language** variable is **Python** that is consisted of 6 characters. As you see, in programming world, we start counting from zero. So, a string like **Python** has 6 characters, but it starts counting from 0.

## type() function

In python, we have a function named **type()**. This function returns the type of a variable. When we run the following code, it will print **<class 'str'>** in the console. str represents string:

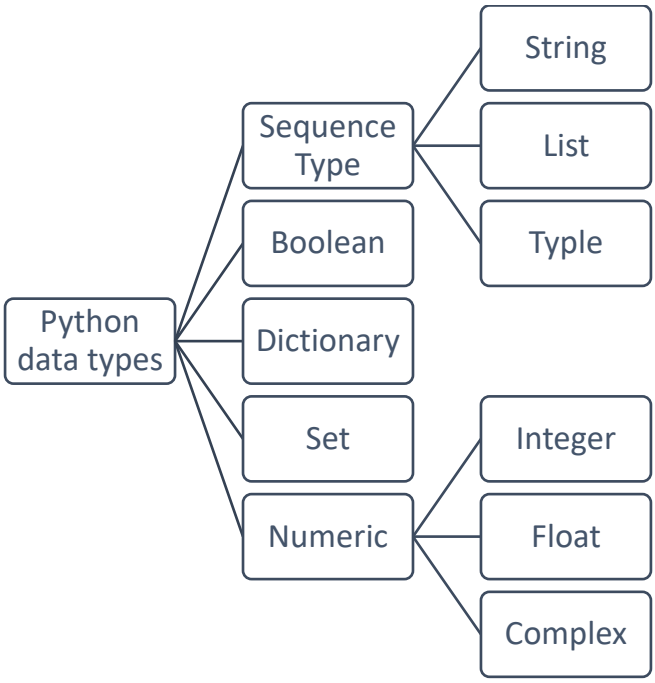



When the Python interpreter wants to execute the line 2, it first executes the **type()** function. As you see, the **type()** function accepts one argument. We can

provide its argument by passing a variable name or even a value. Then the **type()** function returns the type of the **language** variable. And after that, the interpreter executes the **print()** function and prints what the **type()** has returned.

In other words, the result of the **type()** function is an argument for the **print()** function.

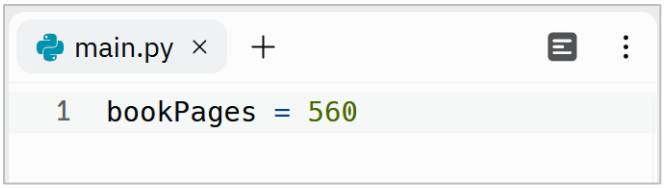
Let's back to the main topic of this section; Data Types. In Python there are 5 main data types including Sequences, Boolean, Dictionary, Set and Numeric:




In this section, I'm going to talk about string, numeric and Boolean.

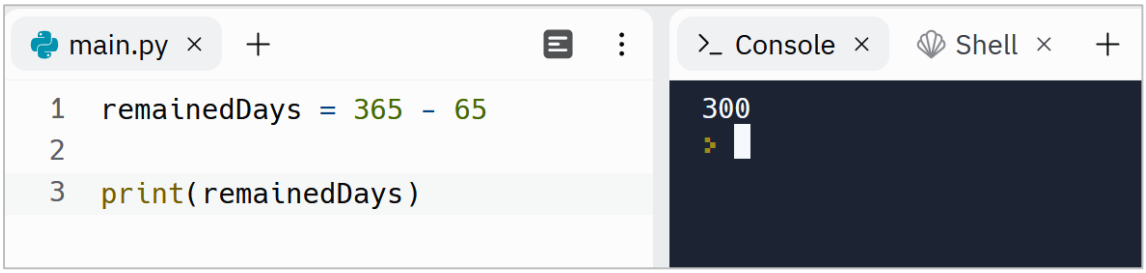
## Integer numbers

We continue with numbers. In real world apps, we have to deal with numbers too. To define a variable that holds a number, we can simply assign our number without using single quotes or double quotes:

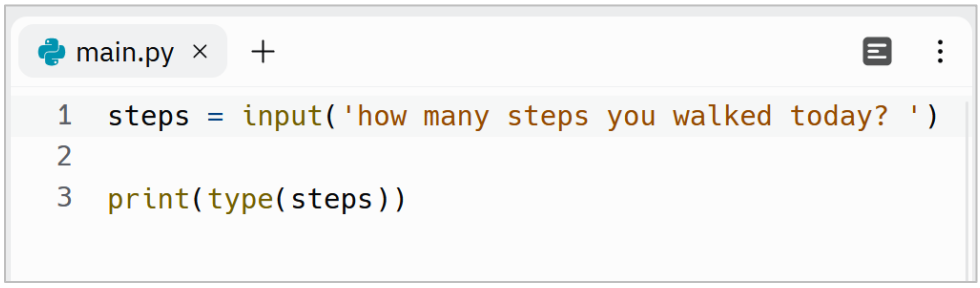



Numbers like 560 that have no floating point are called integer numbers.

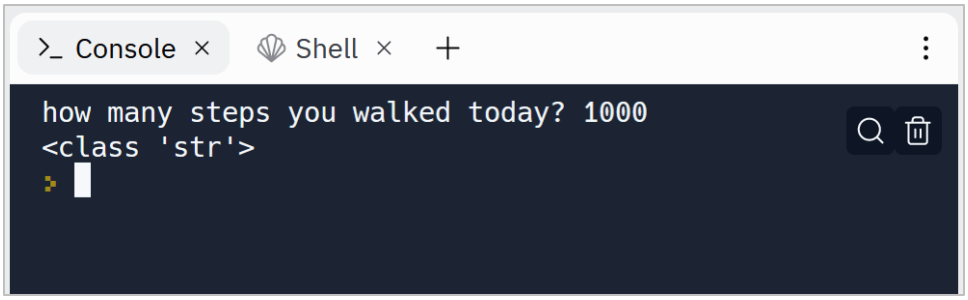
We can also have expressions with numbers. In this example, we have an expression that calculates the remaining days of an account. We can use the – (minus) operator to subtract numbers:




There is a subtle point here. When we use the **input()** function to get a value and enter a number, the returned value is always of type string even when we enter a number:




Let's enter a number. The type of **steps** is string:



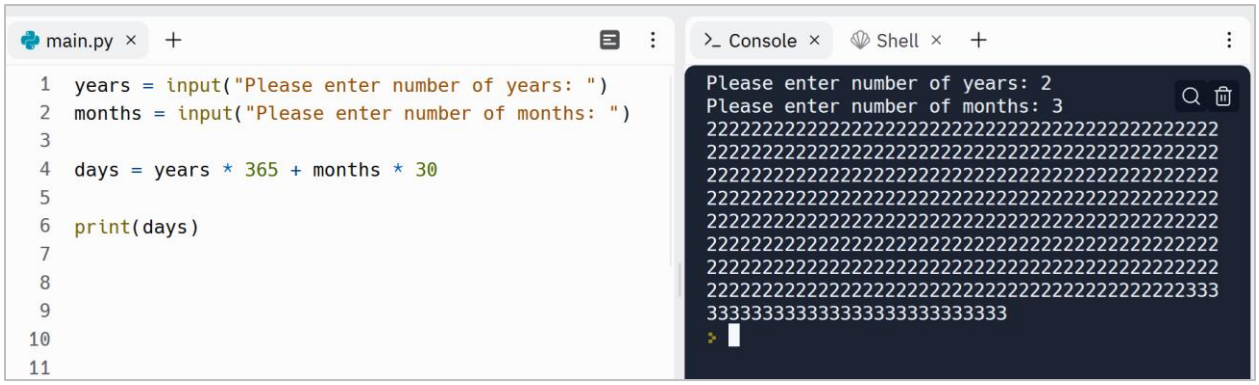

In other words, the input function returns **'1000'** and not 1000.

Let's take another example to explain how to works. Suppose that we want to take two numbers from our user, a number that represents years and another one represents months. Then we need to convert them into days:

*We use triple quotes for multiline comments*

```
"""
Task:
We need to get two numbers from the user
as years and months and convert them to
days.
suppose that each month is 30 days.

Output:
Please enter number of years: 2
Please enter number of months: 3

2 years and 3 months are equal with 820
days
"""
```

To do this task, we need to take two values from the user and do some math operations on the numbers and print the final result:




At lines 1 and 2 we get two values from our user

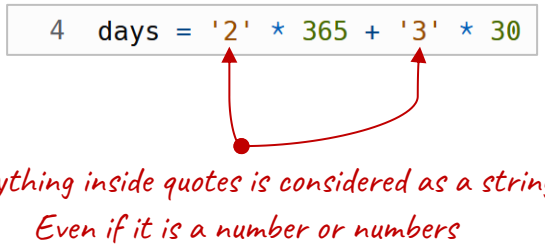
At line 4 we convert years to days and months to days and assign it to the days variable

At line 6 we print the result

But the result is *not* what we expected!

## Type Casting

The previous program prints 2 for 365 times and prints 3 for 30 times! The reason is that *the input function returns a string*. So, if we enter 2 as years and 3 as months, line 4 will be executed like this:




In fact, it displays the string "2" for 365 times and the string "3" for 30 times:

```
Please enter number of years: 2
Please enter number of months: 3
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222222
22222222222222222222222222222222222222222333
333333333333333333333333333
```

That is why we need to convert the string "2" into a number and the string "3" into a number. In fact, if we need to use years and months in mathematical operations, we must convert each of them to a numeric data type. *To convert a string type to an integer numeric type*, we must *use the* **int()** *function*. The **int()** function gets a value of type string and returns a value of type int as the result. In programming terms, we call this operation Type Casting. So, let's rewrite line 4 like this:

```
1  years = input("Please enter number of years: ")
2  months = input("Please enter number of months: ")
3
4  days = int(years) * 365 + int(months) * 3
5
6  print(days)
```

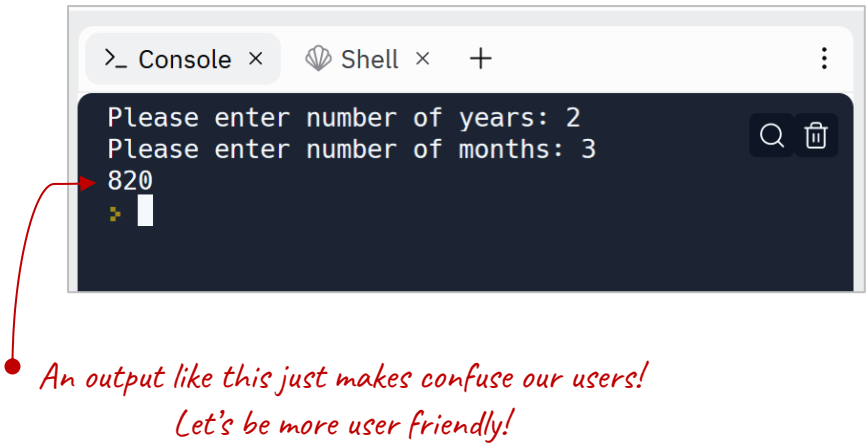
*The int() function accepts a String and converts it to a Number*

So now we can consider line 4 like this:

```
4  days = 2 * 365 + 3 * 30
```

It first multiplies 2 by 365, then multiplies 3 by 30 and then adds them together.

Run the program again, and this time it works correctly:



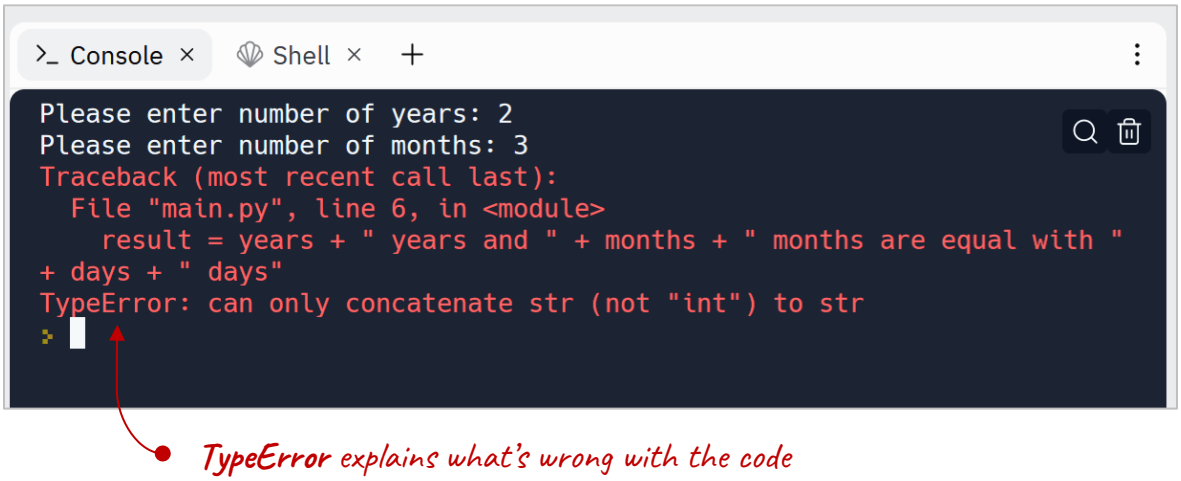

But our task is not completed yet, because the final result is not printed in a meaningful way. As said earlier, we need a message like this:

```
Output:
Please enter number of years: 2
Please enter number of months: 3

2 years and 3 months are equal with 820 days
```

To do so, we can define another variable called **result** and create a meaningful message by concatenating these numbers with some strings:

```
main.py
1  years = input("Please enter number of years: ")
2  months = input("Please enter number of months: ")
3
4  days = int(years) * 365 + int(months) * 30
5
6  result = years + " years and " + months + " months are equal with " + days + " days"
7
8  print(result)
```

But when we run this program and enter years and months, we'll get an error message:




The red lines are an error message. It tells us that hey, Python can't concatenate a *string* to a *number*. In this case, the type of **days** is numeric. If we use the **type()** function to get the type of the **days** variable:

```
print(type(days))
```

we'll get a message like this:

```
<class 'int'>
```

int represents integer. Integer is a number without any floating point. For example, these numbers are integer:

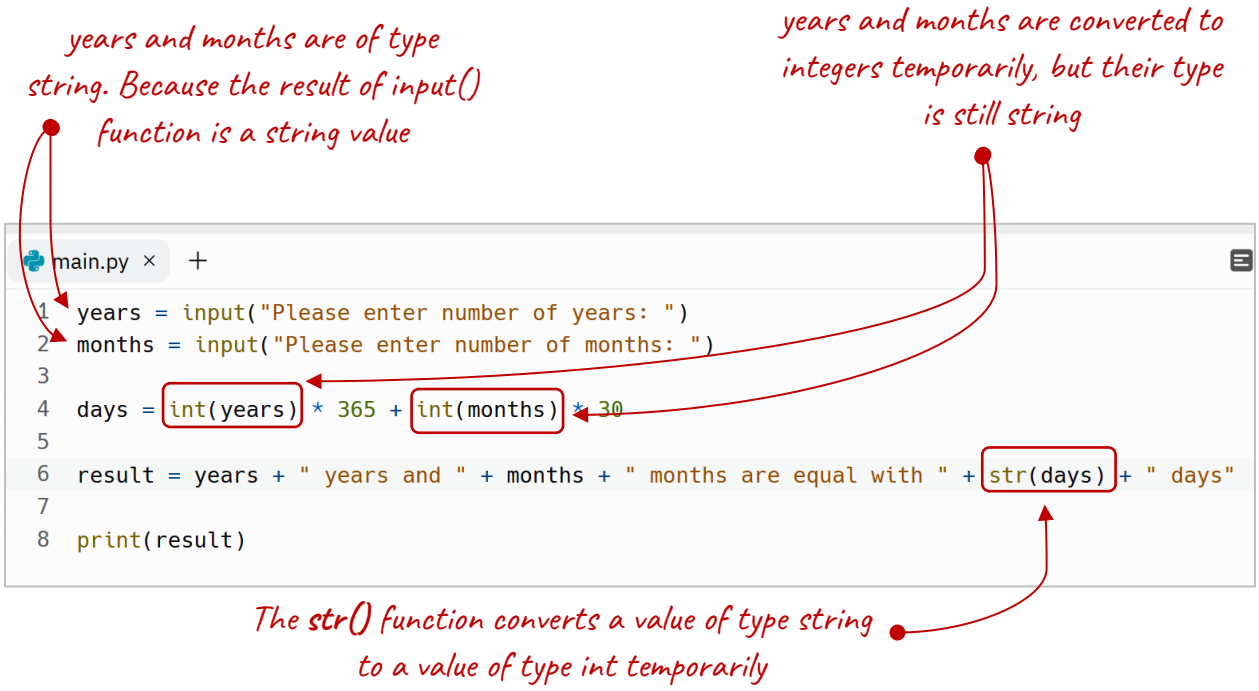
- 100
- -50
- 0

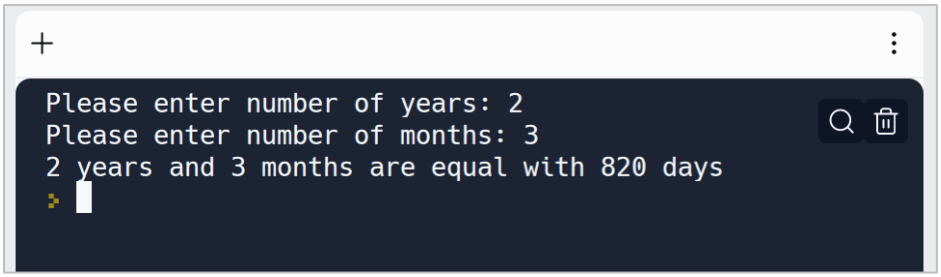
As we saw already, the error message is like this:

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

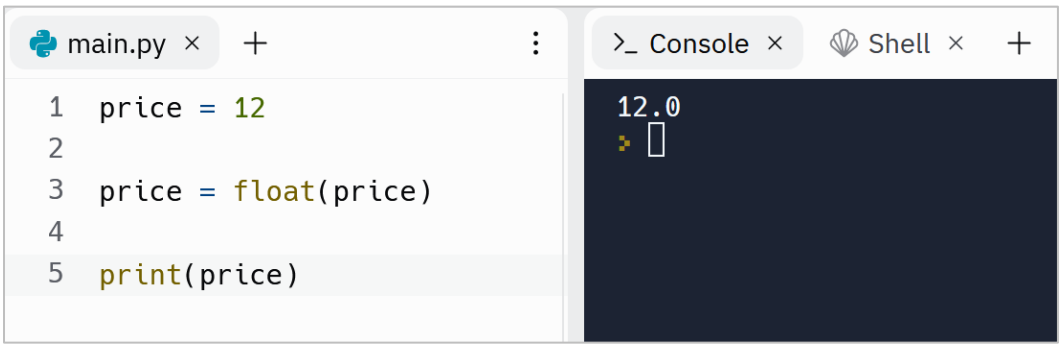
It says: can only concatenate str (not “int”) to str. Python is trying to tell us that it can’t concatenate a string to an integer.

The solution is to convert integers to strings by using the **str()** function. The **str()** function takes a number as an argument and converts it to a string:

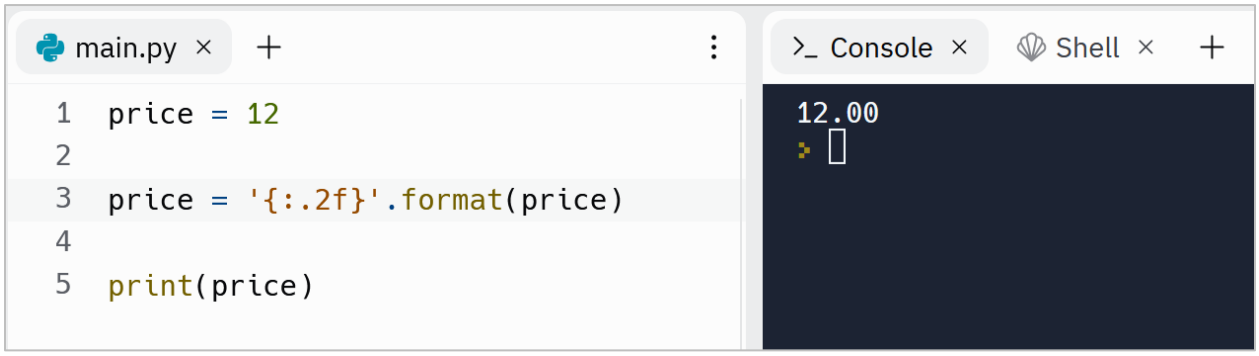





We can also cast an integer into float by using the float() function:




If you need to have a complete control on the floating-point part, for example, you need to have 2 decimal places, then use the format() function:

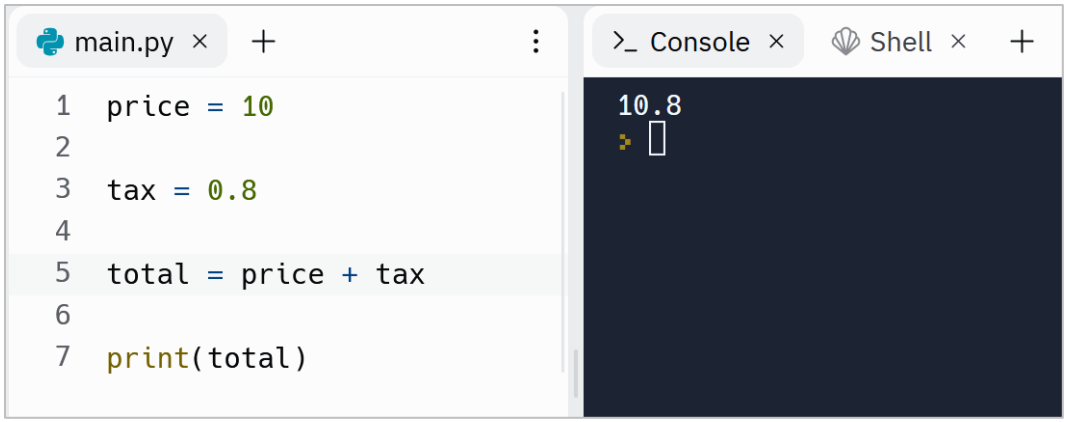


## Type conversion

Until now, you learned about a concept named Type Casting that allows us to use functions like int() or str() to cast a type.

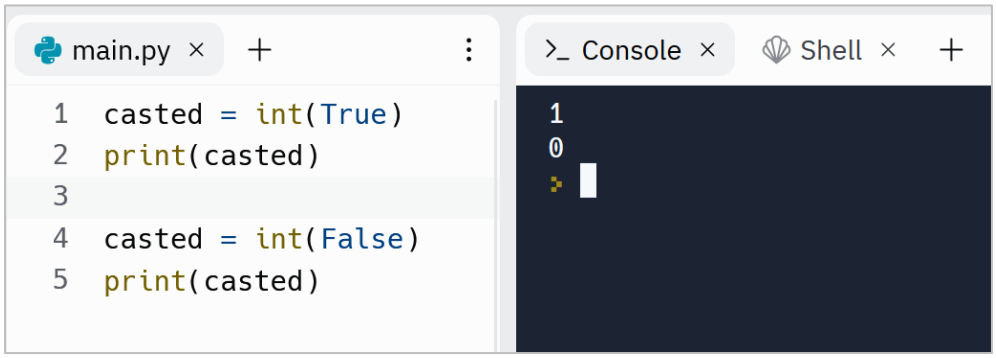
There is also another concept in Python named Type Conversion. In Type Conversion, a data type is converted into another data by the Python.

In this example, Python automatically converts the price data type from integer to float temporarily to do the addition operation:

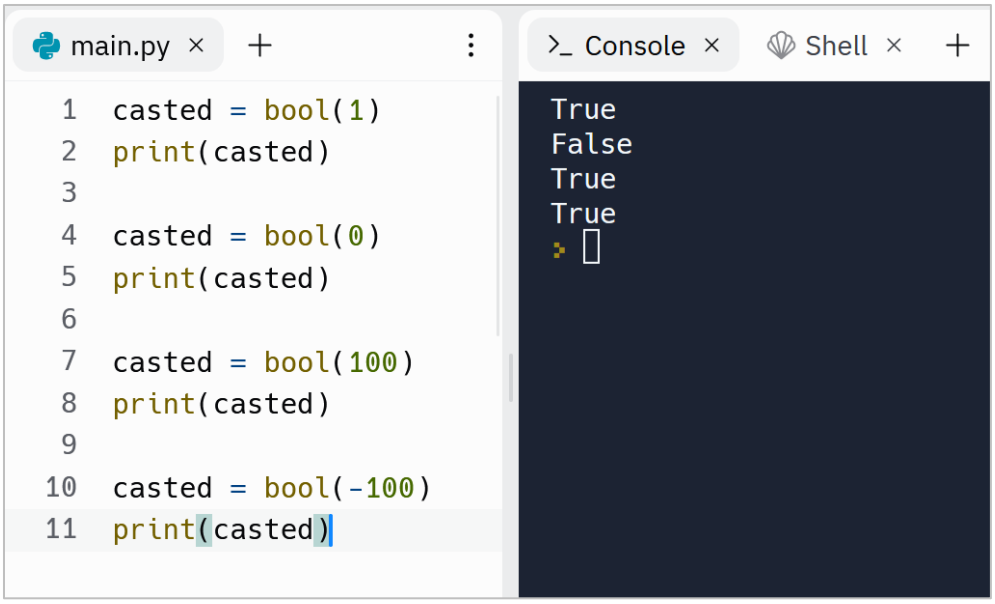



## Casting Boolean into integer

We can also cast a Boolean value into an integer value. This is useful mostly when we need to store Boolean values into database. When we cast a True value into integer, it returns 1 and when we cast a False value into an integer value, it returns 0:




By using the **bool()** function, we can convert an integer into Boolean. This function returns False for 0 and return True for any number except zero:




## Concatenating by using format() function

We can also perform string concatenation by using **format()** function. This function helps us to do string concatenation in a clean way:

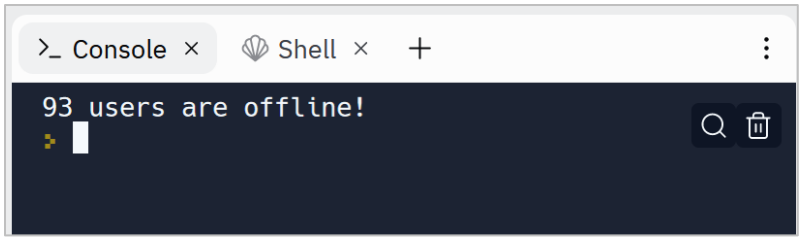

The **format()** function replaces the value of the **followers** variable with {} placeholder.

There is a shorter form of the **format()** function. In this form, we can simply put an **f** before the string and put the variable into the placeholder:

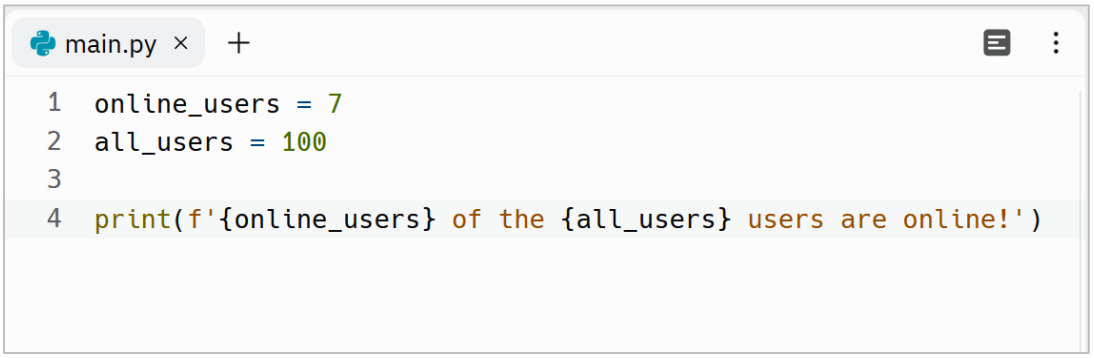

We can hold the result of the **format()** function in a variable and use it in the next lines:


We can even use an expression instead of a variable:


The output is:

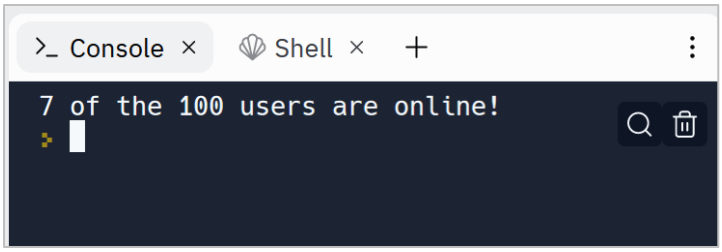



If we need to use multiple variables inside a string, then we can use multiple placeholders:




Also here is another form of the **format()** function for the example:


The output is:




Let's rewrite the year converter example by using the **format()** function:

```
main.py
1  years = input("Please enter number of years: ")
2  months = input("Please enter number of months: ")
3
4  days = int(years) * 365 + int(months) * 30
5
6  result = "{} years and {} months are equal with {} days".format(years, months, days)
7
8  print(result)
```

So all we need is to put placeholders by using **{}** and pass the variables to the **format()** function to replace with the placeholders.

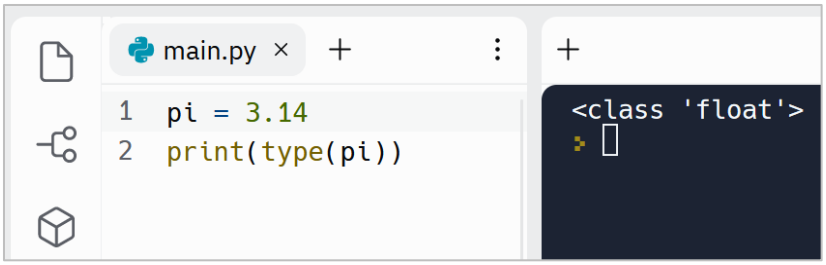
The **format()** function has these advantages over using the **+** (plus) operator:

- It is more readable
- It doesn't need any type conversion

## Float data type

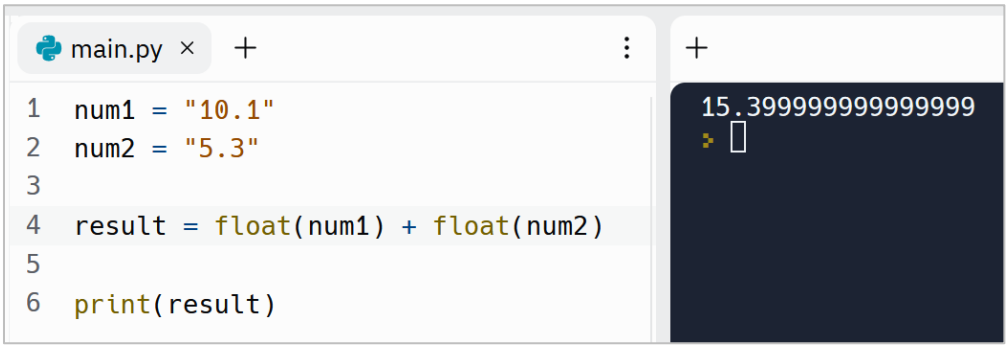
This data type is used to create variable that hold floating point numbers like:

- **3.14**
- **0.02**
- **100.00**




As you see, the **pi** variable is of type float.

Another point about floating point numbers is that, if a string contains a floating point number, we can use the **float()** function to convert the string into float:

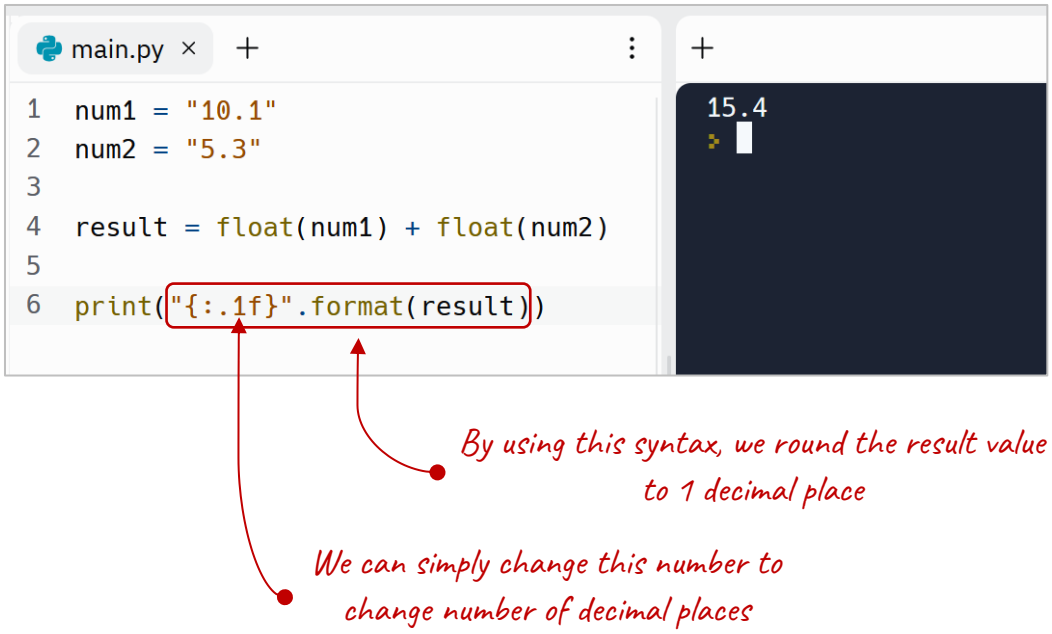



But we expect 15.4 instead of 15.399999999999999! What's going on? 🤔

## Rounding numbers

This is a common problem in computer programming. Because floating point values don't have an exact binary representation. That's why we may experience some loss of precision and unexpected results.

To solve this, we need to use the **format()** function like this:




This function replaces the result with the placeholder {}. But before replacing, it formats the number by using the format we've defined inside the curly braces. In this case, we've defined that format our number in a way that displays only 1 decimal place.

So the **format()** function has two usages for us:

- Concatenating strings
- Formatting numbers

## Boolean data type

Until now, you learned about string and integer data types. Now let's talk about Boolean data type. There is another data type in Python named Boolean. A variable of type Boolean can only accept two possible values, True or False without quotes. We use this data type to hold two state situations like:

```
main.py
1  eligible_to_vote = False
2
3  is_load_completed = True
```

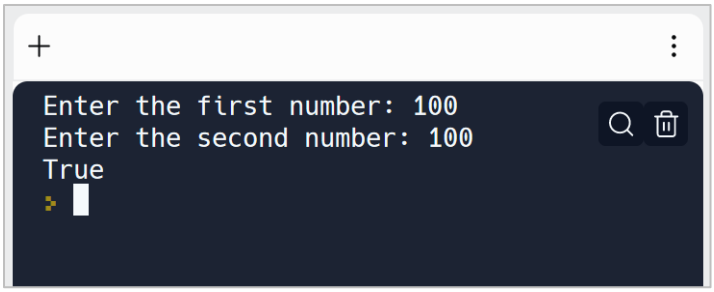
As another example, suppose that we're going to take two numbers from our user and check if they are equal or not. We check this equality by using the equal to **==** (equal to) operator. So first of all, let's get two numbers by using the **input()** function:

```
1  num1 = input("Enter the first number: ")
2  num2 = input("Enter the second number: ")
```

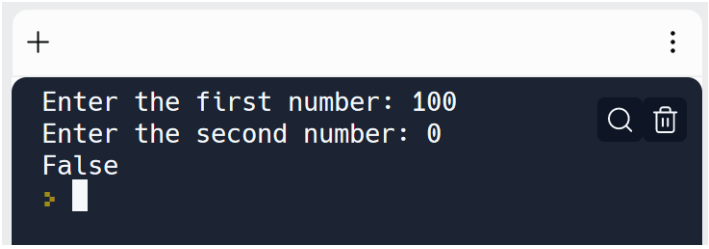
Now, let's check if the two variables are equal and then print the result. Note that we need to convert **num1** and **num2** to integer and then compare them by using the **==** operator:

```
main.py
1  num1 = input("Enter the first number: ")
2  num2 = input("Enter the second number: ")
3
4  areEqual = int(num1) == int(num2)
5
6  print(areEqual)
```

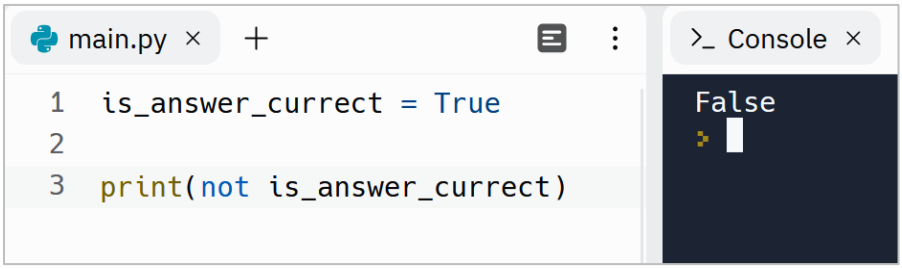
When we run this code, and enter two numbers that are equal, we will get True:




And when we enter two different numbers, then we will get False:




We can reverse a Boolean value by using the not operator. In other words, not makes True to False and makes False to True:




In the next sections, you'll see how the == (equal to) operator and other comparison operators help us to run our code based on conditions.

# Python Data Types Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

Python Data Types Exercises – Dejavu Code Part 1

Python Data Types Exercises – Dejavu Code Part 2

Python Data Types Exercises – Dejavu Code Part 3

Python Data Types Exercises – Dejavu Code Part 4

# Section 5

# *Operators in Python*

In mathematics and computer programming, an operator is a symbol or character that performs a specific operation. The important operators in Python are:

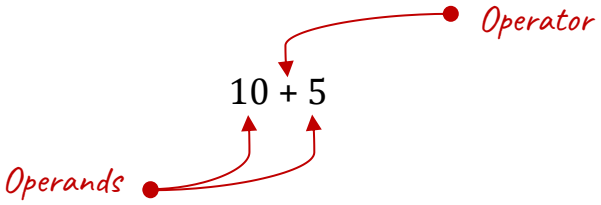
- Arithmetic operators
- Assignment operators
- Comparison operators
- Boolean (Logical) operators

## Arithmetic operators

Sometimes we need to perform arithmetic calculations in our programs. To do so, we use these commonly used Arithmetic operators:

| Operator | Description | Example |
|---|---|---|
| + | Addition | 10 + 5 = 15 |
| - | Subtraction | 10 – 5 = 5 |
| * | Multiplication | 10 * 5 = 50 |
| / | Division | 10 / 5 = 2.0 |
| % | Modulus. This operator returns the remainder. | 10 % 5 = 0 |
| ** | Exponentiation | 2 ** 3 = 8 |

Each Arithmetic operator needs two operands. The value(s) that an operator operates on is called the operand(s):




Let's see the results in practice:

```
1  addition = 10 + 5
2  subtraction = 10 - 5
3  multiplication = 10 * 5
4  division = 10 / 5
5  modulus = 10 % 5
6  exponentiation = 2 ** 3
7
8  print("Addition: " + str(addition))
9  print("Subtraction: " + str(subtraction))
10 print("Multiplication: " + str(multiplication))
11 print("Division: " + str(division))
12 print("Modulus: " + str(modulus))
13 print("Exponentiation: " + str(exponentiation))
```

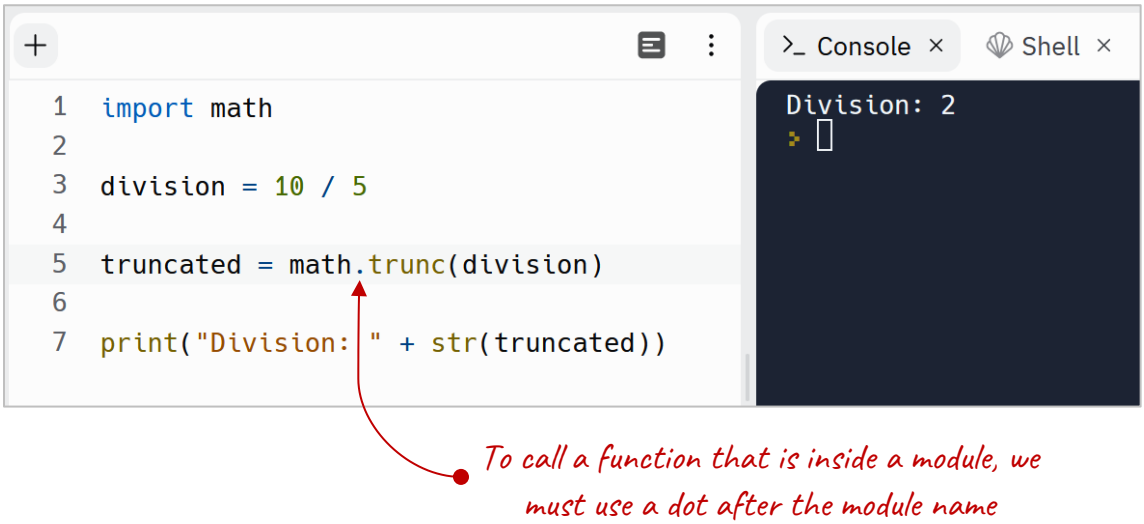
```
Addition: 15
Subtraction: 5
Multiplication: 50
Division: 2.0
Modulus: 0
Exponentiation: 8
```

The return value of the division operator is of type float

There are two points about the code:

1- The return value of the Arithmetic operators is of type numeric. That is why we used the **str()** function to print the result of each calculation.
2- The return value of the division operator is of type float. However, we may don't need to display the floating point.

To remove the floating point, we can simply use a function named **trunc()** that is a part of math class. And as the math module is not accessible in our code, we must import it first (at line 1):




Let's explain what's happening in each line:

At line 1 we import the math module into our program

In other words, the **trunc()** function is located inside a module called math that by default this module is not available in our program. So we need first import the math module into our Python program, then we can access the **trunc()** function.

At line 3 we divide 10 by 5 and assign the result into a variable called **division**

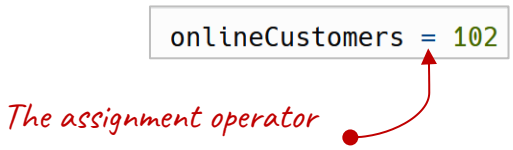
At line 5 we use the **trunc()** function that is inside the math module to truncate the integer part of the division variable.

At line 7 we convert the **truncated** variable into a string, so that we can concatenate it with the string **"Division: "**

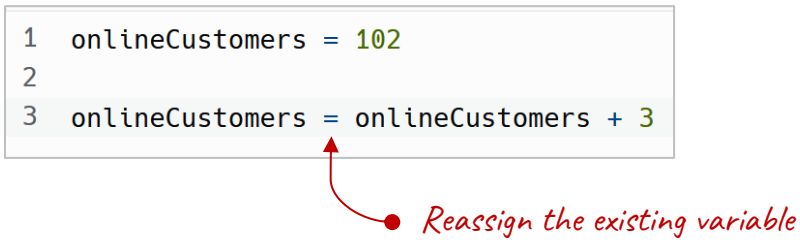
Now you may ask what are modules? In Python, a module is like a library that contains functions. For example, the math module provides functions that contains mathematics functions. As you see at line 1, to access a module's functions, we need to import it into our program. After that, we can access the functions that are inside that module. *To call a function that is inside a module, we must use a dot after the module name*.

## Assignment operators

You already know how the = (assignment operator) works. It assigns a value into a variable for later use:




However, there're times that we need to reassign a variable with a new value:




Python offers the Assignment operators. So, we can simply add 3 to the **onlineCustomers** by using **+=** operator without any repetition:

```
1  onlineCustomers = 102
2
3  onlineCustomers += 3
```

Let's take a look at common Assignment operators in Python:

| Operator | Example | Same As |
|---|---|---|
| = | x = 10 | x = 10 |
| += | x += 2 | x = x + 2 |
| -= | x -= 2 | x = x – 2 |
| *= | x *= 2 | x = x * 2 |
| /= | x /= 2 | x = x / 2 |
| %= | x %= 2 | x = x % 2 |

As you see, each Assignment operator like the Arithmetic operator needs two operands to work on.

## Comparison operators

Comparison operators compare two values (operands) and evaluate down to a Boolean result, True or False. You already seen the == (Equal to) operator:

```
1  myShoeSize = 43
2
3  shoeSize = 43
4
5  isShoeFit = myShoeSize == shoeSize
```

*The equal to == operator evaluates down to a Boolean result. In this case, the result is True*

In Python, there are other Comparison Operators:

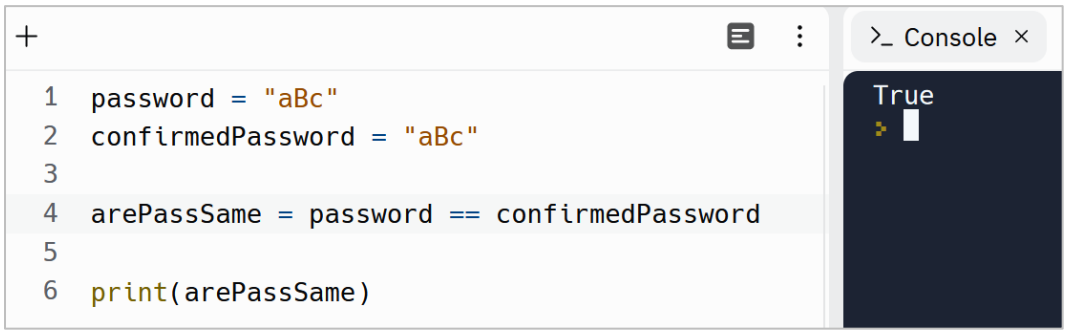
| Operator | Meaning | Example | Result |
|---|---|---|---|
| == | Equal to | 10 == 10 | True |
| != | Not equal to | 10 != 10 | False |
| < | Less than | 10 < 10 | False |
| > | Greater than | 10 > 10 | False |

| | | | |
|---|---|---|---|
| **<=** | Less than or equal to | 10 <= 10 | True |
| **>=** | Greater than or equal to | 10 >= 10 | True |

The **!=** (not equal to) evaluates to False when the operands (values on both sides) are the same.

Another point about the comparison operators is that the operands can be Numbers or Strings or Boolean values. You already see how the comparison operators work with Numbers. Let's see how they work with Strings.

When the operands are of type String, the comparison operators compare the operands (Strings) letter by letter:




The result of **arePassSame** is True because the two operands are the same. In other words:

```
1  password = "aBc"
2  confirmedPassword = "aBc"
```

'a' is equal to 'a'

and 'B' is equal to 'B'

and 'c' is equal to 'c'

So, when all the characters are the same, the final result is True. Otherwise, the result is False. For example, in this case, as ‘c’ is not equal with ‘d’, the result is False:


Even the following program will print False, because ‘B’ is not equal with ‘b’ as Python is a *case sensitive* programming language:


So, the Comparison Operators compares Strings letter by letter.

## Boolean (Logical) operators

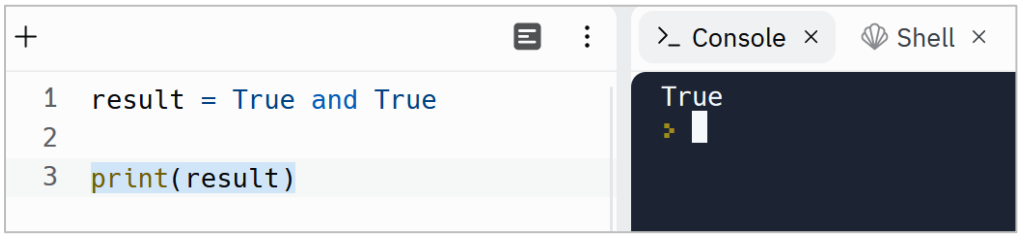
Sometimes we have Boolean values to compare. To do so, we usually use these two Boolean operators:

- and
- or

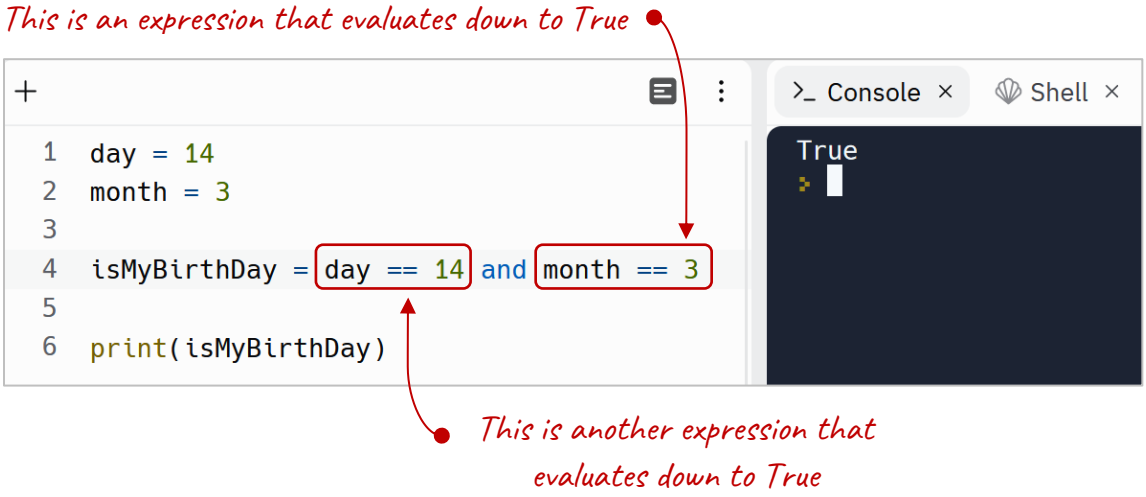
Boolean operators return True or False.

| Operator | Description |
|---|---|
| **and** | Evaluates down to True if both the operands are True |
| **or** | Evaluates down to True even if one of the operands is True |

The and operator takes two Boolean values or expressions and evaluates down to True if both the operands are True. Otherwise, it returns False:




Note that the Boolean operators can work with expressions too. *An expression is a combination of operands and operators*:

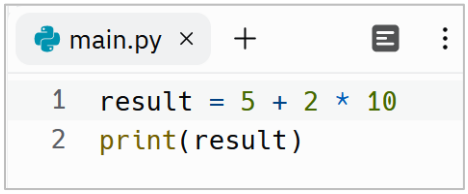



The or operator takes two Boolean values or expressions and evaluates down to False if both the operands are False. Otherwise, it returns True:

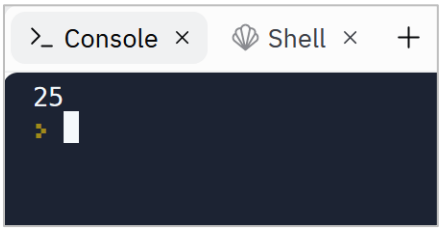

## Understanding the Operator Precedence

In simple expressions that have one operator and two operands, the order is not important. But when we have multiple operators and operands in one expression, then we must be aware about a concept called Operator Precedence. *This precedence simply defines the order of execution.*

To make it clear, I ask you to guess the output of this program:




The result is 25:




Why? Because Multiplication has a higher order than Addition. So, it first calculates 2 * 10 and then adds the result with 5.

Here is the order of operators in Python:

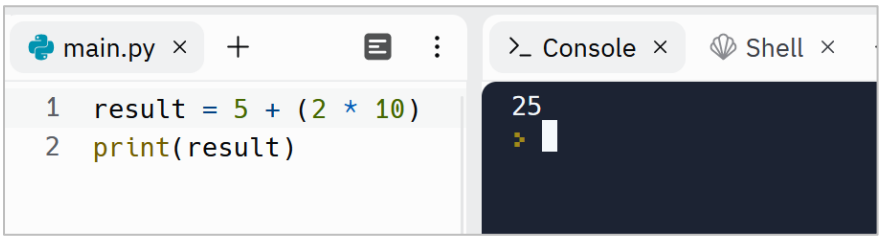
| Operator | Description |
|---|---|
| **()** | Parentheses |
| **** | Exponent |
| *, /, //, % | Multiplication, Division, Floor division, Modulus |
| +, – | Addition, Subtraction |
| ==, !=, >, >=, <, <= | Comparisons |
| **and** | Logical AND |
| **or** | Logical OR |

Note that the operators with the same precedence have left-to-right associativity. For example, in an expression like this:

5 + 2 – 3

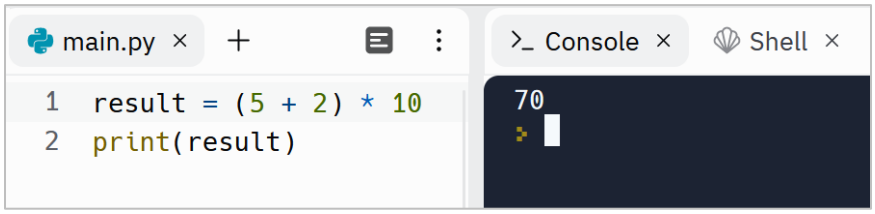
the Addition and Subtraction have the same precedence. So, it evaluates the expression from left to right.

Also, you may have noticed that Parentheses have the highest precedence. We can use this feature of parentheses to simplify expressions:




So now we, as humans, have a better understanding on how line 1 will be executed.

Also, we can use parentheses to change the default precedence:


main.py
Console
Shell
1 result = (5 + 2) * 10
2 print(result)
70

# Python Operators Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

Python Operators Exercises – Dejavu Code Part 1

Python Operators Exercises – Dejavu Code Part 2

Python Operators Exercises – Dejavu Code Part 3

Python Operators Exercises – Dejavu Code Part 4

# Section 6

# *Important built-in functions in Python*

So far, we've worked with functions like **input()**, **print()**, **format()**, **int()** and **str()**. But there are lots of them in Python. Let's talk about some other important functions in Python.

## Important functions to work with Strings

As Strings are used widely in Python, so I'm going to introduce you more functions that work with Strings.

## len() function

**len()** is a function that works with Strings. Suppose that we need to check if a password is strong enough or not. And one important feature of a strong password is that the length of it is equal or greater than 8 characters. We can get the length of a

String by using the **len()** function. In this program, we get the length of a password and check if it is strong or not:

```
main.py
1  password = input('Please enter a password: ')
2  passwordLength = len(password)
3
4  isPasswordStrong = passwordLength >= 8
5
6  print('Is password strong: {}'.format(isPasswordStrong))
```

At line 1 we get a password from our user

At line 2 we get the length of the password

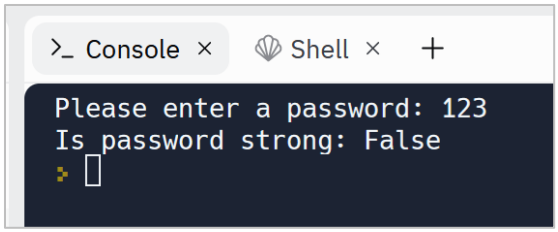
At line 4 we check if the length of the password is greater than or equal to 8

At line 6 we print the output

The output of this program is as follow:

```
Console  Shell
Please enter a password: Dejavu1234
Is password strong: True
```

And another running with a weak password has this output:




## lower() and upper() functions

The other useful functions that work with Strings are **lower()** and **upper()** that helps us to convert a String to lowercase or uppercase. Suppose that we need to get an email two times from our user and check if the emails are the same. We *can't* perform this task like this:

main.py

```
1  email = input("Please enter your email: ")
2  confirmedEmail = input("Please enter your email again: ")
3
4  areEmailsTheSame = email == confirmedEmail
5
6  print("Are emails the same: {}".format(areEmailsTheSame))
```

You can think about this code. Why this code doesn't meet our needs?

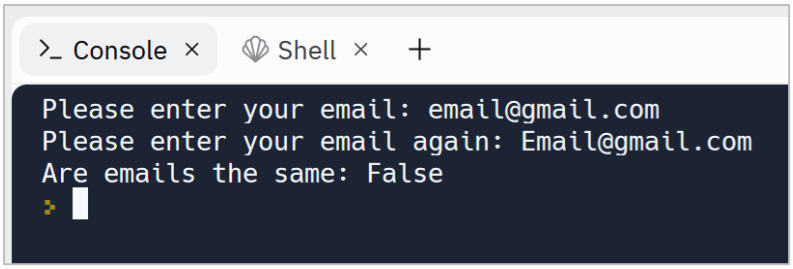
Because, our user may enter her/his email in different formats:

email@gmail.com

Email@gmail.com

Email@Gmail.com

And as we know, Python is Case-Sensitive. So, Python considers the emails as three different texts:

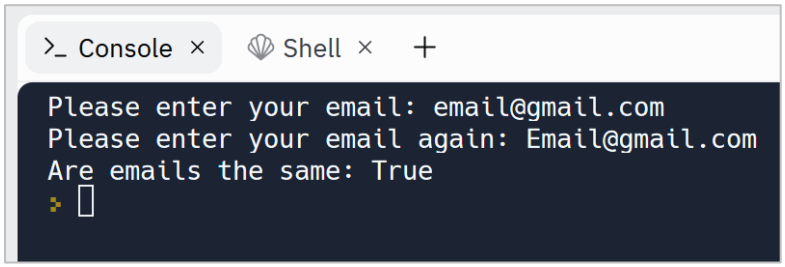



So we need something like **upper()** or **lower()** to uppercase the inputs or lowercase them and after that check if they're the same:

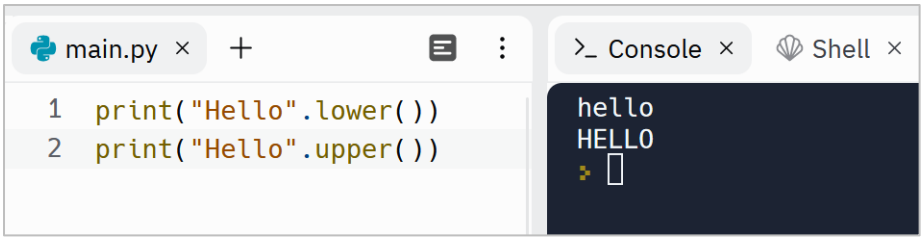
main.py

```
1  email = input("Please enter your email: ")
2  confirmedEmail = input("Please enter your email again: ")
3
4  areEmailsTheSame = email.lower() == confirmedEmail.lower()
5
6  print("Are emails the same: {}".format(areEmailsTheSame))
```

Now our program works fine:




Let's see another example that directly prints the result of the **lower()** and **upper()**:




You may have noticed that some functions comes after a period, like **lower()** and some are called like **len()**.

## find() function

This function is used to search inside a String. This function returns the index of first occurrence of a substring from the given String:


The output is 6, because first occurrence of 'there' is at index 6. As said before, in Python counting starts from zero.

In other words, Python has found 'there' at the 6th character in this string.

Please note that this function is Case-Sensitive. So, for example, it can't find 'There' as T is in uppercase. When it can't find anything, then it returns -1:


main.py
Console
Shell
1 sentence = "Hello there!"
2
3 result = sentence.find("There")
4
5 print(result)
-1

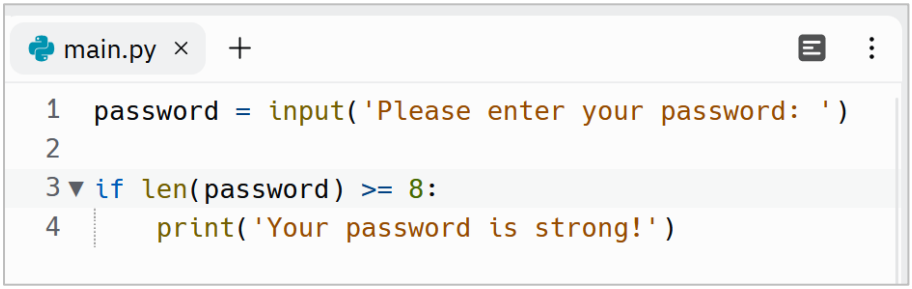
# Section 7

# *Conditional Execution*

It is time to learn how to make decisions in our programs. As you've seen so far, the Python interpreter normally executes the code in order from the top to the bottom. But We can use conditional statements to execute different code blocks based on different conditions.

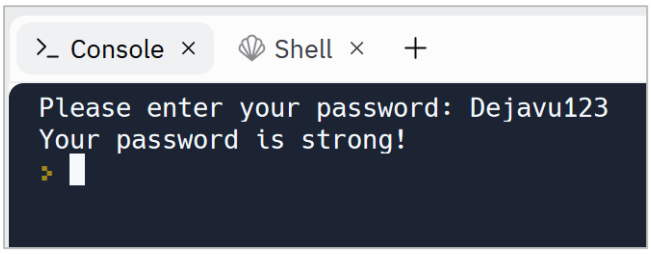
## The if statement

The simplest decision-making statement is if statement. if is the most common flow control in Python. Let's get a password from our user and if it is a strong password, then prints 'it is a strong password':

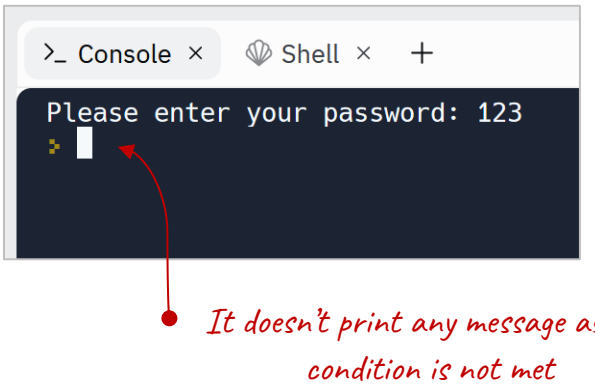


We can read line 3 like this: if the length of the password (variable) is greater than or equal to 8 then execute line 4 else skip line 4.

Run this program two times. The first time enter a password that its length is greater than or equal to 8 characters. It will print a message that says 'Your password is strong!'

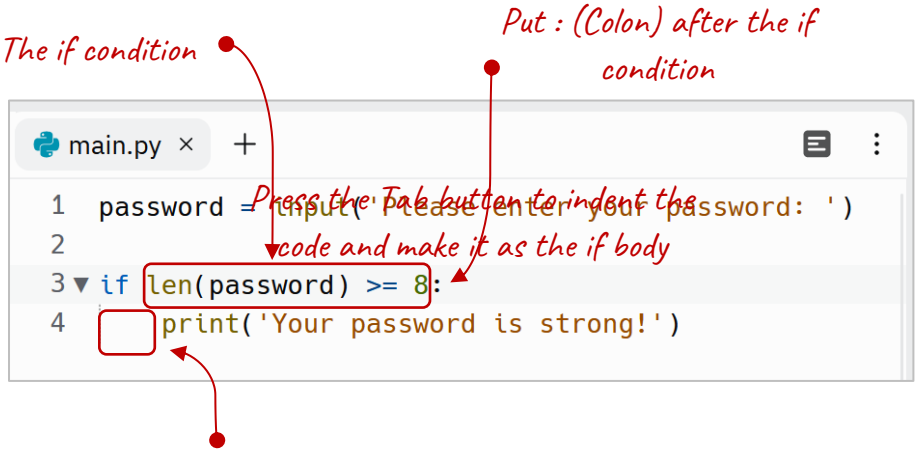



And the second time, enter a password that its length is less than 8 characters:




As you see, it doesn't print anything!

Let's review the syntax of the if statement:




We need to write a condition after the if keyword. This condition usually checks if two values are the same or not, so it must return a Boolean value, True or False. Comparison operators including ==, !=, >, >=, <, <= are used to compare two integers or floats or Booleans or strings. Don't forget to put colon after the condition. In the next line, press the Tab button and write the code that you wish to be executed when the condition is met.

When the result of the condition is True, it executes the if body. *We can indicate the if body by the indentation. A body can be a single line or multiple lines that are indented.*

And when the condition is not met, Python will execute the code after the if body. That is why when we enter a password that its length is less than 8 characters, it won't print the message.

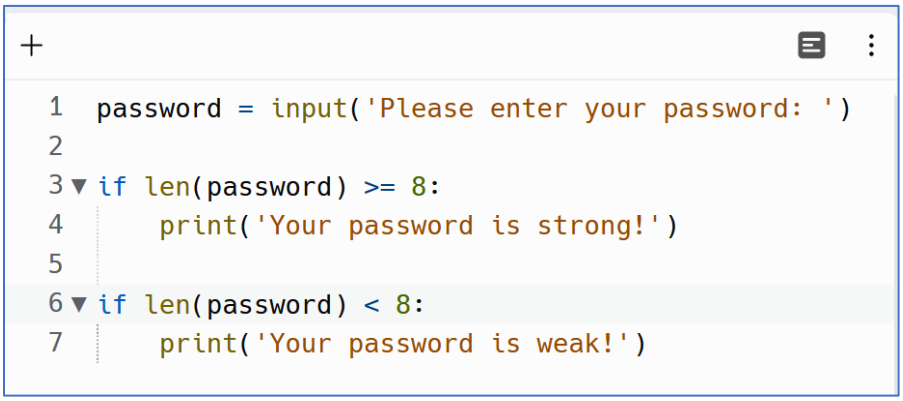
We can hold the result of the condition in a variable and use the variable as the condition:

When **isEligible** is True, it runs line 5, otherwise it skips line 5.

As said earlier, an if body can be a single line or multiple lines. Here is a two-lines body:

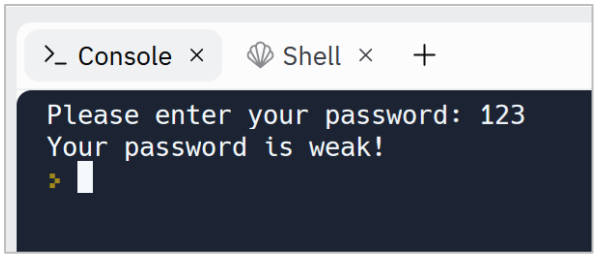

## The else part

The else part is an alternative code block that is executed if the previous condition is not True. Suppose that we need to print another message when the password is weak. So, change the code like this:

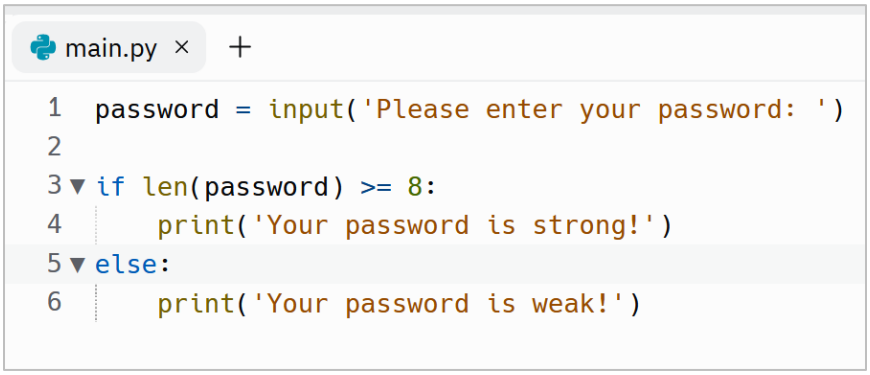



Now we have two if statements. One of them checks if the password is greater than or equal to 8 characters and prints the appropriate message when its conditions is met and the other if statement (at line 6) checks if the password length is less than 8 characters and if so, it prints another message.

So, when we run this program and enter a password that is less than 8 characters, then we'll get an appropriate message instead of getting nothing:




Now our program is user friendlier! Because she/he knows what's going on. But this program needs some improvements. We can simply use else to combine the two if statements into one:




We can read this code like this: if the condition is met, then execute the if block (line 4), else execute the else block (line 6).

When we run this program, we'll get the same result. But this program has a better performance. Because it doesn't need to check two conditions. It only checks one condition.

## Multiple Conditions

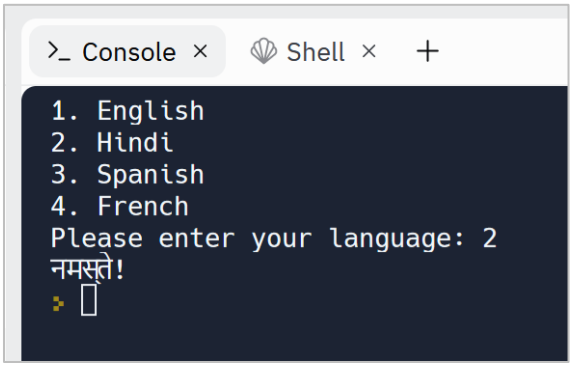
Sometimes the program logic requires to check multiple conditions. In such a case, we can use elif to add extra conditions to the if statement:

main.py

```
print('1. English')
print('2. Hindi')
print('3. Spanish')
print('4. French')

language = input('Please enter your language: ')

if language == '1':
    print('Hello!')
elif language == '2':
    print('नमस्ते!')
elif language == '3':
    print('Hola!')
elif language == '4':
    print('Bonjour!')
else:
    print('Your input is not valid!')
```

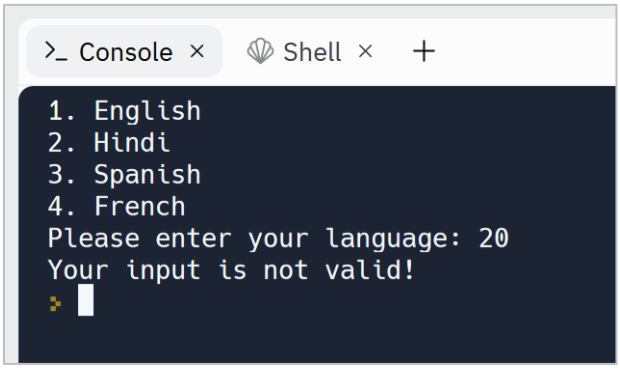
We can have only one if or else statements, but it is possible to use elif statement as many times as we need. Each elif statement executes its code block if the previous conditions were False.

In this example, it checks what number has entered and based on the input number, it will display a hello message. Our user can enter a number between 1 to 4, otherwise it will print a warning message instead of hello:



Note that we need to check user input with a String like '1' or '2' instead of 1 or 2. Because the input function (at line 6), will return a String. So, the type of the language will be String and we need to check it with another String.

Also, when none of the if conditions are met, the else part will be executed:

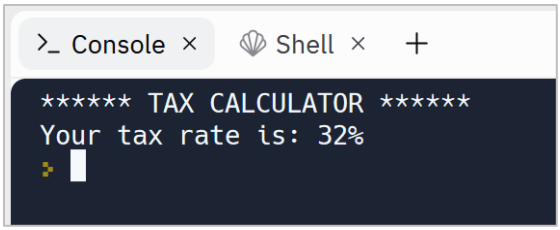



## Inner conditions

Also, there are times that we need to check multiple conditions in one if condition. For example, suppose that we need to check tax rate like this:

```
1  print('****** TAX CALCULATOR ******')
2
3  isMarried = True
4  hasChildren = True
5
6  taxRate = 0
7
8  if isMarried == True:
9    if hasChildren == True:
10     taxRate = 32
11   else:
12     taxRate = 38
13
14 print('Your tax rate is: {}%'.format(taxRate))
```

The output of this program is like this:




At line 6 we define a variable named **taxRate** and initialize it with 0 temporarily, we will reassign it with an appropriate value at line 10 or 12

At line 8 we check if **isMarried** is True, if so then it checks (at line 9) *also* that if **hasChildren** is True too, if so, it sets **taxRate** to 32

Line 12 will be executed only when the condition at line 9 is not met

The if statements at line 9 is called an inner if statement.

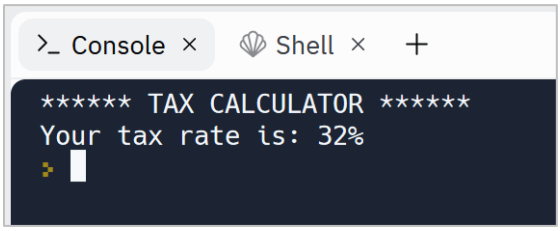
## Combine conditions by using Boolean Operators

We can simplify the previous program by using the Boolean Operators (and, or). In other words, we can combine the if statements at lines 8 and 9 into one if statement. To do so, rewrite the program like this:

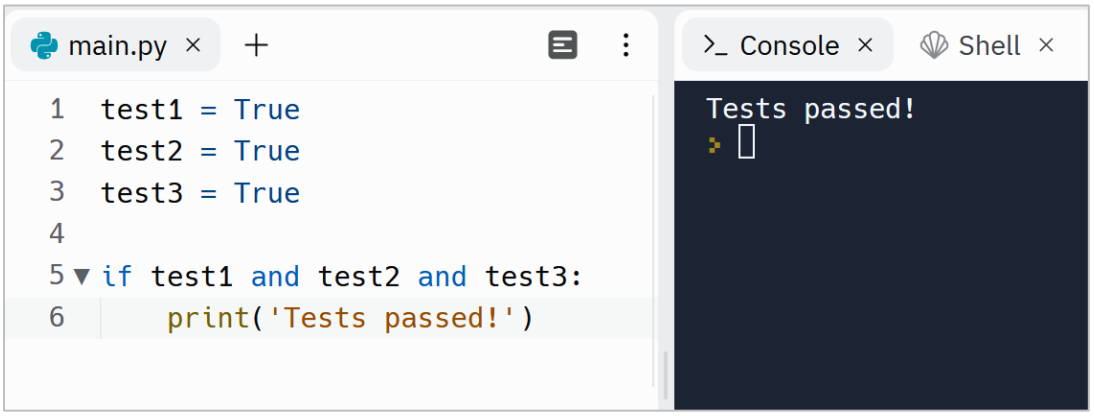
```
1  print('****** TAX CALCULATOR ******')
2
3  isMarried = True
4  hasChildren = True
5
6  taxRate = 0
7
8  if isMarried == True and hasChildren == True:
9    taxRate = 32
10 else:
11   taxRate = 38
12
13 print('Your tax rate is: {}%'.format(taxRate))
```

So, line 9 will be executed only if both the conditions are True, otherwise it executes line 11.

The output of this program is the same as the previous one:




We can have as many conditions as we need. In this example, we have three tests to use in the condition:

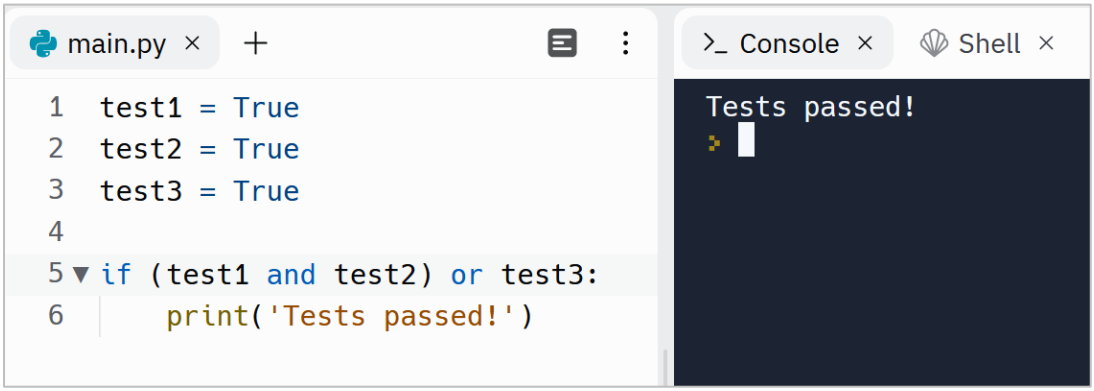



So, we use and when all the conditions must be True. Also, there are times that we need to run a code when either one of the conditions is True. In that case, we combine the conditions with or operator:


Like and, we can use as many or operators as we need.

We can also use both the Boolean operators (and, or) at the same time. In this example, the condition will be True if test1 and test2 are True or test3 is True:


We can use parenthesis to increase readability:



# Python Conditionals Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

Python Conditionals Exercises – Dejavu Code Part 1

Python Conditionals Exercises – Dejavu Code Part 2

Python Conditionals Exercises – Dejavu Code Part 3

Python Conditionals Exercises – Dejavu Code Part 4

# Section 8

# *Write your own functions*

Now you know that, there are built-in functions which are part of Python Standard Library. In other words, functions like **print()**, **input()**, **str()**, **int()** are written by Python developers. *All we need as programmers, is to call them.*

*Besides that, we can define our own functions. Functions that we need to write according to our needs.*

So, there are two categories of functions:

- Built-in functions written by other developers. All we need to do is to call them.
- User-defined functions written by us as developers. We must first write the function and then call them.

Let's see how to write user-defined functions in Python. Functions that belong to us! Let me give an example to understand what are user-defined functions and what are their usages.

Suppose we need to get a country name from the user and print its capital. So, we may need something like this:

```
country = input('Please enter a country name: ')

capital = ''

if country == 'France':
    capital = 'Paris'
elif country == 'Germany':
    capital = 'Berlin'
elif country == 'Italy':
    capital = 'Rome'

print('The capital of {} is {}'.format(country, capital))
```

## Write your first function

Now suppose that, we need to get another country from the user and convert it to a capital name. So, what should we do now? Should we duplicate the same code?

```
country = input('Please enter a country name: ')

capital = ''

if country == 'France':
    capital = 'Paris'
elif country == 'Germany':
    capital = 'Berlin'
elif country == 'Italy':
    capital = 'Rome'

print('The capital of {} is {}'.format(country, capital))

country = input('Please enter a country name: ')

capital = ''

if country == 'France':
    capital = 'Paris'
elif country == 'Germany':
    capital = 'Berlin'
elif country == 'Italy':
    capital = 'Rome'

print('The capital of {} is {}'.format(country, capital))
```

*No never!*

*We must, convert this code to a user-defined function and then use the function as many times we need.* In Python, a user-defined function declaration begins with the keyword **def** followed by the function name, a pair of parentheses and a colon:

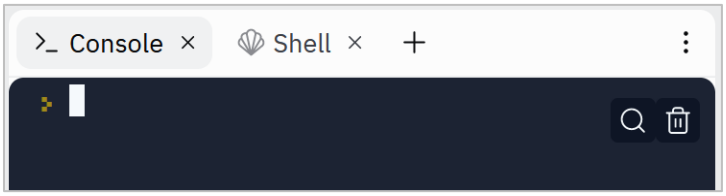
```
1 ▼ def country_to_capital():
2       country = input('Please enter a country name: ')
3
4       capital = ''
5
6 ▼     if country == 'France':
7           capital = 'Paris'
8 ▼     elif country == 'Germany':
9           capital = 'Berlin'
10 ▼    elif country == 'Italy':
11          capital = 'Rome'
12
13      print('The capital of {} is {}'.format(country, capital))
```

At line 1 we defined a function named country_to_capital. Use the snake case convention for function names, in which all the words are written in lowercase and each space is replaced by an underscore (_)

At lines 2 to 13 we implemented the body of the function. Yes, like the if statement, user-defined functions have body. We can simply select all the code we want to indent and press the Tab key

So once again, by using indentation we say to Python that which codes belongs to the **country_to_capital** function.

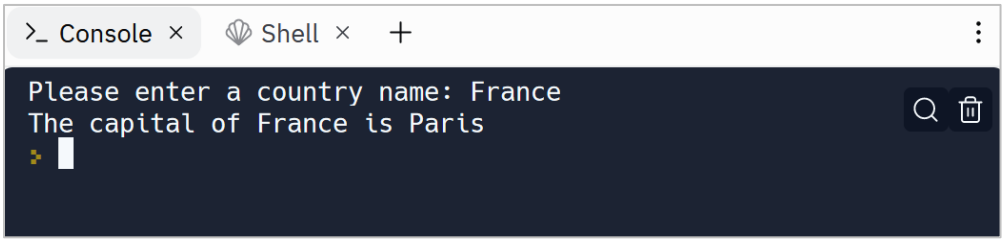
Now let's run this program. But when we run it, nothing happens:




Why? Because at lines 1 to 13, we only *defined* the function. To run this function, we need to *call* it. To do so, simply write the function name followed by open and close parentheses:

```
1  def country_to_capital():
2      country = input('Please enter a country name: ')
3
4      capital = ''
5
6      if country == 'France':
7          capital = 'Paris'
8      elif country == 'Germany':
9          capital = 'Berlin'
10     elif country == 'Italy':
11         capital = 'Rome'
12
13     print('The capital of {} is {}'.format(country, capital))
14
15 country_to_capital()
```

So, at line 15, we called the function. Please note that there is not any indentation. Now run it again. This time Python knows that it must call the **country_to_capital()** function. In other words, it must run the body of this function:

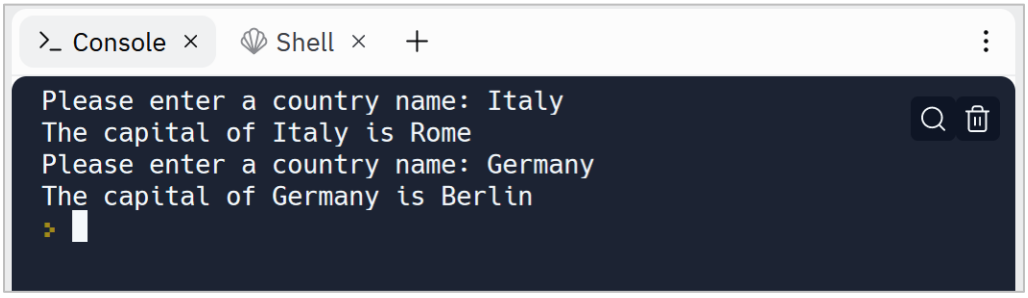



In a nutshell, to use user-defined functions, we first define the function and implement its body and then call it as many times as we need.

Let's call our function two times (lines 15 and 16). So, call the function once again at line 16:

```python
def country_to_capital():
    country = input('Please enter a country name: ')

    capital = ''

    if country == 'France':
        capital = 'Paris'
    elif country == 'Germany':
        capital = 'Berlin'
    elif country == 'Italy':
        capital = 'Rome'

    print('The capital of {} is {}'.format(country, capital))

country_to_capital()
country_to_capital()
```

Now the body of the **country_to_capital()** will be executed two times:



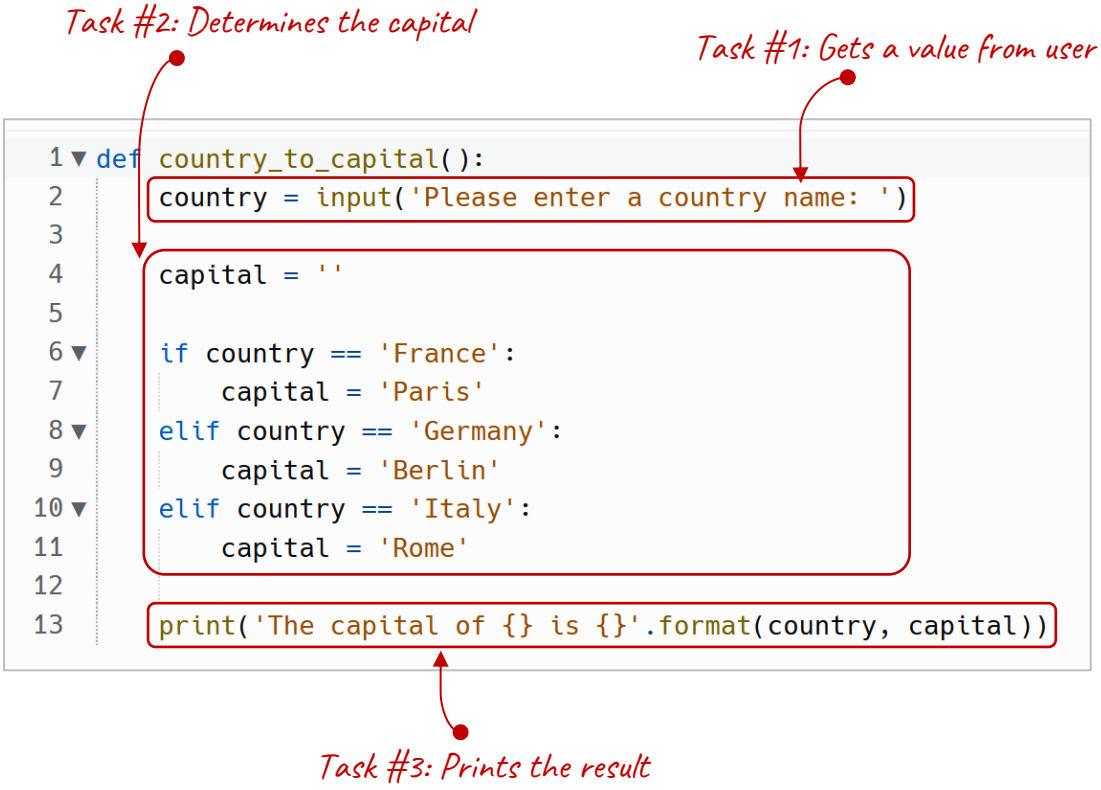

So, it is clear that writing our own functions are sometimes useful. They help us avoid code duplication. Also, functions help us to make our code more organized.

## Return a value from a function

*One of the advantages of functions is that they are written once and could be called multiple times.*

But, in software design, it is a bad practice to write a function that has various responsibilities. The function we wrote in the previous example, has three tasks:

1- It gets a country name,
2- then determines the capital based on it,
3- and finally prints the result:




*So, this function should be broken into three separate functions:*

1- A function that gets a country name (Task #1)
2- Another function that determines the capital (Task #2)
3- Another function that prints the result (Task #3)

Let's start with Task #1:

```
1  def get_country():
2      country = input('Please enter a country name: ')
3      return country
4
5  def country_to_capital():
6      country = get_country()
7
8      capital = ''
9
10     if country == 'France':
11         capital = 'Paris'
12     elif country == 'Germany':
13         capital = 'Berlin'
14     elif country == 'Italy':
15         capital = 'Rome'
16
17     print('The capital of {} is {}'.format(country,
   capital))
18
19 country_to_capital()
```

At lines 1 to 3 we defined a function named **get_country()** that gets an input from user and assigns it to the **country** variable (line 2) and returns the value of the **country** variable (line 3) A function can return any type of value including strings, numbers or Booleans. We can even return the result of input() function directly without using the country variable, but it is not recommended as it decreases code readability.

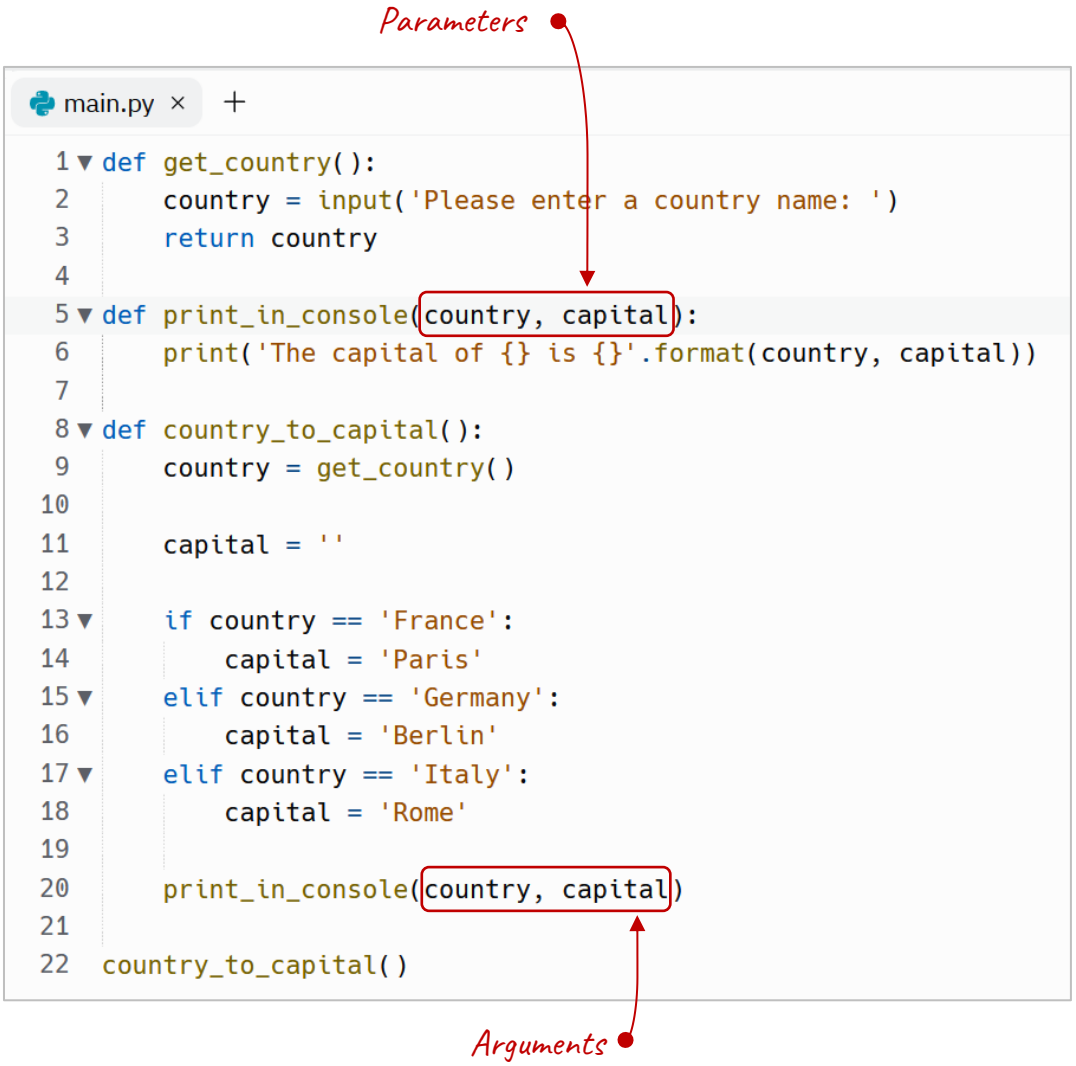
At line 6 We first calls the **get_country()**, so it executes this function and after the execution, returns the value of the **country** variable. Returns to where? It returns to the place that **get_country()** is called.

So, a function can return a value by using the **return** keyword. *We usually put the return at the end of a function.* Because a return ends the function execution and returns the result to the caller.

When we run this program, we will get the same result. Now let's extract Task #3 into a new function.

## Pass arguments to a function

So, we should define another function that its task is to print something. But we must pass the values (arguments) it needs:




At lines 5 to 6 we defined a function named **print_in_console()** that accepts two values (parameters) and prints them in the output

At line 20 we simply call the **print_in_capital()** function and pass it the values (arguments) that this function needs

You already are familiar with the term argument. Argument is the value or values that are sent to the function when it is called.

And Parameter is the variables we define inside the parentheses in the function definition. Some people mistakenly use the terms interchangeably, so don't confuse.

When we run this code again, we will get the same result. But this time, our program is more organized.

*Also, when each function has its own specific task, it will increase testability of our programs.*

Besides that, when we follow this principle, it will improve the expandability. For example, suppose that we want to print the result by using a real printer. So we can simply write another function that do this and replace it with **print_in_console()**.

Let's review what happens when we execute this code.

When we run the program, Python finds out that there are three function definitions (at lines 1, 5, 8) and one function calling (at line 22).

So now Python knows that it must execute the body of the **country_to_capital()** function line by line.

At the first line of this function (line 9), it must calculate the expression at the right-hand side of the **=** (assignment operator) and assigns it to the **country** variable. So, it executes the first line of the **get_country()** function, gets a String from the user and assigns it to the **country** variable (line 2). In the next line (line 3), it returns what is inside the **country** variable to the line 9.

So now the **country** variable at line 9 knows which country our user has entered.

Then Python continues running next line (line 11), and initializes the capital variable with a temporary value (line 11), determines the capital (at lines 13 to 18) and finally executes the **print_in_console()** function, but before executing, it passes country and capital to the **print_in_console()** function, so now **print_in_console()** function knows what should be printed.

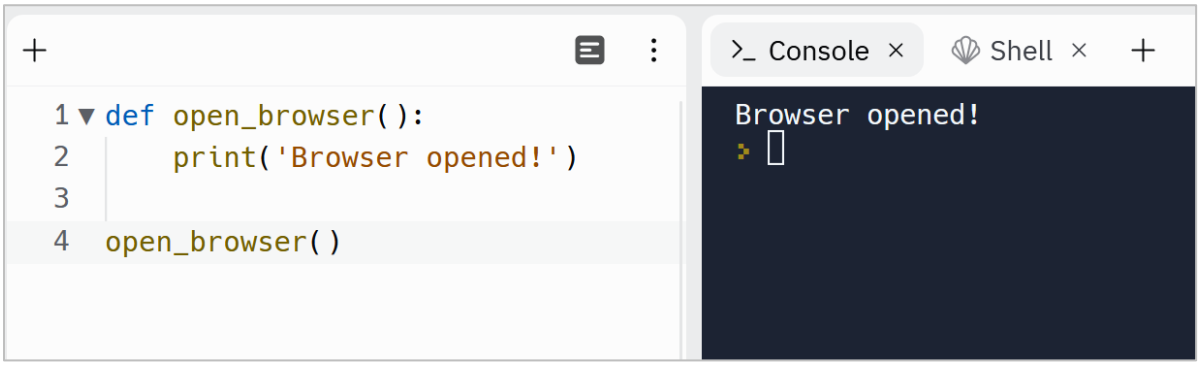
*One point that you may have noticed by now is that a function may or may not accept value or values. Also, a function may or may not return a value. It depends on your program's logic.*

## Previously on functions

If you are still confused about functions, this is for you. Let's review it!

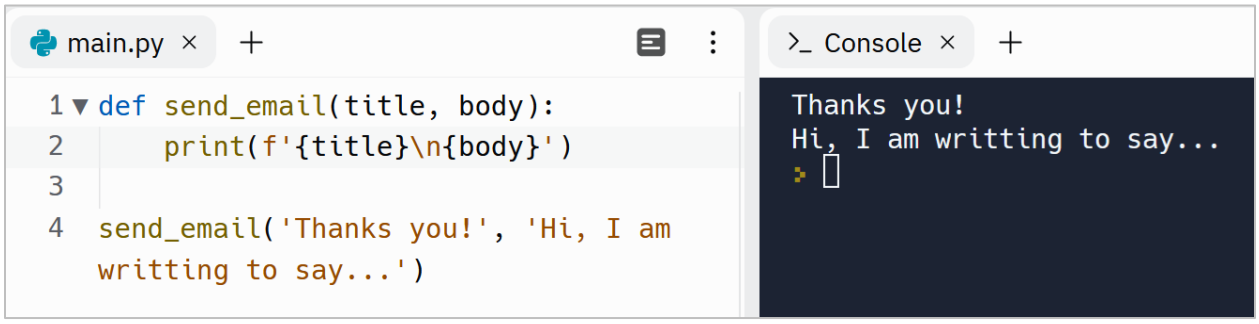
Python has some pre-written functions like str(). But there are times we need to have our own functions that are called user-defined functions. So, if we need to execute the same task again and again, we group its code into a function, assigns a name to the function and call the function name again and again, instead of duplicating the code. However, we also split our code into functions to keep our program's modularity.

In its simplest form, a function only is written to executes a task, without accepting parameters or returns a value:



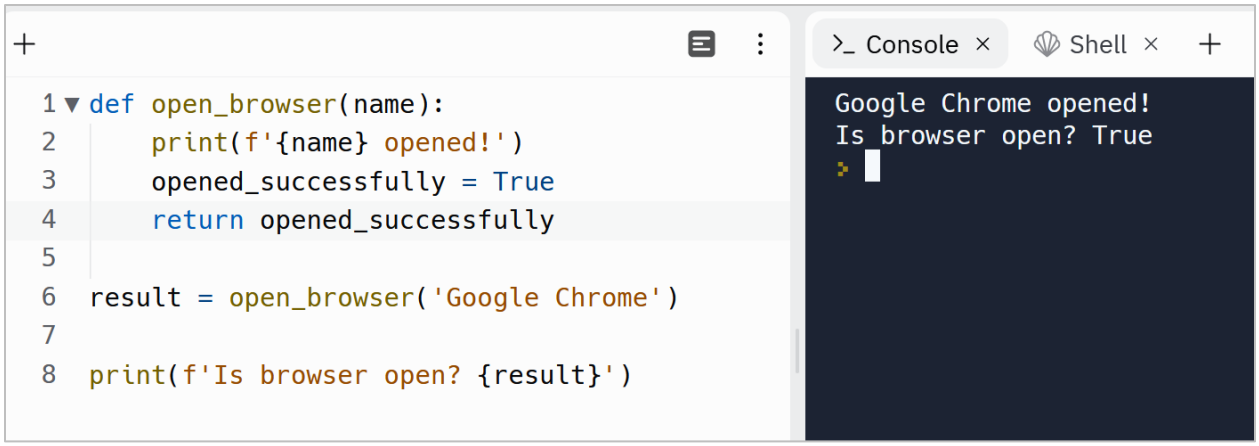

Also, there are times that we need to pass value or values to a function, so that the function can work based on that value or values. In this example, we pass the browser that we need to open:


If we need to pass multiple values to a function, we need to separate parameters by using comma:



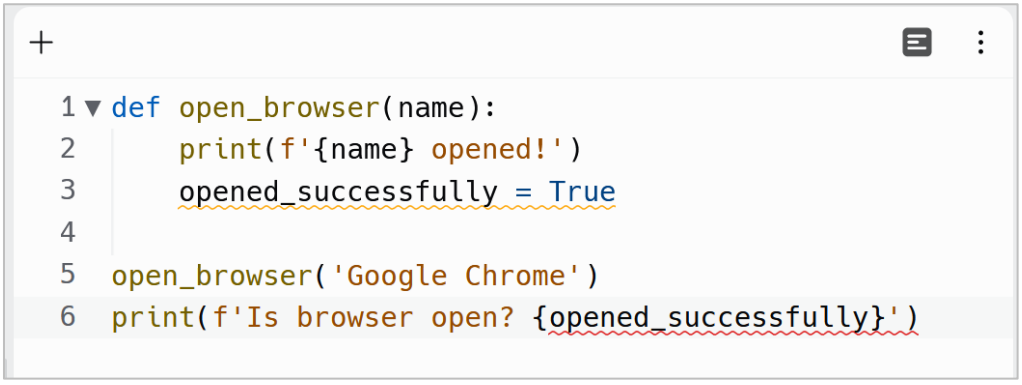

So, we can send as many parameters as we need by separating them by using comma. Just keep in mind that the order of arguments provided to a function matters. The order of the arguments must be the same as the order of the parameters.

And there are also times that we need to return a value (usually the result of the function). This value can be of any type like strings, numbers or Booleans. In this example, we return a Boolean value and hold the returned value in the **result** variable at line 6:

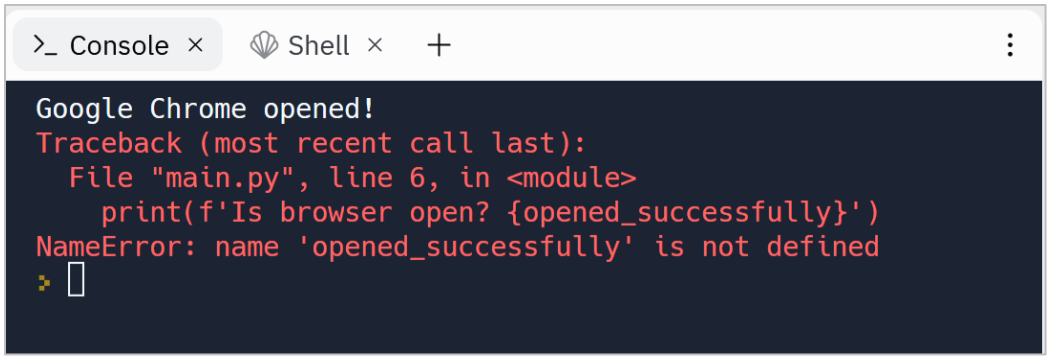



## Local scope vs Global scope

Note that we cannot access a variables defined inside a function somewhere else. Because they have a *local scope*. It means, when we define a variable inside a function, it is only available in the function that created it. For example, we cannot access **opened_successfully** outside the **open_browser** function:

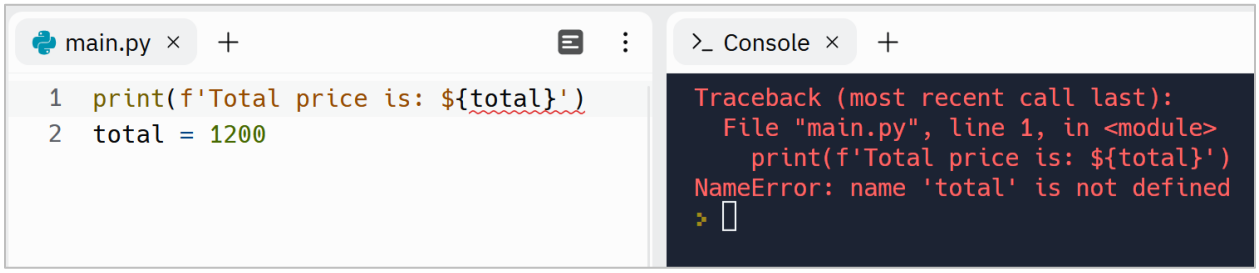



So, it generates an error message, because cannot find the variable:




On the other hand, variables that are defined outside a function, are *global scope*. It means they are accessible everywhere in the code. In this example, we have defined baseTax variable in the global scope, so it is accessible in the functions (lines 4 and 7) and outside of the functions (line 9).

One important point to note is that we cannot access a variable before we create it:




## Pro tips

And finally choose meaningful names for your functions. It helps us and other developers that read our code to quickly understand the purpose of each function. As functions are actions, the name can start with a varb like:

- send_email
- print_report
- build_database
- insert_into_database
- delete_file
- ...

Also, if your function returns a value, use verbs like:

- get_email_address
- calculate_profit
- generate_report
- compute_delay

And if your function returns a Boolean value, then it is recommended to name the function in a short question form:

- is_internet_connected
- are_emails_valid

## Python Custom Function Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

[Python Custom Functions Exercises – Dejavu Code]

# Section 9

# *Data Structures in Python*

*In Python and other programming languages, a Data Structure defines the way that a group of related data can be stored under one name.* In simple words, until now, we defined variables to store one value. *But by using a Data Structure, we can store multiple values into one variable.*


In Python, we usually use these Data Structures:

- List
- Tuple

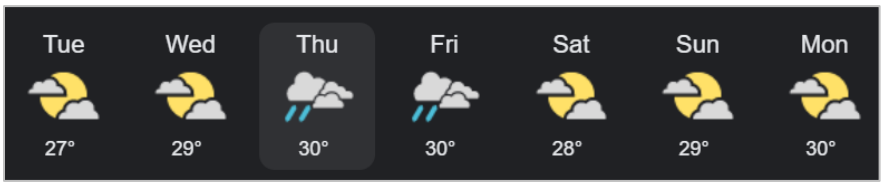
- Set
- Dictionary
- Array

Each of these Data Structures has its own way of storing and retrieving data. Let me clarify one thing now. We have multiple Data Structures because each of them has its own way of storing and retrieving data. For example, Tuples in Python consumes less memory than Lists. So, when we have a large amount of data needs to be stored and *we know the exact number of these data*, it is better to use Tuples over Lists. *But sometimes we don't know how many values we need to store. In this case, Lists are a good replacement for Tuples.*

## List Data Structure in Python

Lists are one of the main Data Structures in Python that we need to know.

Let's define a List that represents the temperature in Miami in the last 7 days (last week):




main.py

```
1  tempInMiami = [27, 29, 30, 30, 28, 29, 30]
```

Now we have a variable named **tempInMiami** that have 7 values! **In technical terms, we have a list with 7 items.** So, to create a list we need to enclose the items in **[]** (square brackets) and each item needs to be separated by a **,** (comma).

But we can store these 7 values into 7 variables! Is there any advantage with Data Structures like Lists?

Using 7 variables instead of using a List would be something like this:

main.py

```
1  tempInMiamiDay1 = 27
2  tempInMiamiDay2 = 29
3  tempInMiamiDay3 = 30
4  tempInMiamiDay4 = 30
5  tempInMiamiDay5 = 28
6  tempInMiamiDay6 = 29
7  tempInMiamiDay7 = 30
```

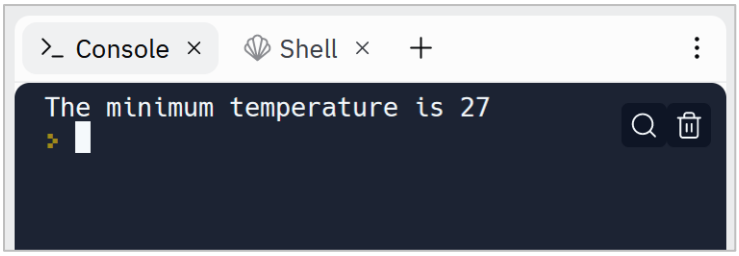
Now suppose that we need to know what the lowest temperature was in the week?

Well, if we've stored values into a List, we can simply call a function named **min()** and pass it the list of temperatures to find the lowest value:

main.py

```
1  tempInMiami = [27, 29, 30, 30, 28, 29, 30]
2
3  minTemp = min(tempInMiami)
4
5  print('The minimum temperature is {}'.format(minTemp))
```

The output is:



But if we have stored values into multiple variables, we may need something like this:

main.py

```
tempInMiamiDay1 = 27
tempInMiamiDay2 = 29
tempInMiamiDay3 = 30
tempInMiamiDay4 = 30
tempInMiamiDay5 = 28
tempInMiamiDay6 = 29
tempInMiamiDay7 = 30

minTemp = tempInMiamiDay1

if tempInMiamiDay2 < minTemp:
    mintemp = tempInMiamiDay2
elif tempInMiamiDay3 < minTemp:
    mintemp = tempInMiamiDay3
elif tempInMiamiDay4 < minTemp:
    mintemp = tempInMiamiDay4
elif tempInMiamiDay5 < minTemp:
    mintemp = tempInMiamiDay5
elif tempInMiamiDay6 < minTemp:
    mintemp = tempInMiamiDay6
elif tempInMiamiDay7 < minTemp:
    mintemp = tempInMiamiDay7

print('The minimum temperature is {}'.format(minTemp))
```

The output of this code will be currect too.

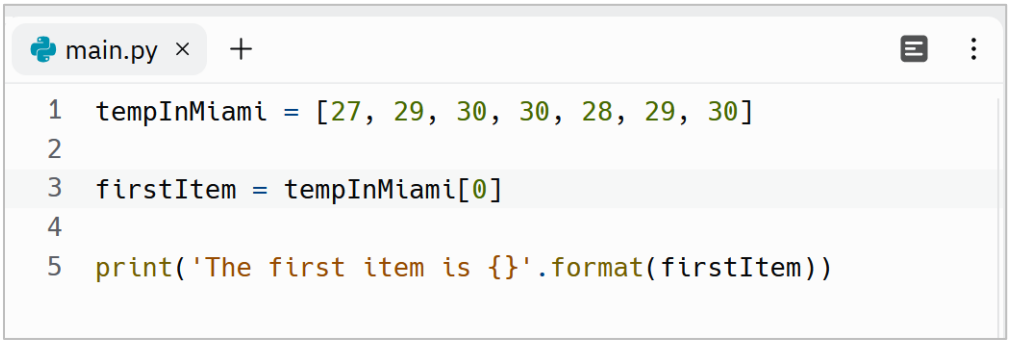
But as you see, by using a Data Structure like List, our data are organized and easy to work with. Imagine what a mess it will be if we need to work with temperatures of one year or even more!

### List Indexing

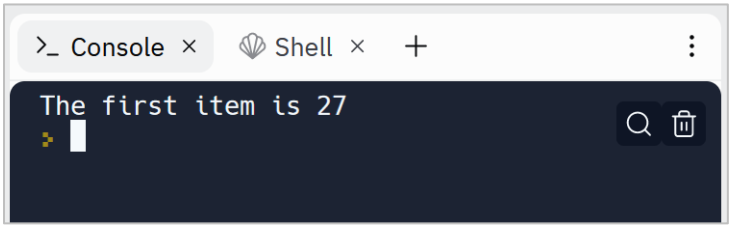
I talked already about indexes in Python. For example, the **find()** function is used to search inside a String. This function returns the *index* of first occurrence of a substring from the given String:


The output is 6, because the first occurrence of 'there' is at index 6. As said before, counting starts from zero in Python.

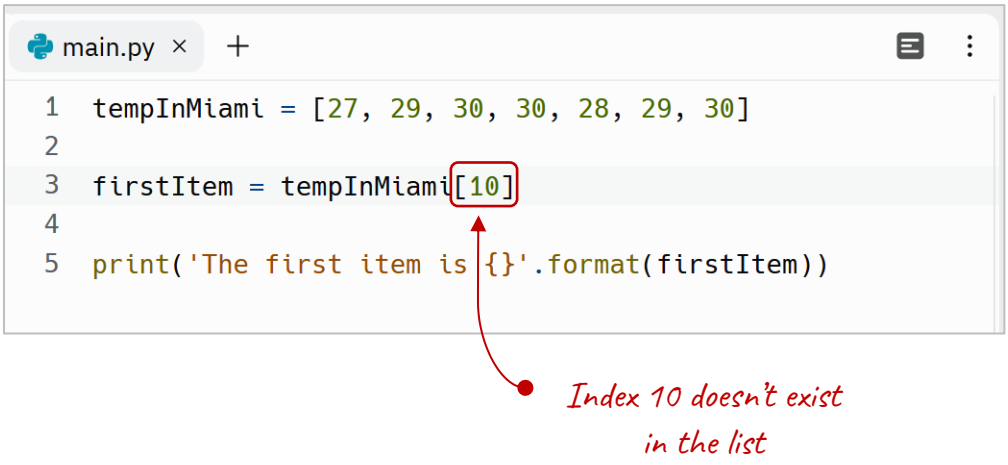
Now let's talk about how indexes work in Lists. To access each item of the List individually, we can use the index of that particular item. For example, to access the first value of the **tempInMiami** List, we need to use its index inside **[]** (square brackets):



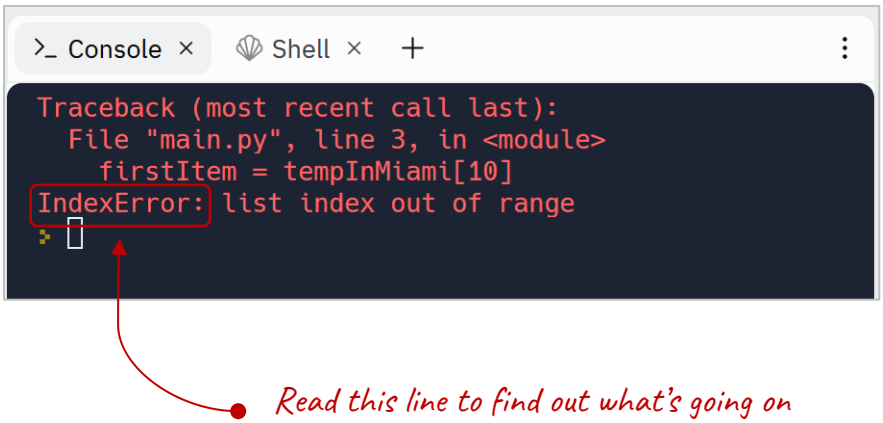

The output is:




If we try to use an index that doesn't exist in a list:




We will get an error message like this:




This error is of Type **indexError**. Its error message is quite self-explanatory: "list index out of range". The range of this list is between 0 to 6. So, we're not allowed to use an index like 10 for this list.

### Adding New Items to a List

**Lists in Python are mutable (changeable).** It means we can change list's items after creating it. We can add, update and remove an item or items from a list.

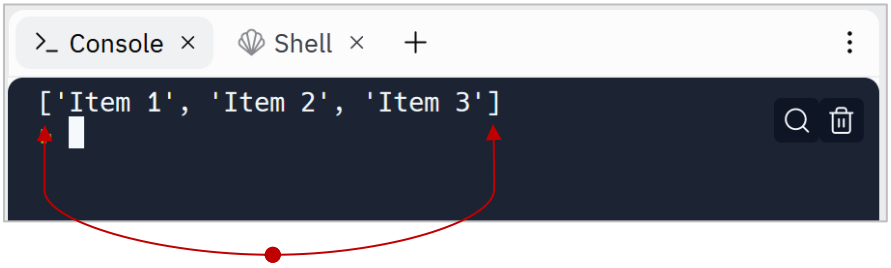
To add a new Item to an existing List, we can simply use the **insert()** function. The **insert()** function needs two arguments, an index and the new value we need to insert in the List:

```
main.py
1  items = ['Item 1', 'Item 2']
2
3  items.insert(2, 'Item 3')
4
5  print(items)
```

At line 1 we defined a List named **items** with two values of type String

At line 3 we inserts a new value 'Item 3' at index 2 of this List

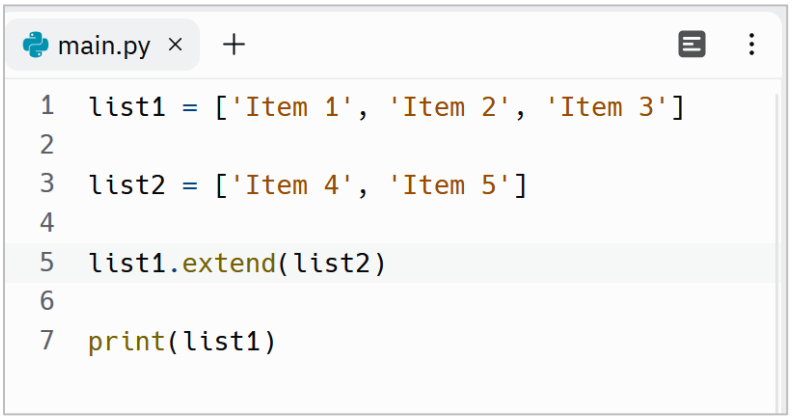
So when we use the **print()** function to print this list, the output will be like this:



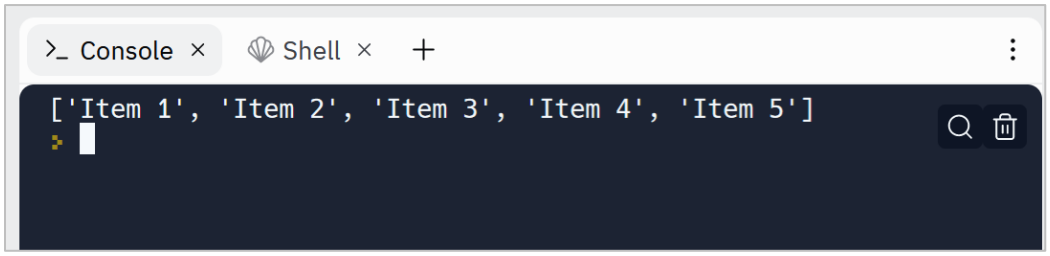

*The console tries to show us that the output is the result of showing a list*

Or we can insert a new item at the beginning of the list:

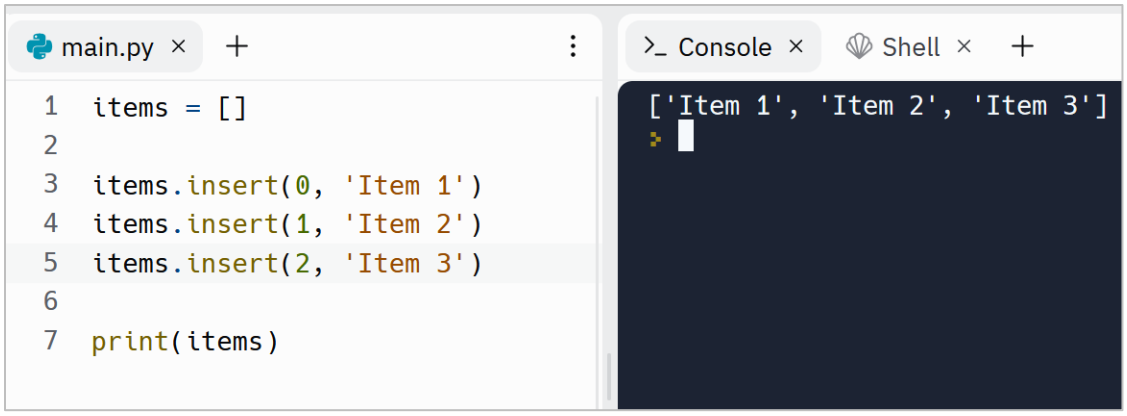

We can also add multiple items at the same time to a List. To do so, we first need to create another List and then extend the first List:




And the output:



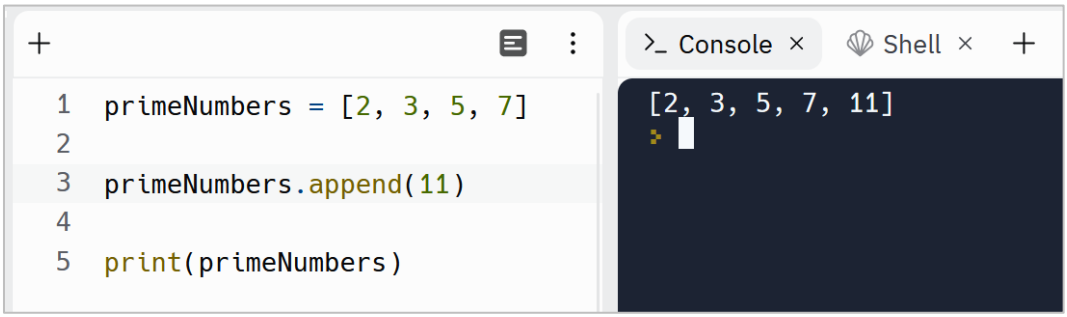

We can also create an empty List and then add items to it:




At line 1 we defined an empty list named **items** with no default items

### Append a new Item to a List

We can simply append a new item to the end of a List by using the **append()** function:




### Removing an item from a List

We can simply remove an item from a List by using a function named **remove()**. This function takes an item and removes it from the List:

```
main.py
1  items = ['Item 1', 'Item 2', 'Item 3']
2
3  items.remove('Item 2')
4
5  print(items)
```

The output is exactly what we expect:

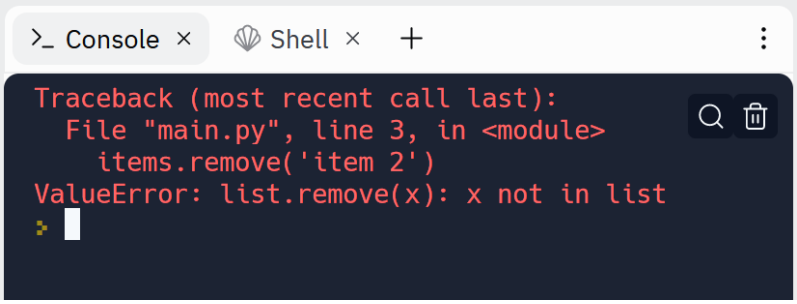
```
Console  Shell
['Item 1', 'Item 3']
```

Note that Python is Case-Sensitive. So, we will get an error message if we don't consider it:

*i is in lowercase here, but uppercase in the List*

```
main.py
1  items = ['Item 1', 'Item 2', 'Item 3']
2
3  items.remove('item 2')
4
5  print(items)
```

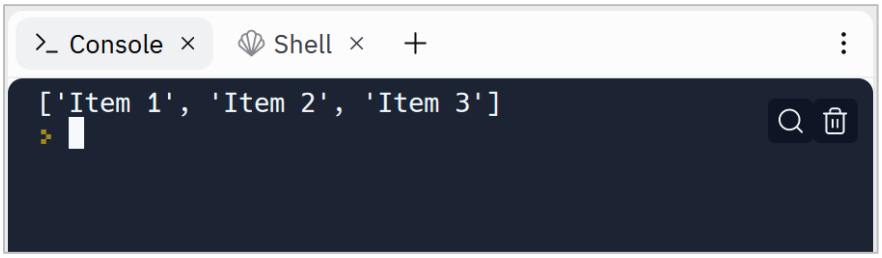
So, we will get an error that says you're going to remove something from a list, but that thing does not exist in that list:




**Updating an existing item in a List**

Here is how we can update an existing item in a List:


We can simply use index of an item to reassign or update its value. So, line 3 means assign 'Item 3' with the value that exists at index 2 of the **items** List:



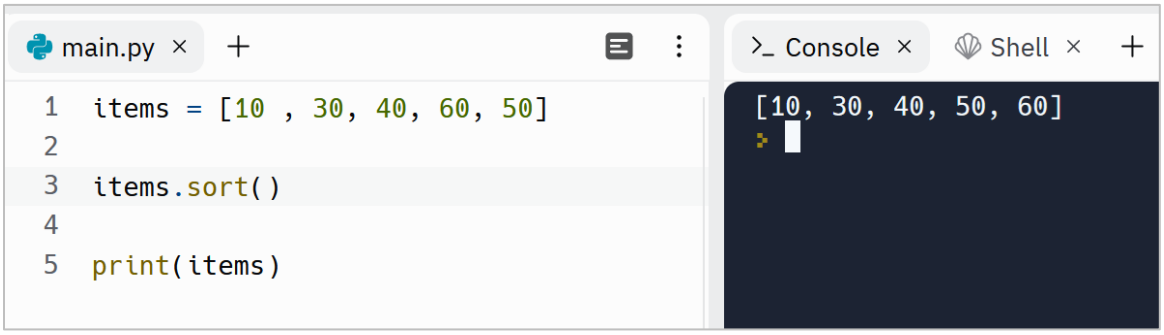

**Clear a List**

To remove all Items in a List at the same time, we can call the **clear()** function:

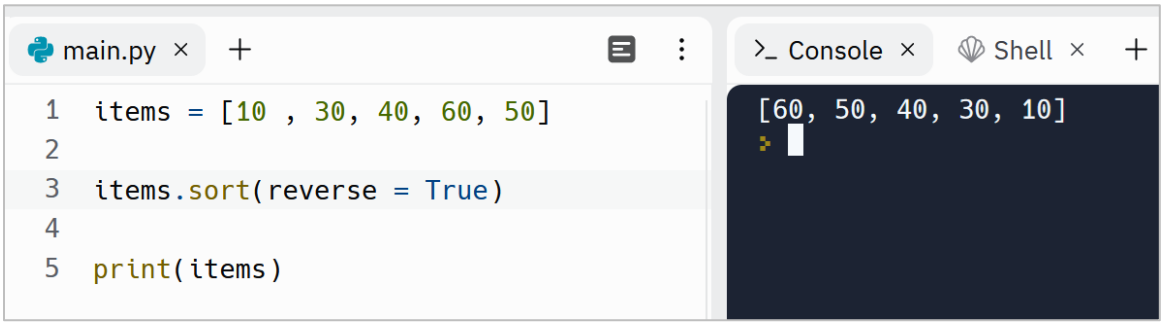

[] represents an empty list.

### Sort the items of a List

To sort the items of a List, we can call the **sort()** function:




We can also sort items in descending order by reversing the result. All we need to do is to pass **reverse = True** to the **sort()** function:



Note that the **sort()** function also works with Strings and Characters. By default, the elements of a list of strings are sorted in alphabetical order:

```
main.py
1  items = ['Python', 'C++', 'Java', 'C#', 'Javascript']
2
3  items.sort()
4
5  print(items)
```

The sorted items are as follow:

```
Console  Shell
['C#', 'C++', 'Java', 'Javascript', 'Python']
```

### Count appearance of an item in a List

If we need to count the number of times that a specified item appeared in a List, we can simply use the **count()** function. For example, to see how many days the weather was 30°C, we use the **count()** function like this:

```
main.py
1  tempInMiami = [27, 29, 30, 30, 28, 29, 30]
2
3  result = tempInMiami.count(30)
4
5  print('{} day(s) was 30°C'.format(result))
```

### Getting the length of a List

We can also find out how many items are in a List by using the count() function:

```
1  items = ['Python', 'C++', 'Java', 'C#', 'Javascript']
2
3  numberOfItems = len(items)
4
5  print('There are {} item(s) in the list'.format(numberOfItems))
```

And the output:

```
There are 5 item(s) in the list
```

### in keyword

By using the in keyword, we can find out if an item exists in a list or not. If the specified item exists in the list, it returns True otherwise it returns False. In this example, False exists in the list, so the in keyword returns True and the if statement executes its body (line 4):

```
main.py
1  tests_result = [True, True, False, True]
2
3  if False in tests_result:
4      print('You have not passed all the tests!')
```

```
Console  Shell
You have not passed all the tests!
```

### Slicing a List in Python

We can also slice a List by using the **:** (colon slicing) operator. In other words, we can access a range of items in a List by using this operator:

```
main.py
1  items = [31, 32, 30, 29, 28, 30, 31]
2
3  sliced = items[1:3]
4
5  print(sliced)
```

```
Console  Shell
[32, 30]
```

So, at line 3, we sliced the **items** List, that its start index is 1 and the end index is 3. In other words, Python slices the **items** List from index 1 *until* index 3:

```
items = [31, 32, 30, 29, 28, 30, 31]
          0   1   2   3   4   5   6
```

There is also another form of List slicing. We can start slicing from a specified index to the end:

```
main.py
1  items = [31, 32, 30, 29, 28, 30, 31]
2
3  sliced = items[4:]
4
5  print(sliced)
```

```
Console  Shell
[28, 30, 31]
```

```
items = [31, 32, 30, 29, 28, 30, 31]
         0   1   2   3   4   5   6
```

### Copy a List by using List Comprehensions

There are multiple ways to create a new List based on an existing List. The simplest way is to create an empty List, iterate through the original List and append its items to the empty List one by one. We do this by using a for loop. You will learn about loops later. In this example, we have a list of prices in US dollar and we want to create a new list that holds the prices in Euro:

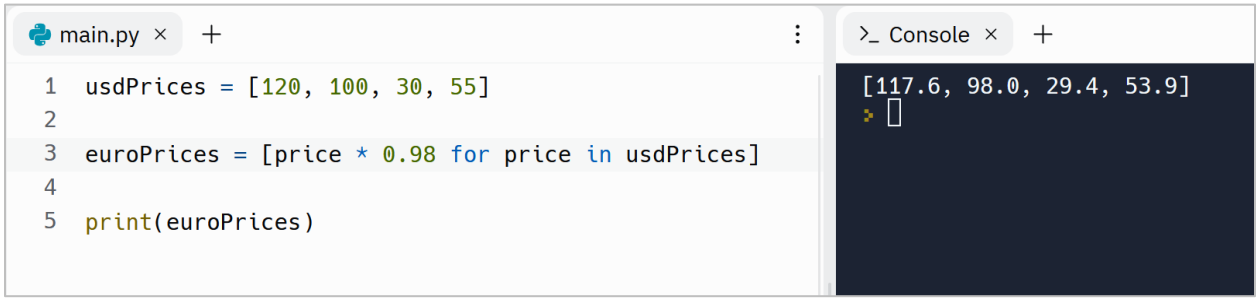
```
main.py
1  usdPrices = [120, 100, 30, 55]
2
3  euroPrices = []
4
5  for price in usdPrices:
6      exchanged = price * 0.98
7      euroPrices.append(exchanged)
8
9  print(euroPrices)
```

```
Console
[117.6, 98.0, 29.4, 53.9]
```

A for loop is used to iterate over a sequence like List. In this example, we have 4 iterations. Because the usdPrices List has 4 items. In each iteration, the for

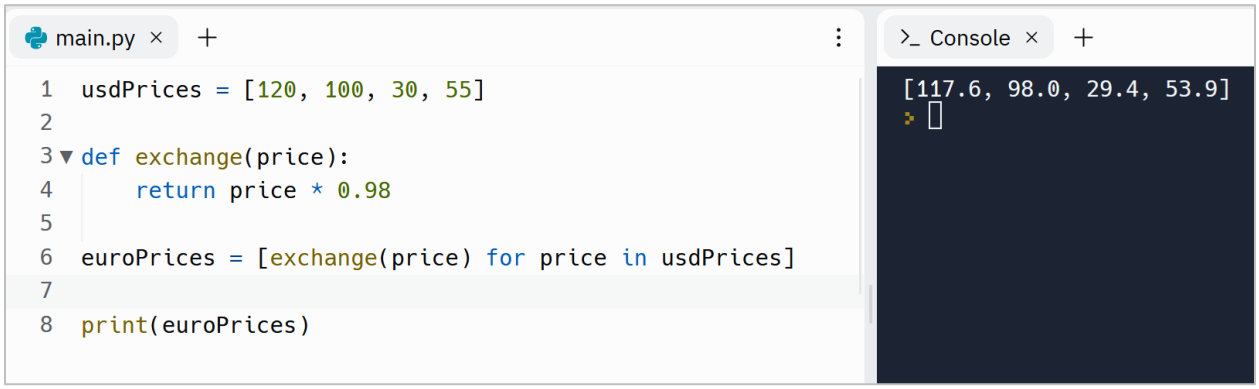
loop assigns the current item to the price variable. For example, in the first iteration, price is 117.6, in the second iteration, the value of the price variable is 98.0 and so on.

The second way to copy a list is to use a concept named List Comprehension. It works like the for loop in the previous example:



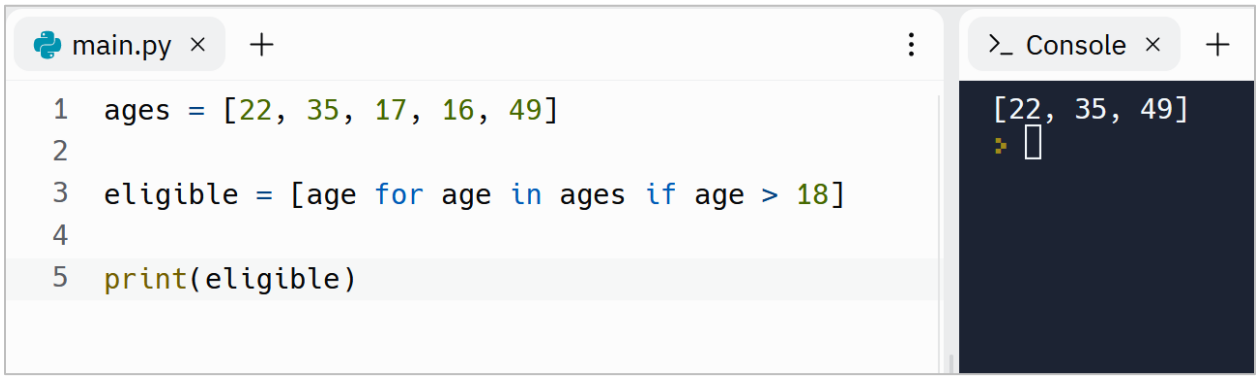

A List Comprehension consists of a pair of brackets. Inside the brackets, we define the expression. The expression works like the previous example. We can consider it like a for loop. The first part is the expression that calculates the exchanged value, then the for keyword followed by a variable name that keeps the value of the elements one by one.

We can even define a function that calculates the exchanged value and use the function inside the brackets:



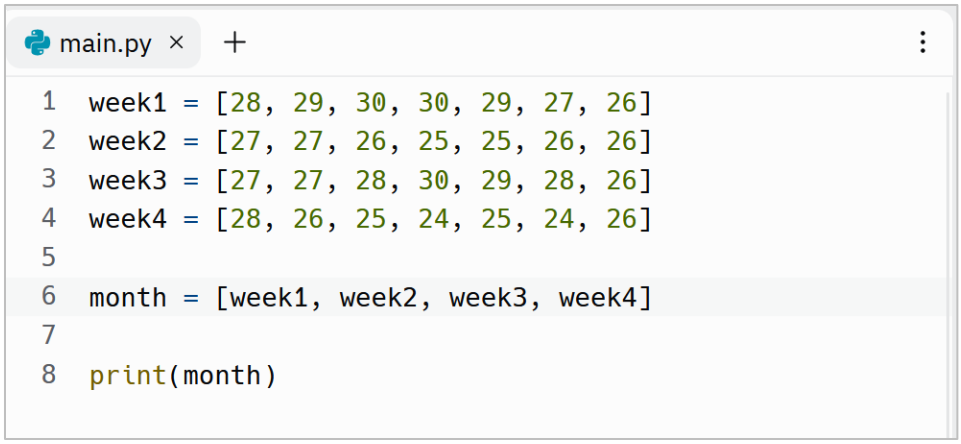

### Filter a List

We can also filter a list by using List Comprehension. To filter a List, we need to add an if statement. In this example, we copy the ages that are greater than 18 in a new List:

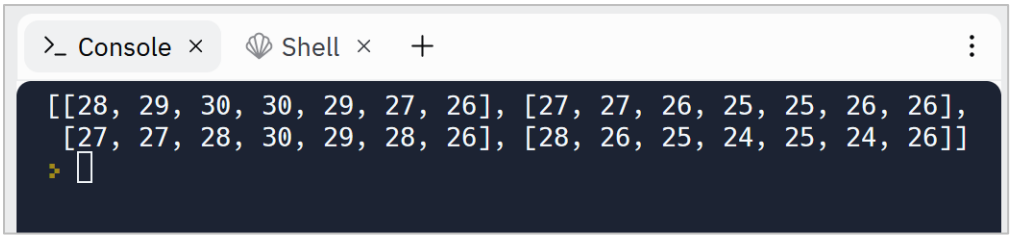



### List of Lists in Python

Now you know that lists can have items. *The beauty is that an item can be a List too. So, we can have a List of Lists*. For example, we can store the temperature of the last month like this:




We will get an output like this when print the month:



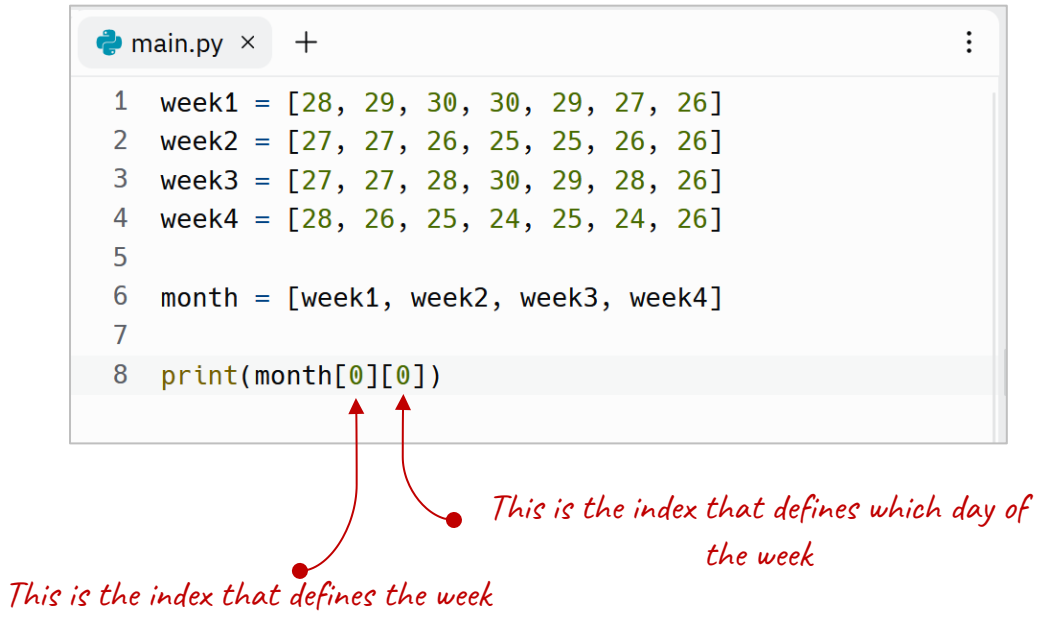

There is a subtle point here. The console tries to tell us that there are 4 Lists inside another List. It does this by using [] (square brackets) like this:

[[...], [...], [...], [...]]

So, at line 6 we have a List that contains 4 Lists as its items. Now the question is how we can access the items of the inner Lists (week1, week2, week3 and week4) through the main List (month)?

For example, how we can access the first item of the first List?

It is simple, all we need is to use [] (square brackets) two times. In other words, in this case that we have Lists inside a List, we need to define two indexes, the first index that defines which List and the second index the defines which item of that List:

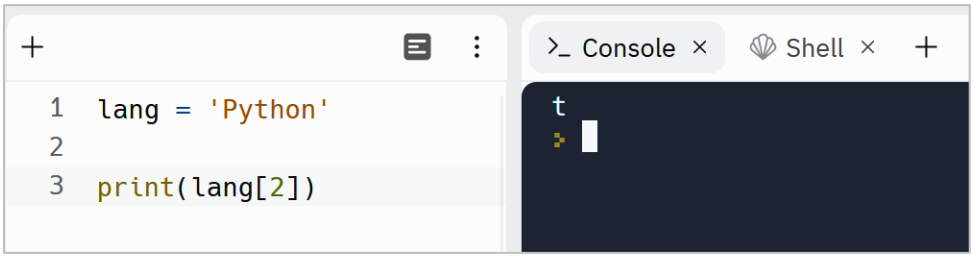



The output is 28, because the first item of the first List is 28!

So, by using Lists as items, we can organize our data even more!

### A String is like a List of Characters

We can consider a String like a List of Characters and access each item of a String by using its index:




However, we can convert a String into a List of characters by using the **list()** function:


As you see, we have a List of Characters now.

## Tuples in Python

In Python, Tuples are like Lists. It means we can do what you learned about Lists on the Tuples. *The only difference is that Tuples are immutable (unchangeable). In other words, once a Tuple is assigned, we can't modify the items of it.*

To define a Tuple that holds the temperature of the last 7 days, we can use this syntax:

```
main.py
1  items = (31, 32, 30, 29, 28, 30, 31)
2
3  print(items)
```

And the output:

```
Console  Shell
(31, 32, 30, 29, 28, 30, 31)
```

So, we use () parentheses to create a new Tuple. That's why the Console uses parentheses to display the result.

As said earlier, Tuples are immutable (unchangeable). So, we can't use functions that edit a Tuple like:

- insert()
- extend()
- remove()
- clear()
- sort()

But we can use the same functions that are used for retrieving items of a Tuple like:

- len()
- count()
- min()
- max()

We can also slice a Tuple by using the : (colon slicing) operator and there is no Tuple comprehension.

## Sets in Python

Sets are also kind of Data Structures that can hold multiple values into one variable. But Sets have their own advantages. *The strength of Sets is that all items in a set must be unique:*

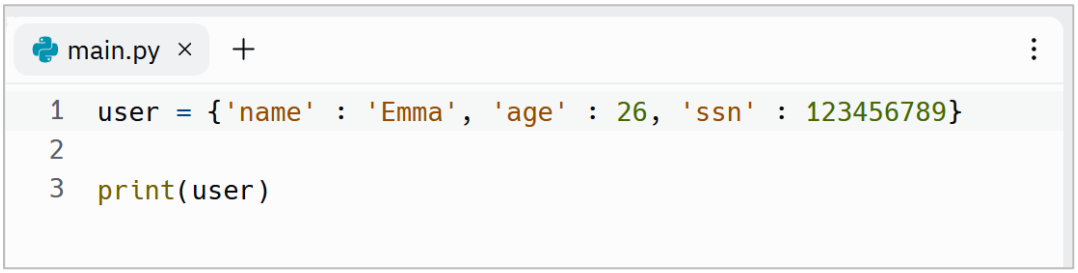

We can define a Set by using {} (curly braces). As you see, we can't have any duplicate items in a Set.

Sets comprehensions is available like what you see in the List comprehensions.

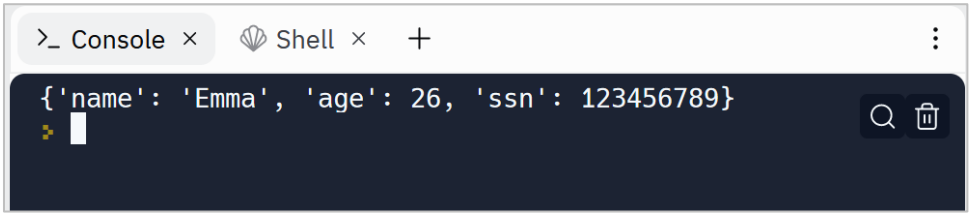
Also Sets have powerful methods that are not exist in Lists or Tuples.

## Dictionaries in Python

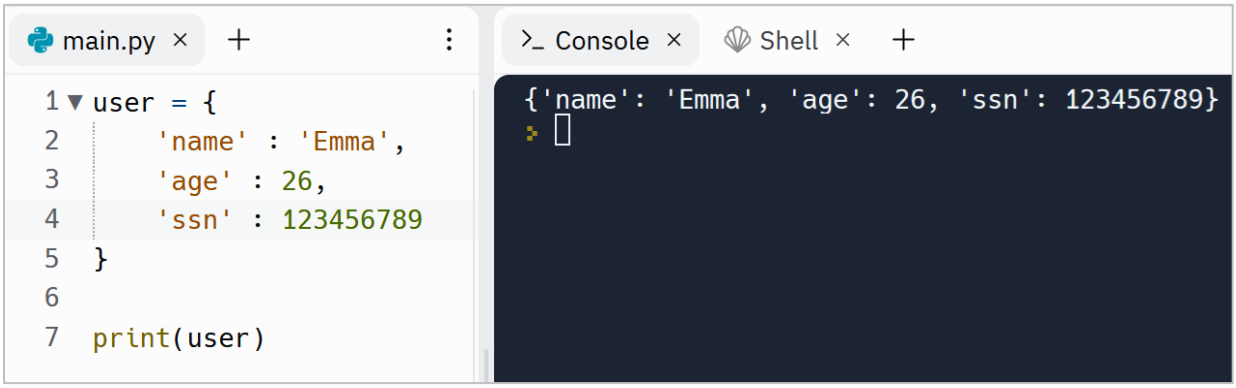
Dictionaries are used to store a collection of key-value pairs. We use : (colon) to separate a key from its value and , (comma) to separate key-value pairs from each other:



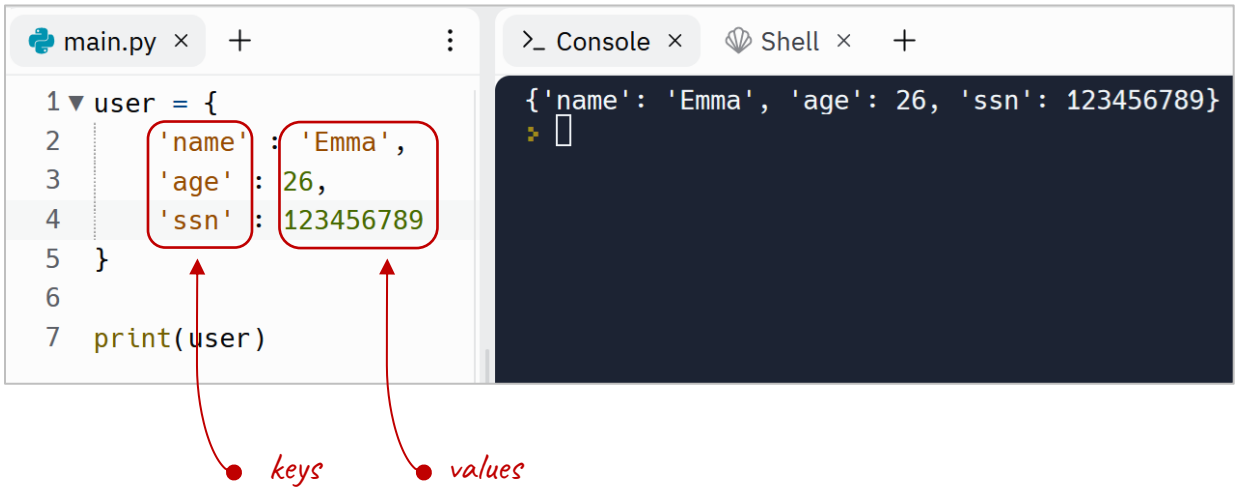
This is the output of line 3:




We can also format line 1 like this:




In this example, the **user** Dictionary has 3 key-value pairs:



Now we can access the values by using the keys:

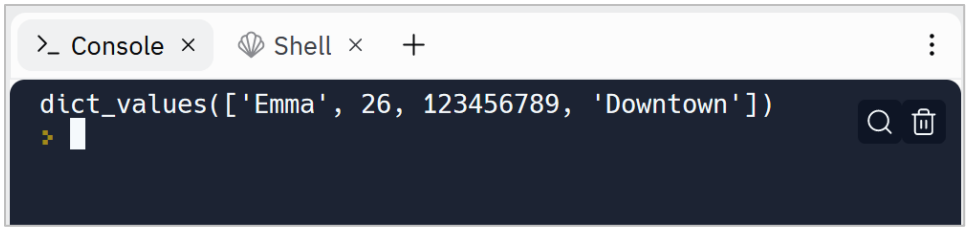

```
1 user = {
2     'name' : 'Emma',
3     'age' : 26,
4     'ssn' : 123456789
5 }
6
7 print(user['name'])
```

**Adding or updating new Items to a Dictionary**

We can add a new item to a Dictionary by using this syntax:

```
1 user = {
2     'name' : 'Emma',
3     'age' : 26,
4     'ssn' : 123456789
5 }
6
7 user['address'] = 'Downtown'
8
9 print(user.values())
```

Dictionaries are mutable (changeable), so we can add or update items. What happens at line 7 is that if a key named 'address' exists, then update its value to 'Downtown' otherwise add a new key-value pair item at the end of Dictionary:




### Dictionary comprehension

Like Sets and Lists, we can have Dictionary comprehensions. The simplest form is like this:

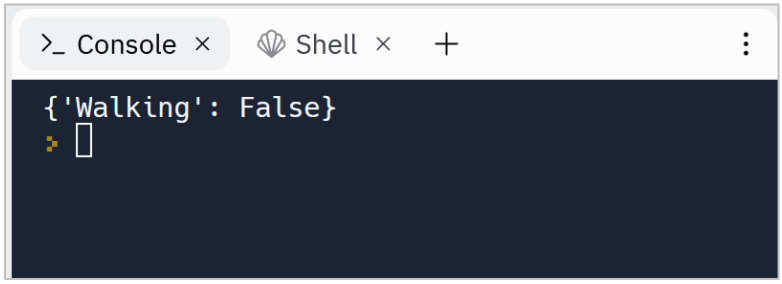

As you see, we have created a copy of the **user** dictionary by using dictionary comprehension. When we print the **copy** dictionary, we will get the same result.

However, we can use Dictionary comprehension to filter a dictionary. In this example, we have a dictionary contains tasks and their status. To filter this dictionary, we can use the if statement. For example, we can filter the dictionary to display the tasks that are False (are not yet done):

main.py

```
tasks = {
    'Reading' : True,
    'Walking' : False,
    'Playing with kids': True
}

#TODO
filtered = {key:value for (key, value) in tasks.items() if value == False}

print(filtered)
```

When we print the filtered dictionary, it displays the Walking tasks, as its value is False:




### Other Dictionaries useful functions

There are some functions associated with Dictionaries. For example, by using the **keys()** function, we can get all the keys of a Dictionary:


Also, by using the **values()** function, we access all the values of a Dictionary:


Also by using the **clear()** function we can remove all the elements from a Dictionary.

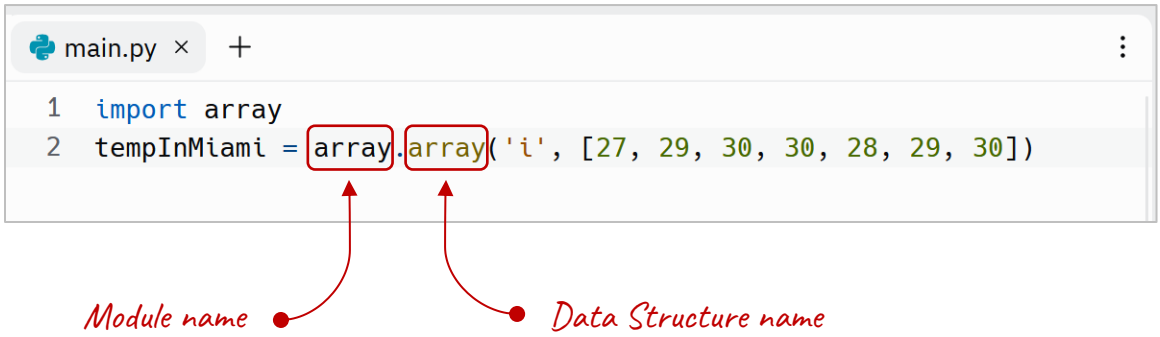
## Arrays Data Structures in Python

Arrays are Data Structures that can be used to store values of the same data type. Arrays can hold a fix number of values. Also, these values must be of the same Data Type:

```
main.py
1  import array
2  tempInMiami = array.array('i', [27, 29, 30, 30, 28, 29, 30])
```

Now we have a variable named **tempInMiami** that have 7 values. Also please note that we need to import a Module named **array** (line 1). *Because Python doesn't support Arrays natively.*

array.array means there is a Data Structure named array (the second array after the dot) that is located inside a module named array (the first array):



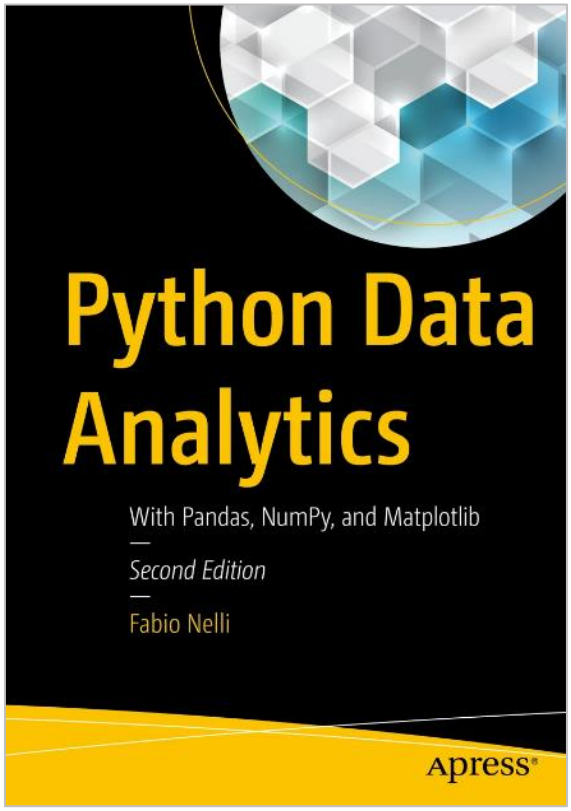

This may confuse us. Because the names are the same. array in array?! What the heck! So, let's use a keyword named **as** in Python. **We use it to assign an alias name to modules:**

```
main.py
1  import array as arr
2  tempInMiami = arr.array('i', [27, 29, 30, 30, 28, 29, 30])
```

Now we can access the array module by its alias name **arr**. But in Python, we usually use Array's replacements like **Pandas** and **NumPy**.

## Pandas and NumPy

In Data-Driven programs, **Pandas** and **NumPy** is recommended. There are libraries that adds more powerful Data Structures to Python. So, if Data Analytics is your first priority in your programs, then try to learn these libraries:




In a nutshell, if you need to do some basic tasks with data, use Python built-in Data Structures like List, Tuple, Set and Dictionary. Otherwise, try to learn Pandas and NumPy. It is good to know that we also call Lists, Tuples, Sets and Dictionaries as iterables too. Because we can iterate over their elements.

## Homogeneous vs Heterogeneous

There is an important note about the Data Structures you learned. All of them are Heterogeneous except Arrays that are Homogeneous.

So, an Array is Homogeneous. In other words, an Array must contain elements of the same data type.

Heterogeneous means items of a Data Structure like List can be a combination of other types like integer numbers, floating-point numbers, Strings and Tuples:

```
main.py
1  items = [2.2 , 'another item', [1, 2, 3], False]
2  print(items)
```

## Casting sequences

We can use list(), tuple(), set() and dict() function for type conversion. For example, to convert a List into a Tuple, we can simply use the tuple() function:


If we have a list of tuples, we can cast it into a dictionary by using the dict() function. In this example, we have a list contains two tuples that each tuple has a key and a value. By using the dict() function, we can convert the list into a dictionary:

# Python Lists Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

Python Lists Exercises – Dejavu Code Part 1

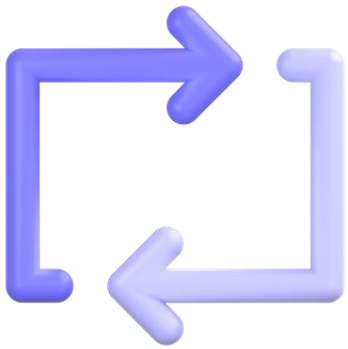
Python Lists Exercises – Dejavu Code Part 2

# Section 10

## *Loops in Python*

We usually use Loop Structures to:

- Execute a code over and over
- Iterate over a List, Tuple, Dictionary, Set and String



Two important Loop Structures in Python are For Loop and While Loop.

## Execute a code over and over

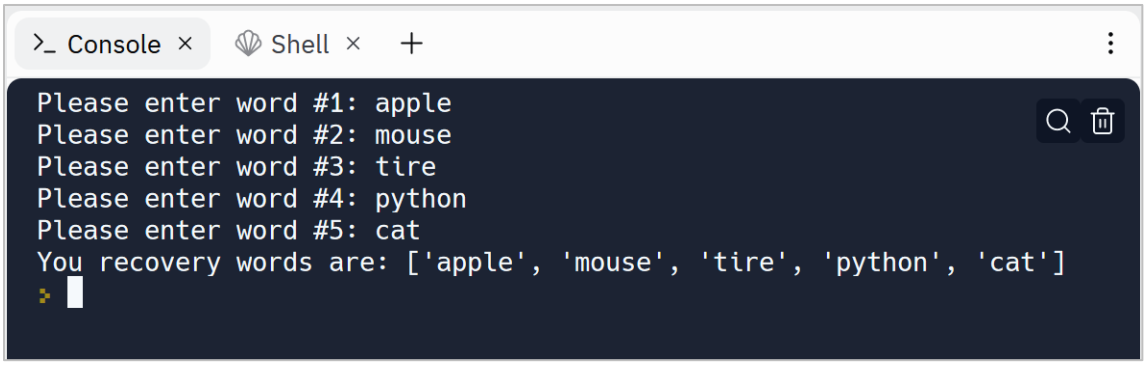
Suppose that we're writing a digital wallet and one of our tasks is to write a code that:

- Accepts 5 words as recovery words
- Store the words into a List
- Displays the List

Without loops, we'll have something like this:

```
listOfWords = []

word1 = input('Please enter word #1: ')
listOfWords.append(word1)
word2 = input('Please enter word #2: ')
listOfWords.append(word2)
word3 = input('Please enter word #3: ')
listOfWords.append(word3)
word4 = input('Please enter word #4: ')
listOfWords.append(word4)
word5 = input('Please enter word #5: ')
listOfWords.append(word5)

print('You recovery words are: {}'.format(listOfWords))
```

The output is:




But let's use the For Loop to make life easier!

## For Loop

This Loop Structure helps us to executes a code as many times as we need. That is why I'm going to rewrite the previous code by using a For Loop:

```
1  listOfWords = []
2
3  for i in range(1, 6):
4      word = input('Please enter word #{}: '.format(i))
5      listOfWords.append(word)
6
7  print('You recovery words are: {}'.format(listOfWords))
```

When we run this code, the output will be the same:


As you see, the For Loop keeps our codes readable and makes it easy to maintain the code.

A For Loop starts with the **for** keyword follow by a **variable** followed by **in** keyword and a function named range and after that the body of the For Loop:

```
3 for i in range(1, 6):
4     word = input('Please enter word #{}: '.format(i))
5     listOfWords.append(word)
```

*Body of the For loop*

So, line 3 means repeat the body (lines 4 and 5) for 5 times. range(1, 6) generates a sequence of numbers in the given range. In this example, it generates numbers 1, 2, 3, 4 and 5 because we said to start from *1 until 6*. Here is how this program works:

1- In the first repetition, the For loop assigns 1 to the **i** variable and runs the body. That is why it asks the user "Please enter word #1: ", because **i** is 1 at the moment
2- In the second repetition, the For loop assigns the next generated number to **i** that is 2. That is why it asks the user "Please enter word #2: ", because **i** is 2 at the moment
3- In the third repetition, the For loop assigns the next generated number to **i** that is 3. That is why it asks the user "Please enter word #3: ", because **i** is 3 at the moment
4- In the fourth repetition, the For loop assigns the next generated number to **i** that is 4. That is why it asks the user "Please enter word #4: ", because **i** is 4 at the moment

5- In the fifth repetition, the For loop assigns the next generated number to **i** that is 5. That is why it asks the user "Please enter word #5: ", because **i** is 5 at the moment

Now you know how to repeat a task as many times as it needed. *In programming terms, we call each repetition an iteration.*

The advantage of using loops is that we can simply maintain our programs. Suppose that we want to change the input message. We can simply change it at one place (line 4) instead of changing it 5 times:

```
1  listOfWords = []
2
3  for i in range(1, 6):
4      word = input('Please enter recovery word #{}: '.format(i))
5      listOfWords.append(word)
6
7  print('You recovery words are: {}'.format(listOfWords))
```

*We can simply apply any necessary changes*

Here is the Table of Truth for this program:

| Iteration | i |
|---|---|
| **1th** | 1 |
| **2th** | 2 |
| **3th** | 3 |
| **4th** | 4 |
| **5th** | 5 |

## The range() function

The **range()** function generates a List of numbers. This function accepts number or numbers and generates a sequence of numbers in a List. **range(1, 6)** generates a sequence of numbers in the given range:

[1, 2, 3, 4, 5]

We can also pass only one argument to this function, for example

**range(6)** that generates a sequence of numbers that starts from 0 until 6:

[0, 1, 2, 3, 4, 5]

So if we pass only 1 argument to the **range()** function, the default start value will be 0.

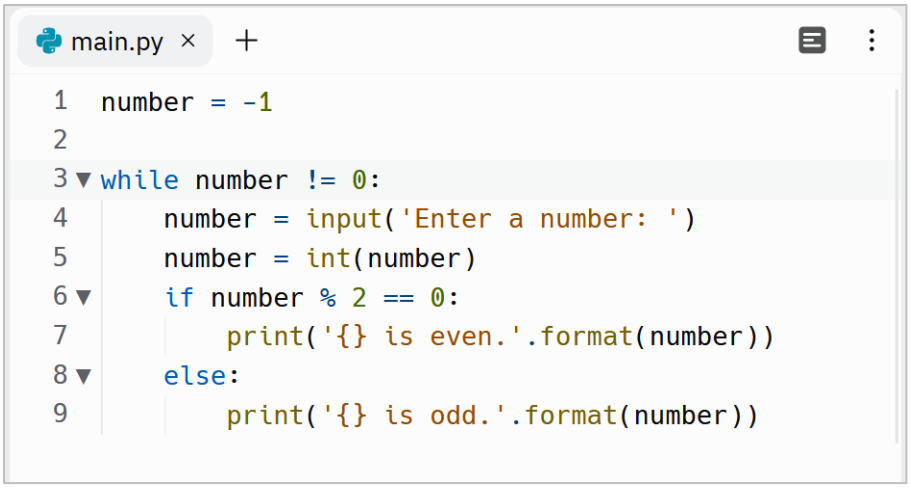
And also we can pass three arguments to the function, for example **range(1, 10, 2)**. That the first argument defines start, the second argument defines the end and third argument defines the step. So the output of this **range()** function is:


In other words, the step defines the incrementation. If we don't specify the third argument, the default value will be 1.

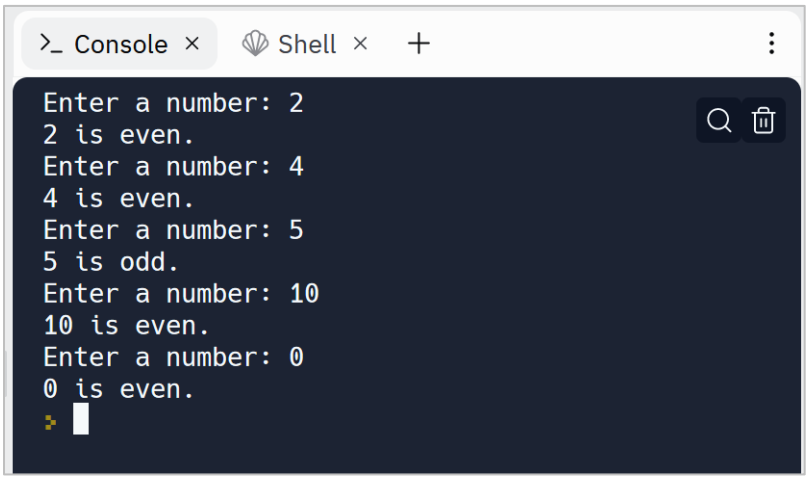
## Execute a code until a condition is true by using the While loop

By using a For Loop, we can define how many times a code must be executed. But sometimes, there are situations that we don't know the exact number of

repetitions. In such a case, we should use While Loop. A While Loop repeats a block of code as long as its condition is True:




This program accepts a number. If this number is even, then it executes line 7, and if the number is odd, it executes line 9. This program runs indefinitely, unless we enter 0. Because at line 3, we have defined while the number is not 0, run the While's body (lines 4 to 9). Here is a sample output of this program:




When the condition of a while loop stays True forever, it executes its body forever and is called infinite loop. So be careful about the condition.

*In a nutshell, the While Loop helps us to run a block of code until a condition is met. But if we need to iterate over a List, Tuple, Set or Dictionary, For Loop is the better option.*

## Iterate over a collection of values

We can also use the For Loop and While Loop to iterate over a List, Tuple, Set and Dictionary.

## Iterate over a List

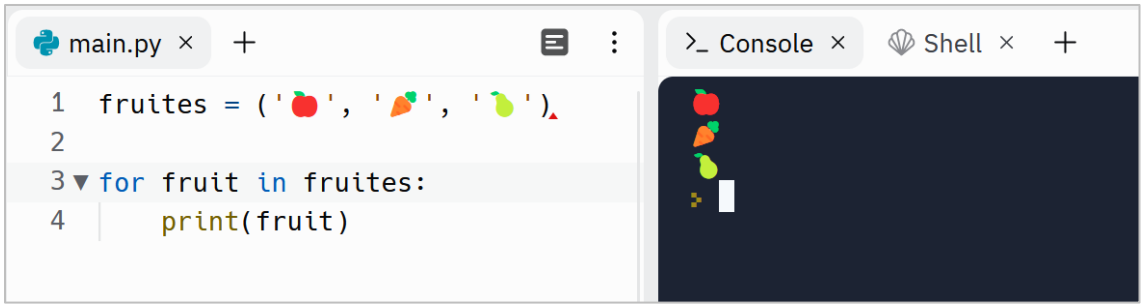
You already know how to iterate over a List. Because the **range()** function generates a List and we iterated over it in the previous examples.

So, we can simply iterate over a List by using a For Loop. The For loop executes its body for as many elements as are in the list. In this example, the list contains 3 items, so it executes the For body (line 4) three times:


We can press **Ctrl + .** to insert an emoji. *Consider each emoji as a character.*

## Iterate over a Tuple

Iterating over a Tuple is like Iterating over a List:




## Iterating over a List of Lists

We can also iterate over:

- a List of Lists
- or a List of Tuples
- or a Tuple of Lists
- or a Tuple of Tuples
- ...

The same pattern is true for Sets and Dictionaries.

In this example, we have a List of Tuples and want to iterate over it. As you see, this List las 2 Tuples as its items:


So, we use an outer loop (line 3) to iterate over tuples and use an inner loop (line 4) to iterate over items of each Tuple.

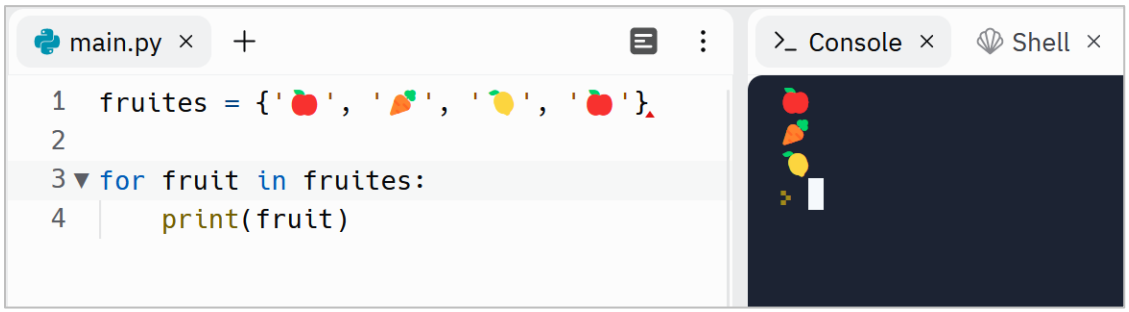
Here is the Truth Table for this example:

| Iteration | a | b |
|---|---|---|
| **1th iteration of outer loop** | ('🍎', '🍏') | '🍎' |
| **1th iteration of outer loop** | ('🍎', '🍏') | '🍏' |
| **2th iteration of outer loop** | ('🍊', '🍋') | '🍊' |
| **2th iteration of outer loop** | ('🍊', '🍋') | '🍋' |

In other words, the outer Loop executes its body (line 4) two times and in each execution (iteration), the inner loop executes its body (line 5) two times.

## Iterating over a Set

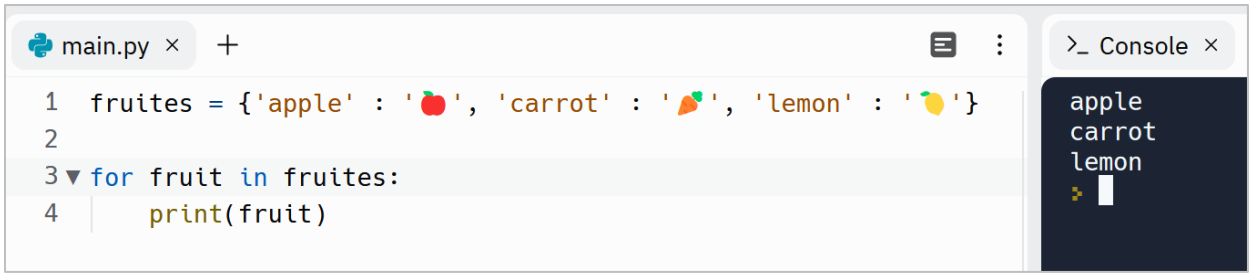
Iterating over a Set is like a List or Tuple:




As you remember, a Set removes duplicate items automatically.

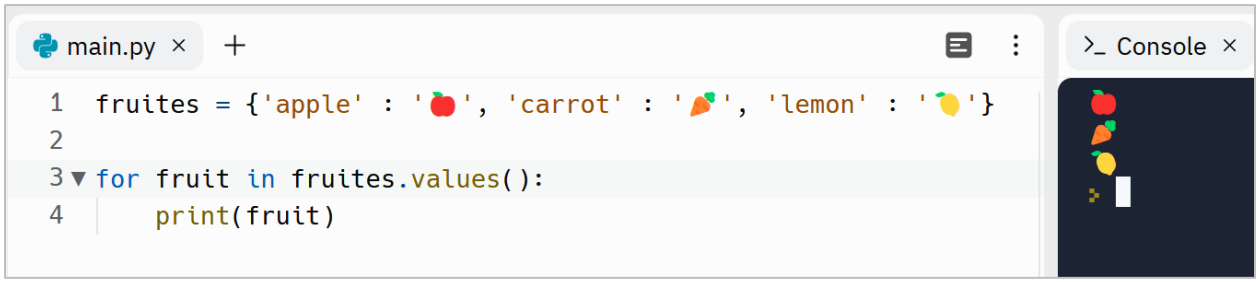
## Iterating over a Dictionary

As you know, a Dictionary is a collection of key-value pairs. When we use a For Loop to iterate over a Dictionary, the For Loop assigns the key to the variable in each iteration. That is why in this example it prints only the keys:




We can get values by using keys. To do so, all we need to do is to use the key name to get the value (line 4):


Or we can directly iterate over the values by using the **values()** function:

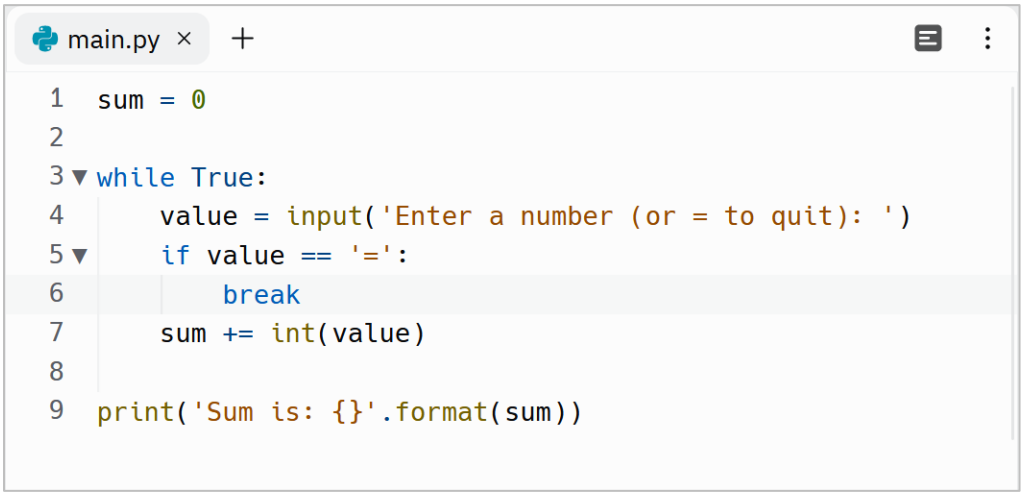



## Python Break and Continue

We can use Break and Continue to terminate or alter the normal flow of a Loop.

## Break a Loop

We can use the *Break* statement to terminate a *For Loop* or *While Loop* immediately if a certain condition is met. In this example, we use the Break to finish an endless loop. This program accepts unlimited numbers from us and adds them together. If we enter =, then it terminates the endless loop by using the Break statement:



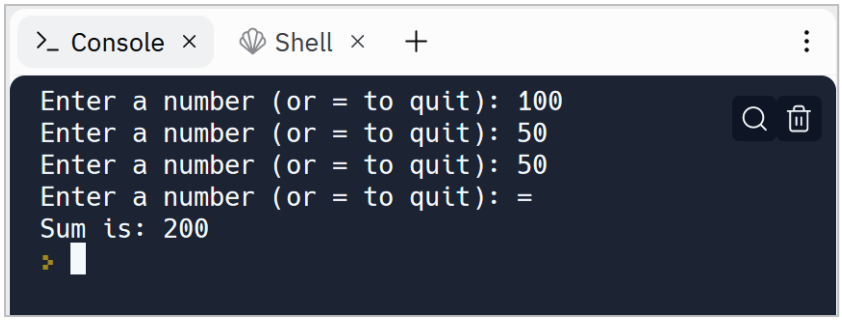

At line 1 we defined a variable named sum that will hold the sum of the numbers

At line 3 we defined an infinite loop. Because the condition is always True

At line 4 we get a value from the user

At line 5 we check if the value is =, then we must end the loop by using the **break** (line 6). Otherwise, add the value to the current value of the sum

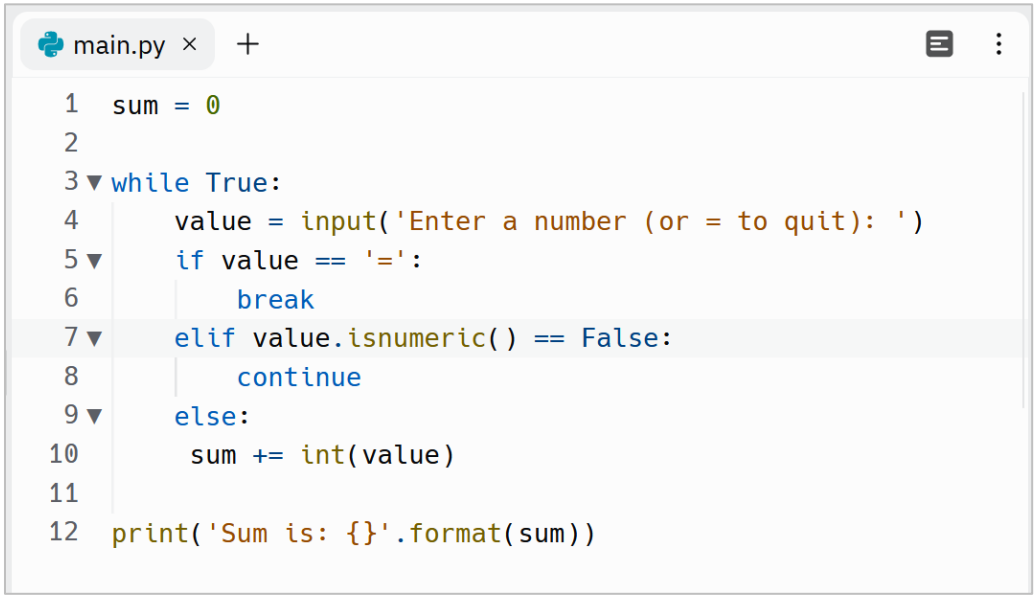
This is the output of this app:



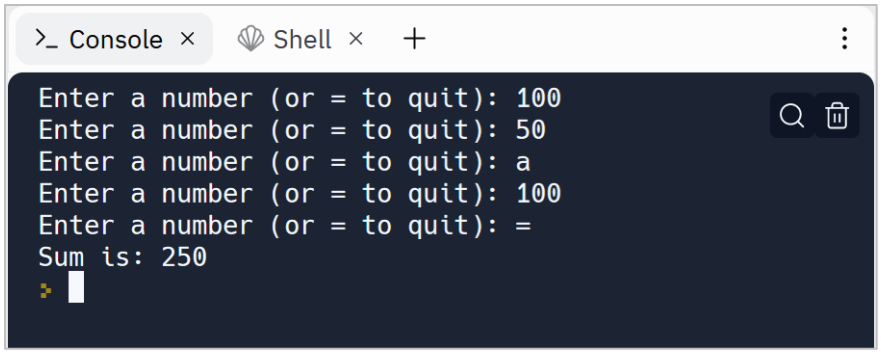

*So, when a Break happens, the Loop ignores the remained iterations and terminates the Loop.*

## Continue a Loop

The Continue statement ignores the remained block code and returns the loop execution to the beginning of the loop. For example, let's check if the value is not a number, ends the current iteration and returns to the beginning of the loop. We use **isnumeric()** method to check if all the characters are numeric:




The output is like this:

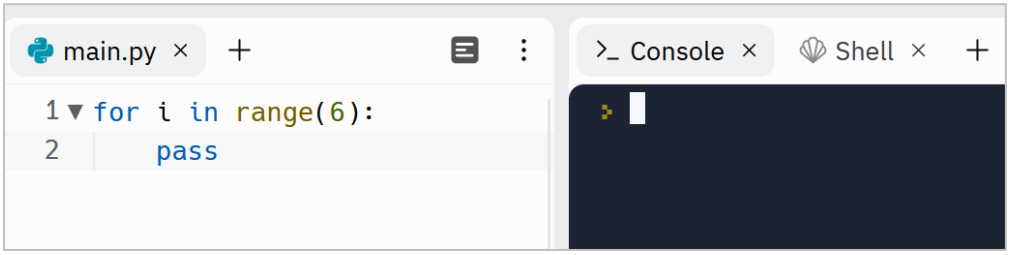



At line 7 we check if the user enters a value that is not numeric, then we ignore the rest of that iteration and moves to the next iteration by using the Continue statement (line 8).

*In a nutshell, the Break statement ends a loop completely, but a Continue statement ends the current iteration and moves to the next iteration if exist.*

## The Pass Statement

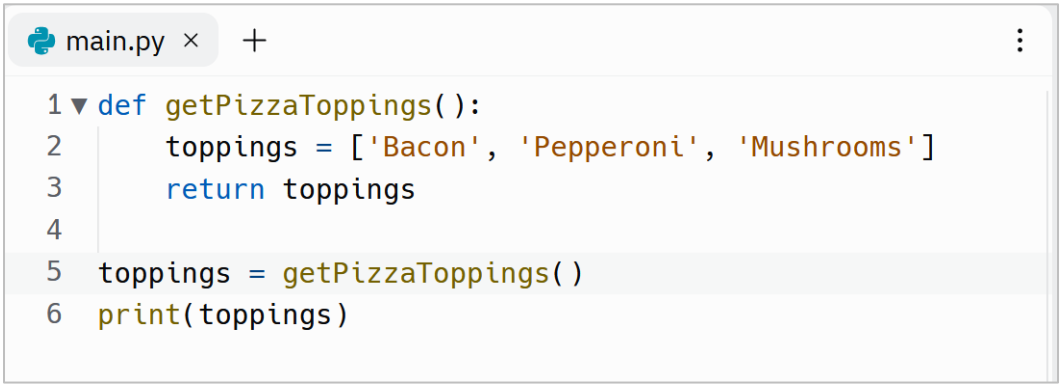
We can use the Pass statement as a placeholder for a functionality we add later. In other words, we know we need a body (for a function or loop), but for now, we don't care about the implementation:



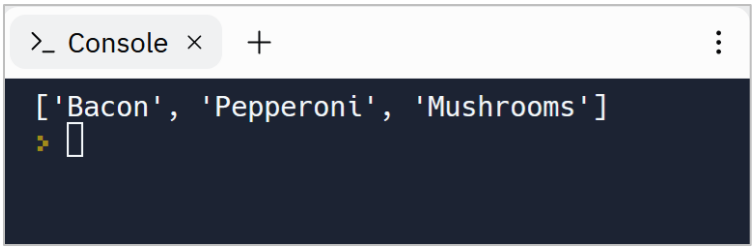

So, when the pass is executed, nothing happens. This is useful when we need to implement the main structure of our program and then implement the details.

## Return multiple values from a function by using sequences

If we need to return multiple values from a function, we can return a sequence like List, Tuple, Set or Dictionary. For example, the following function returns multiple temperatures by using a List:




By doing so, we can simply return multiple values instead of one value from a function:




It is also possible to send sequences as arguments:

main.py

```
1 def printFavorites(name, foods):
2     print(f'{name} is interested in {foods}')
3
4 printFavorites('Taylor', ['Pasta', 'Pizza', 'Rice'])
5
6 printFavorites('Alice', ['Potato', ['Vegetables']])
```

The output is

Console

```
Taylor is interested in ['Pasta', 'Pizza', 'Rice']
Alice is interested in ['Potato', ['Vegetables']]
```

# Python Loops Exercises

Are you ready to level up your Python skills? Dive into a world of hands-on learning with the exercises I've carefully crafted just for you.

Why just read about Python when you can put your knowledge into action? These exercises are designed to reinforce your understanding of key concepts, sharpen your problem-solving abilities, and ignite your creativity.

Remember, practice makes perfect. By actively engaging with these exercises, you'll not only build proficiency but also develop a robust programming mindset. Embrace the thrill of discovery, experiment with different approaches, and unlock the true potential of Python.

Don't just be a passive reader—be an active learner!

[Python Loops Exercises – Dejavu Code]

## What's next?

Looking to get a deeper understanding of Python programming? Look no further than my exclusive Premium Python Course! Join now and unlock a wealth of expert insights and cutting-edge concepts.