**The Buzzword "Data Science"**

**Data Science hype**

When Harvard Business Review stated in 2012 that, “the shortage of data scientists is becoming a serious constraint in some sectors,” they also said that the title had been around for only a few years. The term *Data Science* has been used interchangeably for statistics, analytics, business analytics, and business intelligence. We have heard differing views on it. So, what actually is Data Science?

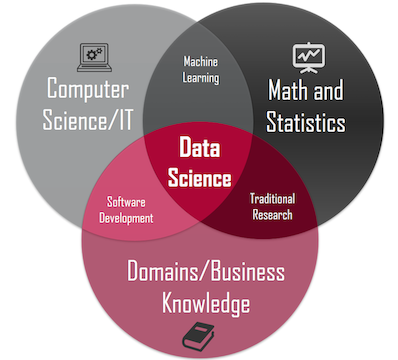


Image from https://towardsdatascience.com/introduction-to-statistics-e9d72d818745

**Data Science** is simply the study of data. It is a multi-disciplinary field that aims to extract knowledge and insights from structured and unstructured data. It makes use of statistical and mathematical methods, computer science tools, and knowledge from related fields in order to draw conclusions from the data and make decisions and predictions based on these conclusions.

**Data Science vs. Business Intelligence**

Data Science and Business Intelligence (BI) are often confused with each other. While both focus on gaining insights from the data, they differ in the kind of questions they answer. BI helps monitor the current state of business data to understand the historical performance of a business. BI is mainly used for reporting or descriptive analysis while Data Science is used for both descriptive and predictive analysis.

BI provides new information on previously known things, using some formula that is available. Data Science works with the unknown, answering data questions that nobody has answered before, without formula in hand.

**Why everyone should learn Data Science?**

We have only started to realize in the last few years the importance of data in solving problems. Data can answer questions for us in every field be it business, health, agriculture, education, sports, or many more.

For professionals who are not statisticians and programmers, they have the domain/business knowledge, but it can become a little hard for them to tap into the data and generate meaningful insights from it. Therefore, everyone should learn how to effectively use the vast amounts of data that we have for solving our business problems.

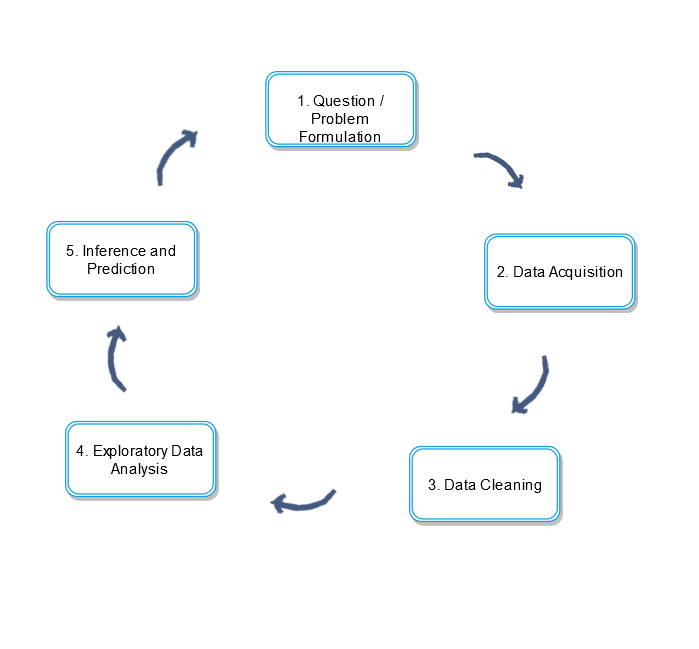
So now that we know what Data Science as a discipline is and what its objectives are, we will look at how Data Scientists go about finding meaning in data in the next lesson.

**Data Science Lifecycle**

We know that the objective of Data Science as a discipline is to extract insights and meaning from data. To achieve this goal, data scientists follow a process that is known as the **Data Science Lifecycle.**

**The Data Science lifecycle**

The **Data Science lifecycle** involves 555 steps as shown in the following figure:



**1. Problem formulation**

The lifecycle starts with a question or a problem that we face. This can be a business question or a genuine curiosity of finding the relationships between different events. For instance, Data Science has been previously used for:

* Predicting and catching fraud
* Matching organ donors to patients
* Optimal staff scheduling
* Churn prediction
* Analyzing the performance in sports
* Increasing sales for businesses.

**2. Data acquisition**

Once the problem is identified, the next step is to gather data. This requires answering some of these questions:

* What kind of data do we need for our problem?
* Do we have any data already?
* From what sources we will collect data?
* How will we manage data during and after gathering?

**3. Data cleaning**

This is a crucial step in the lifecycle. Almost all the data that we gather is untidy (contains heterogeneous values, missing values, or large errors) and full of inconsistencies. Or we may have unnecessary data that we do not need. This step takes a lot of time in the lifecycle.

**4. Exploratory data analysis**

This step is where we really get to know our data. During exploratory data analysis we find the relationships and biases in the data. This includes visualizations as well. Visualizations involve producing images that communicate relationships among the represented data.



**5. Inference and prediction**

This is where all of the statistics and machine learning comes into play. We infer from the data and make predictive models that help us in decision making.

These steps keep repeating since it is a lifecycle. All of the lifecycle from step 2 is done using different tools like *Excel*, *R*, and *Python*. In the next lesson, we will look at which tool is best for Data Science.

# Python for Data Science

## Python as a tool for Data Science

Python is one of the most popular languages. It is being regarded as the language of Data Science these days. It has overtaken other languages and tools such as R, SAS, Excel, and RapidMiner. So, what makes it so popular?

* **General-purpose language**: It is a general-purpose language and is lightweight and fast. Therefore, it can be used for many different purposes.
* **Support for Data Science**: Python was not built for Data Science but its general-purpose nature and the support of the community in providing third-party tools like Numpy, Pandas, matplotlib, and Scipy has made it the most popular in data science.
* **Flat learning curve**: Python has a very simple and easy to learn syntax and programming requirements which make it a popular choice for people who are not programmers. It is known for its usability.
* **Large active community**: Python has a very big active community of users. If someone faces an issue while working, there is a high probability that there will be solutions for it easily available on the Internet.

## Python for Spreadsheets

Most of the data that professionals and non-programmers use in their day to day work are stored in spreadsheets. Most of them use Excel for their data analysis needs. So, why switch to Python from Excel?

#### Large volumes of data

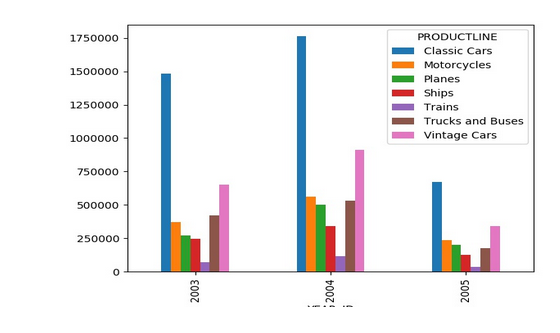
These days we have very large volumes of data at our disposal and we have the opportunity to tap into the data and draw useful conclusions from it. But using Excel for very large datasets is not feasible as Excel can get slower and messier while handling large amounts of data. On the other hand, Python is efficient in handling and cleaning large amounts of data. Therefore, Python is preferred when we have a large amount of data.

#### Reproducibility and automation

Python code is easier to reproduce. In Data Science, a lot of times, repetitive analysis is required. Repeating tasks in Python is hassle-free. The fact that Excel has a Graphical User Interface makes it more appealing, but that can be a hindrance when you have to repeat the same analysis. Even if you use macros and VBA, Python will still provide a smoother experience.

#### Advanced custom visualizations

Making custom visualizations that are easy to read and convey the analysis properly to the audience is a critical task in Data Science. Python is very good at making custom visualizations with its matplotlib package. Excel can not match the level of customization that Python provides.



#### Better at fixing and finding mistakes

Mistakes are inevitable in any kind of work specifically in Data Science but Python will make it easier for us to find errors and mistakes. In contrast, tracing your steps in Excel can be difficult and time consuming.

#### Open source community

Python has a big open source community that is always active and aimed at improving the experience and usability of Python. People have been using Python for the analysis of spreadsheets a lot in the last few years. Solutions and workarounds for most issues that you might face are easily available on the Internet.

#### Advanced statistics and machine learning

Python is miles ahead of Excel in providing advanced machine learning and statistics capabilities. Machine learning and statistical methods are necessary for Data Science in conducting predictive analysis and decision making.

In this course, we will be using Python as a tool for Data Science. We will look at the basics of Python in the next chapter to start our journey with Data Science.

**Hello World**

**Syntax of a language**

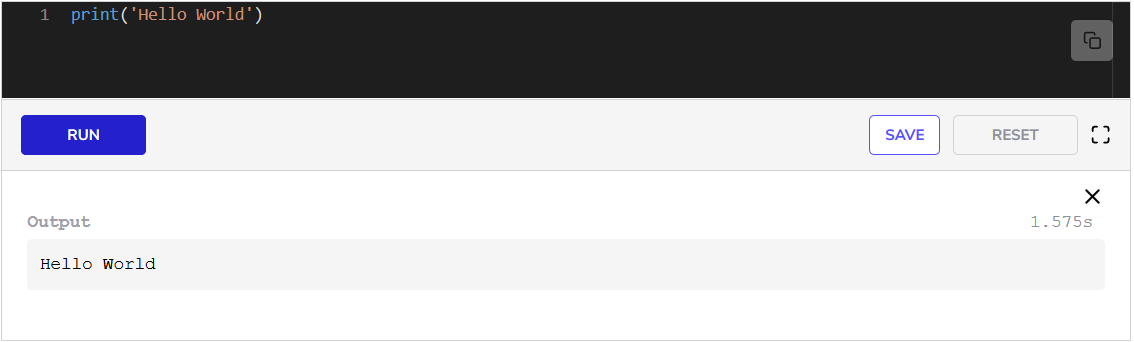
When we speak or write something in English, we follow certain rules and regulations of the English language. In the same way, there are certain rules that we must follow to write programs in a programming language. The spelling and grammar rules and regulations of a programming language are called **syntax**. Python has a very simple syntax that is human-readable which makes it easy for us to make the computer accomplish what we want it to.

**print statement**

We will start by learning how to display text and numbers on our screen. Every language has a different way of displaying or showing things to the user. Python does this with a print statement. The syntax is simple; you write the keyword print, followed by a parenthesis. In the parenthesis, you write whatever you want it to display.

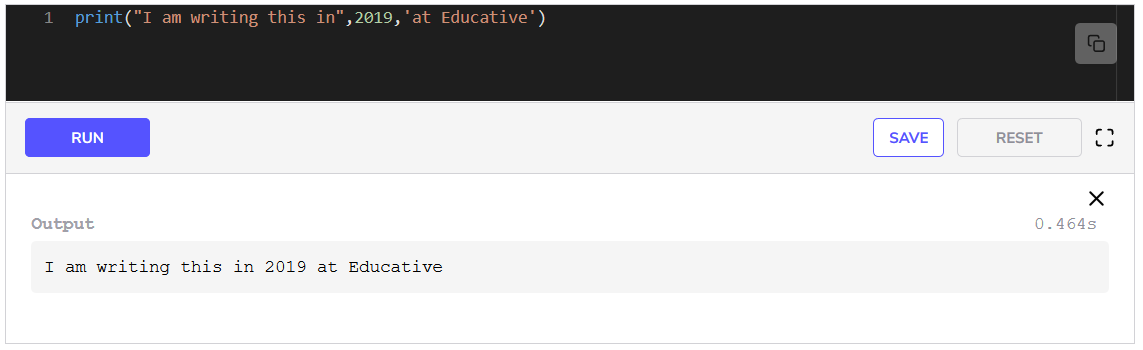
print(data)

We can write any data we want to be displayed inside the parenthesis. Let’s see an example below. If you press the **RUN** button below, you will see **Hello World** written on the screen.

In the above example, we wrote our **Hello World** inside single quotes because it is text. Any text that we want to be printed should be written in quotes. You can use both single quotes and double quotes in Python. But what about numbers?

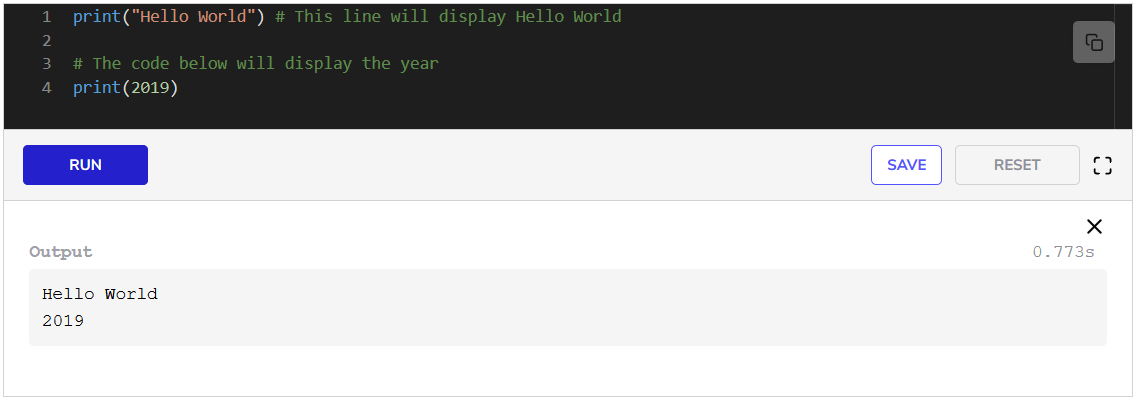
If you run the code now, you will see **777777** printed on the screen. For numbers, we do not have to enclose them in quotes. This was easy. But what if we want to print more numbers and text?

Now when we run the code, we can see the three different statements printed on 3 lines. Each time we use print, the output moves to the next line. We can also print multiple numbers and texts in the same line by separating them with commas.

When we run the code, we can see that everything is printed on the same line.

**Comments**

A very handy feature that all programming languages provide is commenting. We can add comments that explain things alongside our code so if anyone sees our code later, they know what the code is trying to accomplish. In Python, we can add a comment with the **#** character. Everything following the # character is treated as a comment instead of code.

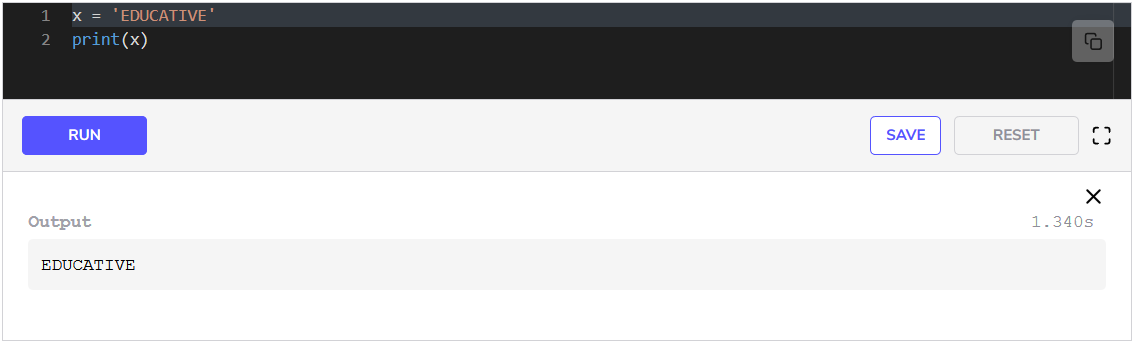
As we see above, the comments did not affect our code whatsoever. But they are of great help when writing code.

This brings us to the end of this lesson. We have written our first program in Python. In the next lesson, we will cover two fundamental concepts, *variables* and *data types*.

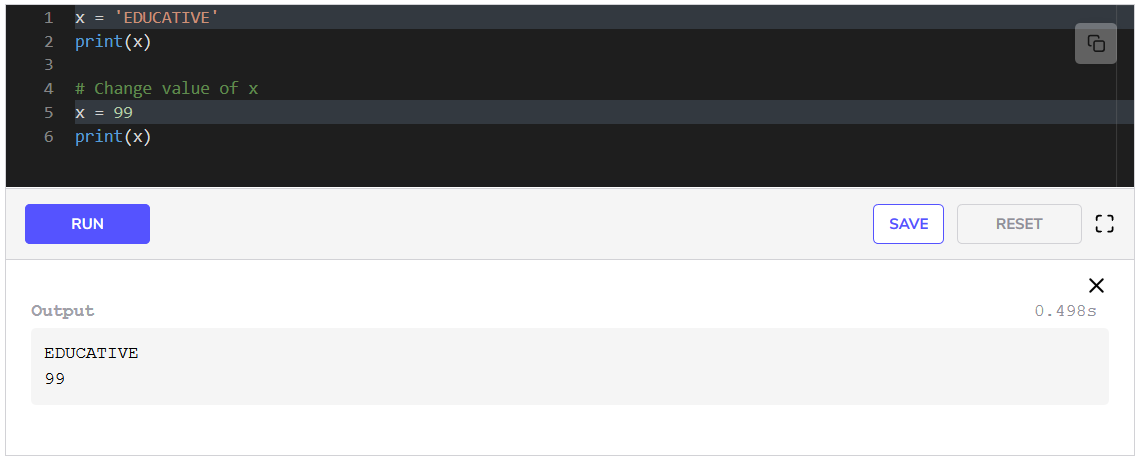
# Variables and Data Types

## Variables

A **variable** is a name to which a value can be assigned. It is basically a placeholder to store information. We can store information in a variable and refer to it later by using the variable name. Variables are assigned values using the = operator.

When you run the above code, you will see **EDUCATIVE** printed on the screen. In **line 1**, we created a variable which we named x and assigned it the value EDUCATIVE. In the next line, when we said to Python to print(x), it printed what was assigned to the variable x.

As the name suggests, the value of a variable can be changed. Let’s see an example of that below.

We have extended the code of the above example. When **line 1** runs the variable x is assigned the value EDUCATIVE. The value of x is printed in **line 2**. But in **line 5**, we assign the value 99 to x. Now when we print x in **line 6**, the changed value of x, i.e., 99, is printed on the screen.

### Naming rules

There are some naming rules and conventions that need to be kept in mind for naming variables. If these rules are not followed, Python will give an error.

* The name can start with an upper- or lower-case alphabet.
* A number can appear in the name, but not at the beginning.
* The \_ character can appear anywhere in the name.
* Spaces are not allowed. Instead, we must use [snake\_case](https://en.wikipedia.org/wiki/Snake_case) to make variable names readable.
* The name of the variable should be something meaningful that describes the value it holds, instead of being random characters.

## Data types

We can have information or data in different forms. For instance, we have numbers, alphabets, alphanumeric values, etc. Therefore, we also have data types in Python. The **data type** of an item defines the type and range of values that an item can have. There are three main data types:

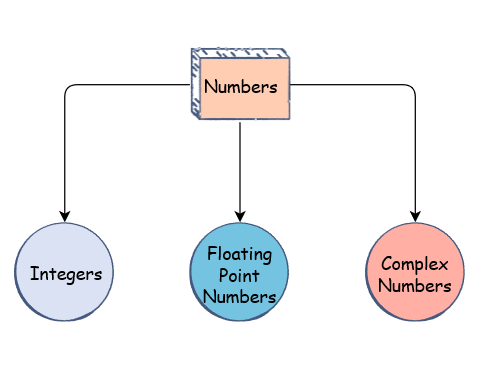
* Numbers
* Strings
* Booleans

Let’s explore these data types one by one.

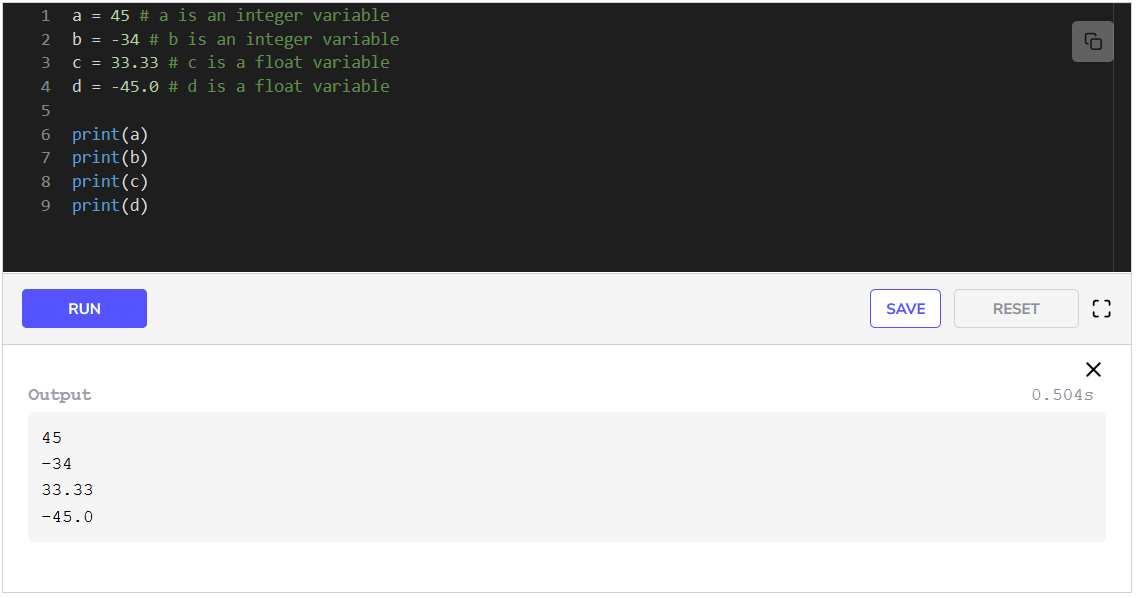
### Numbers

Numbers are divided into three categories in Python.

* **Integers**: These are positive and negative whole numbers including 000.
* **Floating-Point Numbers**: These are numbers with a fractional part.
* **Complex Numbers**: These are complex numbers with a real and imaginary part.



We will focus on Integers and floating-point numbers since we don’t need or complex numbers in this course.

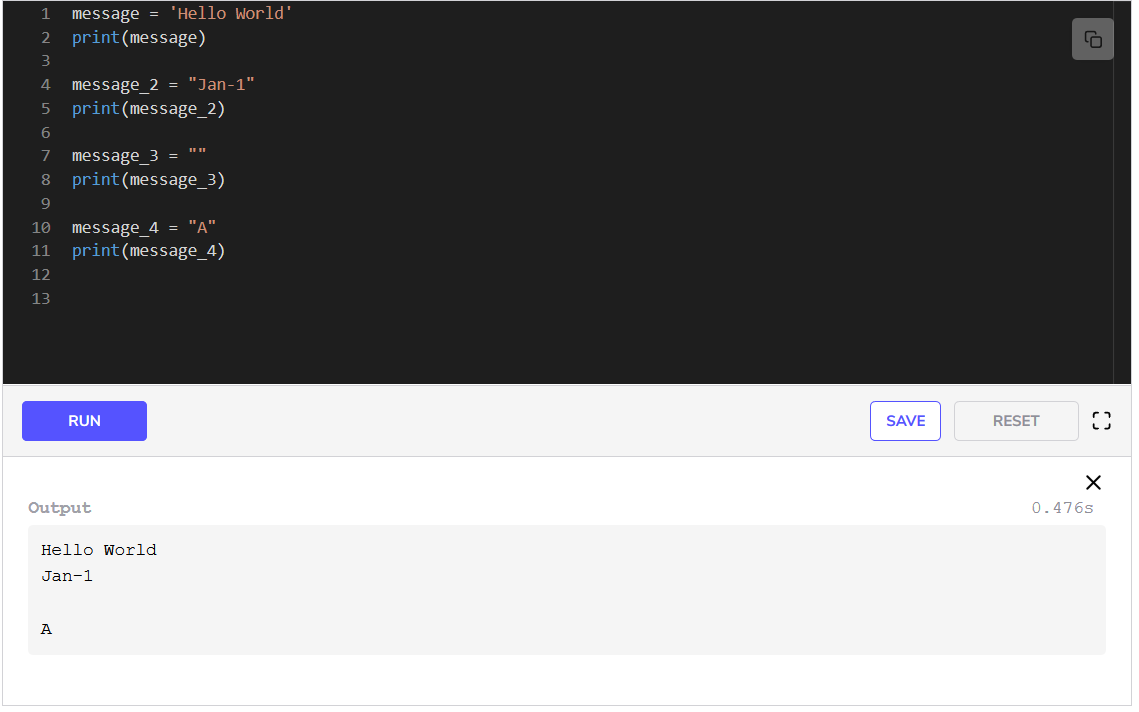




We can see in the above code that a and b are integer variables while c and d are float variables.

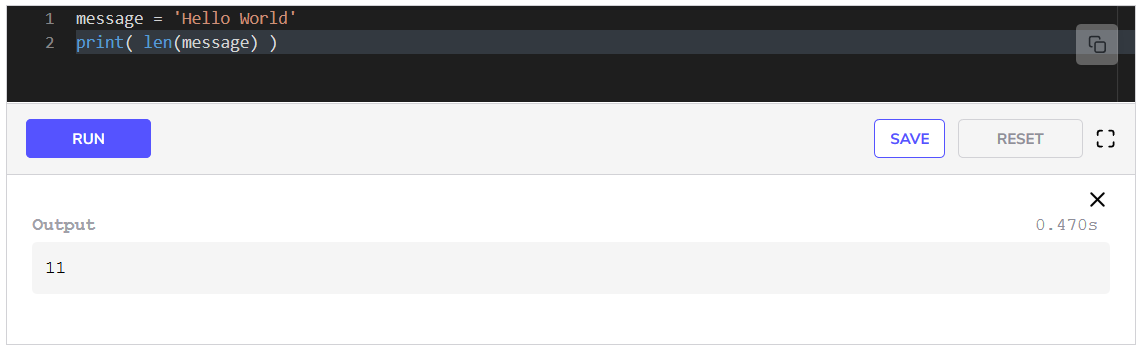
### Strings

A **string** is a collection of characters inside single or double quotation marks. In the previous lesson, when we printed **Hello World** we printed a string.

When we run the above code, we see that at first the value of message, i.e., **Hello World** is printed. Then after that, the value of message\_2 is printed. Strings can contain numbers too, as is the case with message\_2. Then we set the value of message\_3 to an empty string. When we print message\_3 a blank line is printed, which means it considers empty quotation marks as a valid string. Printing message\_4 shows us that single characters are also valid strings.

#### Length of a String

The number of characters in a string, including spaces, is known as the **length** of a string. For instance, the length of the string “name” is 444. The length of a string can be retrieved by using the len statement which has a similar syntax to that of a print statement.

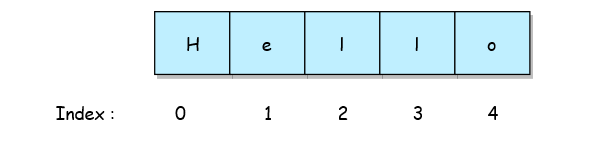
When we run the above code, we see the length of string, i.e., **11**, printed. Focus on **line 2** here. In **line 2**, we have used two statements, the print and the len statement. Let’s look at how this line is executed.

len(message) gives us the value 11, which is printed on the screen because it is inside the parenthesis of the print statement.

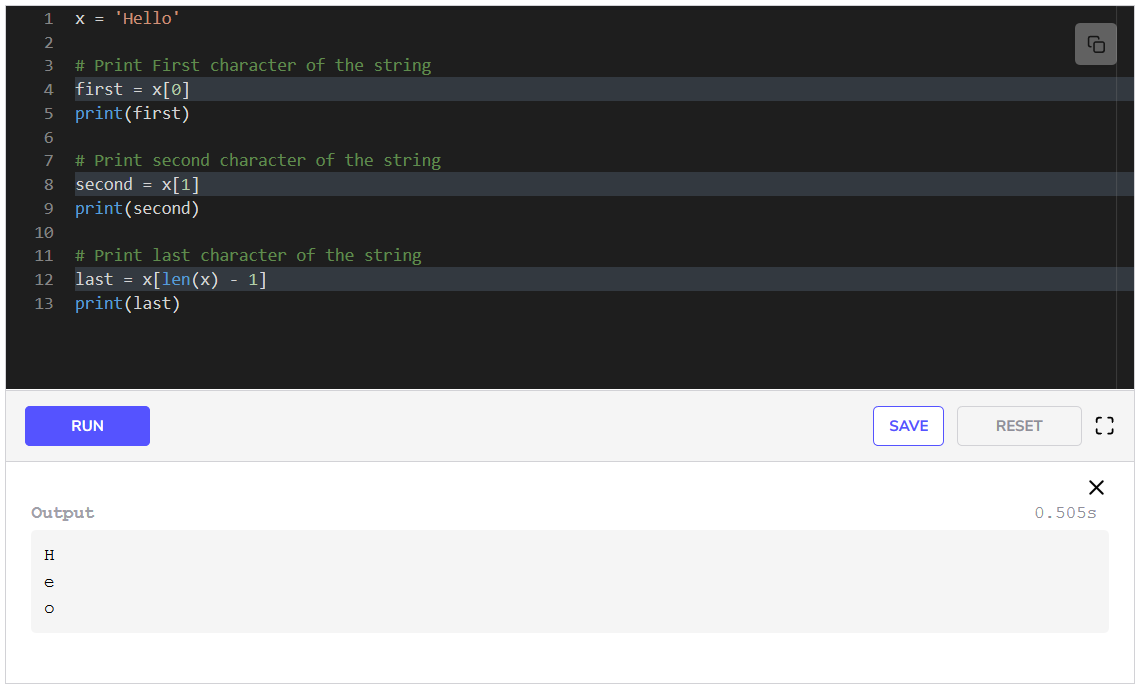
#### Indexing

In a string every character has a fixed position. The position of a character in a string is known as an **index**. We can access a character in a string using its index.

Index values start from 000. This implies that the index of the last character in a string is n−1, where n is the length of the string

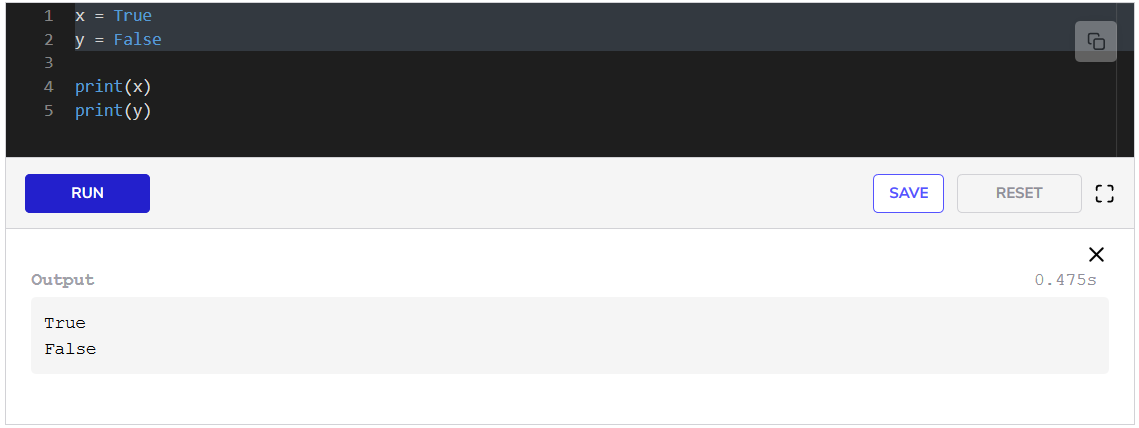


To access a character of a string, the index must be enclosed within [] appended to the name of the string variable. Let’s see an example.

In the above code, we create a string variable, x, with the value Hello. Then in **line 4**, we access the first character of the string, x, by writing x[0]. Here 0 is the index of the first character. We then assign this to a variable first. In the same way, we store the second character, e, in the variable second in **line 8**. To access the last character, we have to give the last index which is length -1. Therefore, we give len(x)-1 as the index and store the last character in the variable last in **line 12**.

### Booleans

**Boolean** data type also known as **bool** is used to store only two values: True and False. In Python, we can simply use True and False to represent a bool. Boolean variables come in handy when comparing different things as we will see in later lessons and chapters.

In **line 1** we assign a value True to variable x. We assign the value False to variable y. We can see from the outputs of the print statements that these are printed as True and False.

This is it for variables and data types. In the next lesson, we will study operators.

**Operators**

**Operators**

**Operators** are used to perform arithmetic and logical operations on data. They use the provided data and give results. Some operators follow the *in-fix* (a **+** b) notation while some follow the *pre-fix* (**-** a) notation. With **in-fix** notation, the operands are placed on the left and right side of the operator. With **pre-fix** notation, the operator is placed before the operand. We will look at the different kinds of operators in Python in this lesson.

**Arithmetic operators**

Arithmetic operators are used to perform arithmetic operations such as addition, multiplication, division, etc. They can be used in conjunction with each other, hence they have an order of *precedence* as well. Some of the most common operators are listed below in order of precedence.

* (): Parenthesis. Whatever is in the parenthesis will be evaluated first.
* \*\*: Exponent (**In-fix**)
* %,\*,/,//: Modulo, Multiplication, Division, Floored Division (**In-fix**)
* +,-: Addition, Subtraction (**In-fix**)

Let’s look at an example.

In **line 1**, we see an example of addition. In **line 2**, the subtraction operator is being used. We can see the printed result of both these lines. Then there is an example of precedence in **line 5**. The expression in parenthesis is being evaluated first and then the rest of the expression is evaluated. The answer is stored in the variable result. Let’s focus on how this expression is evaluated

After evaluating the expression in the parenthesis, there are a bunch of operators that have the same precedence. So, Python decides to go from left to right. It first evaluates floored/integer division //, then multiplication \*, followed by division /.

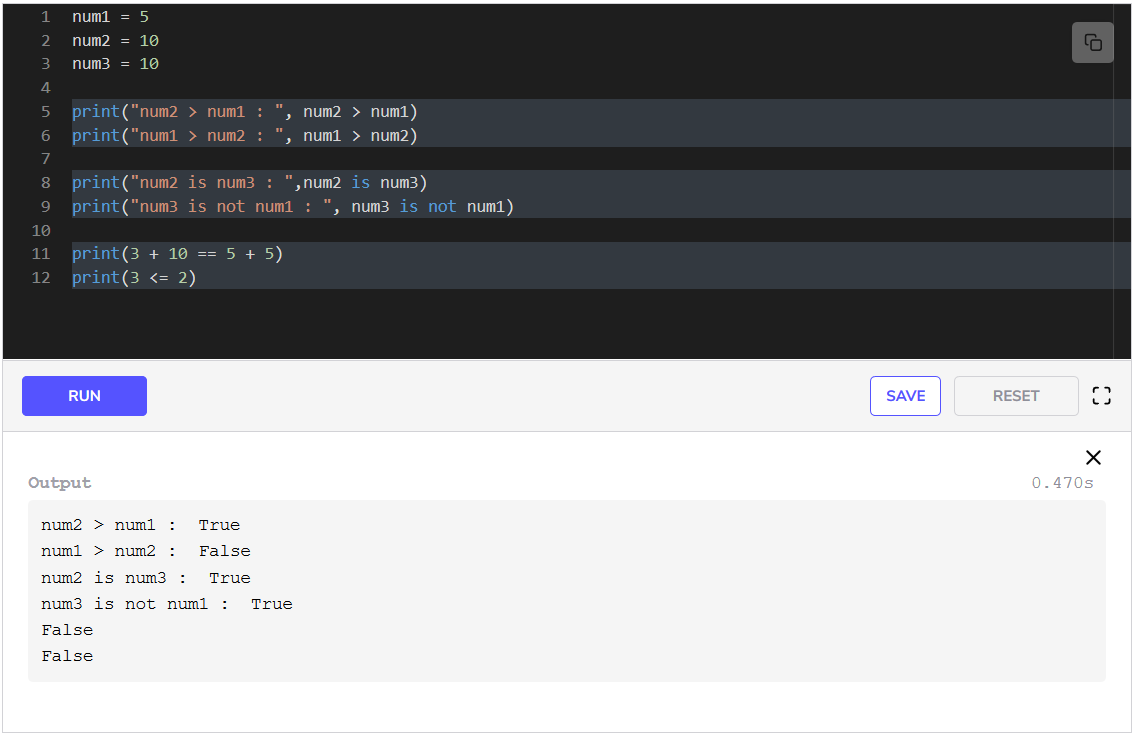
In the next line, we create another variable x and assign it the value 3. In the end, we print result in **line 7** and the result of the addition of result and x in **line 8**.

**Comparison operators**

Comparison operators are used for comparing values mathematically. They give answers in Booleans i.e. True and False. Some of the most common operators are listed below in order of precedence.

* <: Less than.
* >: Greater than
* ==: Equal to
* !=: Not equal to
* <=: Less than or equal to
* >=: Greater than or equal to
* is: equal to
* is not: not equal to

Let’s look at an example.

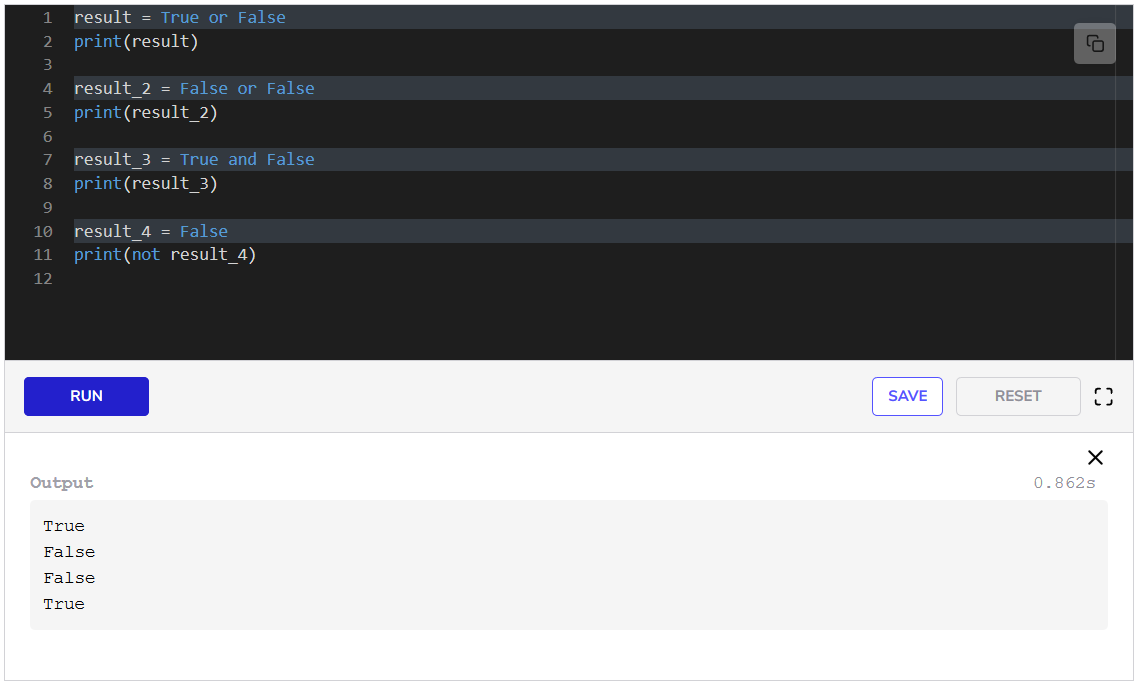
In the first three lines, we create and assign values to variables num1, num2, and num3. In the rest of the code, we evaluate expressions that are either True or False. In **line 5**, we check whether num2 is greater than num1. In **line 6**, we check whether num1 is greater than num2. In **line 8**, we check whether num2 is equal to num3. In **line 9**, we check whether num3 is not equal to num1. In **line 11**, the expressions to the left and right of the == operator are evaluated first. Then equality is evaluated. 13==10 evaluates to False. Hence, False is printed. In the same way, the expression in **line 12** is evaluated to False.

**Logical operators**

Logical operators are used for evaluating Boolean logic expressions. Logical operators consist of:

* and: Evaluate the **AND** between two Booleans (**in-fix**).
* or: Evaluate the **OR** between two Booleans (**in-fix**).
* not: Evaluate the **Not** of a Boolean (**pre-fix**).

Let’s look at an example.

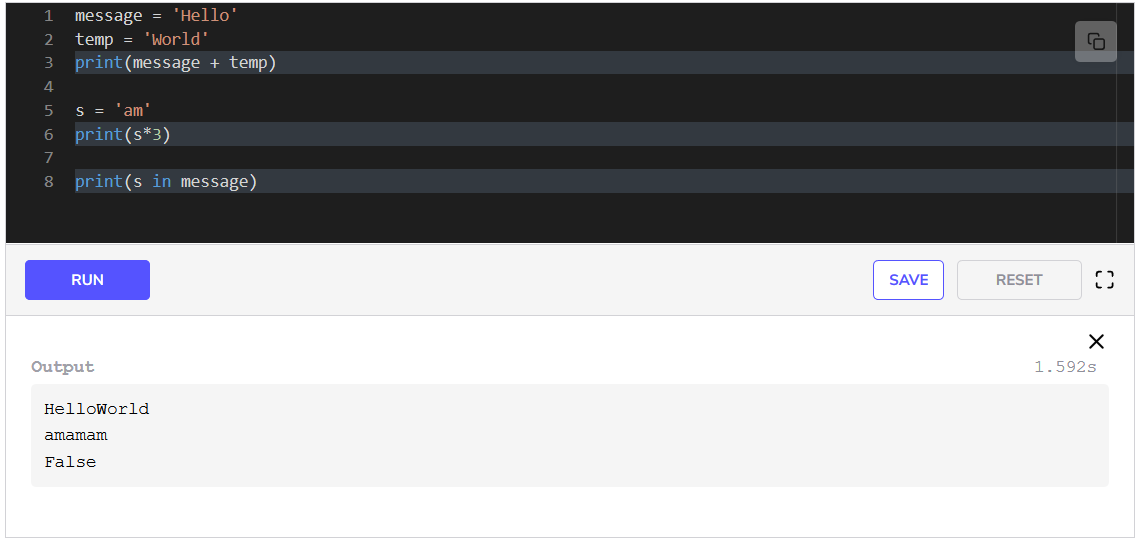
In **line 1**, we create a variable result which stores the result of the expression (True **OR** False). In the next line, we print the result. In the same way, result\_2, result\_3 and result\_4 store the results of the Boolean expression which are printed in the following lines.

**String operators**

Some operators can be used with strings to perform different operations. Some of the most common operators are:

* +: Used to *concatenate* (join) two strings (**in-fix**).
* \*: Used to multiply a string. Repeat the string. (**in-fix**).
* in: Used to search in a string. Returns True/False (**in-fix**).

Let’s see an example.

In **lines 1-2** we create two string variables. We use the + operator between them and print the result. The strings joined to form a single string.

In **line 5**, we create a string. We repeat the string in pattern with the \* operator and show the result in **line 6**. Note that the operator \* does not modify the original string s, rather it creates a copy of it with the repeating pattern.

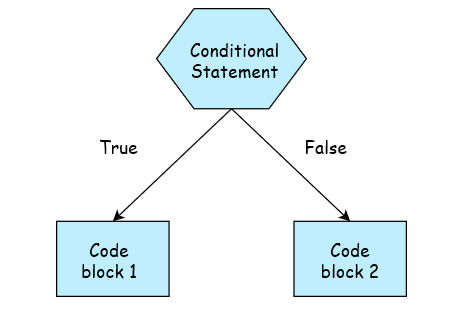
The operator in is used to search a string in the other string. In our case, we see that the string s is not in message, hence False is printed on the screen.

This brings our discussion on operators to an end. In the next lesson, we will look at conditionals.

**Conditional Statements**

**Conditional statements in Python**

A **conditional statement** controls the flow of execution in a program based on a Boolean expression. If the expression evaluates to True, then a different piece of code might execute, If it is False, then some other piece of code will execute. This is also explained in the following illustration.



These statements give the computer the ability to decide the flow of execution of the program based on the information it receives. There are three kinds of conditional statements:

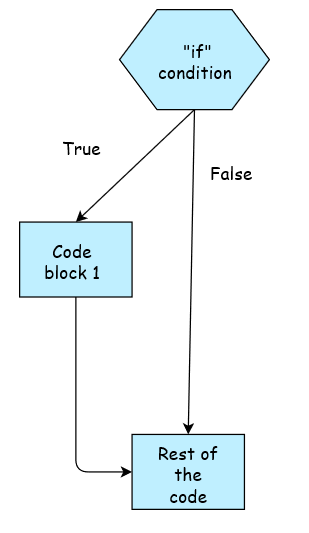
* **if**
* **if-else**
* **if-elif-else**

**if statement**

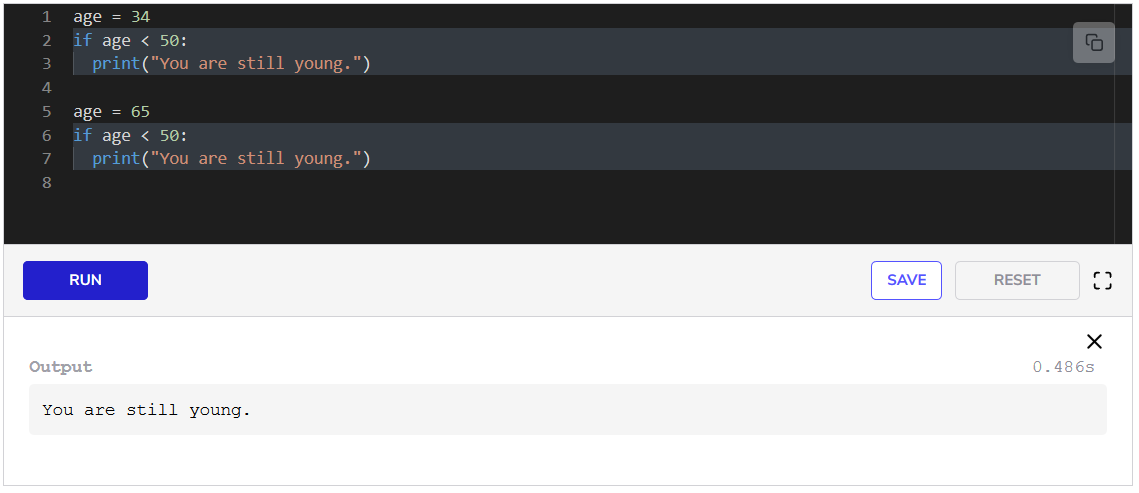
It is the simplest conditional statement. The structure of the statement is:

if (condition):  
  code

If the condition provided is satisfied, then the code is executed, otherwise, it is skipped. The colon, :, after the condition is necessary as it indicates where the condition ends and the code block begins. Note that the code after the : will be indented one tab space to the right. All of the code inside the if block after : should be at the same indentation level. Otherwise, Python will give an indentation error.



We can see the logic of the if statement from the above diagram. Now let’s look at some examples.

In the **first line**, we set age to 34. In the next line we write the keyword if followed by the condition. The condition here is whether or not age is less than 50. When **line 2** is executed and the condition is satisfied, execution jumps to **line 3**. Here the message, “You are still young,” is printed.

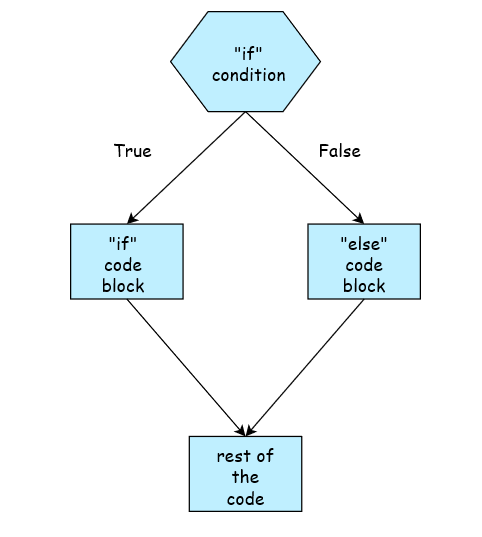
After that, we change the value of age to 65. The same if condition is written in **line 6**, but this time the condition is not satisfied. Therefore **line 7** is skipped.

**if-else statement**

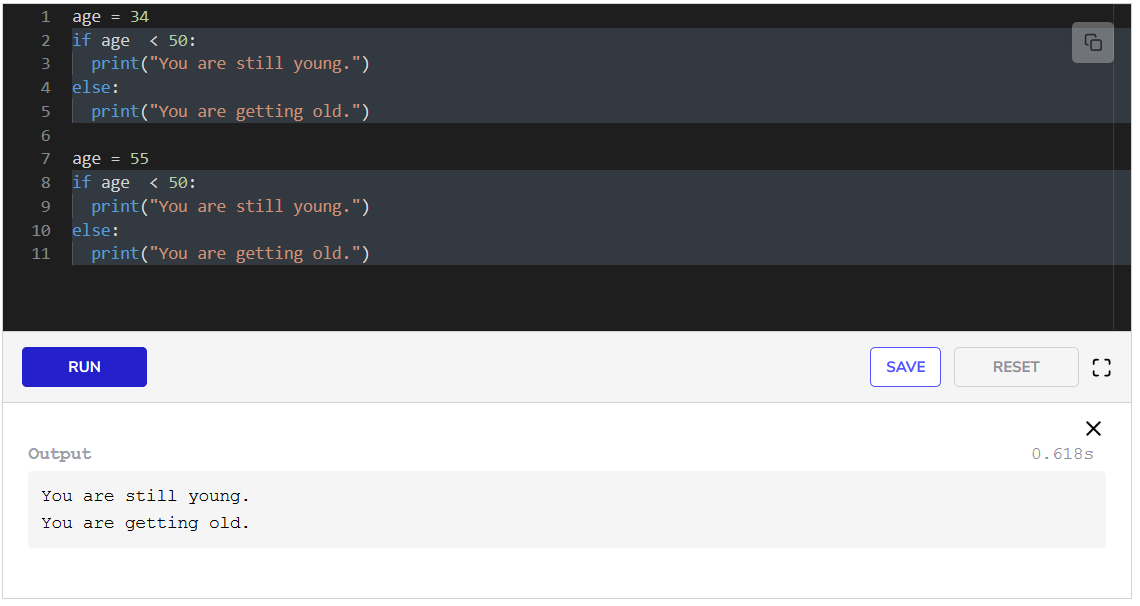
The structure is:

if (condition):  
  code  
else:  
  code

If the condition provided is satisfied, then the code is executed, otherwise the code after else is executed. The else keyword will be on the same indentation level as the if keyword. Its body will be indented one tab to the right just like the if statement.



The illustration above explains the logic of the if-else statement. Now let’s look at some examples in code.

In the **first line**, we set age to 34. In the next line we write the keyword if followed by the condition. The condition here is whether or not age is less than 50. When **line 2** is executed and the condition is satisfied, execution jumps to **line 3**. Here the message, “You are still young,” is printed.

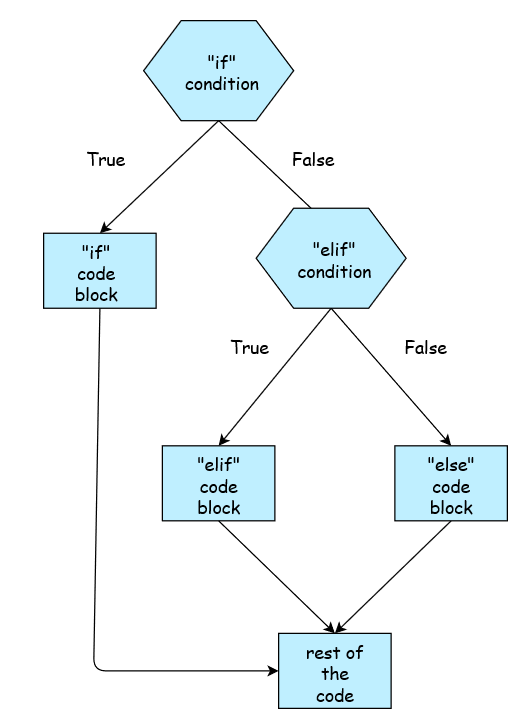
After that, we change the value of age to 55. The same if condition is written in **line 8**, but this time the condition is not satisfied. Therefore, **line 9** is skipped and **line 11** is executed.

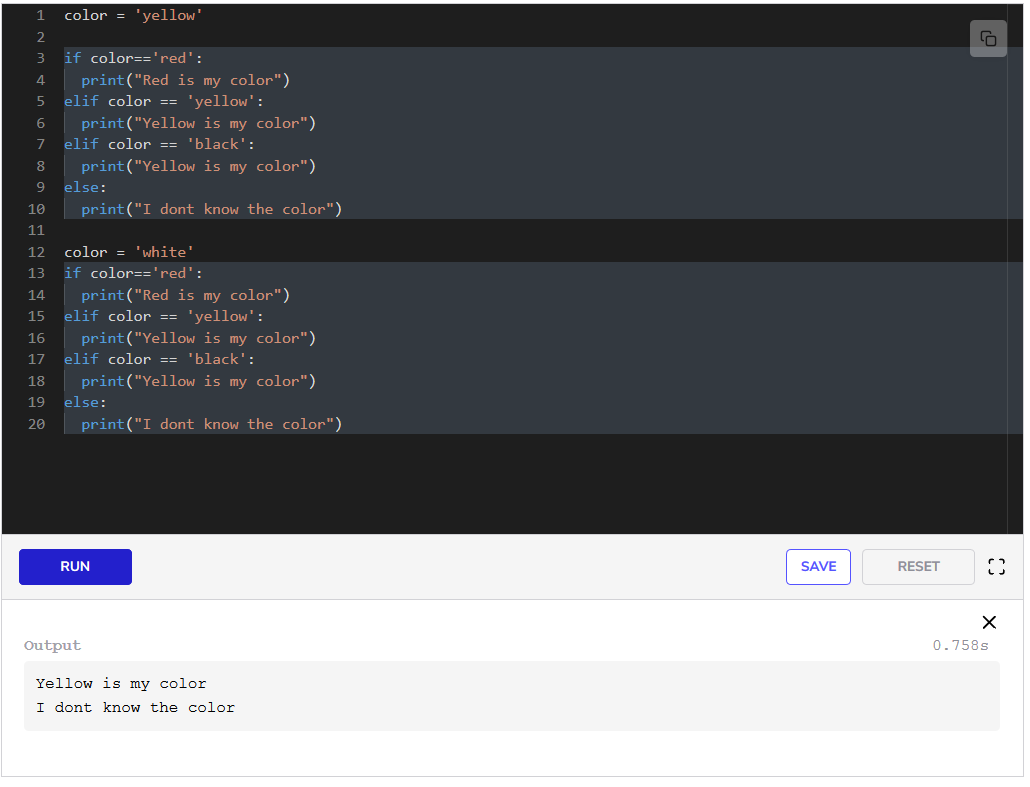
**if-elif-else statement**

The structure is:

if (condition):  
  code  
elif (condition):  
  code  
else:  
  code

If the condition provided is satisfied, then the code in the if block is executed, otherwise the next condition is checked. If that condition is satisfied, the code after elif is executed and the else condition and code is skipped. But if it is not satisfied, then the code inside the else block is executed.



The illustration above explains the logic of the if-else statement. Now let’s look at some examples in code. Note that we can have multiple elif statement blocks between the if and else statement blocks. However, at any time if an elif condition is satisfied, the code inside that block is executed and the rest of the conditions and codes are skipped. Let’s look at some examples. 

In the **first line**, we set color to yellow. In **line 3**, we write the keyword if followed by the condition. The condition here is whether or not color is red. When **line 3** is executed and the condition is not satisfied, execution jumps to **line 5** where another condition is checked to see if the color is yellow. Here the condition is satisfied and **line 6** is executed. After this the execution jumps to **line 12**, where color is changed to white.

Now all the conditions are checked one by one but no condition is satisfied. Hence, the execution jumps to **line 20** in the else block.

Conditional statements can be nested as well. You can try out different sorts of experiments with these. These come in very handy.

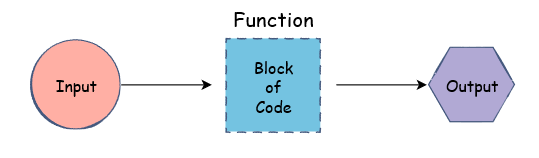
In the last example, we had to write the same piece of code twice. This can be quite annoying when we have to write the same piece of code again and again. The solution is described in the next lesson and is known as a *function*.

**Functions**

**Function**

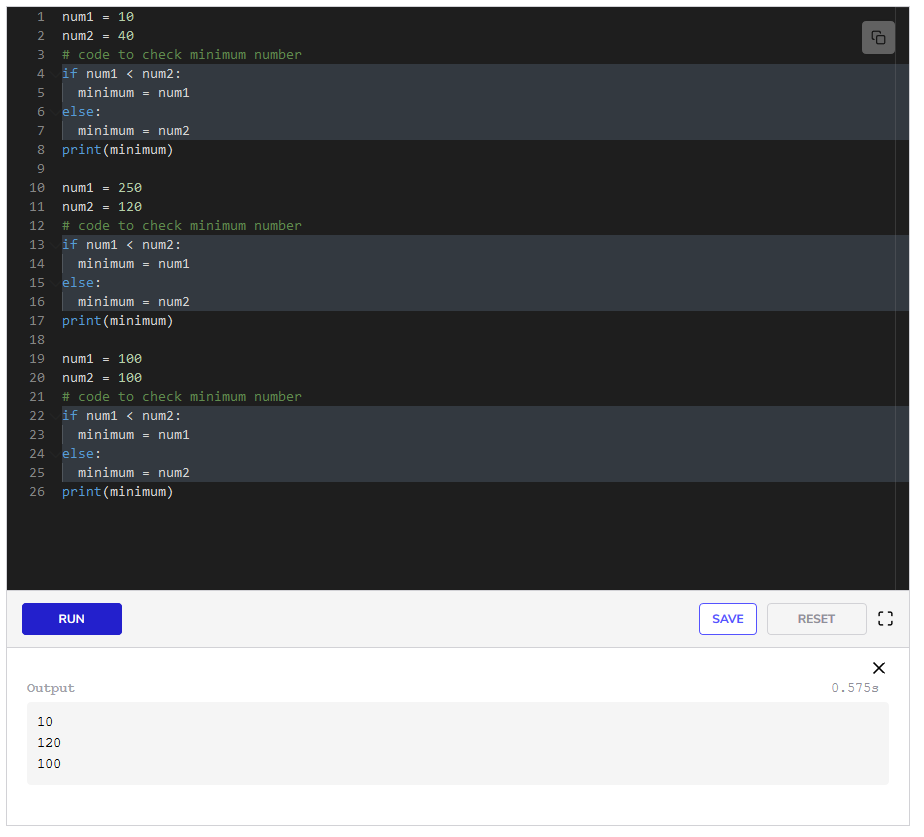
A **function** in a programming language is a reusable block of code that will be executed when called. You can pass data to it and it can return data as well.

Think of a function as a box that performs a task. We give it an input and it returns an output. We don’t need to write the code again for different inputs, we could just call the function again.

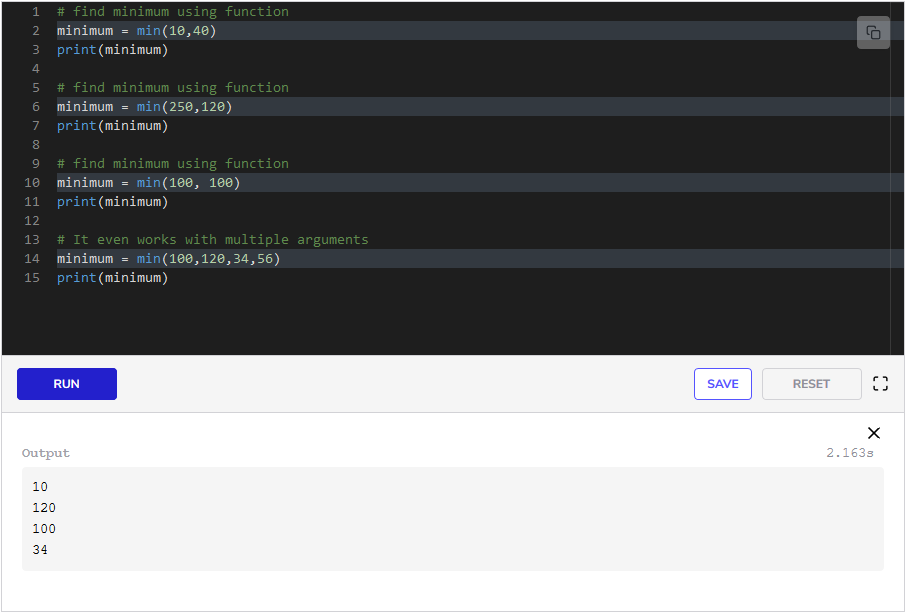


Until now, we have only been using print and len. Both of these are functions. They perform a specific predetermined task. These are called **built-in** functions as they come with the Python language. Similarly, there are many other functions that come built-in.

Let’s say we want to find the minimum of two numbers.



For every new pair of integers, we need to write the if-else statement again. This could become much simpler if we had a function to perform the necessary steps for calculating the minimum. The good news is that Python already has the min function:



We use the min function just like we use print and len; we pass in a list of values separated by commas.

This was an example of using **built in** functions. But we can also create and use our own functions.

**Creating Functions**

In Python, a function can be defined using the def keyword in the following format:

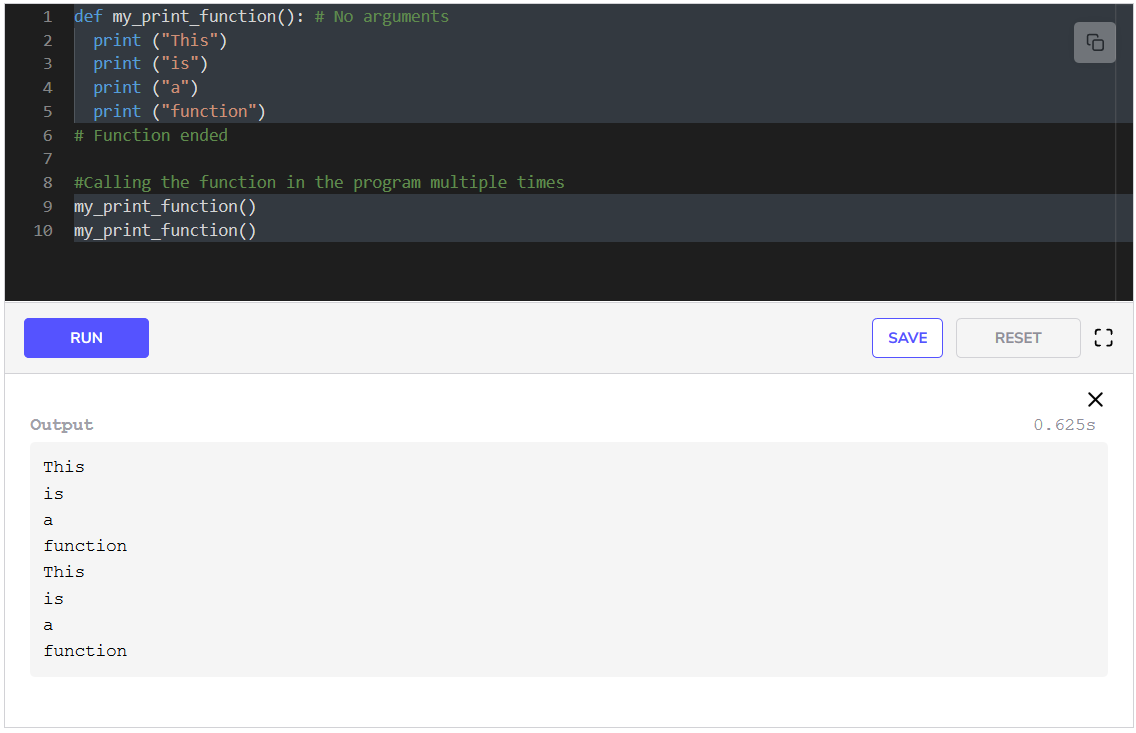
def function\_name (arguments):  
   body

The function\_name is simply the name we will use to identify the function.

The arguments of a function are the inputs for that function. We can use these inputs within the function. Arguments are optional. We’ll get to know more about these later.

The body of the function contains the code for the task that we want the function to perform. This is always indented to the right.

Let’s create our first function.

In **line 1**, we use the keyword def to define our function. Next we write the name of the function. We name it my\_print\_function, then we use parenthesis. But we leave these empty as we do not need any arguments for this function.

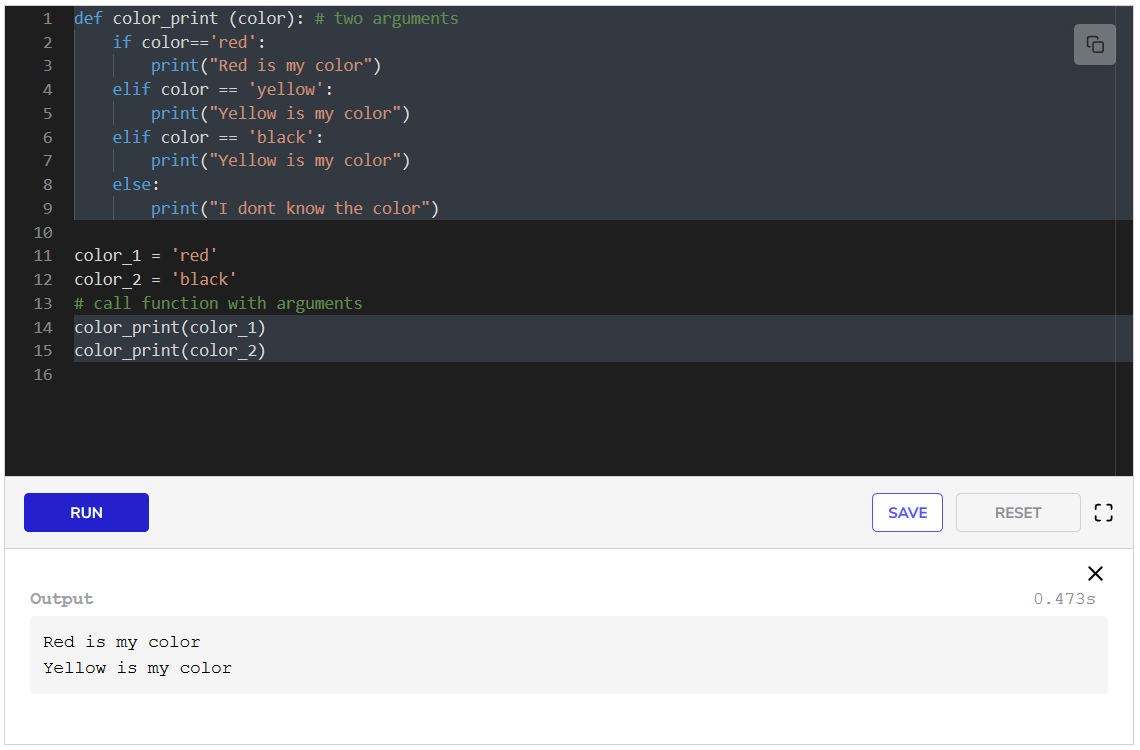
In **lines 9 and 10**, we simply use this function. Using a function is known as **calling** a function. We write the name of the function with parenthesis. In this way, we do not need to write the same code again and again.

**Function Arguments**

**Arguments** are a crucial part of the function structure. They are the means of passing data to the function. This data can be used by the function to perform a meaningful task.

When creating a function, we must define the number of arguments and their names. These names are only relevant to the function and will not affect variable names elsewhere in the code. Arguments are enclosed in parentheses and separated by commas.

Continuing our discussion from the last example, where we had to write the same piece of code twice, we will write a function that will print something on the basis of the argument it gets.



In **line 1**, we use the keyword def to define our function. Next, we write the name of the function. We name it color\_print, then we use parenthesis. We write the argument name. Then in **lines 2-9**, we write the code for printing a message on the basis of the argument we received.

In **lines 11 and 12**, we create two variables, color\_1 and color\_2, and give them some values. We call our color\_print function in **lines 14 and 15**. We give color\_1 and color\_2 as arguments to the functions on **lines 14 and 15** respectively.

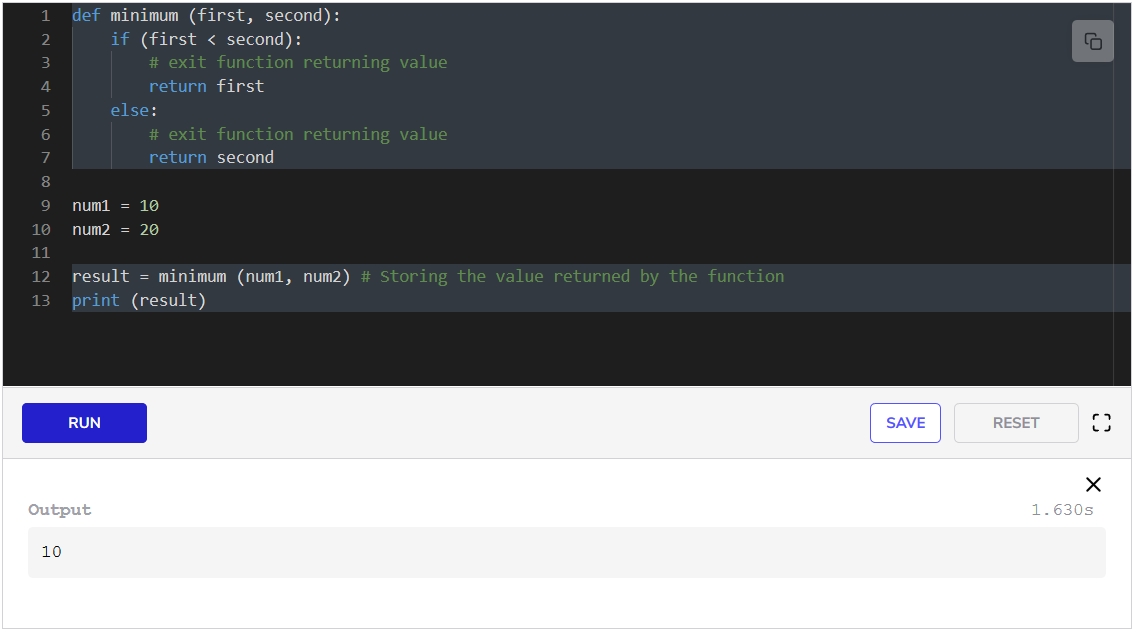
We see that now we do not have to write the same piece of code again and again. This is why functions are always used in programming.

**The return Statement**

So far, we’ve only defined functions that print something. They don’t give anything back to us. But if we think back, functions return values all the time. Just take len for example. It returns an integer that is the length of the list passed to it.

To return something from a function, we must use the return keyword. Keep in mind that once the return statement is executed, the function ends. Any remaining lines of code after the return statement will not be executed.

We saw the min function above. Let’s write our own version of the function minimum to return the smaller value of two numbers. It will work just like the built-in min function with two parameters:

We have defined the minimum function in the same way as we defined the color\_print function above. In **lines 4 and 7**, we return the number by using the return statement.

In **line 12**, we call the function. The positions of the parameters are important. In the case above, the value of num1 will be assigned to first as it was the first parameter. Similarly, the value of num2 is assigned to second. We store the returned value in a variable called result. We print result in the last line.

So now you can write your own functions in Python. In the next lesson, we will look at another fundamental concept, *lists*, in Python.

**Lists**

**Grouping items together**

Until now we have learned that a variable can only hold a single item. But there are many ways to hold multiple values and refer to all of them by a single variable in Python, and the most popular is a *list*. A **list** can hold multiple values at the same time. These values could belong to any data type.

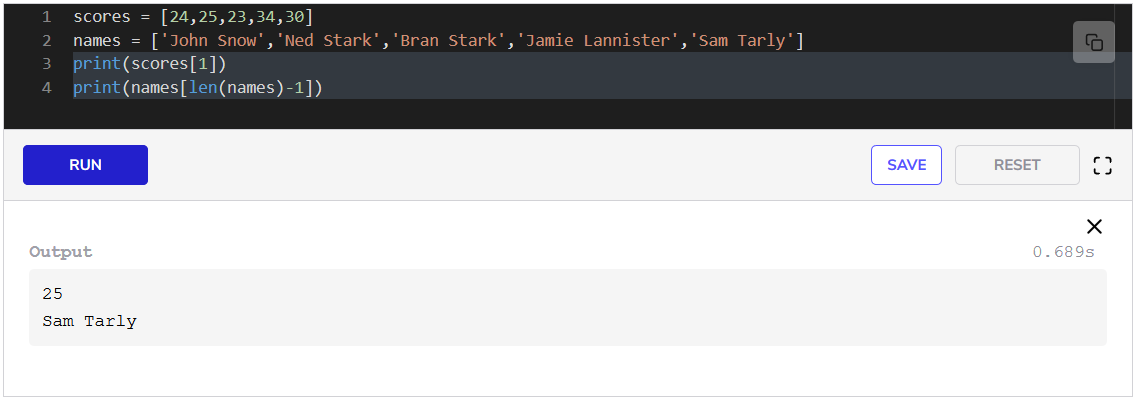
Lists are mostly used to store similar kinds of information. For example, if we want to store the marks of 50 students of a class, we would not want to make 50 separate variables. Instead, we hold them in a list.

To make a list we need to enclose all items separated by commas in [] and assign them to a variable.

In the first line, we create a list called scores, while in the second line we create a list called names. We can print the entire list just by giving its name to the print statement.

**Indexing**

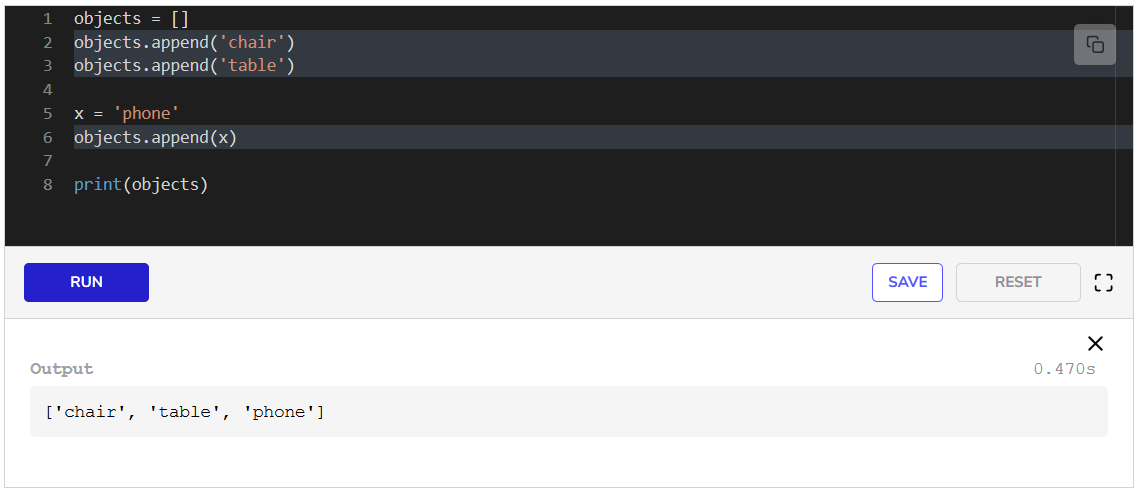
A list is similar to a string in the sense that we can access the elements in the list by *indexing*.

In **line 3**, we access the second element of scores by indexing. In the last line, we access the last element of names. The index of the last element is equal to the length of the list - 1. We retrieve the length of the list by using len.

**Appending a list**

Adding an element at the end of a list is known as **appending**. It is a very common use case in programming where we create an empty list and add items as we go through the program, and then use the items in the list when needed.

We can create an empty list with empty []. To append to a list, we do listname.append(item) where listname is the name of the list, .append is the keyword to append, all followed by a parenthesis. Inside the parenthesis, we specify the variable name of the item that we want to add, or the item directly.

In **line 1**, we create an empty list called objects. In the next line, we append the string chair to it. **Line 3** is the same as it appends table to the list. In **line 5**, we create variable x with the value phone. We append x to the list in **line 6**. In **line 8**, we verify our additions to the list by printing the list.

We will keep learning about lists as we move along this course because they will be used many times. In the next lesson, we will look at *loops*.

**Loops**

**Why loops?**

Many times, during programming, we have to repeat the same task a number of times. For instance, we have a list of numbers and we want to double every number in it. Here, we want to perform the same task for every item on our list. This is where loops come in handy. **Loops** are used to execute the same block of code a fixed number of times.

There are two kinds of loops in Python.

* The for loop
* The while loop

**The for loop**

A for loop uses an **iterator** to traverse a sequence, e.g. a range of numbers, the elements of a list, etc. In simple terms, the iterator is a variable that goes through the list. The iterator starts from the beginning of the sequence. In each iteration, the iterator updates to the next value in the sequence. The loop ends when the iterator reaches the end.

In a for loop, we need to define three main things:

* The name of the iterator
* The sequence to be traversed
* The set of operations to perform

The loop always begins with the for keyword. The body of the loop is indented to the right:

for iterator in sequence:  
   body

The in keyword specifies that the iterator will go through the values in the sequence/list.

**Looping through a range**

In Python, the built-in range function can be used to create a sequence of integers. This sequence can be iterated over through a loop. A range is specified in the following format:

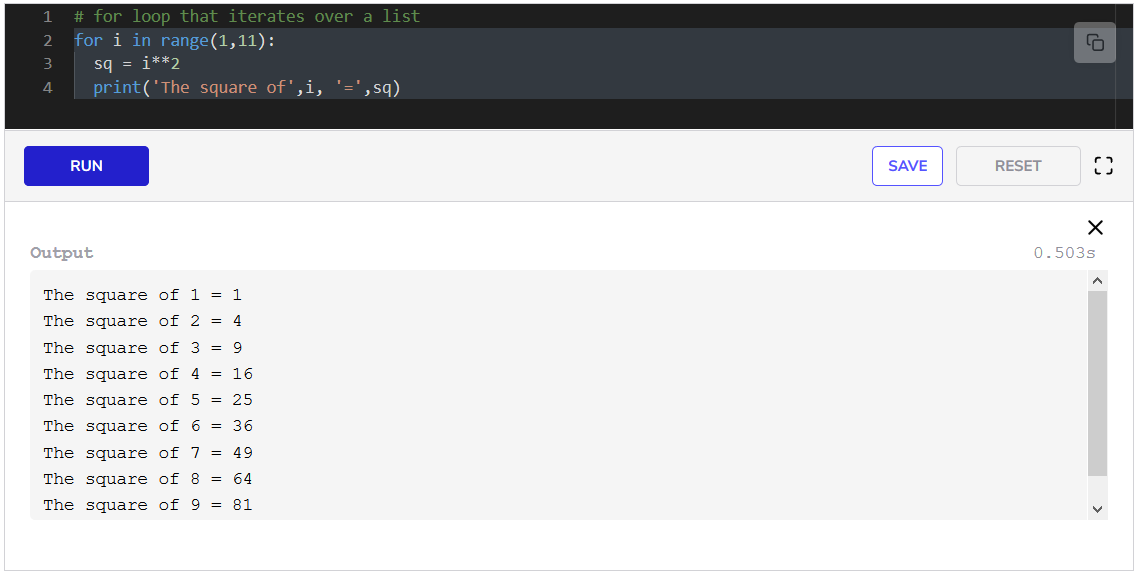
range(start,stop,step)

If the start index is not specified, its default value is 0.

The stop value determines the end of the list and is not included in the list. It also tells where the iterator should stop.

The step decides the number of steps the iterator jumps ahead after each iteration. It is optional and if we don’t specify it, the default step is 1, which means that the iterator will move forward by one step after each iteration.

Let’s take a look at how a for loop iterates through a range of integers:

We use the for loop structure we defined above in the first line. range(1,11) gives us a list of integers [1,2,3,4,5,6,7,8,9,10]. In the first iteration, i gets assigned the value 1. The execution moves to **line 3**. Here a variable sq is created and assigned the value of i squared. Then in **line 4**, the results are printed. After **line 4**, the iterator i is assigned the next value from the list which is 2. **Lines 3 and 4** repeat. Then i is updated to 3 and so on. The loop stops when all the elements of the list have been assigned to i turn by turn.

**The break statement**

Sometimes, we need to exit a loop before it reaches the end. This can happen if we have found what we were looking for and don’t need to make any more computations in the loop.

A good example would be if we are searching in a list for an item. We will exit the loop once we have found it.

In **line 1**, we create a list, names, with 666 names in it. We want to search the list and print a message when we find Joey in the list. We will also print a message if we do not find it. Therefore, we create a Boolean variable found and set it to False. We will set it to True when we find Joey.

In **line 3**, we write the for loop. Iterator i will be assigned every name in the list one by one. When the condition on **line 4** is satisfied, a message is printed in **line 5**. Then found is assigned True in **line 6**. In **line 7**, we exit the loop because of the break statement. In **line 9**, we check if Joey was not found and we print a message.

**The while loop**

The while loop keeps iterating over a block of code as long as a certain **condition** holds True. It operates using the following logic:

*While this condition is true, keep the loop running*.

In a for loop, the number of iterations is fixed since we know the size of the sequence. On the other hand, a while loop is not always restricted to a fixed range. Its execution is based solely on the condition associated with it.

while condition:  
  body

Below we write code for printing the squares of first 10 integers.

In the first line, we create variable i and assign it a value of 1. In **line 3**, we write the keyword while, and specify our condition. Our condition is i < 11. As long as this condition is True, the code inside the loop will keep executing. Inside the loop, we just take the square of i in **line 4** and print it in **line 5**. In **line 6**, we update the value of i. If we do not write **line 6**, the while loop will keep running indefinitely.

We can also use the break statement with a while loop and exit it when a certain condition is fulfilled.

We are now familiar with the very basic features of the language. In the next lesson, we will look at *libraries* and *packages* in Python.

# Packages and Modules

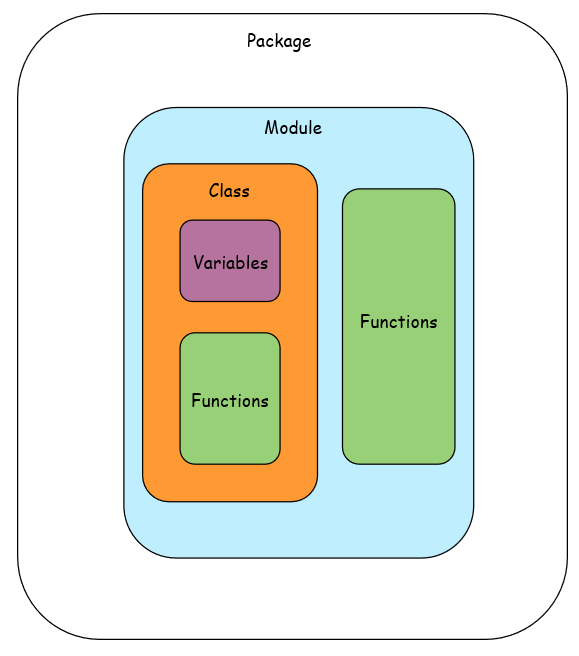
Python is known for its brilliant support for third party resources. The language makers can’t add every functionality to the language, therefore, the open-source community has created multiple packages that can be used when needed. We do not need to write code for everything. People have already done that for us. We only need to use that code in our program.

## Classes and Modules

A **class** in python is a collection of one or more functions and variables. The functions and variables that are defined in a class are called its **members**. Member functions are formally called methods.

A **module** in python is a collection of one or more classes and functions that we can reuse in our programs.

A collection of different modules, classes, and functions is known as a **package**. The illustration below describes this hierarchy



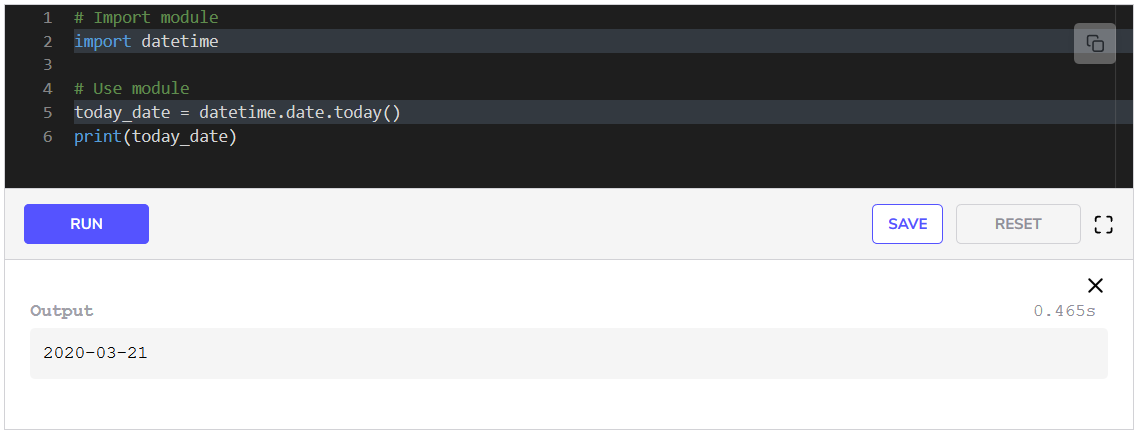
Some popular packages in python are math, numpy, pandas, scipy, etc. We will look at all of these in this course.

To use an outside function in our program, we have to **import** it first. We can import a complete module, a complete class, or a single function. We do that with the import statement. The following are different ways the import statement can be used:

import modulename  
# or  
from modulename import classname  
# or   
from modulename import functionname

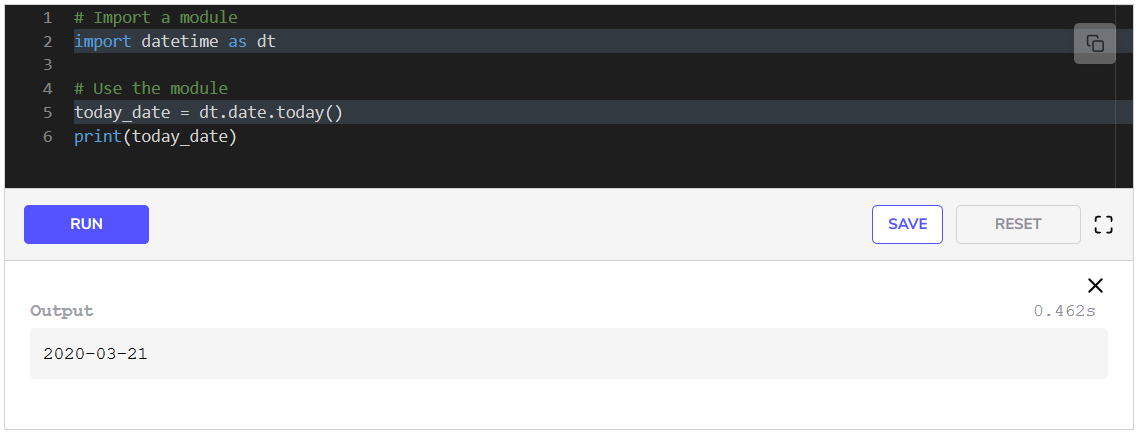
### Importing a Module

Let’s look at an example with the datetime module. This module has functions and classes related to the current date and time, apart from other functionalities.

We import the complete module in **line 2**. We use the module in **line 5**. We access the date class in the module by writing .date. Then we access the today function from the class by writing .today(). This function does not require any arguments. It returns the current date which we print in **line 6**.

#### Naming imports

We can name our imports with the as keyword. We can then refer to them using this keyword.

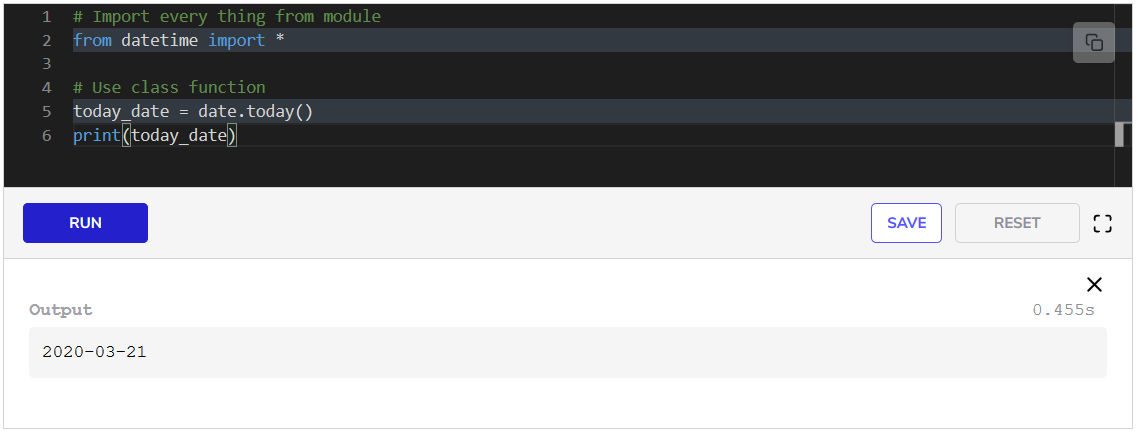
In this instance, we import the datetime module as the alias dt. Everywhere in the program below, we can use dt as we have used in **line 5**.

### Importing a Class

There is a class called date available in the package. We import it in the first line as dt and use its function today in **line 5**.

### Import using \*

We can import everything from a module using the \* and use it in our code.

We can import everything from a module by writing \*. This means that we can use the date class without prepending the name of the package before it, as we have done in **line 6**.

So, we now have the necessary basic knowledge of python to move forward in this course. In the next lesson, you will be tested on the concepts learned in this chapter.

**Exercise: Average of a List**

**Problem Statement**

In this challenge, you must implement the average() function. It takes a list as a parameter and calculates the average of the numbers in the list.

**Sample Input**

list = [2,3,4,-1]

**Sample Output**

2

**Exercise**

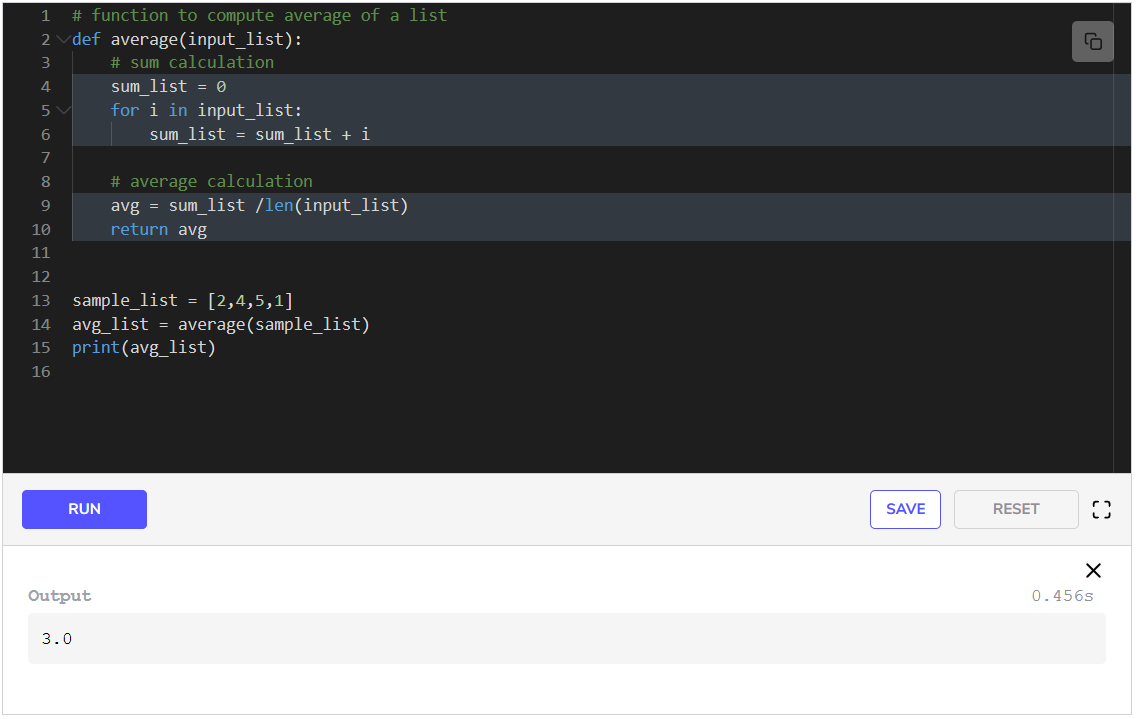
Take some time to understand the logic behind this problem before moving to the implementation. Think about the different concepts we’ve learned so far such as loops, indexing, and lists to write an algorithm.

Write the code below and remove the pass statement. Press the **TEST** button to see if your solution was correct or not.

If you feel stuck, feel free to check out the solution review in the next lesson. Good luck!

**Solution Review: Average of a List**

**Solution**

We define the average function in **lines 2-8**. Let’s break this function down. For computing the average of the list, we need to find the sum of the numbers in the list and the length of the list. We already know we can use the len function to compute the length of the list.

We have found the sum of the list in **lines 4-6**. To find the sum, we need to iterate over the list and add each element to a variable as we iterate through the list. Therefore, we initialize the variable sum\_list to 0 in **line 4**. Then we use a for loop as:

for i in input\_list:

The above line means that variable i will be assigned the value of every element of input\_list one by one, in each iteration.

Then **line 6** is:

sum\_list = sum\_list + i

This line means that in each iteration we add i to sum and store the result of the expression in sum\_list. So, in each iteration, sum\_list is updated.

After the loop ends, we have the sum of the list in sum\_list. Then we just divide sum\_list by the length of the list in **line 9** and store the result in avg. We return the average of the list in **line 10**.

This was the end of your first challenge. In the next lesson, you will be tested on a slightly more difficult challenge.

**Exercise: Factorial of a Number**

**Problem Statement**

In this challenge, you must implement the factorial() function. It takes an integer as a parameter and calculates its factorial.

The factorial of a number, **n**, is its product with all the positive integers smaller than n.

factorial(n)=n∗(n−1)∗(n−2)∗.....∗1factorial(n) = n\*(n-1)\*(n-2) \*..... \* 1 factorial(n)=n∗(n−1)∗(n−2)∗.....∗1

The factorial for 0 is 1 by definition.

**Sample Input**

n = 5

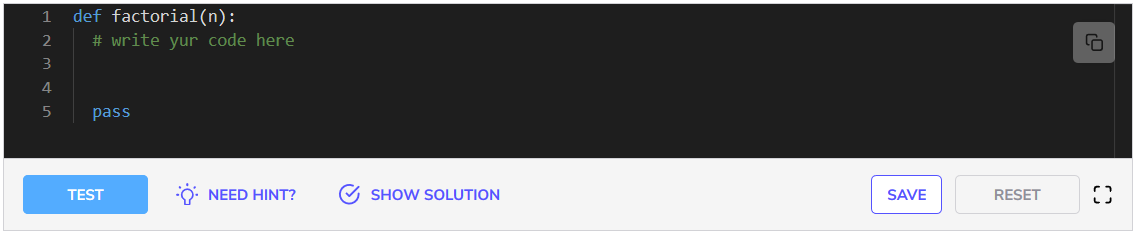
**Sample Output**

120

**Exercise**

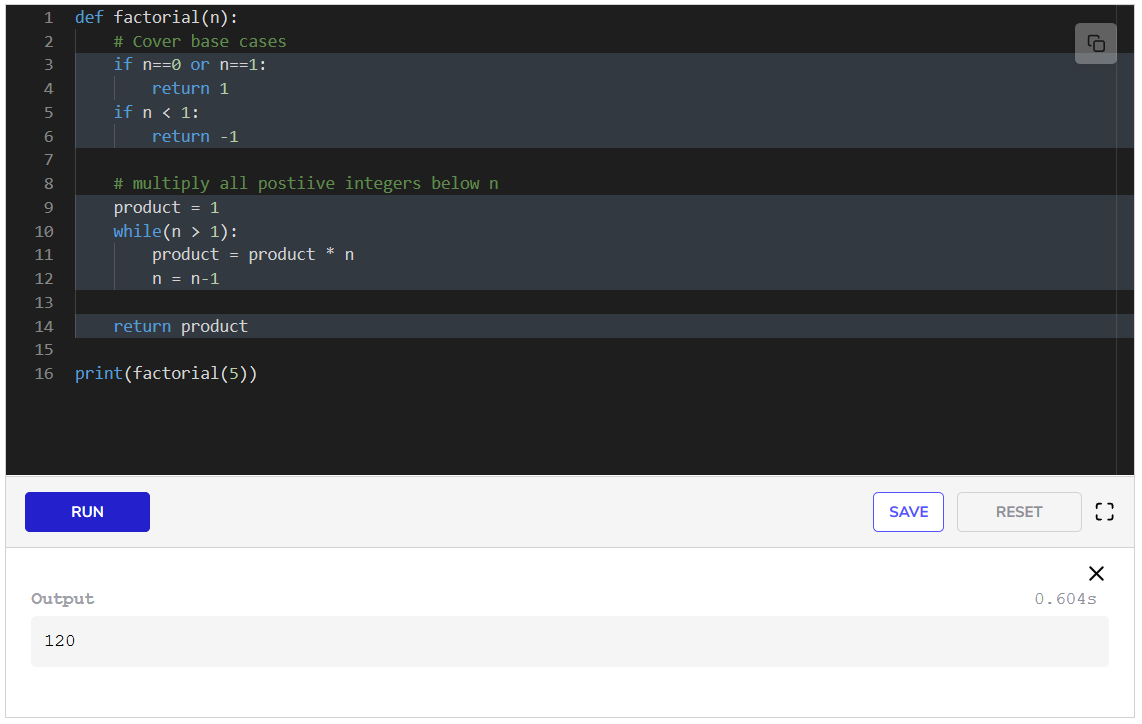
Take some time to understand the logic behind this problem before moving to the implementation. Think about the different concepts we’ve learned so far such as loops and conditional statements and write an algorithm that handles all cases.

The input will always be an integer, so you don’t need to worry about that. If the integer is negative, the function always returns -1.

If you feel stuck, feel free to check out the solution review in the next lesson. Good luck!

**Solution Review: Factorial of a Number**

**Solution**

**Explanation**

The function starts with handling the edge cases. We know that for n==0 and n==1, we need to return 1. Therefore, we write an if statement with an or in between the conditions in **line 2**, so that if any of these is True, we return 1. Then we handle the case if n is a negative number in **line 4**. We return -1 in **line 5**.

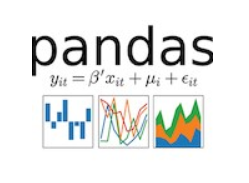
After handling the edge cases, now comes the main part. We initialize a variable product with value 1 in **line 7**. Our aim is to keep multiplying a number to this product and decrease that number in every iteration of the loop. Therefore, we use a while loop in **line 7**. The while loop will keep running as long as n is greater than 1. In **line 9**, we multiply product with n, and store the answer in product. This means the value of product is being updated in every iteration of the loop. We decrease n by 1 in **line 10** so that in every iteration product is multiplied with the updated n.

This brings the end of this chapter. Now you have the basic Python knowledge to move towards handling data in Python in the next chapter.

**Importing Data in CSV Files with Pandas**

**Pandas**

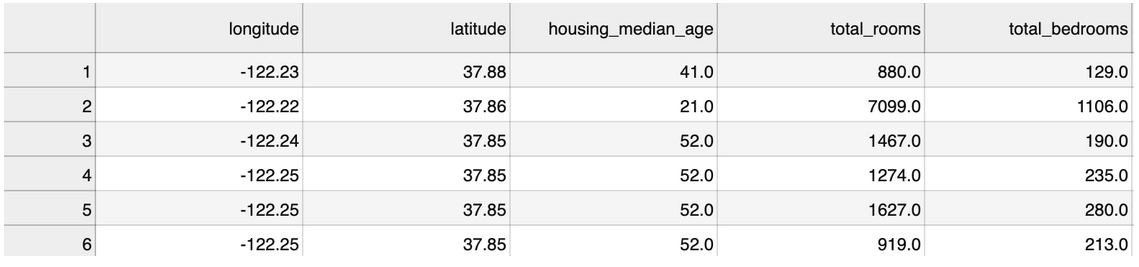
**Pandas** is a very popular Python library that is used for data analysis, which is the second step of the Data Science lifecycle. It offers functionalities to handle and manipulate data very efficiently. It has been widely adopted by people who are not computer scientists or programmers as it makes them move beyond Excel for analyzing data. We will be using Pandas in this course for all of our analyses.



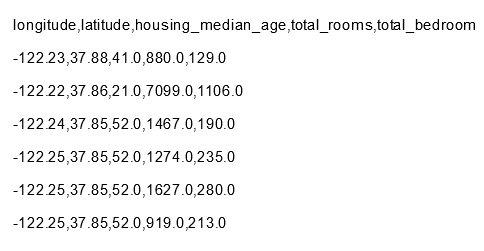
We will start by learning how to *import* CSV data using Pandas.

**Importing CSV files**

These days almost all of the data that is acquired by companies is entered in spreadsheets using different software such as Excel. The data is in the form of tables. This data is stored in CSV (Comma Separated Values) files that have **“.csv”** at the end of their name. A spreadsheet is shown in the below image.

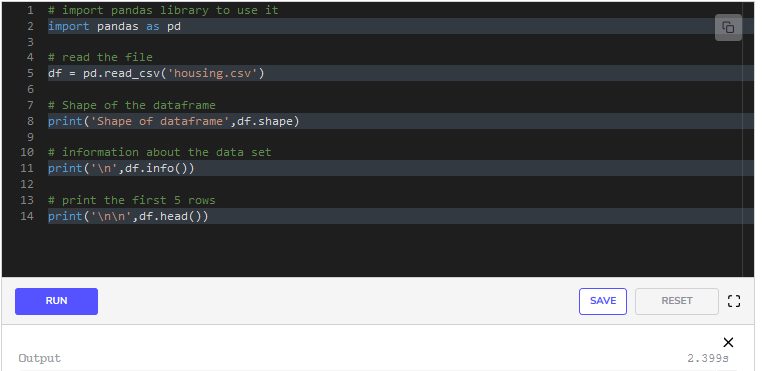


The above spreadsheet can be represented in a CSV file as:



To work with CSV files in Python, we have to import them first. Below is an example of how to import CSV files. The file *housing.csv* is the [Census data of housing blocks in California](https://www.kaggle.com/takedown/complete-tutorial-for-beginners). You can download it to view the contents by clicking on it. Run the code below.

housing.csv



Shape of dataframe (20640, 10)

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 20640 entries, 0 to 20639

Data columns (total 10 columns):

longitude 20640 non-null float64

latitude 20640 non-null float64

housing\_median\_age 20640 non-null float64

total\_rooms 20640 non-null float64

total\_bedrooms 20433 non-null float64

population 20640 non-null float64

households 20640 non-null float64

median\_income 20640 non-null float64

median\_house\_value 20640 non-null float64

ocean\_proximity 20640 non-null object

dtypes: float64(9), object(1)

memory usage: 1.6+ MB

None

longitude latitude housing\_median\_age total\_rooms total\_bedrooms \

0 -122.23 37.88 41.0 880.0 129.0

1 -122.22 37.86 21.0 7099.0 1106.0

2 -122.24 37.85 52.0 1467.0 190.0

3 -122.25 37.85 52.0 1274.0 235.0

4 -122.25 37.85 52.0 1627.0 280.0

population households median\_income median\_house\_value ocean\_proximity

0 322.0 126.0 8.3252 452600.0 NEAR BAY

1 2401.0 1138.0 8.3014 358500.0 NEAR BAY

2 496.0 177.0 7.2574 352100.0 NEAR BAY

3 558.0 219.0 5.6431 341300.0 NEAR BAY

4 565.0 259.0 3.8462 342200.0 NEAR BAY

To use Pandas, we import it in the 1st line. Then we read the file using the function, read\_csv. You only have to provide the name of the file to this function as done above in **line 5**. Pandas stores the file in an object called **dataframe** that stores the data in the form of rows and columns like a 2D array. We name our dataframe df. To find the number of rows and columns, we print df.shape which gives us the number of rows and columns in the format (rows, columns).

To retrieve general information about the columns of a dataset, we use the function info() in **line 11**. The info function gives the name, the number of values, and the data type of each column.

To view the first 555 rows, we have written print(df.head()) in **line 14**. The function head() returns the first 555 rows of the dataframe and print displays them on the screen.

Sometimes because of the long length of the table, it cannot be completely displayed on the screen and it breaks. Some columns are shown together and then the rest of the columns are shown below, as you can see from the output of **line 14** in the above code.

In the next lesson, we will look at how to pick and choose data from a dataframe.

**Indexing and Selection**

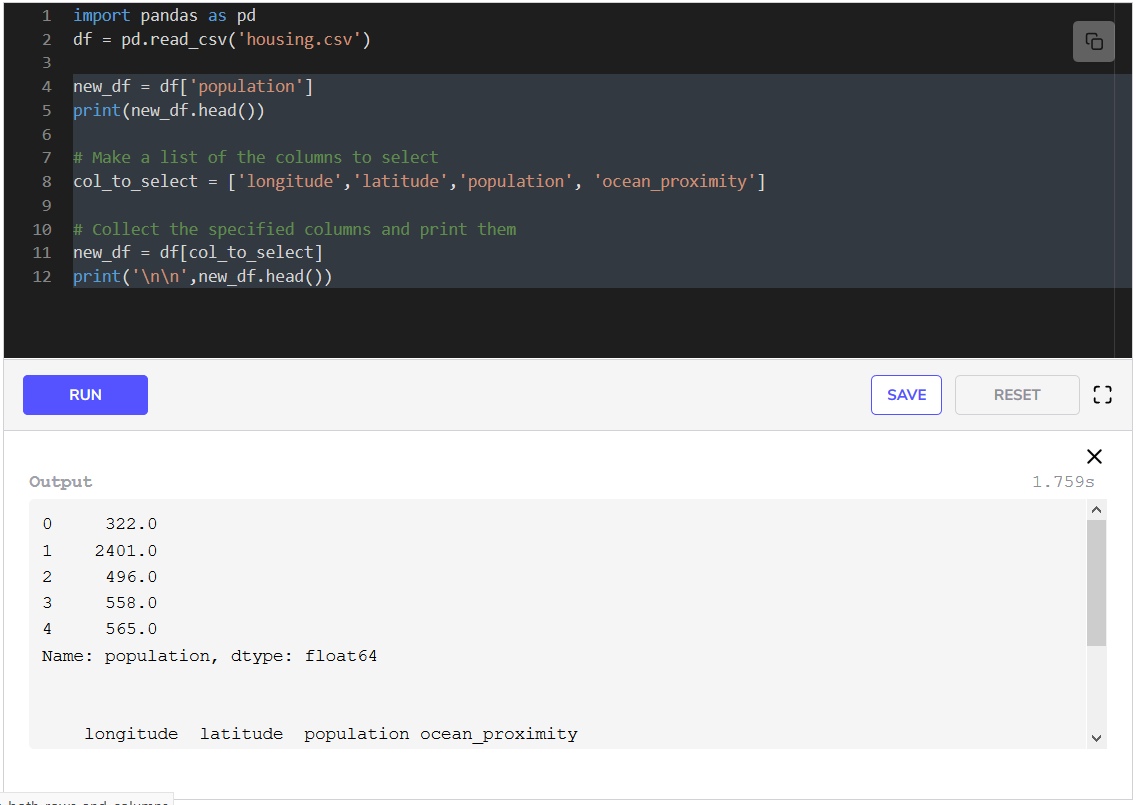
**Indexing** is the technique of efficiently retrieving records from data based on some criteria that the data has been arranged by. As we saw in the previous lesson, the data is organized in rows and columns in a dataframe. So, we can index data using the positions and names of these rows and columns. Now let’s see how to select rows and columns from the data.

**Columns**

To view the names of the columns we use df.columns.values. We will be using the file *housing.csv*.In Machine Learning terminology, a column in a spreadsheet is referred to as a **feature**, while in Statistics it is referred to as a **variable**. It is also referred to as an attribute. We will be using all of these terms interchangeably in this course. 

**Selecting columns**

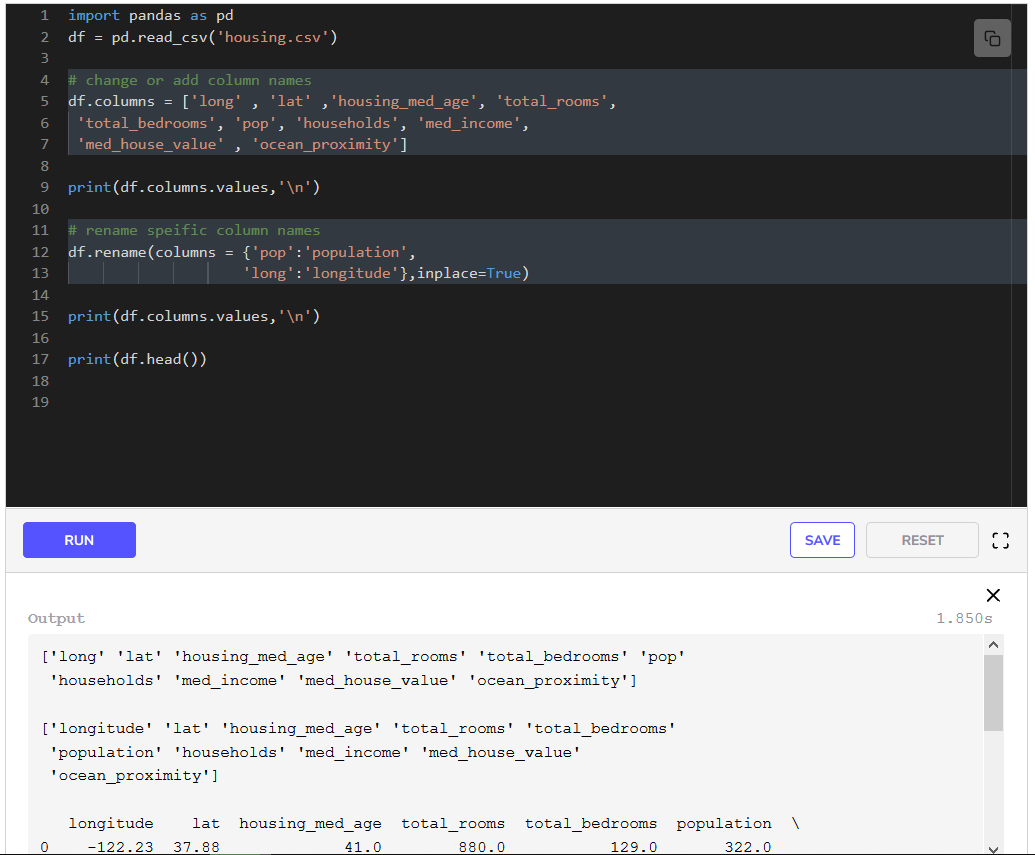
Let’s see how we can select the data of a few columns of *housing.csv*.

We view the values in a column by simply typing their name as we did in **line 4**. In **line 8**, we create a list of columns that we want to select. In **line 11**, we retrieve those columns out of the dataframe df and save it as a new dataframe called new\_df. **Line 12** prints the head of the dataframe.

**Changing column names**

If we want to change the column names of all the columns or add column names that are not already present in the CSV file, we have to provide a list of column names.

We can also change specific column names as well using the rename function. Let’s see an example of these.

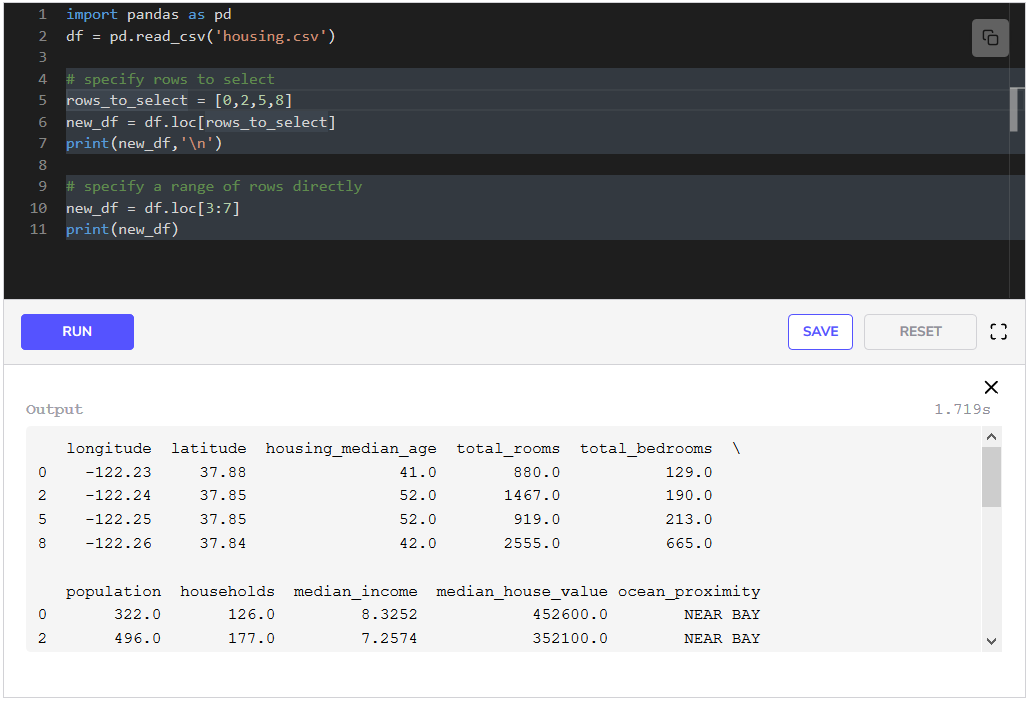


In **line 5**, we add the column names by providing a list of column names. This modifies the names of all columns to what we provide.

In **line 12**, we rename only two specific columns by providing a dictionary with old names and new names. We change the names of two columns, pop and long to population and longitude respectively.

**Rows**

We can select the rows we want by specifying their positions in the dataframe.



In **line 5**, we have specified the rows to select and in the next line, we have retrieved those rows using the loc keyword. We can provide the indices of rows to loc to get the specified rows.

We can also specify a range of rows to retrieve directly, as done in **line 10**. This gives us consecutive rows from the 4th row to the 7th row, i.e., rows with indices 3,4,53,4,53,4,5 and 666. Remember that the row and column numbers start from 0 in Python.

A single row is sometimes called a **record**.

**Indexing both rows and columns**

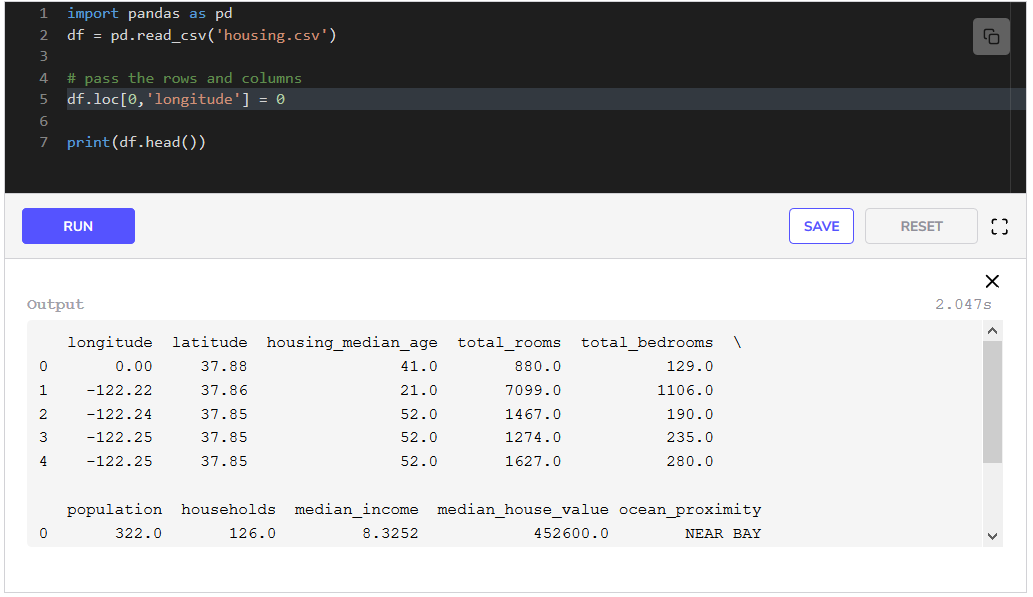
We can specify rows and columns both to select a subset of the dataframe. Let’s look at an example.



In **line 5**, we have specified the columns that we want. In **line 8** we specify both rows and columns to loc. We have written 0:5 to specify that we want the first 555 rows and then a comma, followed by the list of the column names that we want. Then we print the head of the new dataframe in **line 10**.

**Setting values**

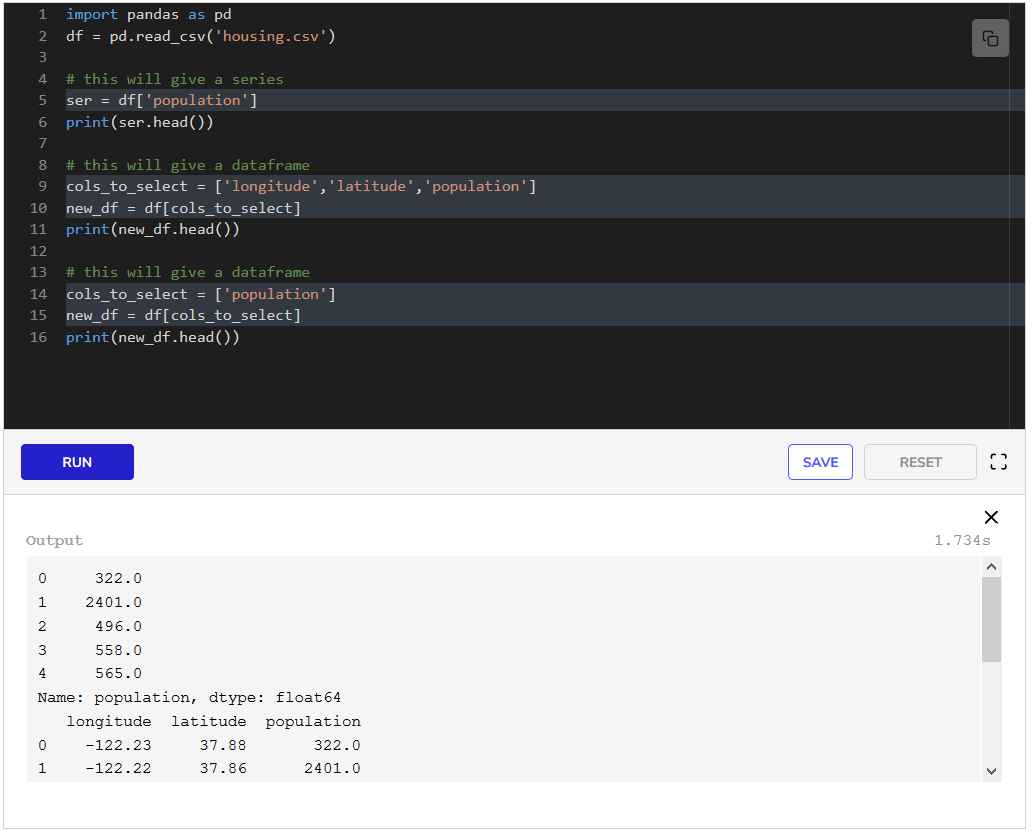
We can set or replace a value in a cell using loc. We select the cell by providing the row index and the column name and then provide the new value.

From the output of **line 7**, it can be verified that the value of longitude for the first row has been changed to 000 as specified in **line 5**.

**Series vs Dataframe**

At this point, it is important to differentiate between a *series* and a *dataframe* in pandas. A **series** in pandas is like a one-dimensional array. It can hold data of a single type. Whereas a **dataframe** is like a two-dimensional array that can hold data of multiple types in the form of rows and columns. Rows, as well as columns, can be named in both series and dataframes. You can think of a series as a dataframe with only a single column.

We know that pandas stores a CSV file in a dataframe. But when we select a single column from a dataframe it gives us a series, and when we select multiple columns from a dataframe by giving a list of column names, we get a dataframe.

In **line 5**, we select the column population by directly providing df the name of the column. This gives us a *series* in ser.

In **line 10**, we select multiple columns by providing a list to df. This gives us a dataframe in new\_df.

In **line 15**, we select a single column by providing the column name in a list to df. This gives us a dataframe in new\_df.

The distinction between series and dataframe is necessary because some functionalities can only be used with series, while some can only be used with a dataframe as we will see in the future lessons.

Now that we know how to do indexing and selection, we will look at how to filter data in the next lesson.

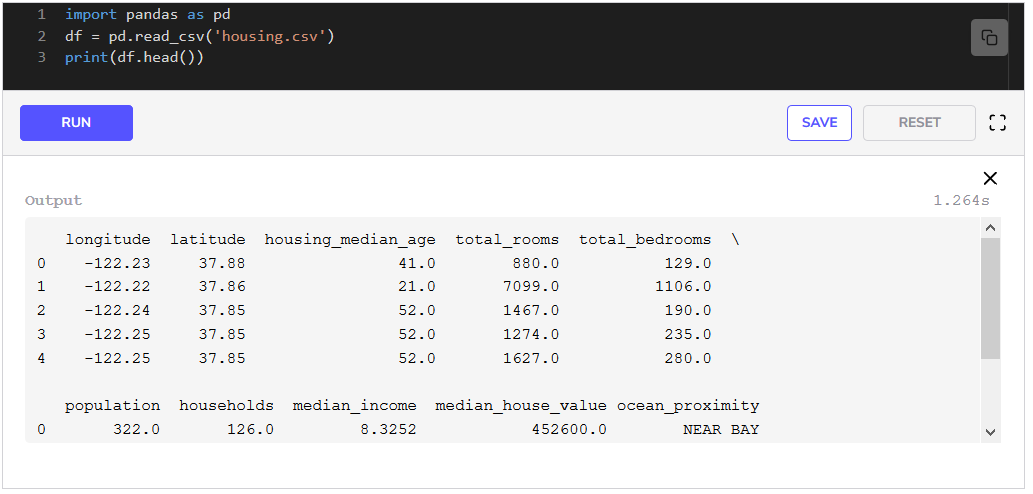
**Filtering Data**

**Filtering**

**Filtering** is the process of extracting a subset of your data based on some condition or constraint. These conditions can be on the values that the data items take. We filter data when we wish to look at a smaller part of the whole data. For instance, we may want:

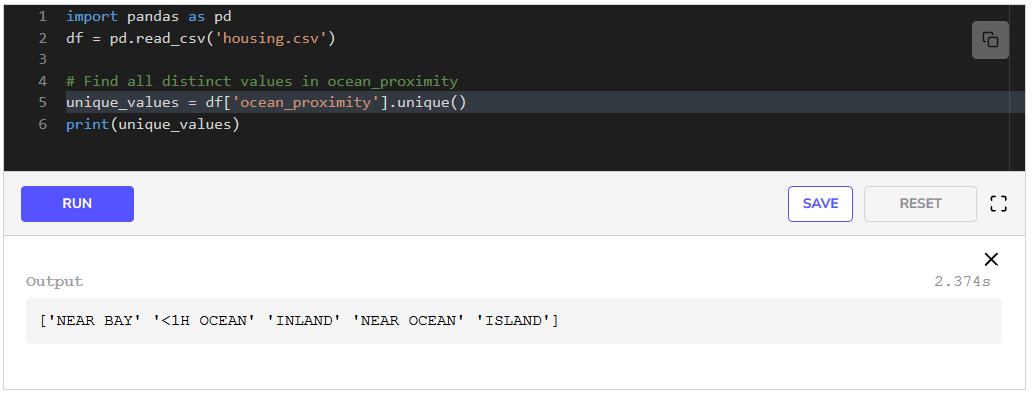
* the data in a particular period of the year
* the data of the highest selling items
* the data for a specific group of items
* to remove extra or useless data

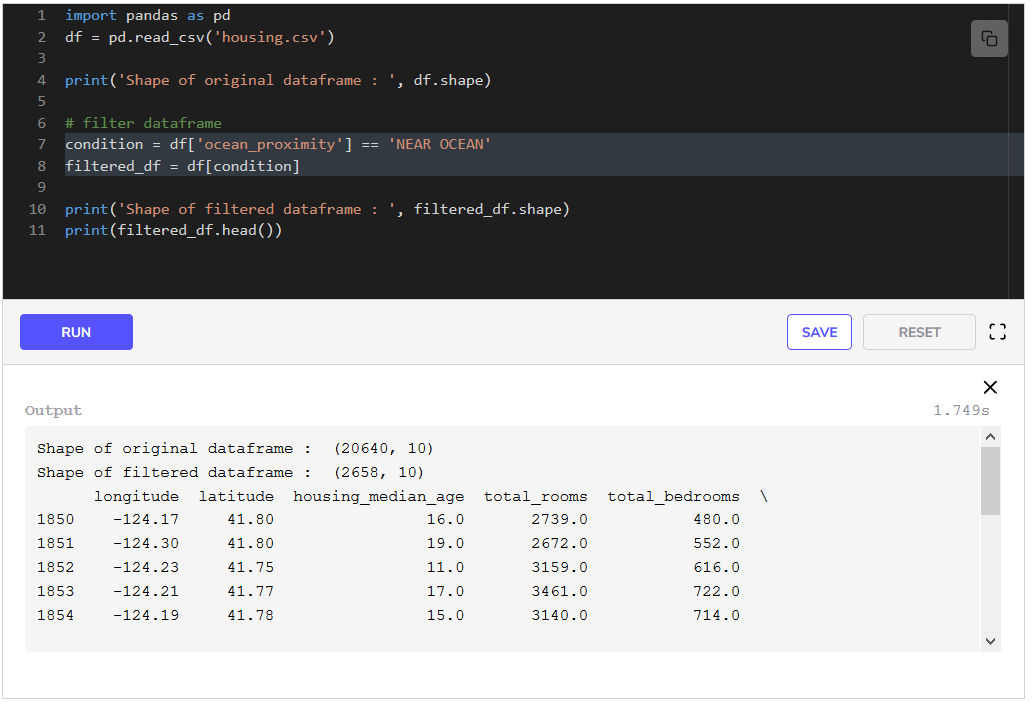
Data filtering is done on almost every dataset before doing any analysis. Let’s look at some examples using our [California Housing Dataset](https://www.kaggle.com/camnugent/california-housing-prices).



The California Housing Dataset

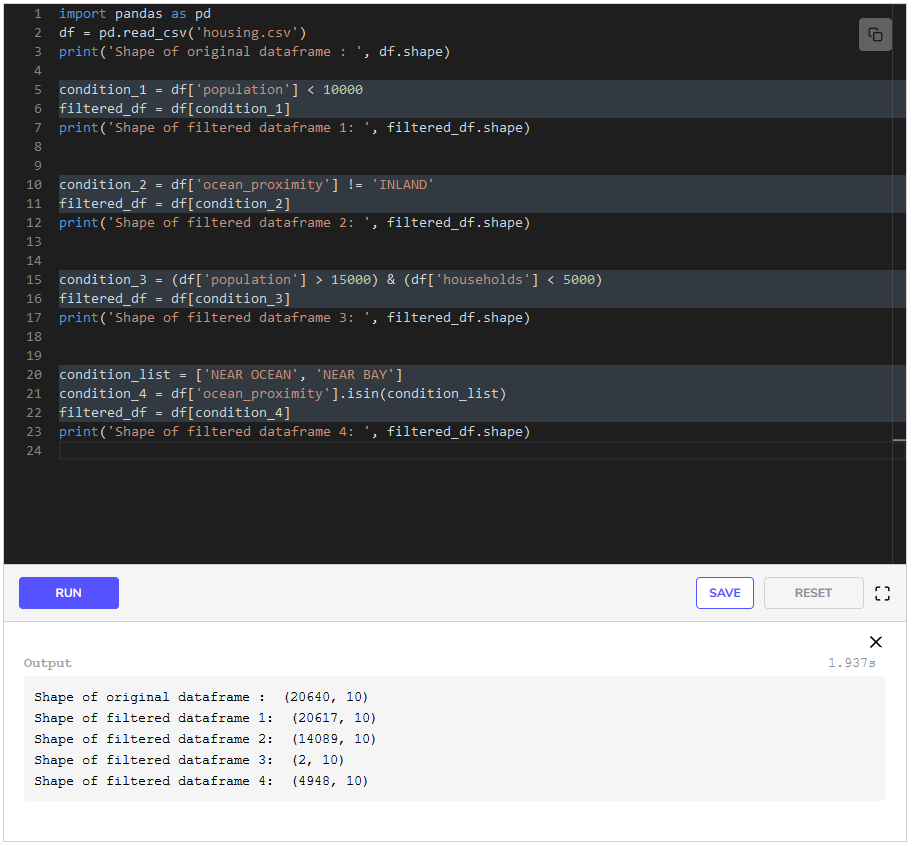
Let’s say we want to see the data for all the housing blocks that are close to the ocean. From the above code block, we know that there is text in the ocean\_proximity column instead of numbers. We will first find out how many distinct values there are in this column and then decide how to filter rows for our requirement.

We have used the function unique() on the ocean\_proximity column in **line 5** to obtain all the unique values in this column. We see that there are 555 distinct values in the ocean\_requirement column, so we have to find those rows that have NEAR OCEAN in them.



To filter, we have to provide a condition to the dataframe which we set in **line 7**. It gives us a list of booleans (True/False) against each row number. The values in the list are True for rows that satisfy our condition. When we provide this list to the dataframe in **line 8**, it gives us all the rows that have True against them, therefore giving us a filtered dataframe. We see from the output of the code that only 265826582658 rows out of total 206402064020640 satisfy our condition.

Now let’s look at some other examples. Can you point out what lines 6,11,16,226, 11, 16, 226,11,16,22 and 262626 give in the code below without looking at the explanation?



In **line 5** we have written the condition that the population column should have values less than 100001000010000. While condition\_2 in **line 10** says that the values in the ocean\_proximity column should not be INLAND.

We can combine two or more conditions using the & operator, which we have done in **line 15**. This condition will filter the dataframe so that all the rows that have population greater than 150001500015000 and households less than 500050005000 will be extracted.

We can also provide a list of values and filter them so that the values of the specified column must be one of the values in the provided list. This is done using the isin() function as we have done in **line 21**. This filtration will extract those rows that have ocean\_proximity values from condition\_list.

Now that we know how to filter we will focus on how to apply functions to data.

**Applying Functions to Data**

During data analysis, we need to use our data to perform some calculations and generate some new data or output from it. Pandas makes it very easy to apply user-defined operations, *a.k.a* *functions*, in Python terminology, on individual data items, rows, and columns of a dataframe.

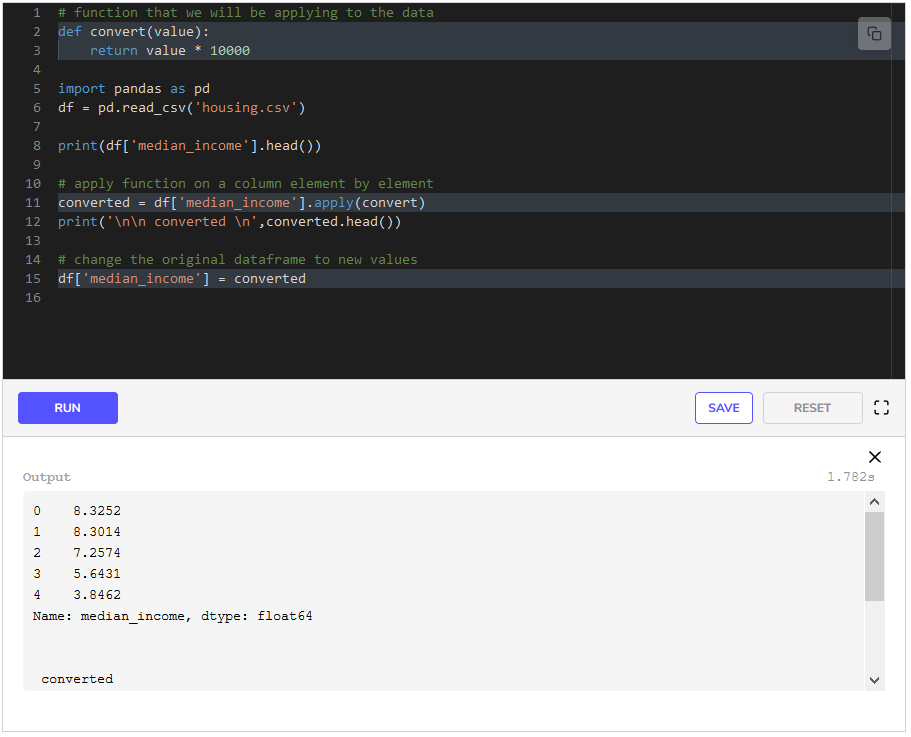
**Functions on individual Items of a column**

Pandas has an apply function which applies the provided function to the data. One of the reasons for the success of pandas is how fast the apply function performs. We will be using the [California Housing Dataset](https://www.kaggle.com/camnugent/california-housing-prices). All of the data is in the *housing.csv* file.

housing.csv

**Example: Converting numbers**

In the Dataset, the field median\_income has values which are written in tens of thousands of dollars. During analysis, we might want to convert this to Dollars. Let’s see how we can do that with the apply function.

We have defined the function that we want to apply in **lines 2-3**. We write our function so that it will receive each value of the column on which it is applied. We write it to operate on a single value. It will then be used on each of the values in the column.

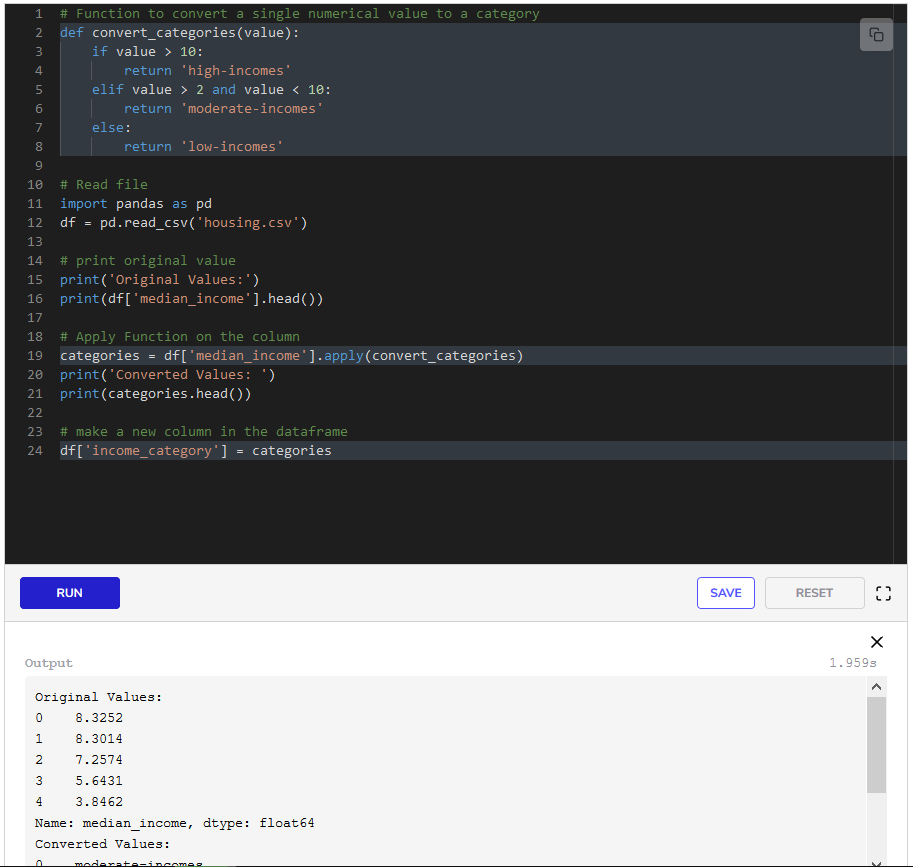
In **line 11** we select the column median\_income and use the apply function. We give the name of the function we defined above to it and store the results as converted. We replace the column in the original dataframe with converted in **line 15**.

**Example: Converting numerical values to categories**

During analysis, sometimes we want to classify our data into separate classes based on some criteria. For instance, we might want to separate these housing blocks into three distinct categories based on the median income of the households i.e.

* High-incomes
* Moderate-incomes
* Low-incomes

We can do that by writing our own function that will check which range the values of the median\_income column fall into.

We have defined our function convert\_categories in **lines 2-8**. Remember the values in the dataset are in tens of thousands of dollars. We can say that if the value is greater than 101010, it is a high-income housing block. If the values are between 2 and 10, it is a moderate-income housing block, and if the value is less than that, then it is a low-income housing block.

In **line 19** we select the column median\_income and use the apply function. We give it the name of the function we defined above and store the results as categories. We add a new column in the original dataframe named income\_category and save our categories in that column in **line 24**.

Now that we know how to operate on individual values in a column, we will see how we can use the whole column at once in the next lesson.

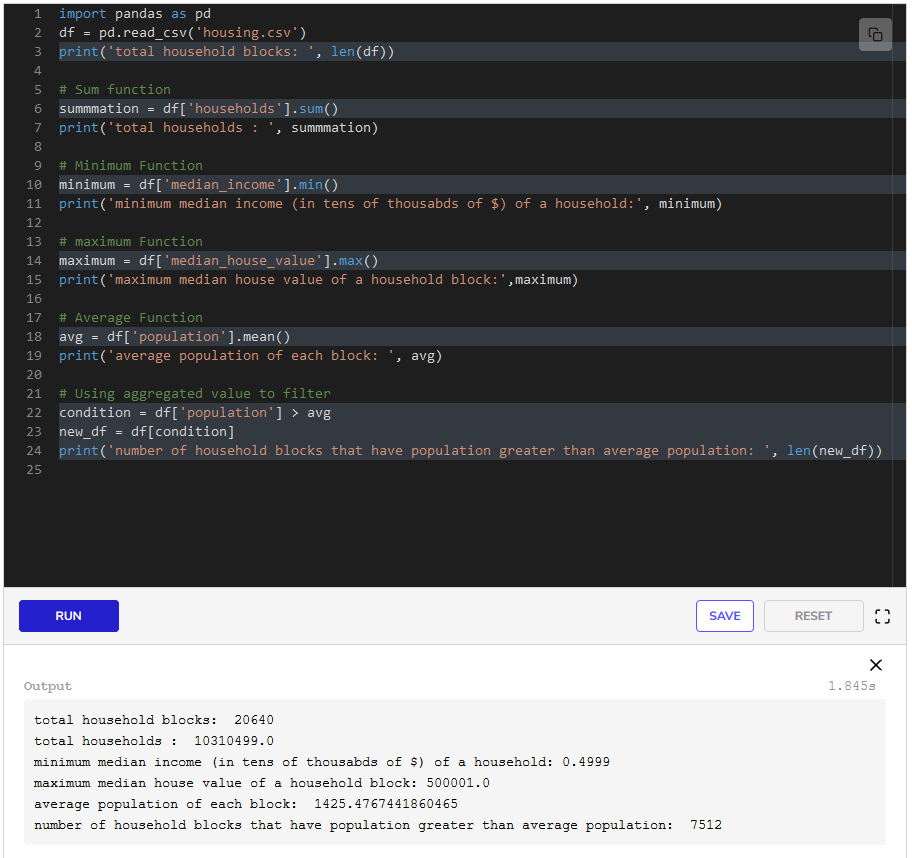
**Aggregating Data**

**Aggregation**

**Aggregation** is the procedure of converting a large number of values, or a dataset, into a single value or quantity aimed to summarize or describe the data. During data analysis, we always want to summarize data in one way or another. When we take a *sum*, *count* the number of items, or take the *average* of some values we aggregate data. Common aggregation methods are:

* *sum*
* *count*
* *maximum*
* *minimum*
* *average*

Aggregation is an essential step in analyzing data as it tells us the nature of the data in a single quantity. Let’s look at examples on our [California Census Housing Dataset](https://www.kaggle.com/camnugent/california-housing-prices) to see how this is true.

In **line 6**, we select the households column and find its sum. This means we have *aggregated* the whole column with just one value which tells us the total number of households in California. In the same way, we have found the mean, minimum, and maximum for different columns in the following lines.

In **line 23** we filter the data using the quantity avg we calculated above in **line 18**. This filtration tells us the number of household blocks with higher than the average population. Then we find the number of blocks that are extracted after this filter using the len function in **line 24**.

This gives us great insight that out of 206402064020640, only 751275127512 household blocks have a population higher than the average population of a household block. This was a use case of how we can use aggregations from data to filter the data.

Now that we know how to aggregate data and can calculate statistics for an individual column, we may want to go a little deeper and gather information about specific types of items in a column. We will see that in the next lesson.

**Grouping Data**

**Grouping**

During analysis, we may want to gather information about specific types of items in a column. For instance, we may want to separate the data for household blocks based on their proximity to the ocean in our [California Housing Dataset](https://www.kaggle.com/camnugent/california-housing-prices) and calculate the total population of household blocks in each type of area.

We simply want to *group* data for each type of area and then aggregate population. This can be easily done with Pandas using the function groupby. Below is an illustration of how grouping is done.

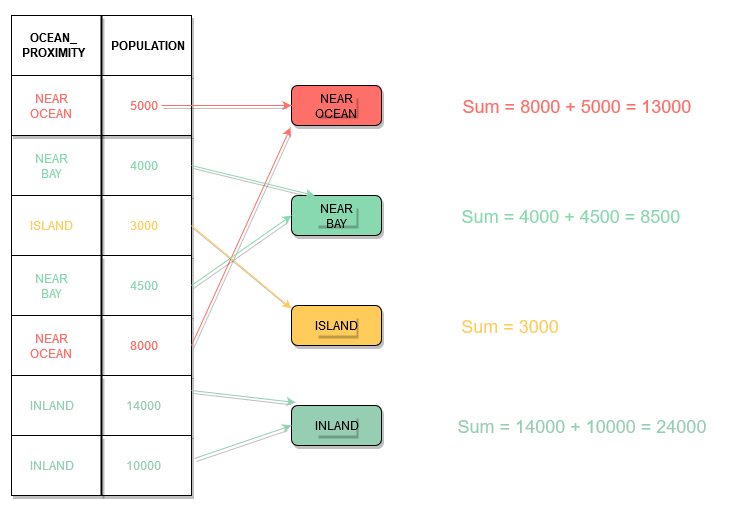
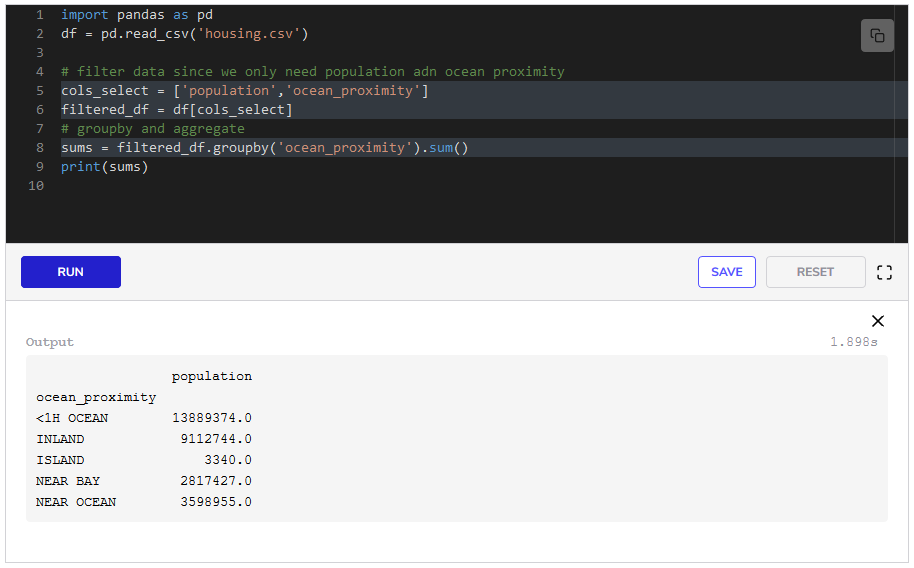


Illustration of how grouping and aggregating works

As we can see in the above illustration, for every row that has the value **NEAR OCEAN** in the ocean\_proximity column, the population values go to the red block and these values are aggregated to calculate a sum.

The same process is followed for all other values of ocean\_proximity column. Groups are created and we say that the data was *grouped by* ocean\_proximity.

Run the code below to see how easy it is to do this with Pandas. All of the data is in the file *housing.csv*.

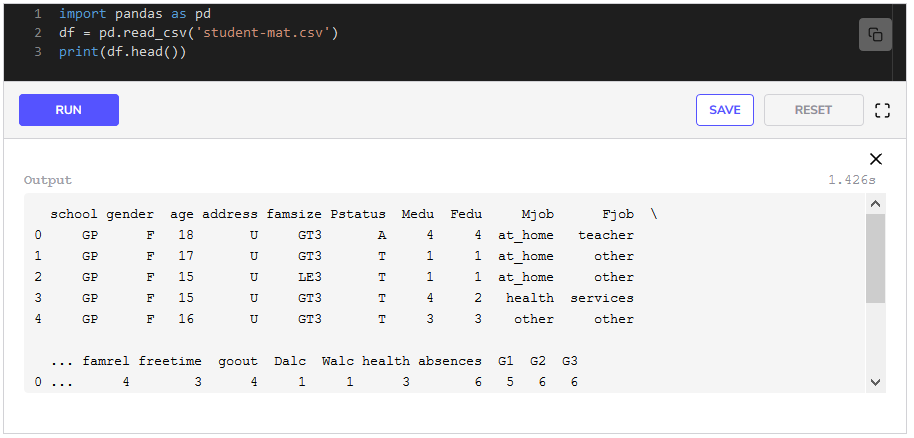
We first filter the data in **line 6** since we only need two columns. Then we use groupby() to group data by ocean\_proximity and take the sum in **line 8**.

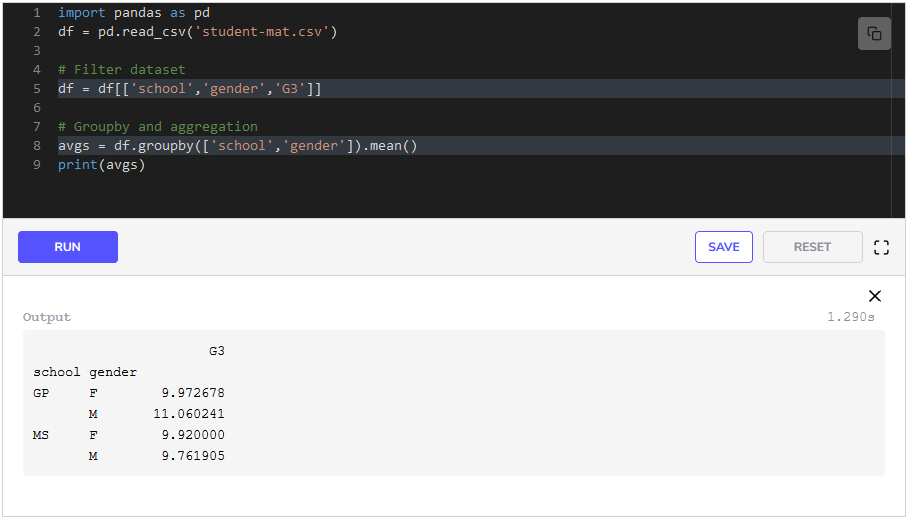
The output of **line 9** gives us another insight that <1H OCEAN areas have the highest population in California.

Usually, we use aggregation functions with grouped data, since we are interested in summarizing data for different groups.

**Grouping by more than one variable**

We will be using the [Student Alcohol Consumption Dataset](https://www.kaggle.com/uciml/student-alcohol-consumption#student-mat.csv). This dataset was made to understand how alcohol consumption and other factors influence the grades of school students. We have grades for math class in the file *student-mat.csv*. Let’s look at the dataset.

We might be interested in finding the average grade for all males and females. The final grades are given in the column G3. We can do that using a *two-level* groupby. We will group the data by school and gender and then find the average of the final grade (G3).

After reading the dataset, we filter the dataset in **line 5** since we only need three columns. In the next line, we use the groupby function and pass it a list of attributes for which we want to group the data. Then we find the average using the mean function. We can see from the output of **line 8**, that 444 groups have been made on the basis of school and gender which are:

* Females from school GP
* Males from school GP
* Females from school MS
* Males from school MS

Now that we know how to group and aggregate data, we will look at *pivot tables* in the next lesson.

**Pivot Tables**

**Pivot Tables**

**Pivot tables** are a summary of the whole data that give us useful information. These tables reorganize the desired data in a different format. These tables can transform data from columns to rows or rows to columns, or group data by any attribute. Because of the reorganization and transformation of the data, these tables were given the name **pivot**. These tables can include statistics such as sum, mean, maximum, minimum, and many more.

**Use cases**

Pivot Tables are used to:

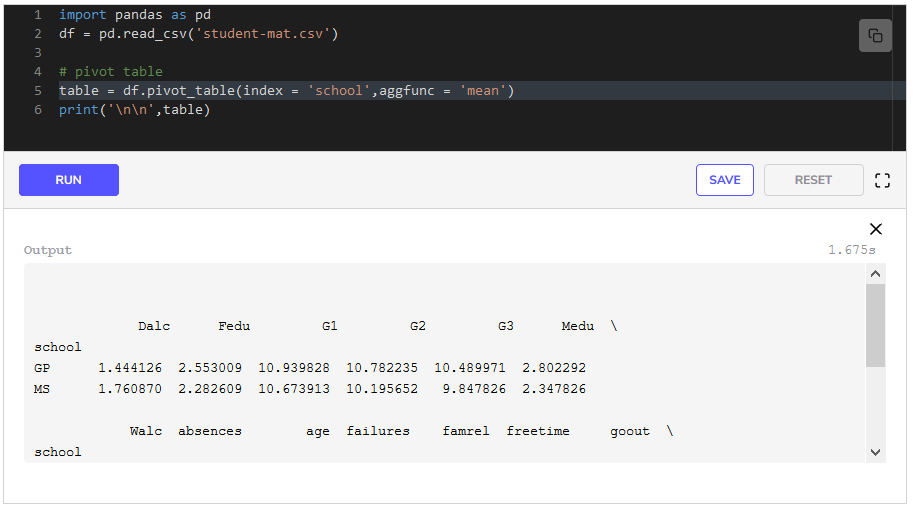
* Group data for business problems such as calculating sales by region or products
* Compare different classes of a data field such as comparing data for males and females
* Easily find out unique values in a field such as finding out different types of products for which we have data.
* Summarize complex tables

**Example: Students alcohol consumption data**

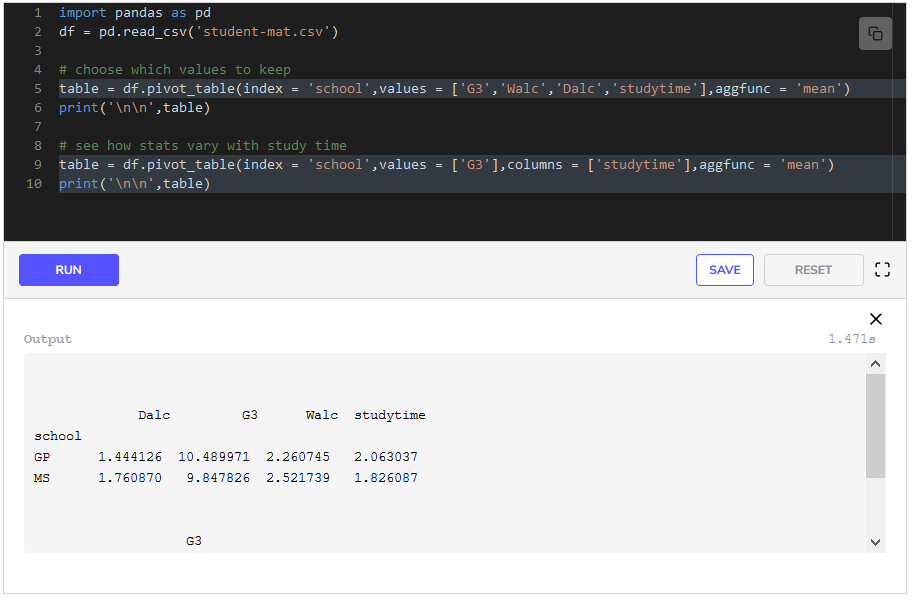
Let’s look at pivot tables with an example. We will be using the [Student Alcohol Consumption Dataset](https://www.kaggle.com/uciml/student-alcohol-consumption#student-mat.csv). This dataset was made to understand how alcohol consumption and other factors influence the grades of school students. We have grades for math class in the file *student-mat.csv*.

student-mat.csv

Now, a natural comparison that we may want to do is between schools. We can do that using the pivot\_table() function in Pandas.

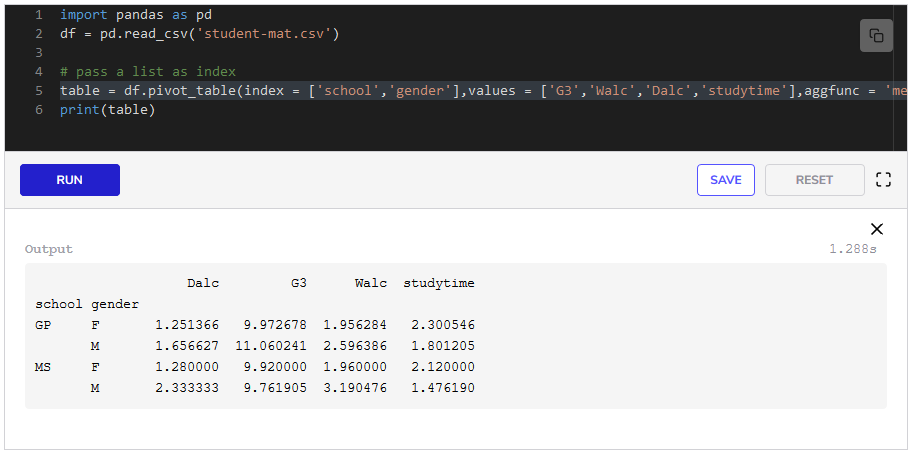
We have used the pivot\_table() function in **line 5** to calculate averages of all fields for the schools. We provide the name of the field for which we want to group our data as index to the function. Then we provide the aggregation function as aggfunc=mean. This averages all the fields.

But what if we are only interested in some of the fields?

In the case where we do not need all data fields, we provide the list of columns that we need to the function as values in **line 5**. We see that the school named GP performs better on average final grades(G3), their students consume less alcohol on weekdays(dalc) and weekends(Walc), and they study(studytime) more.

In **line 9**, we have provided studytime as columns to the function to see the final grades (G3) based on groups formed by studytime and school. By providing the column name studytime in a list, we tell the function to form columns according to the categories of studytime. This will give us the mean grade of all students, for each school, and for each category of studytime.

Now, what if we want to dive deeper into the data and see how different genders consume alcohol and perform in both schools?

We have provided two attributes that we need as index to the pivot\_table function in **line 5**. We get a table that categorizes genders with schools and gives the average statistics for each category.

From the pivot table formed, we can see that, on average the males from school MS consume more alcohol than males from school GP. We can also conclude that females form school GP study the most, on average.

Now that we know how to create and interpret pivot tables in Python, we will focus on plotting the data in the next lesson.

# Plotting Data 1: Univariate Plots

## Plotting

A **Plot** is a way of representing data visually. It can describe relationships between variables. Plots can give insight and information about the data that may not have been easily derived by looking at the data as a table.

Almost every tool that works with spreadsheets can plot data. Python has a complete package dedicated for plotting data known as matplotlib. We will be using this package with pandas to plot data. In this lesson, we will discuss **univariate plots**. These are plots where a single variable is plotted.

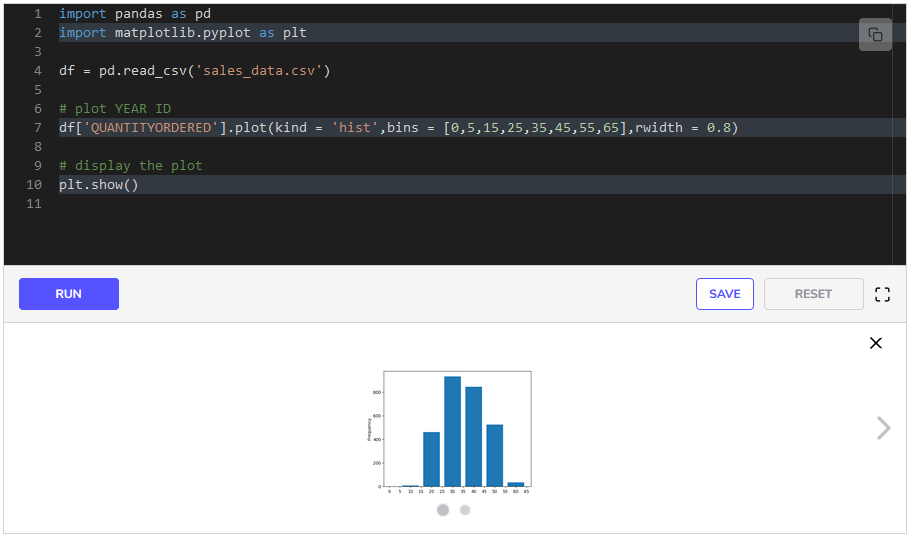
Let’s look at an example of this using the [Sample Sales Data](https://www.kaggle.com/kyanyoga/sample-sales-data). The data is in sales\_data\_sample.csv file. Have a look at the data.

### Plotting individual columns

We can easily plot distributions of a single individual column for our dataframe. Let’s look at an example below. Run the code below and click on the output graph to view it in full screen.

#### Histograms

**Histograms** are plots that inform us of the underlying frequency distribution of the data. They group numbers into ranges and tells us how many values of a variable lie in every range.



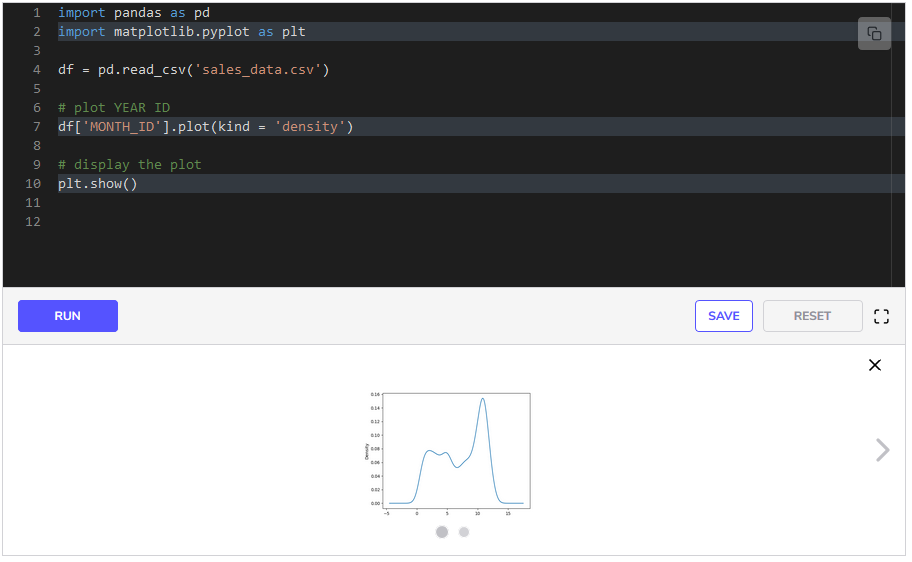
We import matplotlib.pyplot package in **line 2** as plt. We would use the keyword plt wherever we want to use it in the code.

We have plotted the histogram of QUANTITYORDERED in **line 7**. We have set kind to hist. We can provide two more inputs with hist to the function; bins to specify our ranges and rwidth to specify the space between each bar in the graph. If we do not provide these inputs, the function automatically decides some values for the bins and rwidth. Alternatively, we can provide the number of bins to this function instead of a list and it makes the provided number of bins automatically.

From the graph, we can see that the firm received more than 800800800 orders in which the quantity ordered was in the range of 25−3525-3525−35.

#### Density plots

A **density plot** is a representation of the distribution of a numeric variable. It uses a density estimate to show the probability density function of the variable. It is a smoothed version of the histogram and is used in the same concept.

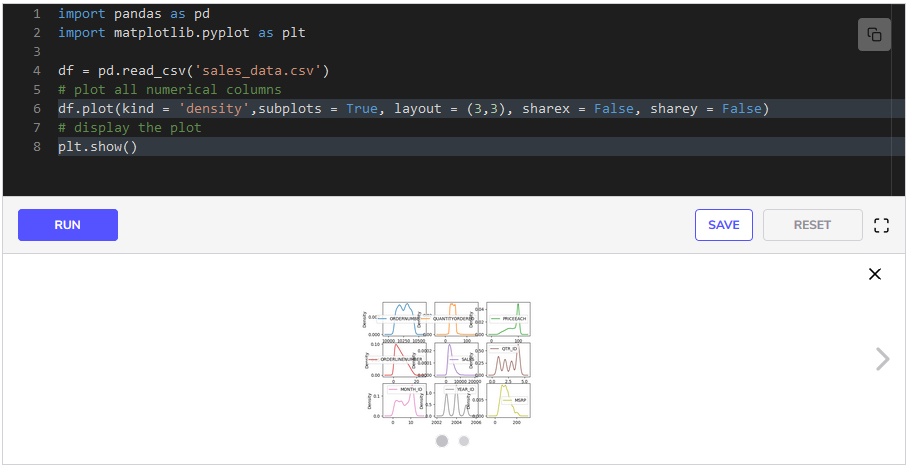
In **line 7** we plot the data in three simple steps:

* Select the column MONTH\_IDby writing df['MONTH\_ID']
* Use the plot function by writing .plot()
* Tell the function the type of graph we want as kind = 'density'

We use plt.show() to display the graph in **line 10**. From the density plot, we can see a continuous curve that explains the frequency distribution of MONTH\_ID. It tells us how much data we have in each month. We see a peak at months 101010 and 111111, which tells us that most orders were made in these months.

### Plotting multiple columns

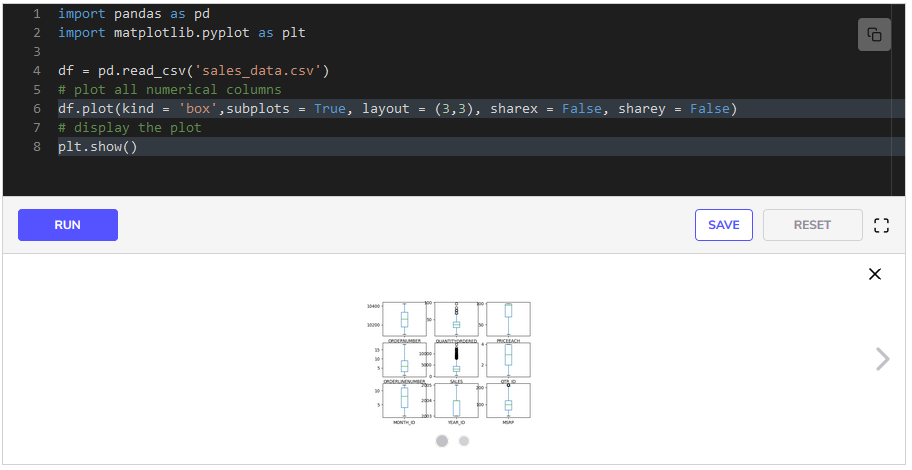
We can plot distributions for all of the columns that have numerical values in a single line with pandas and matplotlib. Let’s see an example of this.

In **line 6**:

* We use the plot function on the whole dataframe by writing df.plot()
* We specify the type of the plots as kind= 'density'
* Since we are using plot function on the whole dataframe, we need to provide some other inputs to the function that will tell it how to draw the plots.
  + We pass subplots = True which indicates that multiple plots are to be drawn.
  + We pass sharex = False and sharey = False so that the plots do not share x-axis and y-axis.
  + We pass layout = (3,3) so that the plots are arranged in a grid and look visually better. This is an optional argument.

Finally, we display the plots in **line 8**. The plots show us the distribution of variables by density plots.

Similarly, we can plot box plots for the dataframe by passing kind = box to the plot function. Run the below code to see them.

Box plots give us information about the spread of the data. They tell us about the mean and the quartiles of each variable.

We can now plot univariate plots in Python. In the next lesson, we will learn how to plot bivariate plots.

**Plotting Data 2: Bivariate Plots**

**Bivariate plots**

**Bivariate plots** are plots that aim to graph the relationship between two variables. These plots can help us establish relationships and patterns in the data. Some common bivariate plots are:

* *scatter plots*
* *bar plots*
* *histograms*

Let’s look at some of these plots and how we can plot them using Pandas and matplotlib. We will be using [Sample Sales Data](https://www.kaggle.com/kyanyoga/sample-sales-data). The data is in *sales\_data\_sample.csv* file. Have a look at the data.

sales\_data.csv

**Bivariate plots with dataframe.plot**

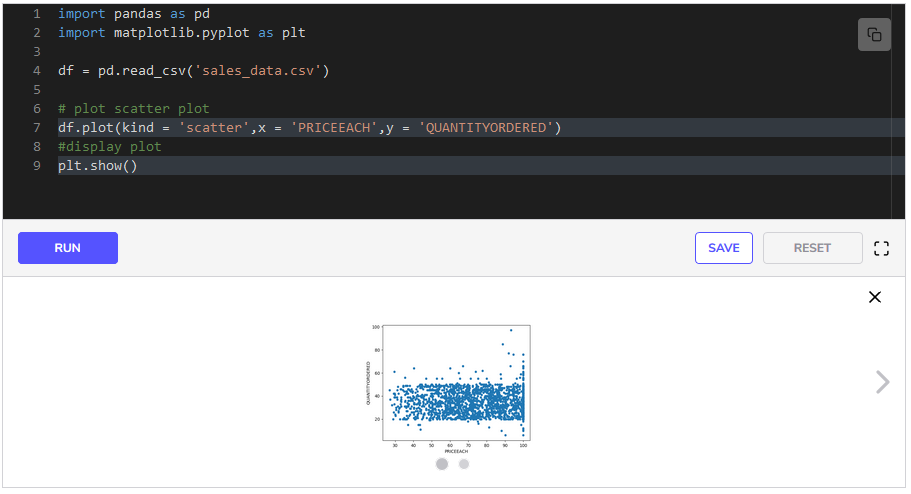
In Pandas, we only need to use the plot function with the dataframe to plot bivariate plots. The function expects the following arguments:

* kind: the type of plot, i.e., scatter, bar etc.
* x: name of the column to place at x-axis.
* y: name of the column to place at y-axis.

Let’s look at some of these bivariate plots below.

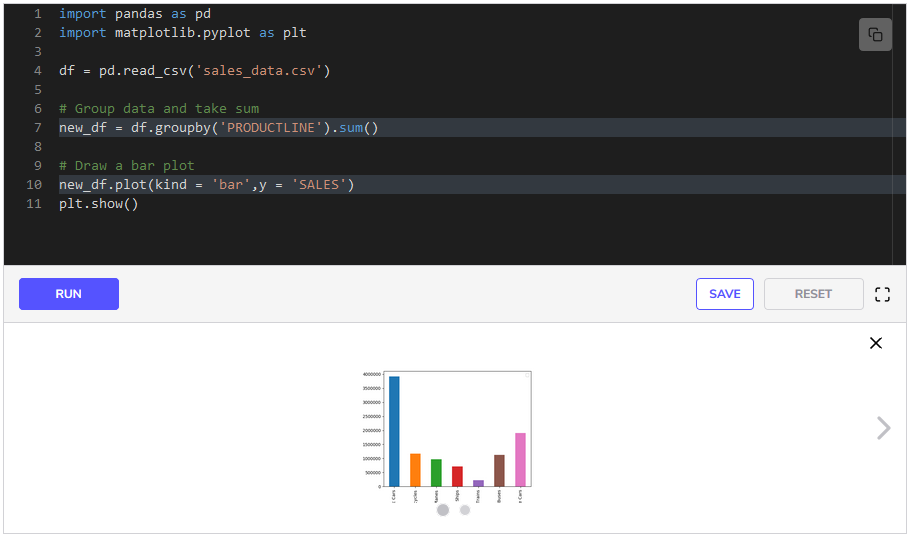
**Scatter plot**

A **scatter plot** is the simplest kind of plot between two variables. One variable is plotted on the x-axis while the other is plotted on the y-axis. The pattern of the plot reveals a general trend or any correlation present between the two variables.

In the code above, we have plotted two quantities, PRICEEACH on the x-axis and QUANTITYORDERED on the y-axis in **line 7**. The graph shows that the market in which this firm is operating favors products in the price range of 60−8060-8060−80, as that part of the plot is very dense.

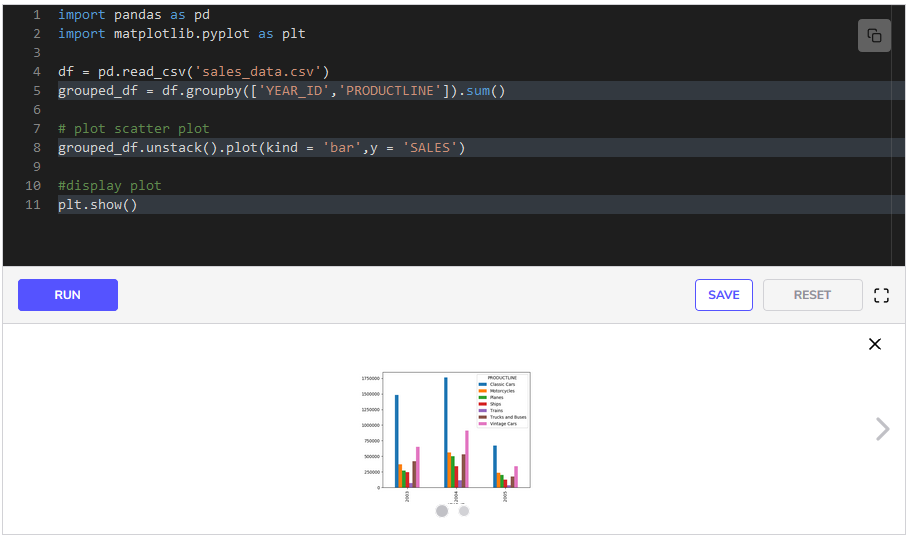
**Bar plot with groupby**

A **bar plot** is similar to a *histogram*, but it is a bivariate plot. The length of the bars represents the quantity on the y-axis. Bar plots are usually used with a grouped-by variable that is plotted on the x-axis. Let’s look at an example.

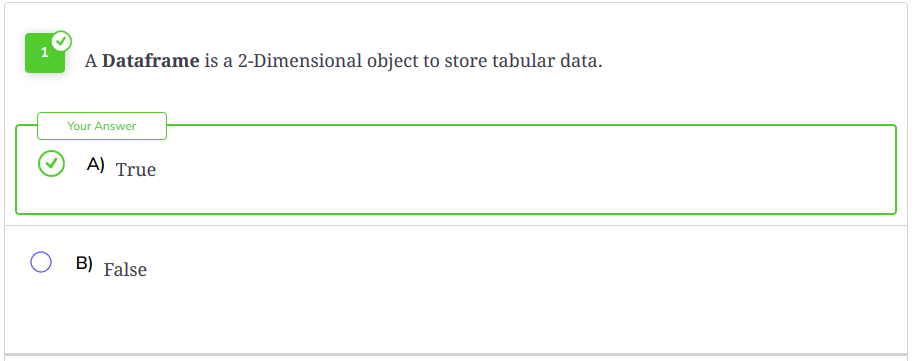
In **line 7**, we grouped the data by the variable PRODUCTLINE and then added all the sales for them so that we have total sales for each product line. Then we plot in **line 10**. Here we do not need to provide the x input as it is already a grouped dataframe and it is implied that the grouping variable, PRODUCTLINE, will be on the x-axis. This plot gives us a good comparison of the total sales of each product line.

**Bar plot with two-level groupby** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/NE8JN1wjry6#bar-plot-with-two-level-groupby)

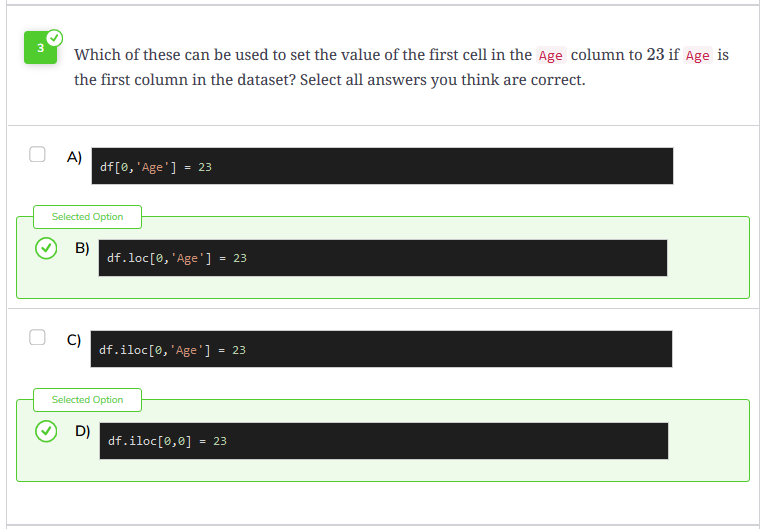
We know that we can group data by two variables instead of just one which can give us great insights into the data. We can also plot this two-level grouping. For instance, we want to see the total sales for each product line, year wise. Let’s plot this below.

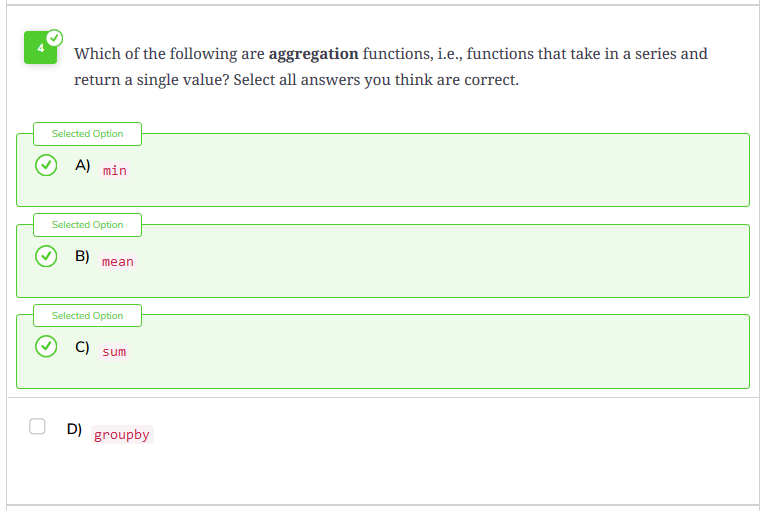
In **line 5** we group data by YEAR\_ID and PRODUCTLINE and take the sum since we want to calculate total sales. In **line 8** we plot it. But before using the plot function, we have used the unstack function. unstack, changes the hierarchical structure of the grouped dataframe. If we had plotted without using unstack then the bars for a single year would not have been grouped together as they are in the plot.

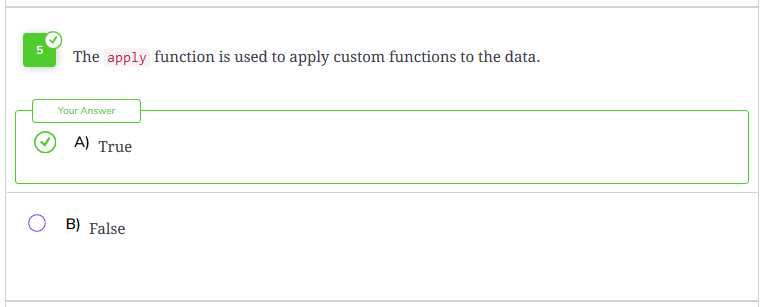
Now that we know how to plot and handle CSV data in Python, we will move to the next chapter where we will learn how to clean our data for analysis. But before that, you can take a quiz in the next lesson to test yourself on the concepts learned in this chapter.

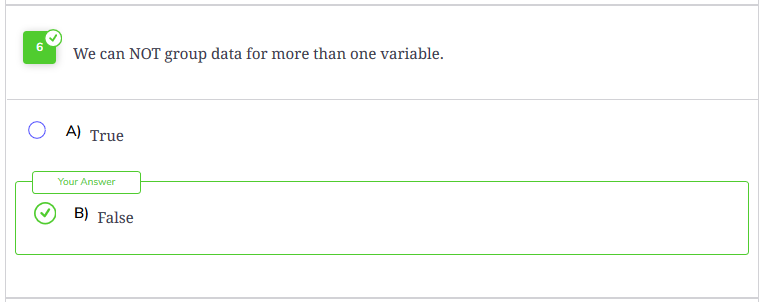


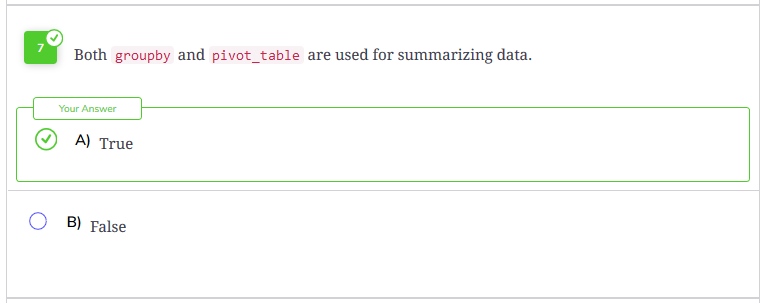


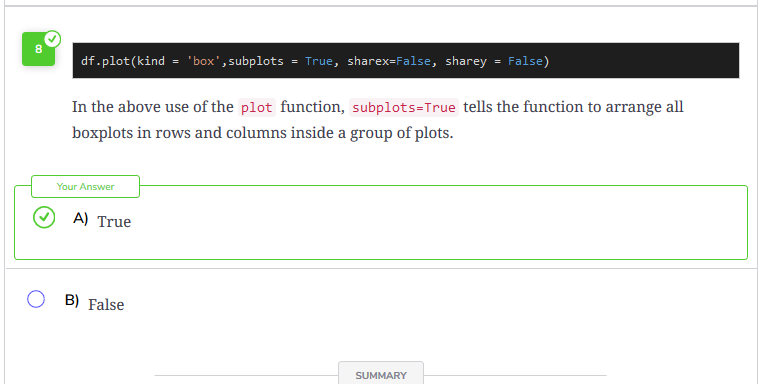












**Introduction to Data Cleaning**

In this chapter, we will look at the third stage of the Data Science Lifecycle - *Data Cleaning*. But before we look at what steps are involved in Data Cleaning, a question arises; why do we need to clean data?

**Why clean data?**

The data that we receive and use is not perfect. Numerous factors such as data collection from multiple sources, or data corruption while storing or retrieving data, human errors in entering data, data loss while transferring data on some network, etc, can lead to incomplete, inconsistent, and incorrect data. If we use data as received in our analysis, then we will perform incorrect analysis and any conclusion drawn from the data will be wrong. Therefore, data cleaning is a necessary step before doing any analysis on the data.



Cartoon by Mark Anderson, www.andertoons.com.

**Cleaning data**

**Data cleaning** or **cleansing** is the process of detecting and correcting inconsistent, incorrect, and extraneous data. Data cleaning involves dealing with

* Missing data
* Duplicated data
* Outliers in the data
* Extra data that might not be needed
* Inconsistent data
* Converting data into a standard format so that it is easy to work on

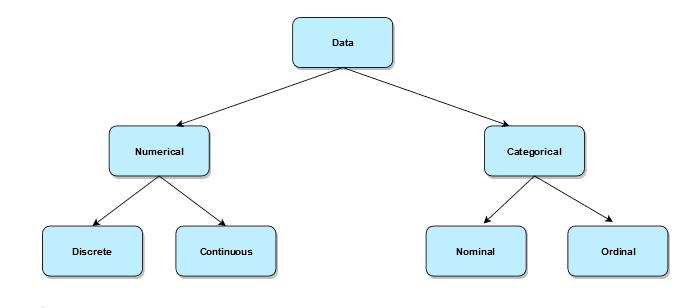
We will look at all of these aspects in the upcoming lessons. But before that, we need to know *data types*. We will explore them in the next lesson.

# Data Types

## Introduction to Data types

Data is basically raw information. It can be in any form. We will look at data from the perspective of a data scientist who is going to clean and analyze the data. We need to know in what form the data is present to analyze it properly and apply different statistical methods on it. To a data scientist, data can take two basic forms:

* **Numerical**
* **Categorical**



### Numerical data

Numerical data is data that has some meaning as a measurement, such as the height of a person, the price of a product, the IQ of a person, the number of lessons in this course, etc. It is also known as Quantitative Data. It can be broken down into two types.

#### 1. Discrete data

Discrete data is data that can take separate and distinct values. It can take only a certain number of values. It cannot be divided into smaller meaningful parts. For instance, the number of heads in 100 tosses of a coin flip, the number of students in a classroom, the number of cars in a showroom, etc.

#### 2. Continuous data

Continuous data cannot be counted, but it can be measured. It represents measurements. It includes quantities that do not have an end to them such as money, the height of a person, the amount of rainfall, the speed of a car, etc. It can be divided into further meaningful parts.

We can use statistical methods such as mean, median, quartiles, Box plots, and Histograms to describe numerical data. 

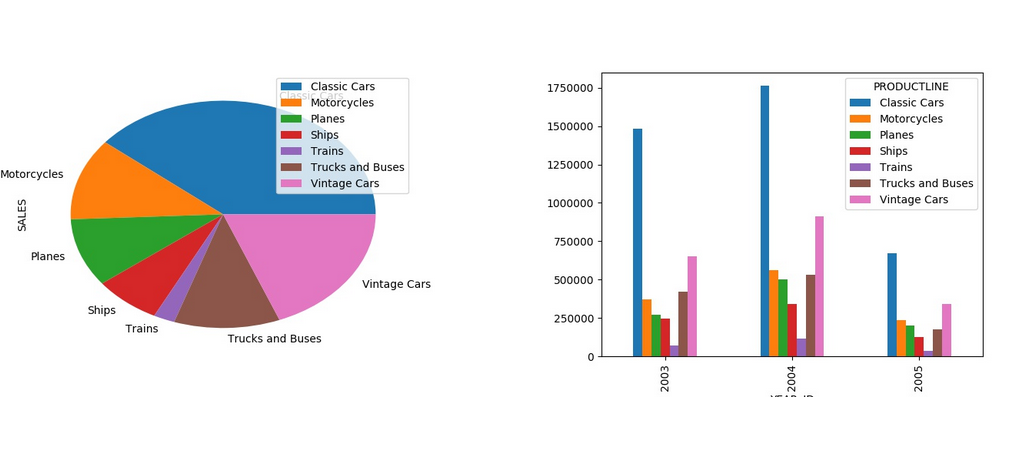
Example of box plot and histogram

### Categorical data

Categorical data as the name suggests, represent categories or characteristics such as gender, language, level of education, marital status, the genre of a movie, etc. It is also known as Qualitative Data. We can associate numerical values with categorical data, but they would not have any mathematical meaning, e.g., 0/1 for male/female.

#### 1. Ordinal

Ordinal data is categorical data that has a sense of order to it. For instance, the happiness level of a customer, level of education, or rating of a movie on a scale of 0−50-50−5.

We can summarize ordinal data with percentiles, frequencies, median, mean, etc. For visualization, we can use pie charts and bar charts. 

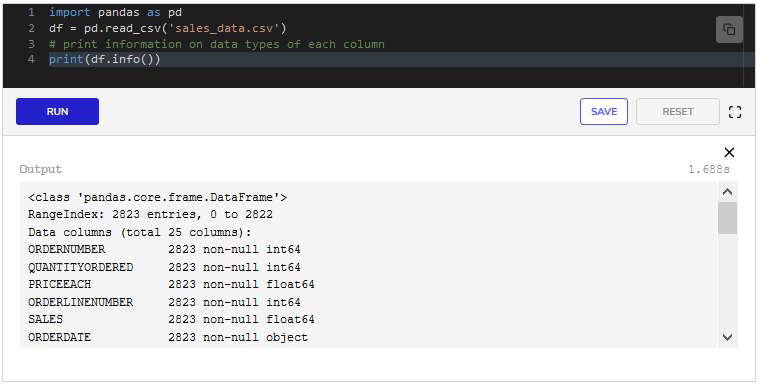
Example of pie chart and bar plot

#### 2. Nominal data

Nominal data is categorical data that has no order. It can be thought of as labels. For instance, the gender of a person as male orfemale, the language a person speaks, etc. Nominal Data can be dealt with using frequencies, proportions, pie charts, bar plots, etc.

### Figuring out Data types in Pandas

In pandas, it is very easy to figure out the data types of the variables. We can use the info function on our dataframe.

The output of df.info tells us the count and type of each column that we have in our dataset. int64 means it is an integer, which means it is a discrete or ordinal variable. float64 means it is a number with a fractional part, therefore, a continuous variable. We can infer from object that it will be a nominal variable.

However, sometimes looking at the output of df.info is not enough. There are instances when nominal data is written in numbers, such as gender is written as 1/21/21/2 or 0/10/10/1 instead of “male” and “female”. So, we have to look out for those cases as well.

Now that we know how our data is available to us, in the next lesson, we will dive into analyzing individual variables to get insights.

# Missing Values

## Missing values

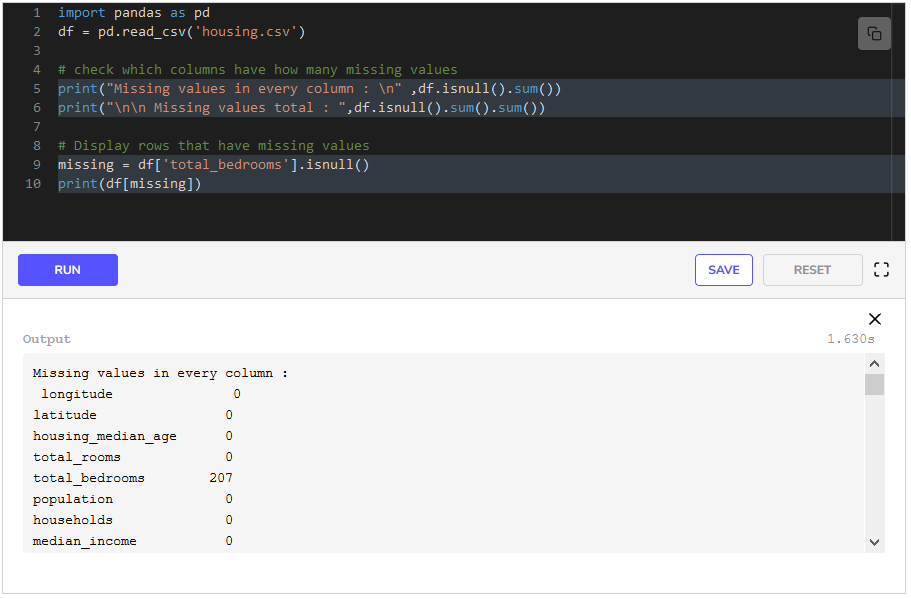
During data collection and entry, it is possible that some values are missed, or data was not available for some entries. Hence, missing data is very common among data science applications.

Pandas makes it very easy to work with missing data. It does not include missing values in all of its different calculations such as sum, mean, etc. by default.

Pandas writes the value NaN(Not a Number) when it finds a missing value.

### Detecting missing values

We can detect missing values using the function isnull. It returns True wherever there is a missing value, and False, otherwise.



In **line 5**, we use the function isnull and then use sum on it. This gives us a list of all columns with the number of missing values in them. From the list, we see that total\_bedrooms has 207207207 missing values. If we take another sum then we have the total number of missing values in the whole dataset, as done in **line 6**.

In **line 9**, we use isnull on total\_bedrooms column and get a list of True and False. In the next line, we index using this list and display all the rows that have missing values in the total\_bedrooms column.

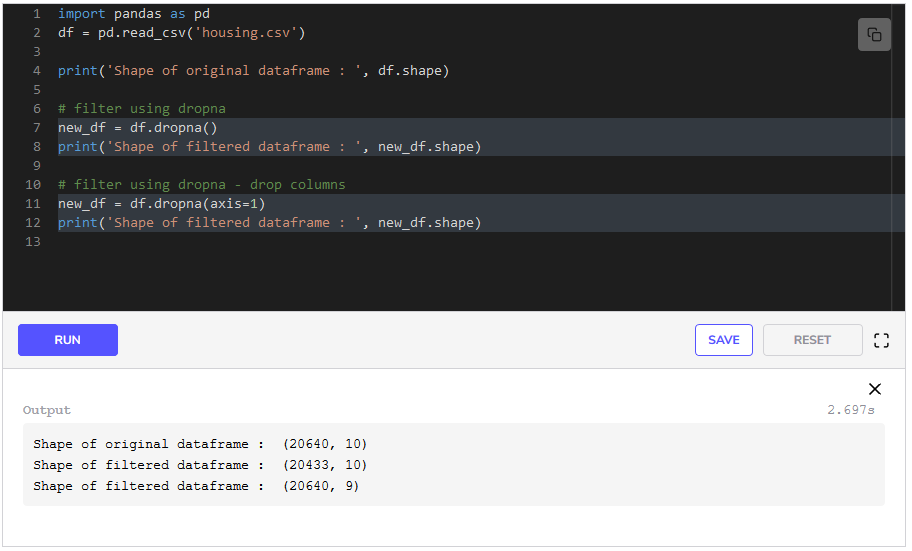
#### Passing custom keywords for missing values

The function isnull detects NaN or empty cells. But what if at the time of data entry, someone wrote a custom value where values were missing, such as NA, missing, or N/A. We can pass a list of these keywords when reading the files to read\_csv as missing\_values, and Pandas will perceive these keywords as missing. For instance, if our data has the terms missing and N/A for missing values, we can write

pd.read\_csv('filename.csv',missing\_values = ['missing','N/A'])

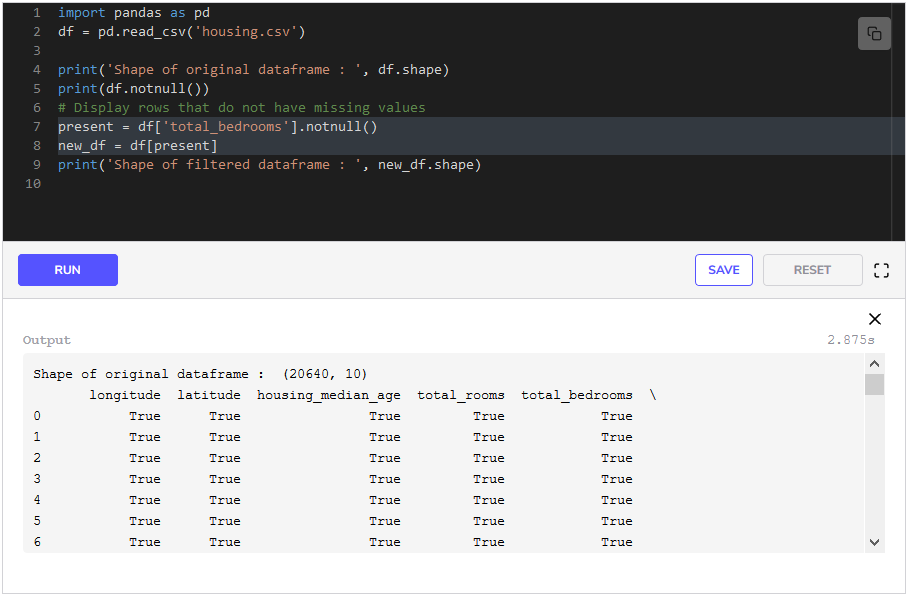
### Filtering missing values

Pandas has a function dropna which filters the rows containing missing values from the dataframe.

**In line 7**, we create a new dataframe which has filtered rows that had missing values. We can see from the output that new\_df has fewer rows. If we pass axis=1, then every column that has a missing value will be dropped. As it can be seen from the output of **line 12**.

#### notnull

As an alternative to dropna(), we can use the notnull() function to remove rows that have missing values in one or more columns.

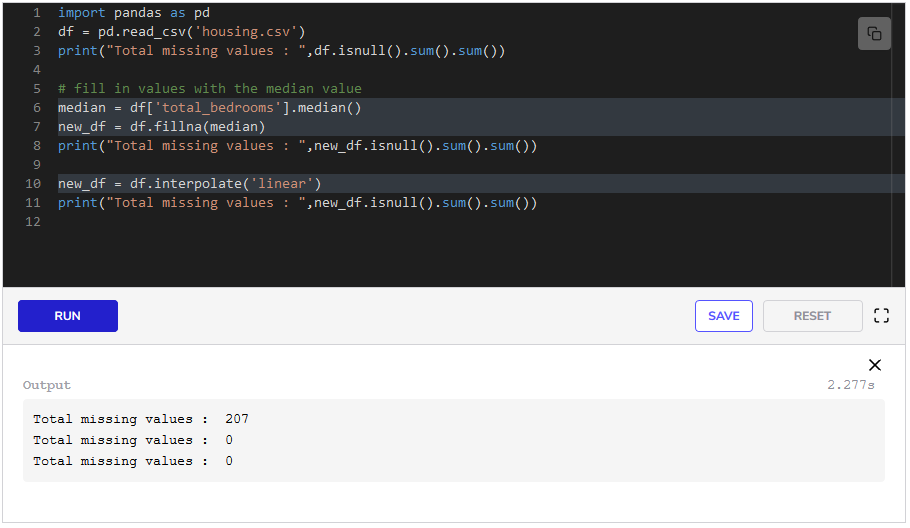
In **line 7**, we use the function notnull on the total\_bedrooms column to get a list of rows that do not have missing values for total\_bedrooms. In the next line, we filter using this list. As we can see from the output of **line 9**, that the new dataframe has fewer number of rows.

### Filling numerical missing values

The strategy behind filling missing values can vary from problem to problem. In some cases, filling in with the mean or median value works very well, while in some cases **interpolation** works better. Interpolation is a mathematical technique of constructing new points from a range of points.

Pandas provides two very easy ways to fill in missing values.

* We can provide any value to the function fillna to fill in the missing values with.
* We can specify the type of interpolation we want to the interpolate function.

We first find the median in **line 6** and then pass it to fillna to fill the missing values.

We only provide the name of the interpolation to the interpolate function in **line 10**.

### Filling non-numerical missing values

For non-numerical missing values, we can simply write missing because we cannot use mathematical methods like mean, median, or interpolation to estimate these.

If the variable in question is a very important variable for which we need values, then it may be better to simply disregard the observation.

We have learned how to detect, filter and fill missing values. In the next lesson, we will learn how to deal with duplicates.

**Duplicates**

**Duplicates**

Repeated data rows in the dataset are called **duplicates**. These can arise from a number of ways. The most common are:

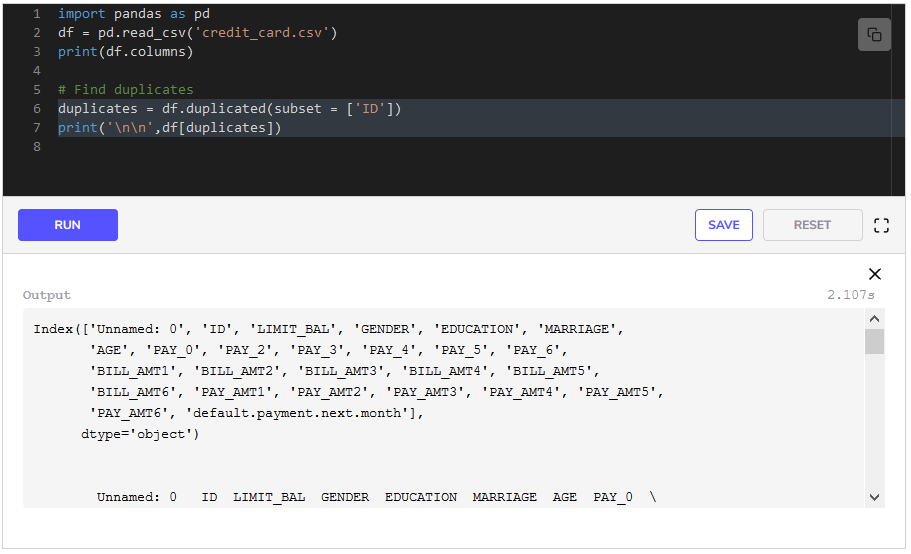
* The same data is entered twice by accident, such as the same article is scraped twice or booking for an online product is made twice.
* If data is being collected in online forms or surveys and the user presses the submit button twice.
* If data is collected from multiple sources.

**Detecting duplicates**

In every dataset, there are some or one attribute that makes the records unique from each other. For instance, the order ID in the sales data, the student ID in the data of students, the longitude and latitude in the Census data, etc. We can search for duplicates using these key variables.

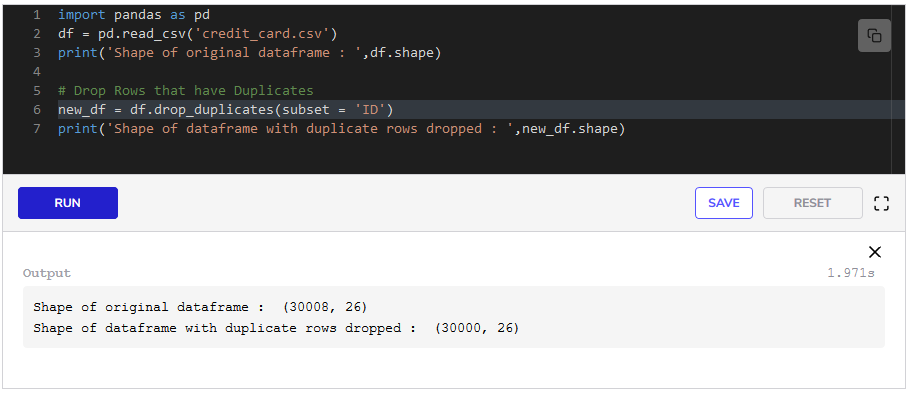
Pandas has a function, duplicated for finding duplicates. We specify the column names, and it gives us a list of Booleans (True and False) putting True against rows that have duplicates. It puts False for the first occurrence of a duplicate.

We will be using the [Credit Card Default Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset).

In **line 6** we use the function duplicated. We provide a list of column names as subset for which we want to find duplicates and it gives us a list. We index using that list in **line 7** and get the duplicated rows.

**Removing duplicates**

Duplicates can be removed by using the function drop\_duplicates. It has to be provided the list of column names in the same way as we provided to duplicated above.

We drop the duplicated columns in **line 6** and verify that by looking at the shape of the new dataframe. 888 rows have been dropped.

This was how to deal with duplicates in data. In the next lesson, we will focus on how to deal with inconsistent data.

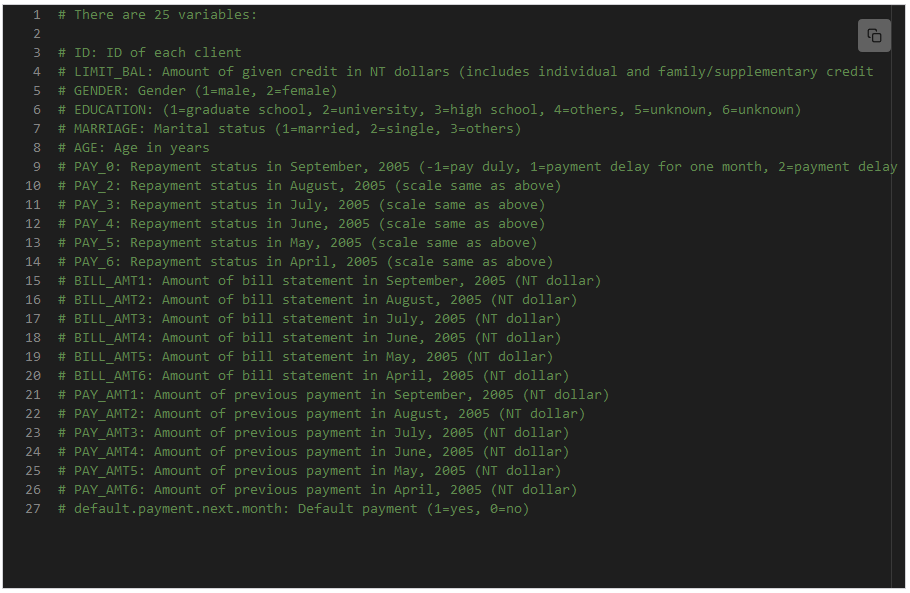
# Inconsistent Data

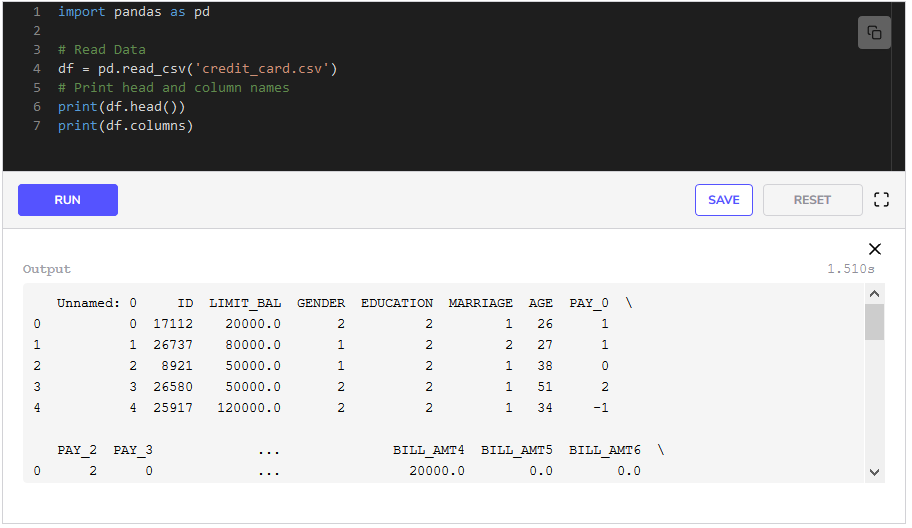
**Inconsistency** in data arises due to errors in collecting data. For instance, if the data was collected from multiple sources, or if the data was collected by multiple people who did not follow the same format of collecting data, then there is a high chance of inconsistencies in the data.

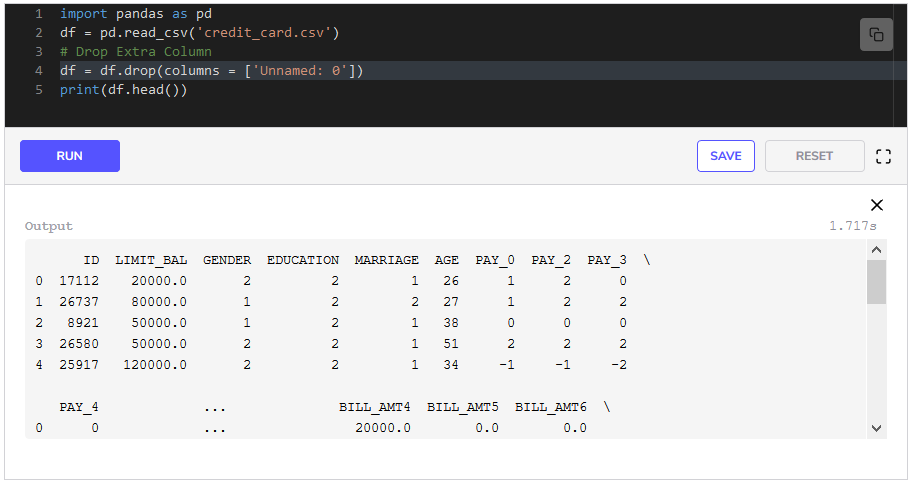
In this lesson, we will be cleaning the [Credit Cards Default Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). This dataset is a very good example of the kind of inconsistencies that are present in most datasets.

## Credit cards default dataset

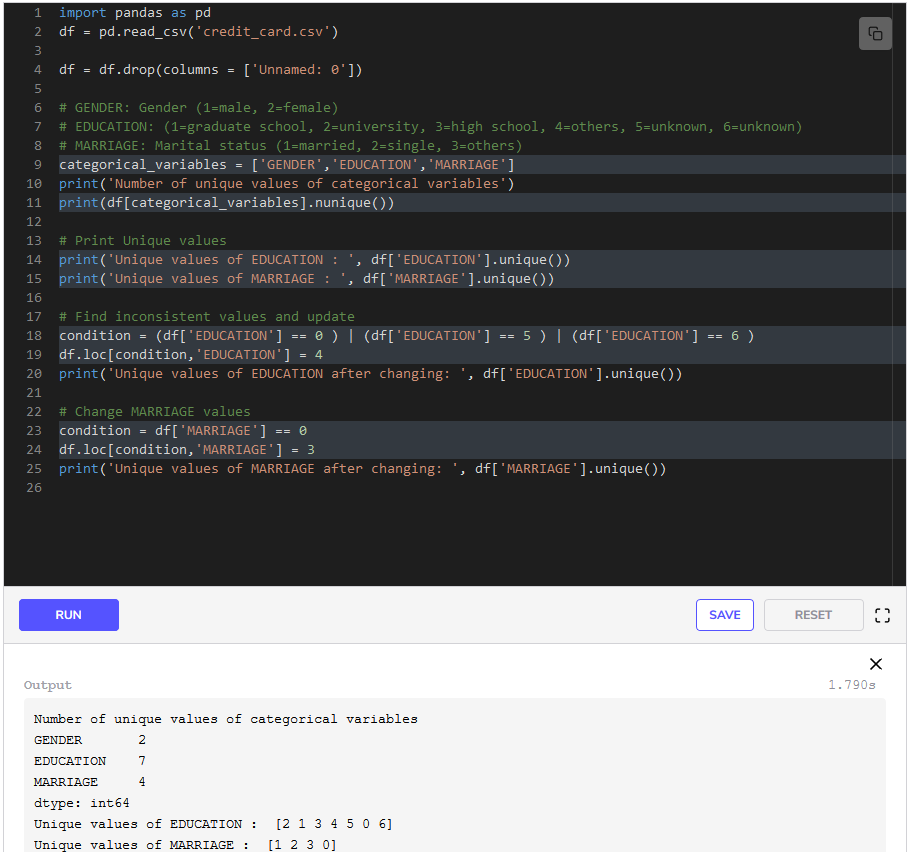
The documented details of individual columns are mentioned below. But we will see that our dataset will not be consistent with this format.

Let’s load the dataset.

Just by looking at the output, we can see that pandas keeps serial numbers for us automatically, and since we have IDs in the ID column, we do not need the first column, we can remove it. We can use the drop function to drop columns by specifying their names.

We have used the drop function on **line 4** and dropped the extra first column as can be seen from the output of **line 5**.

### Categorical variables

The usual check for inconsistent data is usually the number of unique values that categorical variables take. 

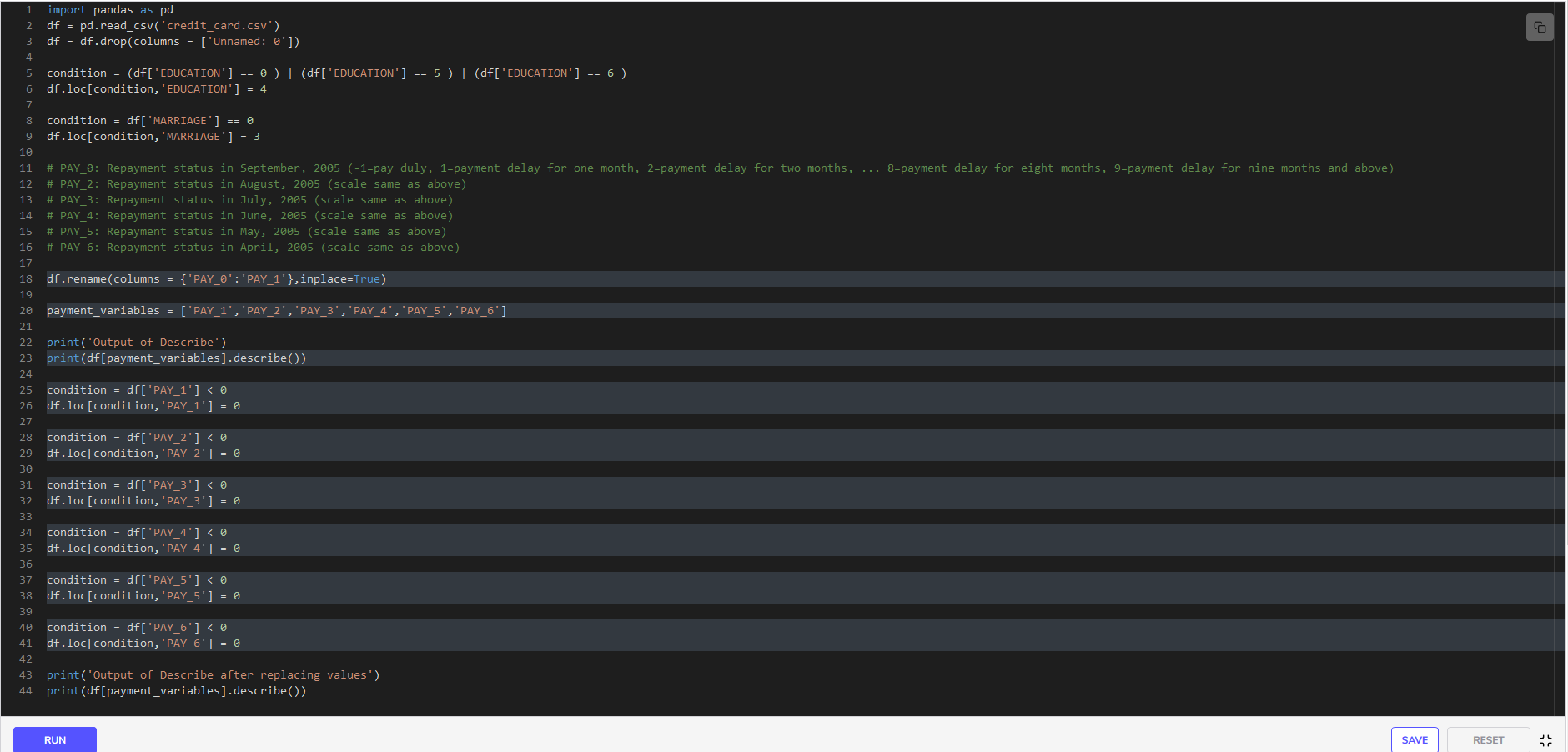
In **line 11**, we selected the categorical variables that we had written in a list in **line 10**, and used the function nunique to output the number of unique values that each variable takes. By looking at the output of **line 11**, we see that EDUCATION and MARRIAGE have more unique values than specified. Therefore, we print these unique values by using the unique function in **lines 14-15**.

In the case of EDUCATION, the label 0 is undocumented and 5 and 6 indicate unknown values. We can replace all three of these values with 4 as that will indicate unknown or other levels of education than 1(graduate school), 2(university), and 3(high school). To accomplish this, we define condition to be the rows where the values of EDUCATION is 0, 5 or 6 in **line 18**. The operator | means that we will select the row if any of the three conditions is true. In **line 19**, we use loc to specify the location where we will be making the changes. We give condition to specify the rows and EDUCATION to specify the column. We place 4 at these locations.

In the same way, in **line 24** we have placed 3 at the places which had 0 in the MARRIAGE columns as 0 is undocumented. In **lines 20** and **25**, we check the unique values in EDUCATION and MARRIAGE respectively to confirm our changes.

#### Payment delay variables

Now let’s check the payment delay variables for inconsistencies. We can use the describe function to see if the columns have values outside the scale by looking at the minimum and maximum values instead of looking at the unique values for each column.

When we look at the names of the payment variables we see an inconsistency that the names follow a pattern. Therefore, we change the name of the variable for September from PAY\_0 and to PAY\_1 in **line 18**.

We write the names of these payment variables in a list in **line 20**. Then we use the function describe on these columns in **line 23**. We see that the minimum value for all of these columns is −2-2−2, which probably indicates that the payment was duly paid. To deal with this, we decide to replace all negative values in these columns with 0.0.0.

We locate rows that have negative values in the PAY\_1 column on **line 25**, and then set such cells to 0 on **line 26**. The same process is repeated for PAY\_2, PAY\_3, …, PAY\_6 columns as well.

This was a lesson on how to deal with inconsistent data. In the next lesson, we will see how to deal with outliers in the data.

**Outliers**

**What are outliers?**

**Outliers** are observations that are significantly distant from other observations. These do not follow the general trend of the data. Outliers can indicate variation or error in the data. Outliers in a single variable/column are called **univariate** while outliers in multiple variables/columns are called **multivariate**.

**Sources of outliers**

Outliers can be caused by a variety of reasons. Some common ones are:

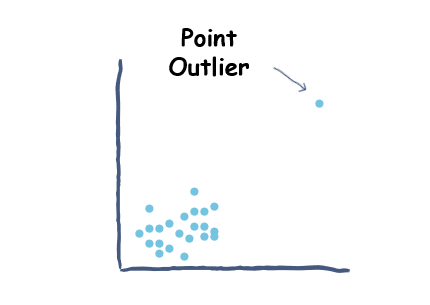
* Errors in entering data.
* Errors in measuring data, e.g., errors in the measuring instrument.
* Errors in collecting and merging data from multiple sources.
* Errors in processing data.
* Natural variance because of some unknown reason.

**Types of outliers**

Outliers can be classified into three broad categories:

**1. Point or Global outliers**

These are observations that deviate from all of the other observations, e.g., if the temperature is recorded as 100 degrees Celsius, or a person who usually spends $100 in a week spends $500 this week.



Point Outlier

**2. Contextual or Conditional outliers**

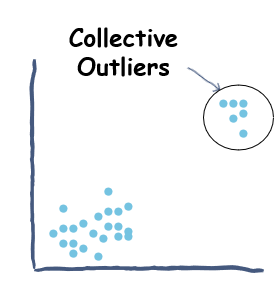
These are data points that are not outliers globally, but are outliers in their own context. If we look at a subset of the data, then we are looking in a context.

For instance, a sudden unusual temperature drop in the summer season is considered a contextual outlier where the context is the summer season.

Another example could be if the price of a good is $15 and its price falls below $10 during the Christmas period. If its price falls below $10 in July, then that would be a contextual outlier with the month of July being the context.

**3. Collective outliers**

These are a group of observations that are outliers globally from the rest of the observations but are not outliers within the group. An example could be a sudden increase in stock transactions of a particular company during a month or unusual delays in shipping orders over a period of three days.



Collective Outlier

Since we are now familiar with the existence of outliers, we will look at the detection and removal of outliers in the next lesson.

**Outlier Detection and Removal**

**Outlier detection**

Detecting outliers is a very important step in data cleaning and exploring. It gives us an idea of the anomalies in the data which can give us valuable insights into the data. So, how can we detect outliers?

Outliers can be detected both visually and mathematically. Some plots are very helpful in visualizing outliers, such as box plots and scatter plots. However, it is sometimes tricky to decide whether or not to remove the outliers. We should remove outliers when we are certain that these outliers were results of some errors.

We will discuss some of the methods to detect and remove outliers. We will be using the [Sample Sales Data](https://www.kaggle.com/kyanyoga/sample-sales-data). The data is in the file *sales\_data.csv*.

sales\_data.csv

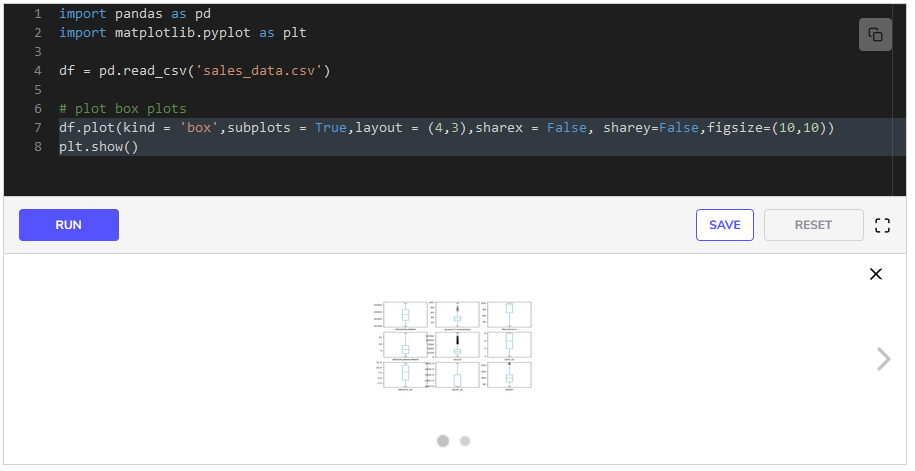
**Box plots and Quantile ranges**

Box plots, by definition, plot outliers as points and group the rest of the observations. The criteria of a box plot for classifying a point as an outlier is if the point is greater than Q3+(1.5∗IQR)Q\_3+(1.5\*IQR)Q​3​​+(1.5∗IQR) or lower than Q1−(1.5∗IQR)Q\_1-(1.5\*IQR)Q​1​​−(1.5∗IQR).

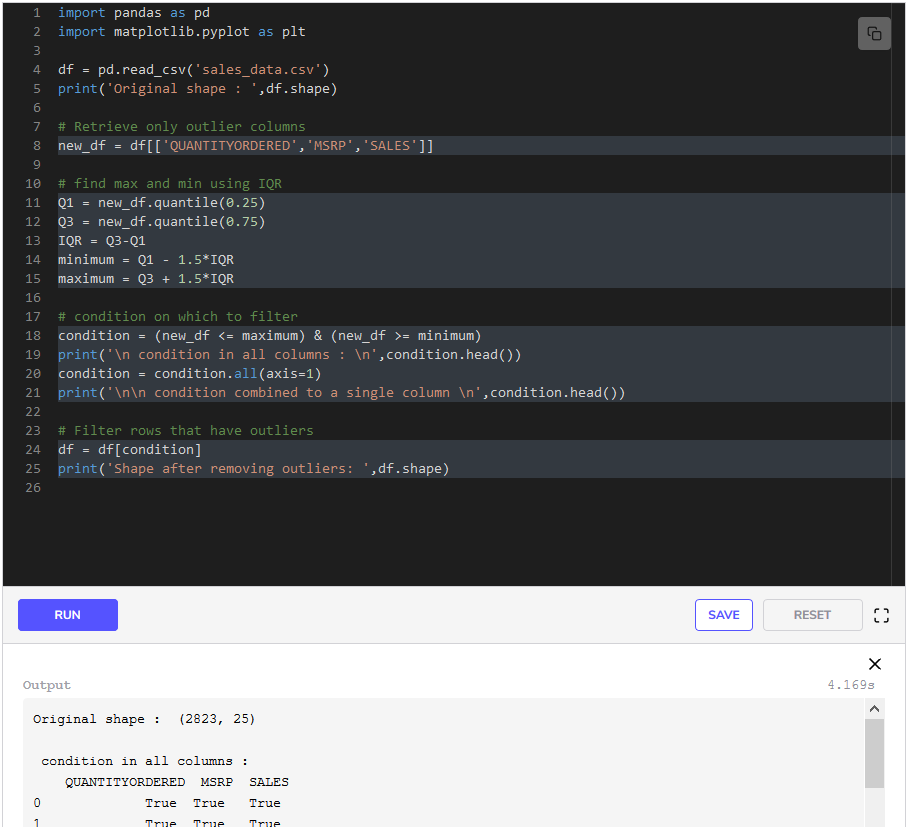
where Q1Q\_1Q​1​​ = First Quartile

Q3Q\_3Q​3​​ = Third Quartile

IQRIQRIQR = Inter Quartile Range = Q3Q\_3Q​3​​ - Q1Q\_1Q​1​​



We plot this in **line 7**. We give the size of the figure to the plot function as figsize = (10,10). This argument is optional but can be given if the plots are not rendering correctly on the screen. From the plot, we see that SALES and QUANTITYORDERED have quite a few outliers while MSRP has one outlier.

We can also use quartile ranges to filter for outliers. Let’s see an example of that below. We will be filtering rows that are outliers in all three variables. 

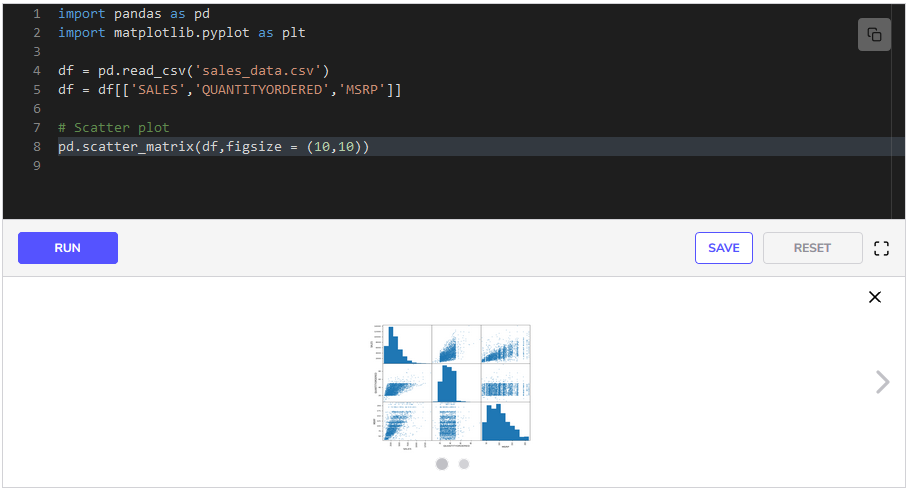
We make a new dataframe, new\_df, with just the three columns having outliers that we identified earlier in **line 8**. We give the list of columns that we need in the new dataframes inside the square brackets appended to df. We find the quantiles using the function quantile in **lines 11-12**. Then we find IQR and the maximum and minimum.

Our goal is to filter the dataframe df and remove those rows that have outlier values in all three columns. To do that, we can provide the dataframe df a list of booleans(True and False) as we learned in the previous chapter.

Therefore, we write our condition on **line 18**. It says that the values should be between the minimum and the maximum boundary values. **Line 18** gives us a dataframe that has True or False for every cell in new\_df based on whether or not it satisfies the condition. We can see this by the output of **line 19**. In the next line, we specify that the condition should be true for all three columns by using the all function with axis=1 argument. This gives us a list of True/False against each row. If a row has all three True values, then it gives a True value to that row. We can verify this by looking at the output of **line 21**. Then we simply filter the original dataframe df in **line 24** using the list we obtained above.

**Scatter plots**

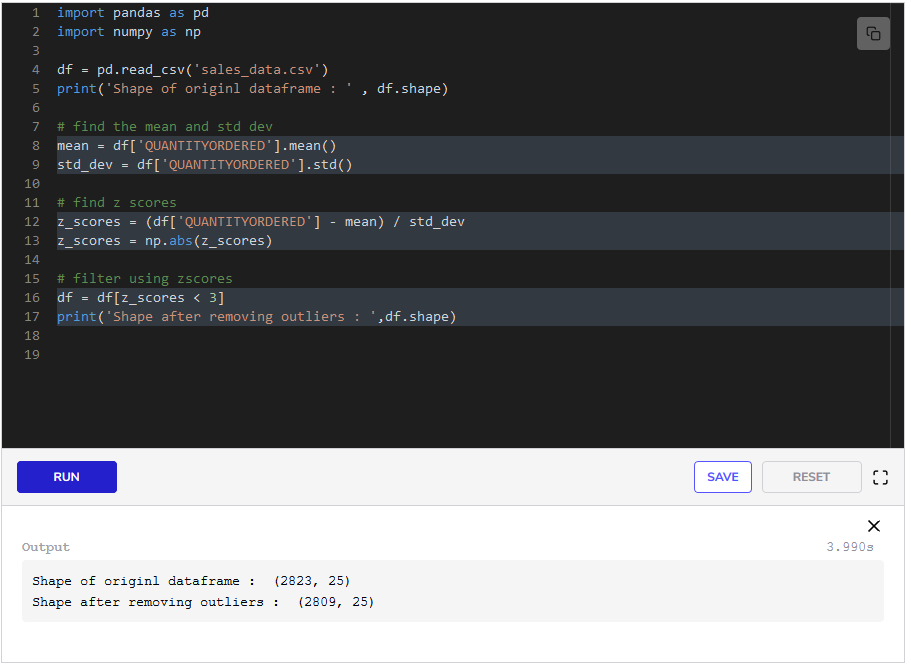
**Scatter plots** are a handy way of looking at point and collective outliers. In scatter plots, the values of two variables are plotted along two axes. Let’s see an example below.

We choose only three variables that will make sense for the scatter plot and draw scatter plots between all of them using the function pd.scatter\_matrix. This gives us a table of scatter plots. We can see a few outliers easily using these scatter plots.

**Z - Score**

**Z score** is the number of standard deviations an observation is above or below the mean of a variable. Z scores are used in statistics to study variance of data. We can use z-scores to filter outliers easily.

We find z-scores by subtracting the mean and dividing the standard deviation.

We removed outliers from the column QUANTITYORDERED. We first find mean and the standard deviation using the mean and std functions. Then we find z-scores in **line 12** by subtracting mean (calculated in **line 8**) and dividing by std\_dev (calculated in **line 9**). We take the absolute values of the z-scores using the function np.abs. We imported numpy on **line 2** to use this function. In the end, we filter using z\_scores. We set the condition that the value of z\_scores must be less than 333. We choose the score of 333 because, in normally distributed data, approximately 97% of the data lies inside 3 standard deviations. So, if a data value has a z-score greater than 3, it is an outlier.

This marks the end of this lesson. In the next lesson, you will be cleaning a dataset yourself as an exercise.

**Cleaning NYC Property Sales**

**1. Change values** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/myLj2qLZEO0#1-change-values)

In this task we had to change the values in the BOROUGH column according to the following rule:

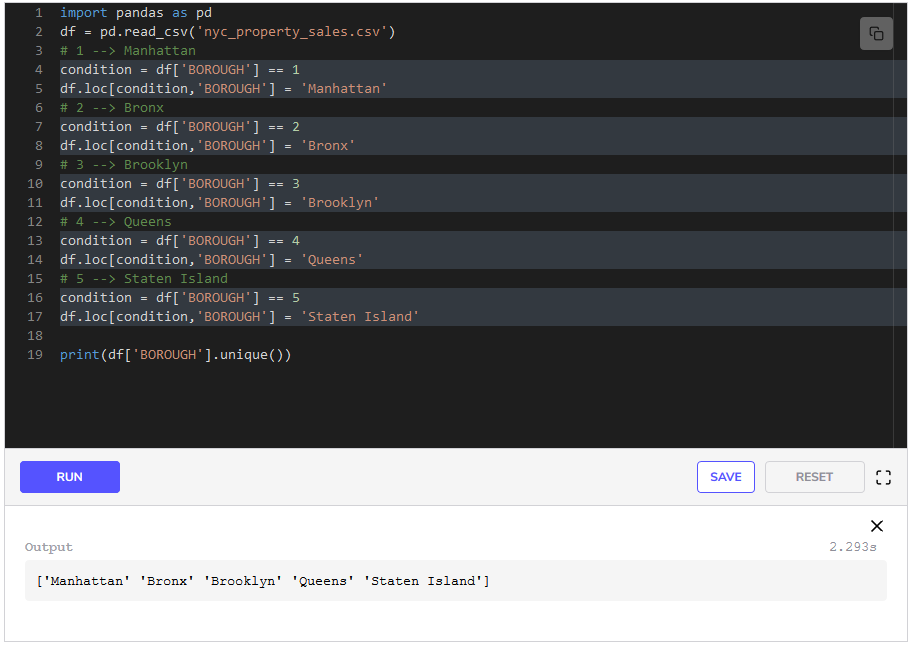
1 --> Manhattan

2 --> Bronx

3 --> Brooklyn

4 --> Queens

5 --> Staten Island

By looking at the problem statement, we can see that we need to write similar code for all 555 categories. We do each category one by one.

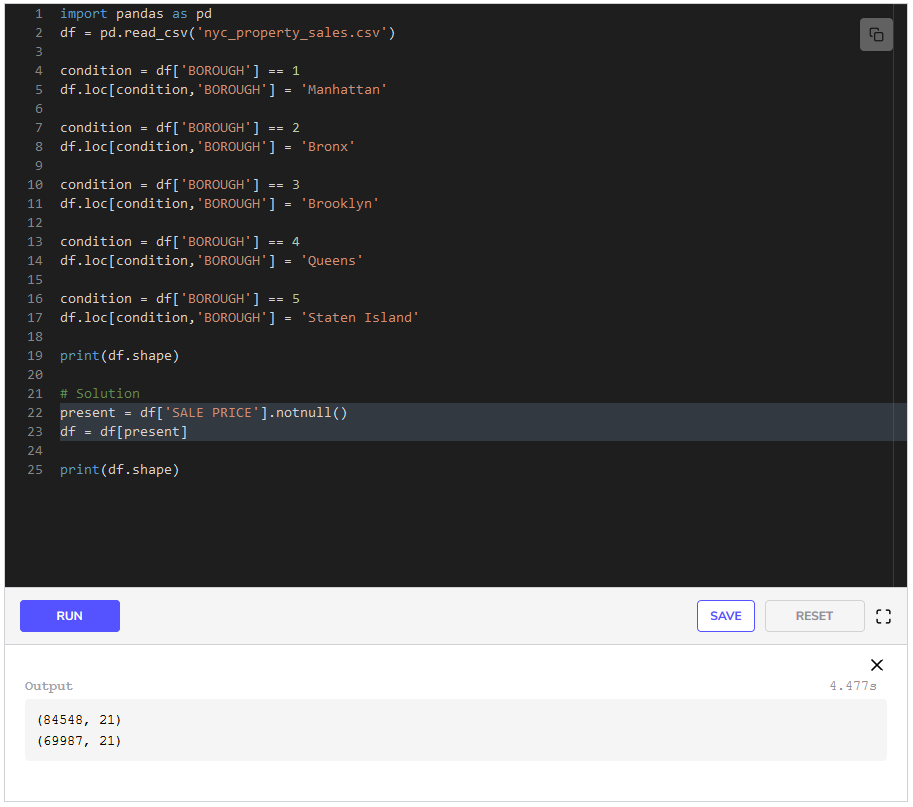
To change all instances of a specific value in a column, first, we need to find the rows where that value is present. To do this, we write our condition in **line 4**. df['BOROUGH'] == 1, gives us a list of True/False against each row. It is true for rows where the value of the BOROUGH column is 111. Now we need to go to all these places and change the value. We do that in **line 5** by using loc. We index the dataframe by the rows we had stored in condition and by the column BOROUGH.

df.loc[condition,'BOROUGH'] gives us all the cells where the value for BOROUGH is 111. So, we set these values to Manhattan. We follow the same steps for all other values.

In the end, we can verify our results by the output of **line 19**. It gives us the unique values that the column BOROUGH takes.

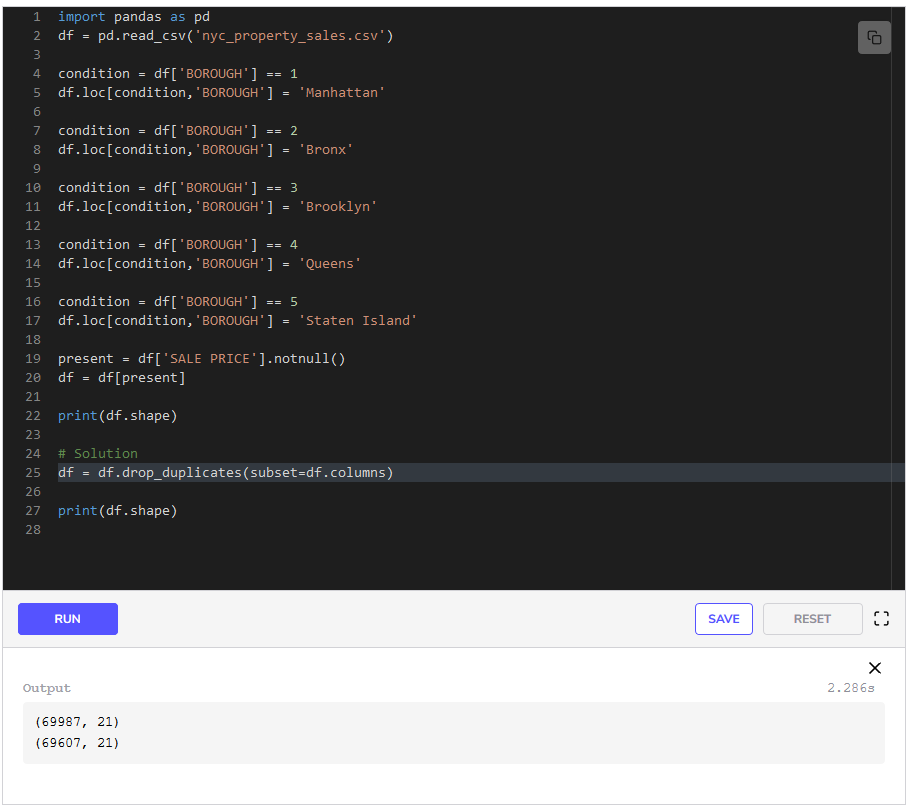
**2. Missing Values**

In this task, we had to remove rows that had missing values in SALE PRICE column.

This task is simple. To remove the rows containing missing values in the SALE PRICE, we just find the rows that do not contain missing values in SALE PRICE by using the notnull function in **line 22**. It gives us a list of True/False against each row. It is true for rows where there is no missing value. We store this list in present. In the next line, we use this list to filter the rows that have missing values. We verify the results by looking at the dimensions of the dataframe before (**line 19**) and after (**line 25**) filtering.

**3. Duplicate Values**

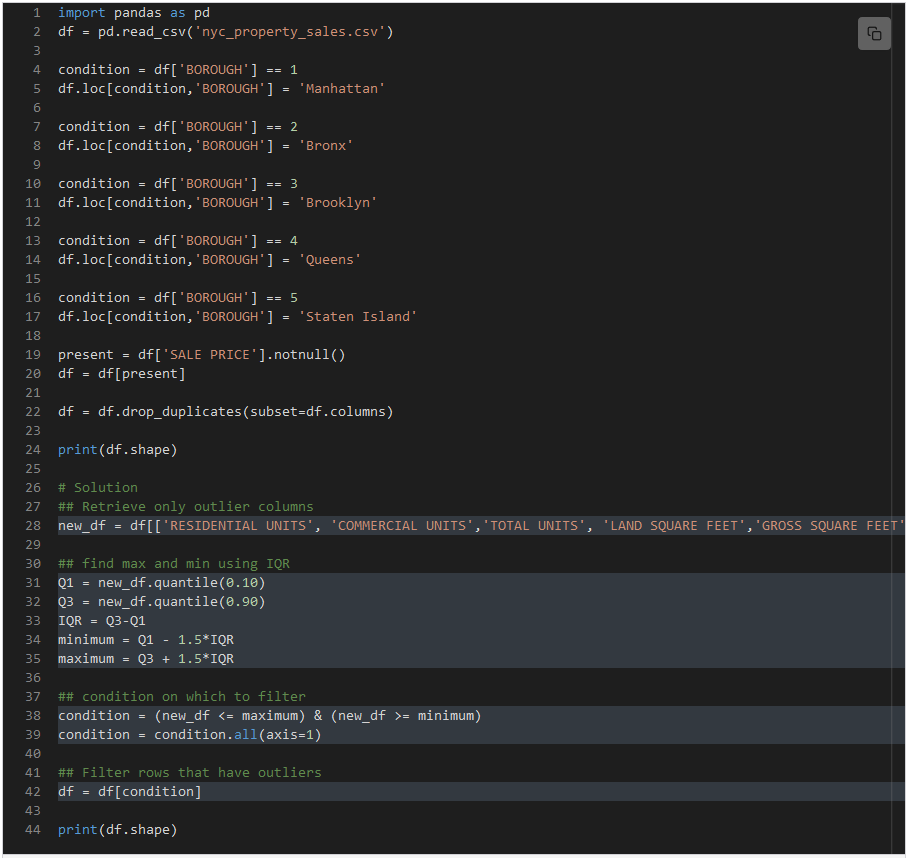
In this task, we had to remove duplicate rows.

Recall that we use the function drop\_duplicates to remove duplicates. We have to provide it a subset of columns for which the function checks if the values are duplicated in all these subset columns. If they are, it removes the duplicates. In our case, we used drop\_duplicates in **line 25**. We provide our whole list of columns, which we access by df.columns, to the function. We verify the results by looking at the dimensions of the dataframe before (**line 22**) and after (**line 27**) removing duplicate rows.

**4. Outliers**

In this task, we had to remove outliers using the Interquartile range, but there was a catch. The quantiles for q1 and q3 were 0.100.100.10 and 0.900.900.90. The columns in which we had to check for outliers were:

* RESIDENTIAL UNITS
* COMMERCIAL UNITS
* TOTAL UNITS
* LAND SQUARE FEET
* GROSS SQUARE FEET
* YEAR BUILT

In this task, we need to

* Find the minimum and maximum boundary values within which values are allowed for each column.
* Find rows that satisfy the boundary values condition for all columns.
* Filter for rows that are not outliers.

First, we create a new dataframe new\_df with only columns for which we have to check for outliers in **line 28**. Step 1 is done in **lines 31-35**. We use the quantile function to get quantiles and find maximum and minimum boundary values.

Step 2 starts on **line 38**. It gives us a dataframe that has True or False for every cell in new\_df based on if it satisfies the condition or not. In the **next line**, we specify that the condition should be true for all three columns by using the all function with axis=1 argument. This gives us a list of True/False against each row. If a row has all True values, then it gives a True value to that row. Here, we have a list of rows with which we can filter. We filter the original dataframe df on **line 42**. We verify the results by looking at the dimensions of the dataframe before (**line 24**) and after (**line 44**) removing duplicate rows.

Before cleaning this dataset, we had 845488454884548 rows in the data. The number reduced to 399023990239902 after cleaning. This just shows how much redundant data we had in the data set.

This was it from this chapter. In the next chapter, we will cover *Exploratory Data Analysis*.

**Introduction**

**What is Exploratory Data Analysis?**

**Exploratory Data Analysis (EDA)** is the process of analyzing datasets with the aim of understanding them more deeply. As the term “exploratory” suggests, during EDA the focus is to *explore* or *understand* the data better. We try to make sense of the data in the context of our problem.

During EDA, we can do any kind of analysis that gives us some insight into our data. Some of the common practices in EDA are:

* Looking at the data types of the variables
* Identifying the most important variables
* Looking at the distributions of the variables
* Summarizing the data
* Finding biases in the data
* Looking at the different trends in data
* Studying relationships among quantities
* Spotting anomalies in the data
* Visualizing the data

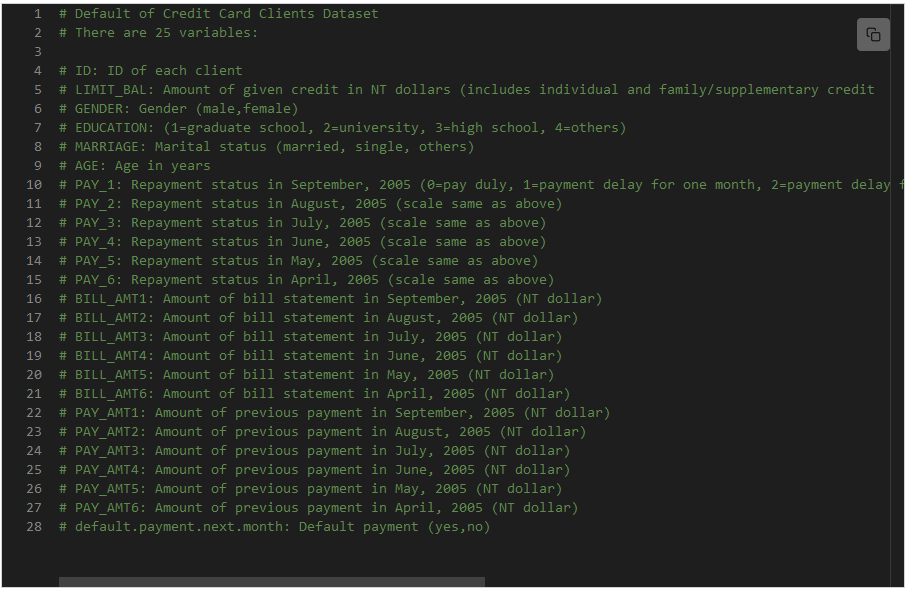


We will use some of the techniques we learned in chapter 2 to perform EDA and learn some new techniques as well. Python is well suited for this kind of work. So, in the next lesson, we will look at analyzing individual columns.

# Analyzing Individual Quantities

Analyzing individual variables is usually the way to start with EDA after figuring out data types. Summarizing a variable or looking at its distribution can be very helpful.

We will be using the [Default of Credit Card Clients Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). This dataset contains information on default payments, demographic factors, credit data, history of payment, and bill statements of credit card clients in Taiwan from April 2005 to September 2005. However, we will use the cleaned version of the dataset from the lesson [Inconsistent Data](https://www.educative.io/collection/page/10370001/4733468011397120/5322007673569280/). The details of individual columns are mentioned below.



Details of all columns in the dataset

## Summary stats

Summarizing a variable can give us useful information which can be used to draw conclusions or make decisions. Some common summarizing statistics are:

* mean
* median
* quartiles

We can use the describe function on our dataframe which summarizes individual columns for us, or we can select the columns that we want and use functions like mean, std, and max on them.

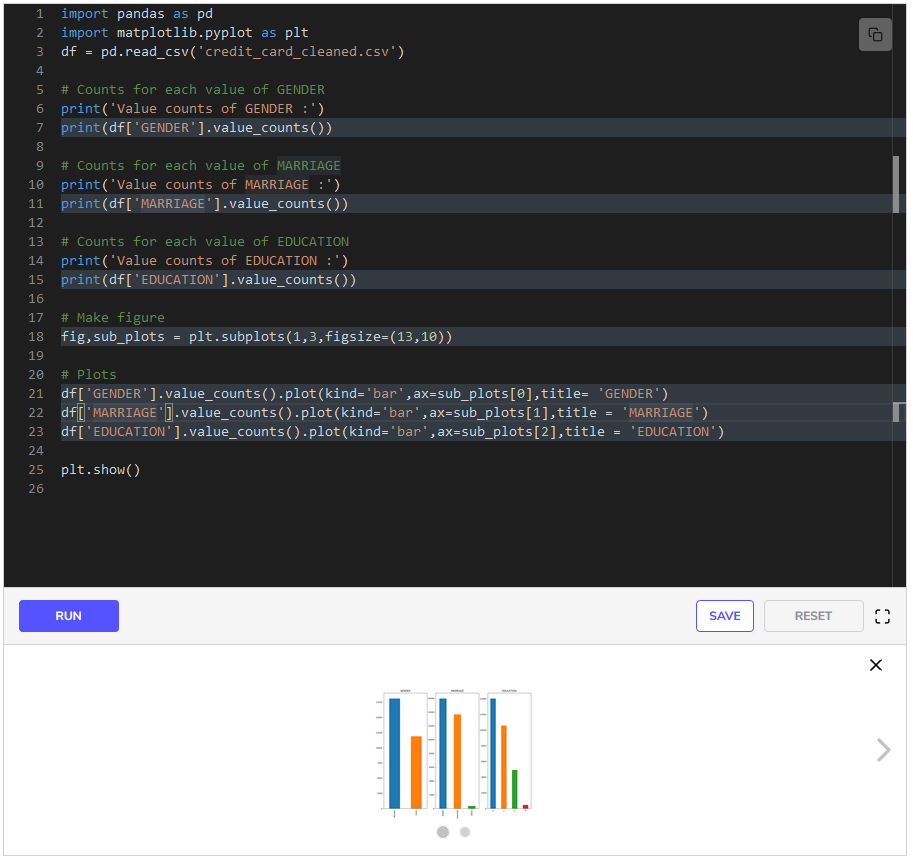
We have selected two variables and then called the function describe on them in **line 4**. The output of **line 4** gives us the count, mean, standard deviation, quartiles, minimum, and maximum.

By looking at the output, we find out that

* The average age is 353535 and the 75th75^{th}75​th​​ percentile of AGE is at 41 which means 757575 percent of the people in the dataset are below 41. This is a useful insight.
* The 75th75^{th}75​th​​ percentile of EDUCATION is 222 which implies that at least 757575 percent of the people in this dataset have been to university.

## Categorical variables

When we look at the dataset for exploration we see three important categorical variables in this dataset i.e. EDUCATION, GENDER, and MARRIAGE. We might be interested in how many males and females are in the dataset or what is the level of education of the majority. Let’s see how the data is distributed among these categorical variables.

We use the function value\_counts to retrieve the number of unique values for a variable on **lines 7,11, and 15** and then we print them. The function value\_counts can only be called on a series object, therefore, if we want to plot the value counts of all three variables separately on a single figure, we have to do some extra work.

We make the customized figure with subplots using plt.subplots on **line 18**. We give the layout of the grid to be 1,3 since we want three plots to be drawn side by side. We also provide the figure size as figsize. We store the figure as fig and the array of subplots as ax.

On **lines 21-23**, we plot the value counts by using the function plot. We specify the kind of the plot as bar. The plot function, when used with a series, can be provided a subplot as ax. Hence, we provide the subplots (sub\_plots) as ax.

sub\_plots[0] means we want this plot to be drawn as the first subplot in the figure, while sub\_plots[1] means we want the plot to be drawn at the second position, and so on. Moreover, we provide the titles of the plots as title to the plot function.

Scroll the graph output to view the text output. By looking at the outputs of **lines 7,11,15**, and the plots, we conclude that:

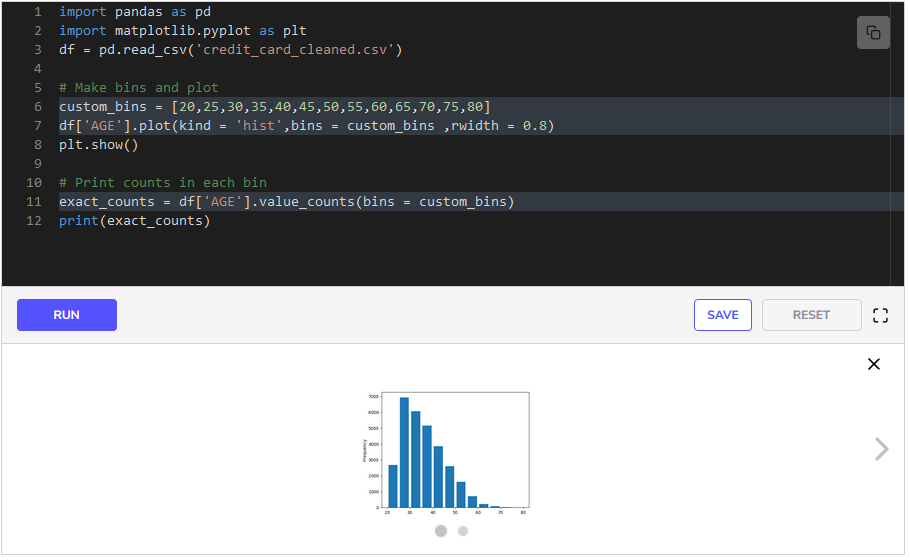
* There are more females in the dataset than males.
* We have an almost balanced set of married and unmarried people overall.
* The greatest number of people have gone to University and then Graduate School and then High School in the dataset.

## Distributions

Looking at the distributions of the variables can be very helpful and can give us key insights.

### AGE

We will be drawing a histogram of the ages of the people in the dataset as an example.

In **line 6**, we create a list that has values that will be used as customized bins for the histogram. In the next line, we select the AGE column and use the plot function with it. We specify the kind of plot as hist and give our custom bins to the function. Moreover, we specify the width between the bars as rwidth.

But what if we want the exact number of people in each bin? To do that, we use the value\_counts function and provide our custom\_bins to it in **line 11**. It gives us the exact number of people in each bin. So, we have some useful insights from this distribution:

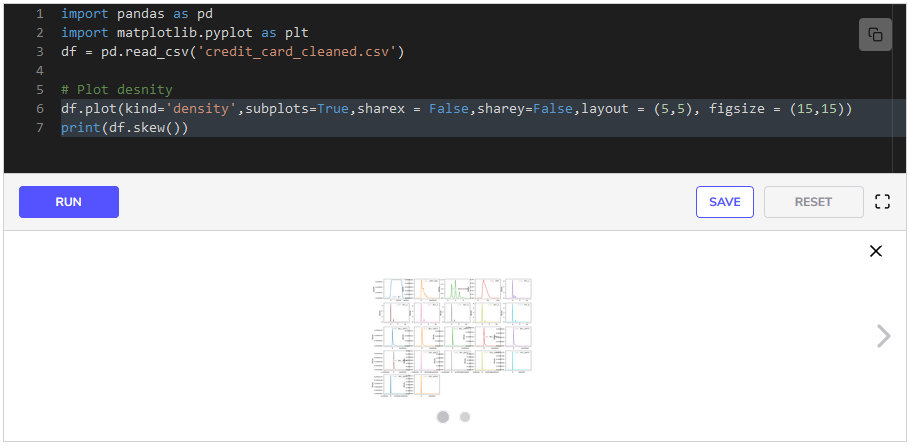
* The greatest number of people lie in the 25−3025-3025−30 age group.
* The majority of the people lie in the 20−4020-4020−40 age group.

## Skeweness

**Skewness** is the measure of the asymmetry of a distribution. In a normal distribution, the mean divides the density curve symmetrically into two equal parts at the median and the value of skewness is zero. When a distribution is asymmetrical, the tail of the distribution is skewed to one side either to the right or to the left.

When the value of the skewness is negative, the tail of the distribution is longer towards the left-hand side of the curve. This simply means there are more values towards the left side of the distribution.

When the value of the skewness is positive, the tail of the distribution is longer towards the right-hand side of the curve. This simply means there are more values towards the right side of the distribution.

We can check skewness both graphically and mathematically. We plot density plots of all the variables in **line 6**. We also use the skew function on the dataframe that gives us a measure of skewness for all variables.

From the output of **line 6**, we see that:

* Most variables have a positive skew.
* The payment variables (PAY\_AMT1,PAY\_AMT2…) are the most skewed variables. This can also be verified from the density plots.

## Quiz

You have been given a quiz on LIMIT\_BAL (the amount of credit that is given to a person) column below. You are also provided an empty code window. You have to answer the quiz questions by writing code and finding answers to the questions.

1

2

3

4

5

6





1

What is the mean of LIMIT\_BAL?

A)

176488

B)

160000

C)

167488

D)

170000

COMPLETED 0%

#### 1 of 4

This was how we can explore individual variables in the dataset. In the next lesson, we will see some techniques for exploring relationships between categorical variables.

# Exploring Categorical Quantities

This lesson will focus on how to explore relationships between different categorical variables in the dataset with examples.

We'll cover the following

* + [Grouping](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#grouping)
    - [GENDER](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#gender)
    - [EDUCATION](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#education)
    - [MARRIAGE with GENDER](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#marriage-with-gender)
  + [Quiz](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#quiz)

Exploratory Data Analysis is all about exploring relationships in the dataset that might be hidden or might not be easy to spot just by looking at the dataset. We will try to explore these kinds of relationships in the [Default of Credit Card Clients Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). We will use the cleaned version of the dataset from the lesson [Inconsistent Data](https://www.educative.io/collection/page/10370001/4733468011397120/5322007673569280). The details of individual columns are mentioned below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28





More specifically, we are interested in finding out how the variable default.payment.next.month is affected by other variables.

## Grouping [#](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#grouping)

As we saw in Chapter 3 of this course, grouping data can give us very useful insights. Let’s see how the categorical variables GENDER, EDUCATION, and MARRIAGE are related to default.payment.next.month.

### GENDER [#](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#gender)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16





We group the data by EDUCATION and default.payment.next.month on **line 6** and use the function size to retrieve the number of males and females. We then use the function unstack in the next line. The function unstack performs two steps here:

* It changes the table into a dataframe
* It names the columns no and yes, the two categories of the variable default.payment.next.month.

We can see the resultant dataframe in the output of **line 8**. We plot the dataframe grouped\_df in **line 11**. We see in the produced bar plot the number of males and females for each category of default.payment.next.month.

A natural question that arises after looking at the bar plot is that out of females and males, which category is more likely to default the next month since the number of females and males in our dataset is not equal? To answer this, we can calculate the probabilities of each gender defaulting the next month. We can calculate the probability of a male defaulting by dividing the number of males defaulting by the total number of males. We can do the same for females. Therefore, we divide the column yes with the sum of both yes and no. We save the probabilities in a new column in the dataframe and name the column prob\_default in **line 14**. From the output of **line 16**, we see that the:

* probability of a female defaulting is approximately 0.200.200.20
* probability of a male defaulting is approximately 0.240.240.24

This means that a male is more likely to default according to this dataset.

### EDUCATION [#](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#education)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16





We have written the same code that we did above except that we have replaced GENDER with EDUCATION in **line 6**. We get a bar plot in **line 11** in which we have counts of people in each category of education. The colors indicate whether or not they defaulted.

We calculate the probability of a person defaulting in each category of education by using the same formula that we used above for calculating the probabilities of males and females defaulting. We find out that the:

* probability of a postgraduate defaulting is approximately 0.190.190.19.
* probability of a university graduate defaulting is approximately 0.230.230.23.
* probability of a high school graduate defaulting is approximately 0.250.250.25.

This gives us a general trend in the data that as people get more educated they are less likely to default.

### MARRIAGE with GENDER [#](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#marriage-with-gender)

We have calculated above the probability for defaulting of males, females, married people, and singles according to our dataset. But we might want to go a level deeper and find how likely single males or single females are to default? So, let’s see how we can do that.

1

2

3

4

5

6

7

8

9

10

11

12

13





We have written the same code that we did above except that we have added three variables (MARRIAGE, GENDER and default.payment.next.month) by which we group by in **line 5**. We get a bar plot in **line 9** in which we have counts of people in each category of education. The colors indicate whether they defaulted or not.

We calculate the probability of a male and female defaulting in each category of marriage status by using the same formula that we used above for calculating the probabilities of males and females defaulting. We find out that

* A single female is the least likely to default with a probability of 0.190.190.19.
* A married male has a probability of almost 0.260.260.26 to default.

Similarly, we can make other combinations using categorical variables and draw plots and calculate probabilities to find out general trends or patterns in the dataset.

## Quiz [#](https://www.educative.io/courses/data-science-for-non-programmers/xlJ2QkRQmgE#quiz)

You have been given a quiz on the MARRIAGE column below. You are also provided an empty code window. You have to answer the quiz questions by writing code and finding answers to the questions.

1

2

3

4

5





1

How many married persons have defaulted in our dataset?

A)

10455

B)

5209

C)

3206

D)

3342

COMPLETED 0%

#### 1 of 4

In the next lesson, we will learn how to explore relationships between numerical quantities.

# Exploring Numerical Quantities

This lesson will focus on exploring the numerical quantities and finding out general trends from these quantities.

We'll cover the following

* + [Scatter plots](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#scatter-plots)
  + [Binning numerical data](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#binning-numerical-data)
    - [AGE](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#age)
  + [Quiz](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#quiz)

A very important part of exploratory data analysis is finding out general trends or patterns in the data. We can find out different relationships between two quantities that can be very helpful in making decisions at the end. We will use the cleaned version of the dataset from the lesson [Inconsistent Data](https://www.educative.io/collection/page/10370001/4733468011397120/5322007673569280). The details of individual columns are mentioned below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28





## Scatter plots [#](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#scatter-plots)

Scatter Plots are a very useful way of visualizing the inverse and direct relationships between two variables. In a **direct** relationship between two quantities, an increase/decrease in one quantity leads to a corresponding increase/decrease in the other quantity, whereas in an **inverse** relationship, an increase/decrease in one quantity leads to a corresponding increase/decrease in the other quantity.

However, in real data, we do not observe strict direct or inverse relationships, rather we observe relationships or patterns that look like direct or linear relationships because there are many external factors that affect a quantity.

1

2

3

4

5

6

7

8

9

10

11

12





We are interested in finding out the spending habits of people from month to month, whether they generally spend the same every month, or if there are some months in which they spend extra, and how the amount of credit given (LIMIT\_BAL) varies with the bills.

We write the billed amount columns (BILL\_AMT1,BILL\_AMT2,…) and LIMIT\_BAL in **line 6** in a list, and filter using this list in **line 7**. Then we use the pandas function scatter\_matrix in **line 8** which draws the scatter plots between all the variables in the dataframe.

Keep in mind that scatter\_matrix is not a function that is called on a dataframe. Rather it is provided a dataframe as an argument.

By looking at the last row of the scatter matrix, we can see that there is a kind of linear relationship between LIMIT\_BAL and all other bill amount variables. As we increase the bill amounts, the credit given is increased. This means that the bank gives more credit to people who spend more usually. But there a few exceptions that can be seen from the plots. There are a few people who spend very little yet are given high credits.

Another observation that we can make from the scatter matrix is that as we increase the amount of bills in a month, we are likely to see an increase in the amount of bills the next month. For instance, look at the plot where BILL\_AMT6 (The amount of Bills in April 2005) is at the x-axis and BILL\_AMT5(The amount of Bills in May 2005) is at the y-axis. We see a pattern similar to a direct relationship. We can see the same pattern between BILL\_AMT5 and BILL\_AMT4, BILL\_AMT4 and BILL\_AMT3 and so on. This tells us that people usually spend similar amounts of money in these months and if they spend a certain amount in one month, they are expected to spend similar amounts in the next month.

## Binning numerical data [#](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#binning-numerical-data)

We can divide our numerical data in bins and see how many people in each bin default.

### AGE [#](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#age)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25





We divide the ages into bins. We use the pandas function cut and give it the column that we want to divide as the first argument, and give the bins we made as bins = custom\_bins in **line 7**. This gives us a series in which we have the age bin against every row. We then add this to our dataframe as a new column (AGE\_BIN) in **line 9**. Now we have both AGE and AGE\_BIN for every row in the dataset.

After this, we group the data by AGE\_BIN and default.payment.next.month, and call size() on the groups to obtain the number of default and non-defaults in each age bin in **line 12**. After this, we use the function unstack to change the groups into a dataframe and name the columns as yes and no.

Then, we calculate the probability of defaulting for each age bin using the simple formula we used in the last lesson in **line 18** and save these as a new column in the dataset named prob\_default. We plot the probabilities of each age group defaulting in **line 23**.

From the plot of the probabilities for defaulting in each age group, it is visible that very young people and very old people are more likely to default.

## Quiz [#](https://www.educative.io/courses/data-science-for-non-programmers/BnQoWZXWmQo#quiz)

You have a quiz below. You are also provided an empty code window. You have to answer the quiz questions by writing code and finding answers to the questions.

1

2

3

4

5





1

How many people lie in the interval (0, 100000] of LIMIT\_BAL who have defaulted?

A)

3684

B)

8817

C)

3454

COMPLETED 0%

#### 1 of 3

These were some techniques to explore numerical data. There is another mathematical way of exploring the relationships between quantities known as correlation which we will study in the next lesson.

**Correlation and Heatmaps**

This lesson will introduce how to calculate and visualize correlations between quantities in python.

We'll cover the following

* + [Correlation](https://www.educative.io/courses/data-science-for-non-programmers/NExx5MrXmxN#correlation)
  + [Heatmap](https://www.educative.io/courses/data-science-for-non-programmers/NExx5MrXmxN#heatmap)

**Correlation** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/NExx5MrXmxN#correlation)

**Correlation** is a mathematical technique that shows how strongly two variables are linked. It quantifies the strength of the relationship. For instance, we know that the weight and height of a person are correlated. Taller people tend to have more weight. Hence, we say that height and weight are correlated.

Correlation is measured in terms of a number called **correlation coefficient**, which ranges from −1-1−1 to 111. The value of 111 or −1-1−1 denotes complete correlation, while 000 indicates that no correlation is present between the two variables. Negative values mean there is an inverse relationship between the two variables, while a positive value denotes a direct relationship.

Pandas has the function corr that can be called on a dataframe. Let’s see an example of this on our [Default of Credit Card Clients Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset).

1

2

3

4

5

6





We just use the function corr with the dataframe df in **line 5**, which gives us a table with correlation values for each pair of variables. This table is known as the **correlation matrix**. We print the correlations in the next line.

Looking at the correlation matrix, it can be very hard to study these values in this printed table. Therefore, we use *heatmaps* to visualize the correlations.

**Heatmap** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/NExx5MrXmxN#heatmap)

A **heatmap** is a graphical representation of data where individual values are represented as colors. The intensity of the colors indicates the values.

Seaborn is a Python module that is used for plotting. It has the function heatmap that we will be using to plot the heatmap. We will be concerned with the following arguments of the function:

* data: The data or matrix from which to plot the heatmap.
* annot - **optional**: Whether to write the values on each cell of the heatmap. Expects True/False. False by default.
* vmin - **optional**: The minimum on the color bar.
* vmax - **optional**: The maximum on the color bar.
* cmap - **optional**: The color map to use.

1

2

3

4

5

6

7

8

9

10

11





We import the seaborn library in **line 3** as sns. Then we create the correlation matrix in **line 7** using the function corr. The magic happens in **line 10** where we use the heatmap function. We give our correlation matrix corr as data. Then we set the color bar scales to −1-1−1 and 111 since we know that correlation coefficients range from −1-1−1 to 1.1.1. We set cmap to coolwarm as this color map makes it easy to study heatmaps.

From the plot, we can clearly infer that the Bill amount variables (BILL\_AMT1, BILL\_AMT2,…) are highly correlated with each other as we suspected from the scatter plots in the last lesson. We can say the same about the payment delay variables (PAY1, PAY2,…) as well.

We also see some positive correlation between LIMIT\_BAL and Bill amount variables (BILL\_AMT1, BILL\_AMT2,…) which implies that people who were given more credit (higher values of LIMIT\_BAL) tend to have larger bills.

Interestingly, there is a slight negative correlation between LIMIT\_BAL and payment delay variables (PAY1, PAY2,…). This implies that people who are given more credit tend to have fewer payment delays. Maybe because they earn more they are given higher credit in the first place.

This is how correlation and heatmaps help us to make sense of the data. In the next lesson, we will explore another dataset as an example and see how different techniques are used on different datasets.

**Exercise: Exploring E-Commerce**

This lesson tests the learners on EDA on an e-commerce dataset.

We'll cover the following

* + [E-Commerce data](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#e-commerce-data)
  + [1. Top 555 customers with the highest number of orders](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#1-top-5-customers-with-the-highest-number-of-orders)
    - [Input](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input)
    - [Output](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output)
  + [2. Top 555 customers with most amount of money spent](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#2-top-5-customers-with-most-amount-of-money-spent)
    - [Input](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-2)
    - [Output](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-2)
  + [3. Top 555 countries with the highest number of orders](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#3-top-5-countries-with-the-highest-number-of-orders)
    - [Input](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-3)
    - [Output](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-3)
  + [4. Number of orders for every month in 2011.](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#4-number-of-orders-for-every-month-in-2011)
    - [Input](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-4)
    - [Output](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-4)
  + [5. Top 10 most ordered products](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#5-top-10-most-ordered-products)
    - [Input](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-5)
    - [Output](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-5)
    - [Some useful tips](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#some-useful-tips)

**E-Commerce data** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#e-commerce-data)

In this lesson, you are going to be tested on exploring E-commerce data. The [dataset](https://archive.ics.uci.edu/ml/datasets/online+retail) was made available on the UCI Machine Learning Repository. This is a transnational data set that contains all the transactions occurring between 01/12/2010 and 09/12/2011 for a UK-based and registered non-store online retail. We will be using a **sample** of it.

In the below exercise, you will be writing functions for every task. The functions will receive a dataframe df as an input argument. Your task will be to perform the required operations to answer a question and return the answer to that.

1

2

3

4

5

6

7

8

9

10

11

12

13





**1. Top 555 customers with the highest number of orders** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#1-top-5-customers-with-the-highest-number-of-orders)

Find the CustomerID of the top 555 customers with the highest number of orders.

**Input** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input)

A dataframe

**Output** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output)

The number of orders against CustomerID of top 555 customers.

**2. Top 555 customers with most amount of money spent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#2-top-5-customers-with-most-amount-of-money-spent)

Find the CustomerID of the top 555 customers with the most amount of money spent.

**Input** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-2)

A dataframe

**Output** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-2)

Amount of Money Spent against CustomerID of top 555 customers.

**3. Top 555 countries with the highest number of orders** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#3-top-5-countries-with-the-highest-number-of-orders)

Find the top 555 Countries from where most orders come from.

**Input** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-3)

A dataframe

**Output** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-3)

Number of Orders against the names of the top 555 countries.

**4. Number of orders for every month in 2011.** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#4-number-of-orders-for-every-month-in-2011)

Find the number of orders for every month in the year 2011.

**Input** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-4)

A dataframe

**Output** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-4)

Numbers of orders against each month.

**5. Top 10 most ordered products** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#5-top-10-most-ordered-products)

Find the top 10 most ordered products.

**Input** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#input-5)

A dataframe

**Output** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#output-5)

Numbers of orders against each product.

**Some useful tips** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nX1pwyQWDB#some-useful-tips)

* You can use size function after groupby to retrieve the number of times a category appeared.
* To sort a series, sort\_values(ascending=True) can be used. For descending order, give ascending=False.
* To sort a dataframe, sort\_values(by=column\_name, ascending=True) can be used. Replace column\_name with the name of the column by which you want the rows to be sorted. For descending order, give ascending=False.
* You can use df.iloc[:5] to retrieve the first 555 rows from the dataframe or series.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20





**Solution Review: Exploring E-Commerce**

This lesson provides solutions to the exercise on exploring E-Commerce Dataset in previous lesson.

We'll cover the following

* + [1. Top 555 customers with the highest number of orders](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#1-top-5-customers-with-the-highest-number-of-orders)
  + [2. Top 555 customers with the most amount of money spent](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#2-top-5-customers-with-the-most-amount-of-money-spent)
  + [3. Top 555 countries with the highest number of orders](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#3-top-5-countries-with-the-highest-number-of-orders)
  + [4. Number of orders for every month in 2011](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#4-number-of-orders-for-every-month-in-2011)
  + [5. Top 10 most ordered products](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#5-top-10-most-ordered-products)

**1. Top 555 customers with the highest number of orders** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#1-top-5-customers-with-the-highest-number-of-orders)

1

2

3

4

5

6

7

8

9





We do this task in three steps:

* First, we group our data with CustomerID and call size to retrieve the number of times each CustomerID appeared in the data in **line 5**.
* Second, we sort the values in descending order using sort\_values in **line 6**.
* In the end, we just take the top 5 customers since they are already sorted.

**2. Top 555 customers with the most amount of money spent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#2-top-5-customers-with-the-most-amount-of-money-spent)

1

2

3

4

5

6

7

8

9

10





We do this task in four steps:

* First, we group our data with CustomerID and call sum since we want the AmountSpent of all orders added up for each customer in **line 5**.
* Then we select the AmountSpent column in **line 6**.
* Then we sort the values in descending order using sort\_values in **line 7**.
* In the end, we just take the top 5 customers since they are already sorted.

**3. Top 555 countries with the highest number of orders** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#3-top-5-countries-with-the-highest-number-of-orders)

1

2

3

4

5

6

7

8

9





We do this task in three steps:

* First, we group our data with Country and call size to retrieve the number of times each Country appeared in the data in **line 5**.
* Second, we sort the values in descending order using sort\_values in **line 6**.
* In the end, we just take the top 5 countries since these are already sorted.

**4. Number of orders for every month in 2011** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#4-number-of-orders-for-every-month-in-2011)

1

2

3

4

5

6

7

8

9





We do this task in two steps:

* First, we have to filter the data to keep entries for only 2011. For this, we specify our condition in **line 5**, then we filter using it in the next line.
* Second, we group our data by PurchaseMonth and call size to retrieve the number of times each month appeared in the data in **line 7**.

**5. Top 10 most ordered products** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B89RZmBOk0N#5-top-10-most-ordered-products)

1

2

3

4

5

6

7

8

9

10





We do this task in four steps:

* First, we group our data with Description and call sum since we want the Quantity of all orders added up for each product in **line 5**.
* Then we select the Quantity column in **line 6**.
* Then we sort the values in descending order using sort\_values in **line 7**.
* In the end, we just take the top 10 products since they are already sorted.

In the next lesson, we will look at how we can perform RFM analysis in Python.

**Business Example: RFM Analysis in Python**

This lesson will focus on how to do RFM analysis in Python with Pandas as an example of the usability of pandas.

We'll cover the following

* + [RFM Analysis](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#rfm-analysis)
    - [Recency](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#recency)
    - [Frequency](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#frequency)
    - [Monetary](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#monetary)
    - [Putting it all together](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#putting-it-all-together)
    - [Using quartiles to form classes](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#using-quartiles-to-form-classes)
    - [Combining classes into a number](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#combining-classes-into-a-number)
    - [Sort using rfm values](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#sort-using-rfm-values)
  + [Benefits of RFM analysis](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#benefits-of-rfm-analysis)

In this chapter, we have seen how we can extract useful information from raw data very easily using *pandas* in Python. But we have only scratched the surface. A lot more can be done to obtain useful insights from the data. Professionals can use their domain expertise to perform different kinds of analysis on the data. In this lesson, we will explore a dataset from the perspective of a business or a marketing professional. We will be doing an *RFM analysis* of the [Sample Sales Data](https://www.kaggle.com/kyanyoga/sample-sales-data) that we have been using.

**RFM Analysis** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#rfm-analysis)

**RFM** (**R**ecency, **F**requency, **M**onetary) **analysis** is a marketing technique used to determine quantitatively which customers are the best ones by examining how recently a customer has purchased (recency), how often they purchase (frequency), and how much the customer spends (monetary). Using RFM analysis, customers are assigned a ranking number of 1,2,3,41,2,3,41,2,3,4 (with 444 being highest) for each RFM parameter. The three scores together are referred to as an RFM “cell”. The data is sorted to determine which customers were the **best customers** in the past, with a cell ranking of 444444444 being ideal.

So, let’s start coding.

1

2

3

4

5

6

7

8

9





We read the data in **line 4**. Since we are doing RFM analysis, we will only need four columns. CUSTOMERNAME to group customers, ORDERDATE to calculate recency, ORDERNUMBER to calculate frequency, and SALES to calculate monetary. Therefore, we write these column names in a list in **line 7** and filter the dataset in **line 8**.

**Recency** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#recency)

**Recency** is a measure of how recently a customer has purchased a product. We find recency for every customer by finding out the date of their last order. Then we subtract this date from the date of the most recent order in the dataset to find the number of days between these two dates. This difference gives us the recency values. For a recent customer, this value will be smaller. For a customer who has not purchased a product for a long time, this value will be higher.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16





First of all, we change the data type of ORDERDATE to the datetime datatype of pandas in **line 2** so that we can easily compare dates. Then we find the date of the last order of every customer in **line 5**. We group the data by CUSTOMERNAME. Then we only retrieve the ORDERDATE column from the grouping. Then we use the max function which gives us the latest ORDERDATE for every customer. To find the most recent ORDERDATE, we get the max of the ORDERDATE column in **line 8**. We name it most\_recent.

Our next step is to subtract the last ORDERDATE of every customer from the most recent date. Therefore, we define a function subtract\_date in **line 11** that takes a date dt, find the difference in days between the most\_recent and dt in **line 12** and returns the difference in **line 13**.

Afterward, we subtract recent\_dates from most\_recent in **line 16** by using the function that we defined. We use apply to apply subtract\_date to recent\_dates. The function subtract\_date is called for every value in recent\_dates. Each value is passed as an argument to the function one by one. The results are stored in recency in **line 16**.

**Frequency** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#frequency)

**Frequency** is a measure of how often a customer purchases a product. We will calculate the total number of times a customer has made an order.

Note that in the dataset, a single order denoted by an ORDERNUMBER can have orders on more than one product. Therefore, a single value of ORDERNUMBER can appear more than once for the same customer if that order had different kinds of products purchased. We want to calculate the number of times they have placed an order. So, if a single order has 333 different products, we will count it as a single order.

1

2

3

4





We calculate frequency in 2 steps:

1. We group by using CUSTOMERNAME and ORDERNUMBER and then use size as aggregation function in **line 2**. This gives us the number of times each ORDERNUMBER appeared for every customer. If we print head of frequency after **line 2**, we will get the following output:
2. In the second step, we group the above table by CUSTOMERNAME and use size as the aggregation function in **line 4**. This gives us the number of times each customer appears in the above table, which is the number of times a customer has placed an order. If we print the head of frequency after **line 4**, we will get the following output:

The values in this table will form our frequency values.

**Monetary** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#monetary)

It is a measure of how much the customer has spent. It is the sum of all the sales made to the customer. It can be calculated by grouping the data by CUSTOMERNAME and summing the SALES.

1





We use groupby to group by CUSTOMERNAME, then we select the SALES column, and use sum as the aggregation function. This gives us the sum of all sales made to the customer.

**Putting it all together** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#putting-it-all-together)

Now we will combine all three recency, frequency, and monetary into a single table.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





**Using quartiles to form classes** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#using-quartiles-to-form-classes)

Now that we have raw values for recency, frequency, and monetary, we can convert these values into classes. We will convert each value into a class based on which quartile it falls into.

First, we will obtain quartile values using the quantile function. Then we will assign the class values according to the quantile values. For instance, if the quartiles of recency are (80,125,230)(80,125,230)(80,125,230), then a customer with a recency value of 707070 can be given a class value of 4 while a value of 120120120 can be assigned a class value of 3 and so on. Let’s implement this below.

1





This line will give us the quantiles of all three columns in a dataframe that we call quantile\_df. The quantile\_df will look like:

We can use this dataframe to assign classes to our variables. To accomplish that we define a function quantile\_classes that we will apply to all three columns.

Recall that when we apply functions using apply, the function is called on every value of the column one by one. The applied functions expect only one argument, which is the value in the column. But the function we define below expects three arguments.

* The first argument, x, will be the value taken from the column.
* The second argument, quantile\_values, will be the quantile values we calculated above.
* The third argument, attribute, will be the name of the column that we are applying the function on.

So, let’s look at quantile\_classes below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23





The function checks in **line 3** if attribute is recency or not. If it is recency then the execution jumps to **line 4**. The function checks in what quantile the value x falls. It can access the quantile values using the loc keyword. loc can be provided the row name and the column name to access the quantile value. For the column name, it uses attribute since it contains the column name on which the function was applied.

If the value is less than the 25th percentile, it is assigned the class value 4 in **line 5**. If it is between the 25th and 50th percentile, then it is assigned the class value 3 in **line 7**. If it falls between the 50th and 75th percentile, then it is assigned the class value 2 in **line 9**, and if it falls above the 75th percentile then it is assigned the class value 1 in **line 11**.

Now if the attribute is not recency, then the class value assigned to x for being less than the 25th percentile is 1 in **line 15**. In the same way, we assign class values 2, 3, and 4 in **lines 17,19, and 21** respectively.

This change in the order of assigning class values was made because low recency values mark the more recent customers and we want to assign them higher class values. But for frequency and monetary, higher values indicate top customers.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





We write our function in **lines 39-59**. Then we calculate the quantiles in **line 62**. To convert recency into classes, we use the apply function on the recency column in rfm\_table in **line 65**. The apply function is first given the name of the function. Then it is given the extra arguments that the quantile\_classes expects in parenthesis as args. For recency, the arguments are quantile\_df and recency. The output is stored as a new column named r\_class.

We do the same for frequency and monetary in **lines 66 and 67**, respectively. By looking at the output of **line 68**, we can see the classes assigned to each customer.

**Combining classes into a number** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#combining-classes-into-a-number)

Now that we have assigned separate classes to all customers for recency, frequency, and monetary, we can combine these three separate classes into a single class. For instance, if a customer has the values 2, 3, 4 for recency, frequency, and monetary, respectively, the combined value will be 234.

Since the columns r\_class, f\_class, and m\_class are of the string data type, we can join the values by using the + operator. For strings, + joins them together to form a single string. For instance,

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





In **line 70**, we add a new column rfm to rfm\_table that will store the combined classes. We combine the three columns using the + operator. We can see the output of **line 71**, to verify.

**Sort using rfm values** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#sort-using-rfm-values)

To find the top customers we can sort the dataframe using rfm values. But there is an issue with sorting. The rfm column is of string type and we can only sort numerical values. Therefore, we need to convert them to integer values. To do this, we can use the function to\_numeric available in pandas.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





We use pd.to\_numeric in **line 73**, to convert the rfm values into numerical values. Then we sort the dataframe using the function sort\_values in **line 74**. We tell the function to sort the dataframe using the column rfm by giving it as the argument by. We give ascending=False to sort in descending order. We can see the top 20 customers from the output of **line 75**.

**Benefits of RFM analysis** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPlwrq9R9KD#benefits-of-rfm-analysis)

Now that we have arranged our customers according to the rfm values, we can answer a lot of questions such as:

* Who are the top 10 customers?
* Which customers are close to churning?
* Who are the most loyal customers?
* Who are the customers that buy the most?
* Which group of customers do we need to retain?
* On which customers do we need to focus more?

All of these and many more questions can be answered by exploring the dataset using the recency, frequency, and monetary values we calculated. We can filter the dataset using different conditions to answer these questions.

This was an example of using Python for performing a business technique easily. People from other domains can apply their expertise to perform any kind of task on the data to gain more insights.

This marks the end of this chapter. In this chapter, we looked at how to find certain trends and patterns in the data. In the next chapter, we will look at how we can verify our findings.

**The Basics of Statistical Inference**

This lesson will introduce statistical inference and point estimates, along with the central limit theorem.

We'll cover the following

* + [Inferential statistics](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#inferential-statistics)
  + [Point estimates](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#point-estimates)
  + [Sampling distributions](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#sampling-distributions)
    - [The central limit theorem](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#the-central-limit-theorem)

In the last chapter, we learned how to explore data using different graphs and extract information from the data. We discovered relationships between different variables in our datasets, but how do we decide whether these relationships are real or just by coincidence? How do we decide whether we can make a generalization about the dataset or not? This is where *inferential statistics* comes in.

**Inferential statistics** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#inferential-statistics)

Before we see what inferential statistics is all about, we need to understand the concept of a *population* and a *sample*.

A **population** includes all the elements from a set of data that we are studying. For example, if we are interested in finding out how grades of students in a school are affected by alcohol consumption, all the students in that school form a population. A measurable characteristic of the population, such as a mean or standard deviation, is known as a **parameter**.

A **sample** is a subset of the population. In our school example, if we randomly choose a single grade and analyze students in that grade, then those chosen students form a sample. A measurable characteristic of a sample is known as a **statistic**.

**Inferential statistics** is the branch of statistics that helps us make inferences about the general *population* from a *sample*. It aims at gaining insights about the population from which the data was collected, based on the collected samples. There are two main areas of inferential statistics:

* **Estimation**: This involves taking a statistic of a sample and using that to say something about a population parameter.
* **Hypothesis testing**: This involves answering some research questions about a population using a sample of that population.

**Point estimates** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#point-estimates)

**Point estimates** are estimates of population parameters based on sample data. Let’s say we want to measure the average salary of all data scientists in the US, but taking the data of all the data scientists will be cumbersome. Therefore, what we do is take the salary data of 100 random data scientists and take the average of this sample. The average of this sample is known as **sample mean**.

The sample mean is usually not the same as the population mean. The difference could be due to randomness in choosing a sample or biased sampling. Let’s investigate this further using our [Default of Credit Card Clients Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). We will be investigating the AGE column of the dataset.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16





We are considering the entire dataset as our population in this example since we are concerned with only credit card customers. In **line 5**, we calculate the population mean using the function mean, and store it in pop\_mean. Then in **line 8**, we sample from the AGE column. We use the function sample and give it n=200 as the number of samples that we want. We also give it a random\_state. We can give any number as random\_state. This is just to ensure that every time we run the code with the same random\_state, we get the same random sample. In the next line, we calculate the mean of the sample by using the mean function. In **line 11**, we simply calculate the difference between the two means by subtraction. Then we print the three quantities in **lines 13-15**.

We see that the difference is approximately only 0.590.590.59 years. This means that we get a fairly accurate estimate of a population parameter.

**Sampling distributions** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#sampling-distributions)

You might have heard or read somewhere about the normal distribution. In a normal distribution, the data is symmetrically distributed and is gathered in a few standard deviations around the mean. However, real-world data that we collect is not normally distributed. Therefore, when we draw a sample from that data, the sample tends to follow the distribution of the population.

Let’s see if this is true using the above sample that we took from the AGE column. We will be drawing two density plots, one of the population, and one of the sample, on the same figure to compare them.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19





Since we want to draw two density plots on the same figure, we initialize our subplots in **line 5** by using plt.subplots. We specify that we want the layout to be 2,1 so that there is one column and two rows. We retrieve the subplots axes in subplots. In **line 8**, we plot the density plot of AGE. We pass ax=subplots[0], to indicate that the top plot in the figure should be this one. In the next line, we calculate the skewness of the AGE column.

In **line 12**, we draw a sample from the AGE column. We then plot it in the next line and pass ax=subplots[1] to indicate that the second plot in the figure should be this one. Then we calculate skewness in the next line. We print the skewness of both the population and the sample in **lines 17-18**.

From the plot, we can see that the distribution of the sample and the population are almost identical. Both are skewed to the right. We can also verify this by looking at the coefficients of skewness of both the sample and the population.

**The central limit theorem** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7nn4DR41QoO#the-central-limit-theorem)

The above plots imply that we cannot apply techniques that apply to a normal distribution to a sample drawn from a population. This is where the *central limit theorem* kicks in. The **central limit theorem** is a very important theorem in probability theory and statistics. It states that the distribution of sample means known as **sampling distribution** will be normally distributed. It means the techniques we used on a normal distribution can be used on the sampling distribution too. We can treat the *sample mean* as if it was drawn from a normal distribution. Let’s see an example of the theorem.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25





In **lines 6-10**, our aim is to sample the AGE column 500500500 times and then store the mean of each sample in a list. To accomplish this, we first initialize an empty list in **line 6**. We call this sample\_means. Then in the next line, we use a for loop. The for loop will run 500500500 times. In the for loop,

* We first draw a sample of 200 items from the AGE column. We do that on **line 8**.
* Then we take the mean of the sample using the mean function in **line 9** and call it s\_mean.
* Now we store this s\_mean in our sample\_means list by using the append method of the list in **line 10**. Because of the loop, **lines 8-10** will run 500500500 times and we will get 500500500 sample means in the list sample\_means.

Now we want to plot the distribution of the sample means that we have collected in sample\_means. To do this, we first convert our list into a Pandas *Series* in **line 13** by using the function pd.Series. We give it our list and it returns a Series. We call it sample\_means\_series. Then we plot it in the next line. By looking at the plot, we can see that it follows roughly a normal distribution as informed by the central limit theorem.

Another interesting point that the central limit theorem implies is that the mean of the sample means is close to the actual population mean. To observe this, we calculate the means of both the population and the sample means and then their difference in **lines 17-19**. We can see from the output of **line 22** that this difference is minor, and this difference will reduce if we increase the number of samples taken from 500500500. We also observe from the output of **line 23** that the sample means have almost 000 skew.

This is how we can estimate population parameters using samples. In the next lesson, we will continue our discussion and focus on confidence intervals.

**Confidence Intervals**

This lesson will focus on the importance and calculation of confidence intervals.

We'll cover the following

* + [Confidence intervals](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals)
    - [Calculating confidence intervals](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#calculating-confidence-intervals)
    - [Confidence intervals using z-values](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals-using-z-values)
    - [Confidence intervals using t-values](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals-using-t-values)

A point estimate can give us a rough idea of a population parameter. But there is a chance that it contains some errors. We need to take many samples to reduce these errors. But taking many samples is not feasible all the time. As in our example of Data Scientists, it might not be feasible to take many different samples of 100 data scientists. What do we do if we face such a situation? We make *confidence intervals*.

**Confidence intervals** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals)

A **confidence interval** is a range of values above and below a point estimate that might contain the true value of a population parameter. The interval is associated with a *confidence level*. The bigger the confidence level, the wider the range of the interval. The confidence level is decided before calculating the confidence intervals. Confidence intervals are usually reported with point estimates to show how reliable the estimates are.

The intervals are calculated from samples which means for each sample, there will be a different interval. Commonly, a 95% confidence level is used, which means there is a 95% chance that the true population parameter lies in the range. Or in other words, if we take 100100100 samples and calculate confidence intervals for each of the 100100100 samples with 95% confidence, the true population parameter will lie in 959595 of these intervals.

**Calculating confidence intervals** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#calculating-confidence-intervals)

We can calculate confidence intervals from point estimates. We add and subtract a margin of error to the point estimate to calculate both boundaries of the interval. The margin of error depends on the:

* Confidence level chosen
* Size of the data
* Standard deviation of the data which denotes the spread of the data

There are two ways of calculating confidence intervals that depend on whether we know the standard deviation of the population or not.

**Confidence intervals using z-values** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals-using-z-values)

If we know the standard deviation of the population then we can calculate the margin of error as

marginoferror=z∗σnmargin\: of\: error = z^\* \frac {\sigma}{\sqrt {n}} marginoferror=z​∗​​​√​n​​​​​σ​​

z∗z^\*z​∗​​= the critical z-value

σ\sigmaσ = standard deviation of the population

nnn = sample size

The **critical z∗z^\*z​∗​​ value** for a 95% confidence interval is the z-score that tells us the number of standard deviations we must go above or below the mean to obtain an area of 95% in a standard normal distribution.

Let’s see how we can calculate confidence intervals in Python. We will be using the [Default of Credit Card Clients Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). We will sample from the AGE column as we did in the last lesson.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29





We calculate the population mean and standard deviation in **lines 8-9** using the functions mean and std respectively. We set sample\_size to 100 in **line 12**. In the next line, we sample from the AGE column using the function sample by passing it n = sample\_size and a random state for repeatability. Then we take the sample mean in **line 14**.

Now to calculate the confidence interval, we need to have the z∗z^\*z​∗​​ value. We know from the below graph that for a standard normal distribution, the z-score for obtaining an area of 95% is 1.961.961.96.

However, we can calculate this value using the Python library scipy which has a stats module that has functions related to statistics. We import the library scipy.stats as st. Then we use the function norm.ppf to calculate the z-score. The norm is for functions related to the normal distribution. This function norm.ppf expects a quantile. We can calculate the quantile for a confidence interval using the formula

quantile=1−(1−CL)2quantile = 1 - \frac{(1 - CL)}{2} quantile=1−​2​​(1−CL)​​

We set the confidence level as cl to 0.95 in **line 17**. Then, we calculate the quantile using the above formula in the next line and finally use the function norm.ppf in **line 19** to retrieve the z-score. We calculate the margin of error using its formula in **line 21**. To calculate the square root, we have imported the library maths which has the function sqrt that we have used in **line 21**. In the next line, we store the confidence interval values in a *tuple*. Then we print all of our findings in **lines 25-28**.

From the output, we see that the population mean lies in between the confidence interval.

**Confidence intervals using t-values** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JEYpYzYggzy#confidence-intervals-using-t-values)

In most cases, we do not know the standard deviation of the population. In that case, we use the standard deviation of the sample instead of the population. But since using the standard deviation of the sample may induce some error, therefore, to account for the error, we calculate **t-values** instead of z-values. t-values are drawn from a **t-distribution**, which is similar to a standard normal distribution, but it goes wider as the sample size falls. But for greater sample sizes, t-distribution is very close to a normal distribution.

Let’s see an example of calculating confidence intervals using the t-distribution. We will use the same data but this time we can assume that the credit card data that we have is a sample of the population.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29





The code is almost the same as above. The first difference is that we have calculated the standard deviation of the sample in **line 14** and used that in the formula of the margin of error in **line 21**. Also, we have used the t-distribution instead of normal distribution. Therefore, we have used the function t.ppf instead of norm.ppf to calculate the t-values. From the output, we can see that the margin of error is greater, in this case, to account for the variability induced by using the standard deviation of the sample.

So, now whenever we have data that we know is not a population, we can report confidence intervals with whichever statistic we report in our findings. In the next lesson, we will use confidence intervals again to learn about *hypothesis testing*.

**Hypothesis Testing**

This lesson will focus on the basics of hypothesis testing and how to perform different types of hypothesis tests.

We'll cover the following

* + [Hypothesis testing](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#hypothesis-testing)
    - [Null and Alternate hypothesis](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#null-and-alternate-hypothesis)
    - [The testing process](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#the-testing-process)

**Hypothesis testing** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#hypothesis-testing)

During our analysis of the different datasets, we are often concerned with questions like whether males default more than females? Do self-driving cars crash more than normal cars? Does drug X help prevent/treat disease Y? To answer these questions, we can use another statistical technique known as **Hypothesis Testing**.

During data exploration, we discovered interesting patterns hidden in the data. Hypothesis testing enables us to confirm whether these patterns were present in the data by luck or by some real phenomena.

**Null and Alternate hypothesis** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#null-and-alternate-hypothesis)

The aim of the hypothesis test is to determine whether the null hypothesis can be rejected or not. The **null hypothesis** is a statement that assumes that nothing interesting is going on, or no relationship is present between two variables, or that there is no difference between a sample and a population.

For instance, if we suspect that males default more than females, the null hypothesis would be that males do not default more than females. If there is little or no evidence against the null hypothesis, we accept the null hypothesis. Otherwise, we reject the null hypothesis in favor of the **alternate hypothesis**, which states that something interesting is going on, or there is a relationship between two variables, or that the sample is different from the population.

To reiterate, the null hypothesis is assumed true and statistical evidence is required to reject it in favor of the alternative hypothesis.

**The testing process** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2G9pj31JDO#the-testing-process)

The testing process is:

* State both hypotheses.
* Choose the **significance level (α\alphaα)**. It is the probability threshold that determines whether you accept or reject a hypothesis.
* Choose a test and compute the test statistic.
* Make the decision to reject or accept the null hypothesis based on if the probability of getting a result as extreme as the one you observe. If it is **lower** than the significance level, you reject the null hypothesis in favor of the alternative. This probability of seeing a result as extreme or more extreme than the one observed is known as the **p-value**.

Usually, three types of tests are carried out:

* One-Sample t-Test
* Two-Sample t-Test
* Paired t-Test

We will look at these in the next few lessons starting from *one-sample t-test* in the next lesson.

**One Sample t-Test**

This lesson will focus on how to perform one sample t-tests in Python.

We'll cover the following

* + - [One-sample t-test](https://www.educative.io/courses/data-science-for-non-programmers/NEKk4JB61EL#one-sample-t-test)

**One-sample t-test** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/NEKk4JB61EL#one-sample-t-test)

A **One sample t-test** checks whether a population mean differs from the sample mean or not. We can use the function ttest\_1samp from the scipy.stats module.

We will be using the [Student Alcohol Consumption Dataset](https://www.kaggle.com/uciml/student-alcohol-consumption/). We will take a sample from the data and find the *mean* grade. We will check if the population mean grade (μ\muμ) differs from the sample mean grade (x¯\bar{x}​x​¯​​) or not.

**Null hypothesis H0H\_0H​0​​**: x¯=μ\bar{x} = \mu​x​¯​​=μ

**Alternate hypothesis HaH\_aH​a​​**: x¯≠μ\bar{x} \neq \mu​x​¯​​≠μ

We choose α\alphaα to be at 95% confidence level which means α=1−(conf.level)=1−0.95=0.05\alpha =1 - (conf.level) = 1 - 0.95 = 0.05α=1−(conf.level)=1−0.95=0.05

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22





Since there were three separate grades, we combined all of the grades in one column and named it grade in **line 5**. In **line 8**, we filter for the grade column since we are only concerned with the grades at this time. We consider this as our population. We find the mean of the population in the next line. In **line 12**, we sample from the population and find the sample mean in the next line. We print both means in **lines 15-16** to see the difference between the two means. Now our test will tell us whether this difference was by chance or if it is statistically significant.

In **line 19**, we use the ttest\_1samp function. It expects two arguments; the sample and the population mean. We provide both the arguments and store the result in result. Then we print the results in the next line.

ttest\_1samp returns the test statistic and the p-value. The test statistic tells us how much the sample mean deviates from the population mean. ttest\_1samp conducts a **two-tailed test** meaning that the p-value returned to us caters for both positive and negative differences in both the means. From the output, we can see that the p-value = 0.40, which is **greater** than our significance level α\alphaα. Therefore, we cannot reject the null hypothesis and the difference in both the means is by chance and not statistically significant. This implies that if we were to construct a 95% confidence interval with the sample mean, the population mean would be captured in it.

In the next lesson, we will look at the *two-sample t-test*.

**Two Sample t-Test**

This lesson will focus on how to perform a two-sample t-test in Python.

We'll cover the following

* + - [Two-sample t-test](https://www.educative.io/courses/data-science-for-non-programmers/m2ZrM21qE7E#two-sample-t-test)

**Two-sample t-test** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2ZrM21qE7E#two-sample-t-test)

A **two-sample t-test** checks whether means of two independent samples differ from each other. We can use the function ttest\_ind from the scipy.stats module to perform the two-sample t-test.

We will be using the [Student Alcohol Consumption Dataset](https://www.kaggle.com/uciml/student-alcohol-consumption/). We will divide the data into two groups based on alcohol consumption. Then we will find the *mean* grade for both groups. We will check if the mean grades for both groups differ from each other or not.

**Null hypothesis H0H\_0H​0​​**: x1¯=x2¯\bar{x\_1} = \bar{x\_2}​x​1​​​¯​​=​x​2​​​¯​​

**Alternate hypothesis HaH\_aH​a​​**: x1¯≠x2¯\bar{x\_1} \neq \bar{x\_2}​x​1​​​¯​​≠​x​2​​​¯​​

We choose α\alphaα to be at 95% confidence level which means α=1−(conf.level)=1−0.95=0.05\alpha =1 - (conf.level) = 1 - 0.95 = 0.05α=1−(conf.level)=1−0.95=0.05

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20





We combine the grades in **line 5** as we did in the previous example. We also combine the weekend (Walc) and weekday (Dalc) consumption to form a new column, alc. We will divide the dataset into two samples. One group with alc values greater than 555 and the other with alc values less than or equal to 555. We filter in **lines 8-9**. Since we are interested in the grade, we separate out the grade columns. We take the mean grades of both samples in **lines 11 and 12** and print these.

We use the function ttest\_ind which expects both samples as arguments. From the result, we can see that the p-value is less than the significance value of 0.050.050.05, which means that we can safely reject the hypothesis and the difference between the mean grades of both samples is statistically significant. Therefore, we can conclude that the mean grade of students who consume less alcohol is greater than those who consume more alcohol.

In the next lesson, we will look at the *paired t-test*.

**Paired t-Test**

This lesson will focus on how to perform the paired t-test in Python.

We'll cover the following

* + - [Paired t-test](https://www.educative.io/courses/data-science-for-non-programmers/JPk6m9pM99o#paired-t-test)

**Paired t-test** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JPk6m9pM99o#paired-t-test)

A **paired t-test** checks whether the means of the same sample differ at two different times. We can use the function ttest\_rel from the scipy.stats module to perform the paired t-test.

We will check the mean of grades for two exams, G1 and G3. G1 is the result of the exam after the first quarter and G3 is the result of exam at the end of the year. We will check if the mean grades differ for the same sample at different times.

**Null hypothesis H0H\_0H​0​​**: x1¯=x2¯\bar{x\_1} = \bar{x\_2}​x​1​​​¯​​=​x​2​​​¯​​

**Alternate hypothesis HaH\_aH​a​​**: x1¯≠x2¯\bar{x\_1} \neq \bar{x\_2}​x​1​​​¯​​≠​x​2​​​¯​​

We choose α\alphaα to be at 95% confidence level which means α=1−(conf.level)=1−0.95=0.05\alpha =1 - (conf.level) = 1 - 0.95 = 0.05α=1−(conf.level)=1−0.95=0.05

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18





We take the grades for G1 and G3 in **lines 6-7** in sample\_1 and sample\_2 respectively. We calculate their means in **lines 9-10**.

We use the ttest\_rel function in **line 15** that expects both samples, to test the hypothesis. From the results, we can see that the p-value is less than the threshold of 0.050.050.05 which means that we reject the null hypothesis and the drop in performance in G3 is statistically significant.

This brings the end of this chapter. We have learned how to construct confidence intervals and test our hypotheses. In the next chapter, we will move towards the last stage in the data science lifecycle, i.e., *prediction*.

# A Simple Model

This lesson will introduce predictive modeling and will focus on how to construct a simple model with loss functions.

We'll cover the following

* + [Modeling](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#modeling)
    - [Predicting waiter tips](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#predicting-waiter-tips)
    - [Loss functions](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#loss-functions)
      * [Mean squared error](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#mean-squared-error)

In the previous chapters of this course, we have looked at data cleaning, data exploration, and statistical inference. Now it is time to move to the last stage of the data science lifecycle, which is making predictions that help us in decision making. Until now, we have discovered interesting patterns in the data that we know were significant, but how do we use these patterns to predict future events? With this objective, we make predictions using models.

## Modeling [#](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#modeling)

A **model** is a representation of a system. It tries to approximate real-world phenomena. For instance, Isaac Newton gave us a model for tries to approximate gravity. We can make predictions using the model that how far or high a ball will go if we throw it with a certain force. In the gravity model, there are certain factors that affect the outcome such as the force with which the ball is thrown, or the mass of the ball, and so on. In the same way, we can make models to predict whether a certain client will default on the credit card payment next month or not, where the client’s history of payments and some other factors may affect the outcome of our model.

There are many different ways of making models and measuring their effectiveness. But first, let’s start with a very simple model.

### Predicting waiter tips [#](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#predicting-waiter-tips)

We have the data of customers that paid a tip at a restaurant. We will try to make a model that predicts the tip paid by the customer. Let’s load the dataset and look at it first.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17





Now a very simple model can be that the customer always pays **15%** of the total bill as the tip. So mathematically:

predictedtip=0.15∗total\_billpredicted\: tip = 0.15 \* total\\_bill predictedtip=0.15∗total\_bill

Here 0.150.150.15 is called our model **parameter**. If we denote the model parameter with θ\thetaθ, total\_bill value with xxx and the predicted tip with y^\hat{y}​y​^​​ then the above equation becomes

y^=θx\hat{y} = \theta x ​y​^​​=θx

So, our simple model becomes a mathematical function, f(x)f(x)f(x), that takes in an input xxx and gives us the output of predicted tip.

Let’s first create a column to check the percent tips.

1

2

3

4

5

6

7

8

9

10





We calculate the percentage tips in **line 6** by multiplying the tip by 100.0 and then dividing it by the total\_bill. Afterward, we round the values so that they are easy to distinguish in **line 7**. Then we plot the counts of each value of percent\_tip in **line 10**. We first retrieve the counts of each value in percent\_tip by using the value\_counts function and then plot them using the plot function.

By looking at the plot, we find out that our model that always predicted the tip to be 15% does not predict accurately in all cases. So, we can say that our model is not performing well. At this point, we need some way of measuring how far off the predictions are from the actual values. This is where loss functions come in.

### Loss functions [#](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#loss-functions)

A **loss function** is a mathematical function that takes in predicted values that we predicted using our model parameter and a set of actual values (YYY) and tells us how well our model performs on the entire input data (set of xxx values: XXX). Loss functions output a single value known as **loss**, which tells us how much loss or error we are getting when using a certain model parameter. Since loss functions output loss, the smaller the loss value, the better the model performance.

Therefore, we can compare which values of the model parameters perform better than others by making predictions using different model parameters and giving those predicted values to a loss function and then choosing the model parameter that gets the minimum value from the loss function.

Hence, we choose the model that **minimizes** a loss function.

L(θ,Y)L(\theta,Y) L(θ,Y)

But how does a loss function compute loss? There are many ways to compute loss, and several loss functions have been designed for several problems, but perhaps the most common and intuitive is the **Mean Squared Error** loss function.

#### Mean squared error [#](https://www.educative.io/courses/data-science-for-non-programmers/N8p0GN8QwOL#mean-squared-error)

As the name suggests, this loss function computes the mean of squared error. This is a very intuitive way of computing a loss. Let’s see why, with an example.

Suppose that we choose our θ=0.15\theta = 0.15θ=0.15 as before. The first value of total\_bill in our tips dataset is 16.9916.9916.99.

y^=θx\hat{y} = \theta x ​y​^​​=θx

y^=0.15∗16.99\hat{y} = 0.15 \* 16.99 ​y​^​​=0.15∗16.99

y^=2.54\hat{y} = 2.54 ​y​^​​=2.54

If we denote the actual value of tip for the first row by yyy, then the error in the prediction is:

y−y^=1.01−2.54y - \hat{y} = 1.01 - 2.54 y−​y​^​​=1.01−2.54

y−y^=−1.53y - \hat{y} = -1.53 y−​y​^​​=−1.53

We can repeat this for all values of inputs (x1,x2,x3,x4,...,xnx\_1,x\_2,x\_3,x\_4,...,x\_nx​1​​,x​2​​,x​3​​,x​4​​,...,x​n​​) and compute all the errors. Since the errors can be positive and negative, we take the squares of the errors. Now that we have all the squared errors, we take their mean to get the average error in a prediction using the current θ\thetaθ.

L(θ,Y)=mean{(y1−y1^)2,(y2−y2^)2,(y3−y3^)2,.....,(yn−yn^)2}L(\theta,Y) = mean\{(y\_1-\hat{y\_1})^2,(y\_2-\hat{y\_2})^2,(y\_3-\hat{y\_3})^2,.....,(y\_n-\hat{y\_n})^2\} L(θ,Y)=mean{(y​1​​−​y​1​​​^​​)​2​​,(y​2​​−​y​2​​​^​​)​2​​,(y​3​​−​y​3​​​^​​)​2​​,.....,(y​n​​−​y​n​​​^​​)​2​​}

L(θ,Y)=1n((y1−y1^)2+(y2−y2^)2+(y3−y3^)2+.....+(yn−yn^)2)L(\theta,Y) = \frac {1}{n} ( (y\_1-\hat{y\_1})^2+(y\_2-\hat{y\_2})^2+ (y\_3-\hat{y\_3})^2+ .....+ (y\_n-\hat{y\_n})^2) L(θ,Y)=​n​​1​​((y​1​​−​y​1​​​^​​)​2​​+(y​2​​−​y​2​​​^​​)​2​​+(y​3​​−​y​3​​​^​​)​2​​+.....+(y​n​​−​y​n​​​^​​)​2​​)

L(θ,Y)=1n∑i=1n(yi−yi^)2L(\theta,Y) = \frac{1}{n}\sum\_{i=1}^{n}{( y\_i - \hat{y\_i})^2} L(θ,Y)=​n​​1​​​i=1​∑​n​​(y​i​​−​y​i​​​^​​)​2​​

This was the theory behind loss functions. In the next lesson, we will implement this loss function to find out the best model for our problem of predicting tips in Python.

**Model Fitting on a Loss Function**

This lesson will focus on implementing the mean squared error loss function in Python and applying optimization to obtain the best performing model.

We'll cover the following

* + [Coding up the loss function](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#coding-up-the-loss-function)
  + [Getting predictions](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#getting-predictions)
  + [Minimizing the loss function](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#minimizing-the-loss-function)

In the last lesson, we learned how loss functions can be used in theory to find out the best model for our problem. Now, we need to implement the mean squared error function in Python to our dataset.

**Coding up the loss function** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#coding-up-the-loss-function)

First, we will write the MSE (mean squared error) function, and then use that on our dataset.

1

2

3

4





We define a function that will work on two columns of dataframe, i.e., Series objects. We have two input series y\_pred which has predicted values and y\_actual which has the actual values. In **line 2**, we first calculate the error and then take the square by using the \*\* operator. Since these are Series objects, Python will automatically subtract the corresponding y\_actual and y\_pred values. In the next line, we use the function mean on sq\_error and return the loss.

**Getting predictions** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#getting-predictions)

Now we will get predictions using the data that we have for a single model parameter and then compute the loss for that model.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18





We define our function, mse\_loss, in **lines 4-7**. Then we read the data in **line 10**. After that we compute the predicted values in **line 14** by multiplying theta with total\_bill column. This gives us predicted values for each row in the dataset. Next, we compute the loss in **line 17**. We use the function mse\_loss defined above. We give predicted\_values as y\_pred and the column tip as y\_actual to the function. Then, we print the loss returned from the function in the last line.

We get a loss value of approximately $1.19\$1.19$1.19 which means that the average squared difference between the predicted value and the actual value is approximately $1.19\$1.19$1.19.

**Minimizing the loss function** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/myrBOL0Z5O3#minimizing-the-loss-function)

Now that we know how to get loss values for a model, we need to find the model that best predicts our data by comparing losses for different models. We will set up some values for model parameter and compare the losses for all of these values.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





Let’s examine the long piece of code part by part. Our aim was to minimize the loss function by comparing losses for different values of the model parameter and then choosing the model parameter that gave the lowest loss.

We define our loss function in **lines 6-9**, as we did above. Then we store some values in a list for our model parameter, θ\thetaθ in **line 14**, that we will be using later on. We create an empty list and call it losses in the next line. This list will be used to store the losses that we get for all the model parameters.

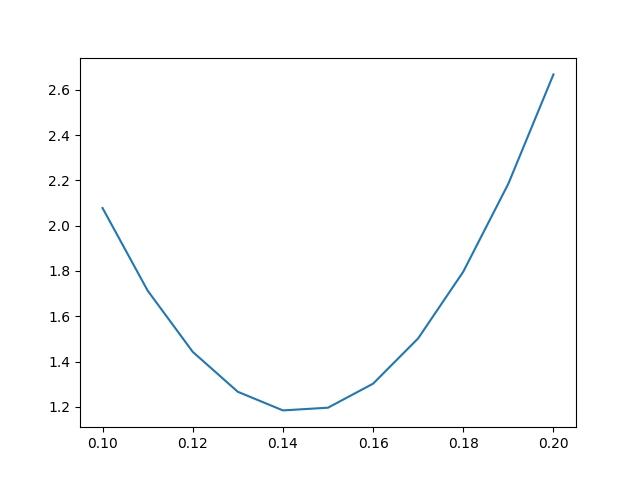
We use a for loop since we want to repeat the same process of computing the loss for different values of θ\thetaθ. In **line 16**, we start the for loop.

for theta in theta\_values

This means that for every iteration of this for loop, the keyword theta will take a value from our list theta\_values. Inside the loop, we compute the predicted values and the loss as we did before. In **line 24**, we store the loss that we calculated in the list of losses by using append.

After the loop finishes, losses will have the loss values for all thetas in theta\_values. Now to find the minimum loss, we use the built-in function min in **line 27**. But we need to get the corresponding θ\thetaθ for it as well. So, we also find out the index of the minimum value from the losses list in **line 28** by using the function argmin from the numpy library that we imported in **line 2**. We index the list theta\_values using the min\_loss\_index and retrieve the corresponding θ\thetaθ.

In **lines 31-34**, we display our findings. It is always a good idea to plot the loss values against model parameters. Since losses are not stored in a dataframe, we use the plot function from the matplotlib library. We give it theta\_values as the first argument and losses as the second argument. It plots the first argument on the x-axis and the second on the y-axis.



We can confirm our results from the plot as well. We see that we encountered minimal loss when the model parameter was 0.140.140.14. It implied that the average squared error in the prediction was $1.18\$1.18$1.18. So, in this case, we got the best model at θ=0.14\theta=0.14θ=0.14 by minimizing the mean squared loss function.

The curve that we plotted above is sometimes called the **error curve** or the **error surface**. The process of finding the best model according to a loss function is called **model fitting**. The goal of model fitting is to find the minimum of the error curve.

But, did you spot any issues with our implementation of model fitting? We will discuss a very important issue in the next lesson that will lead us to a technique called *gradient descent*.

**Gradient Descent**

This lesson will focus on the intuition behind the gradient descent algorithm.

We'll cover the following

* + [Intuition](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#intuition)
    - [Direction of change in θ\thetaθ](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#direction-of-change-in-theta)
    - [Amount of change in θ\thetaθ](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#amount-of-change-in-theta)
  + [Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#gradient-descent)

In the last lesson, we minimized a loss function to find the best model to predict the tip paid by customers. But there was a drawback with the approach. We manually entered the values of the model parameter θ\thetaθ and compared the losses. But this approach of manually choosing the values of the model parameters is not scalable because:

* It works only on predetermined values of θ\thetaθ
* Most models have many model parameters and complex structures of the prediction function, for which it will require a lot of time to choose parameters manually.
* We may not choose the best set of model parameters, and then we will not get to the best model.

We need some approach that chooses the model parameters automatically and then arrives at the best model.

**Intuition** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#intuition)

Since we need a method in which we do not use predetermined values of θ\thetaθ, let’s start by picking a random value of θ\thetaθ and see what our loss is. After this, we will decide whether to increase the current value of θ\thetaθ or decrease it and the amount to increase or decrease. Let’s look at the direction and amount separately.

**Direction of change in θ\thetaθ** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#direction-of-change-in-theta)

Look at the error surface of the example in the previous lesson below.

We have highlighted two points in this curve. The red highlighted point A is the value of the loss at θ=0.10\theta=0.10θ=0.10. If we choose this as our starting point for θ\thetaθ, then at this point, we need to choose a new value for θ\thetaθ that is closer to the minimum of this curve. Let’s look at the slope of the line at this point.

The slope at point A is **negative**, which means that if we increase θ\thetaθ from this point, the loss will decrease. Therefore, we need to *increase* the value of θ\thetaθ from here to reach the minimum of the curve.

Now let’s consider another situation where we start at θ=0.18\theta=0.18θ=0.18. This point B is highlighted in black. Looking at the slope of the line at point B, we can see that the slope is **positive**, which means that if we increase θ\thetaθ from this point, the loss will increase. Hence, we need to *decrease* θ\thetaθ from here to reach the minimum.

Therefore, we establish the rule that if the slope is **negative**, we need to *increase* the value of θ\thetaθ, and if the slope is **positive**, we need to *decrease* the value of θ\thetaθ. So, now that we have established the direction in which we need to move, the question of what amount should we move remains.

**Amount of change in θ\thetaθ** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#amount-of-change-in-theta)

The natural approach would be to increase or decrease θ\thetaθ by the amount of gradient at this point. Since we need to compute the gradient to capture the direction of the slope, why not increase or decrease by the gradient amount itself? Let’s see how we will do this mathematically.

If at instant ttt, we call our model parameter θt\theta\_tθ​t​​, then at the next instant t+1t+1t+1, we call it θt+1\theta\_{t+1}θ​t+1​​. To update the value of θ\thetaθ we do,

θt+1=θt−∂∂θL(θt,Y)\theta\_{t+1} = \theta\_t - \frac{\partial}{\partial \theta} L(\theta\_t,Y) θ​t+1​​=θ​t​​−​∂θ​​∂​​L(θ​t​​,Y)

where L(θt,Y)L(\theta\_t,Y)L(θ​t​​,Y) is the loss function which depends on the values of θ\thetaθ and the set of actual values YYY to give us a loss value. ∂∂θ\frac{\partial}{\partial \theta}​∂θ​​∂​​ means that we are taking the partial derivate of the loss function with respect to θ\thetaθ. We subtracted the gradient instead of adding to apply the rule we made above of going opposite to the direction of the gradient.

We multiply the gradient term with a constant α\alphaα which ranges from 000 to 1,1,1, so that we can control the rate at which we update the model parameter θ\thetaθ. This term is known as the **learning rate**. Hence, the above expression becomes

θt+1=θt−α∂∂θL(θt,Y)\theta\_{t+1} = \theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta\_t,Y) θ​t+1​​=θ​t​​−α​∂θ​​∂​​L(θ​t​​,Y)

Hence, we have come up with a way to update the model parameters based on how they perform. This procedure is known as **Gradient Descent**.

**Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m2q5L0yQ1Vp#gradient-descent)

It is an algorithm to update the values of model parameters to find the best model. It works as

* Start with random initial value of θ\thetaθ.
* Compute θt−α∂∂θL(θt,Y)\theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta\_t,Y)θ​t​​−α​∂θ​​∂​​L(θ​t​​,Y) to update the value of θ\thetaθ.
* Keep updating the value of θ\thetaθ until it stops changing values. This can be the point where we have reached the minimum of the error function.

The process of using algorithms like gradient descent to minimize a function is sometimes called **optimization**.

This was the gradient descent algorithm. In the next lesson, we will implement it in our example of predicting tips given by the customer.

**Optimization with Gradient Descent**

This lesson will focus on how to implement gradient descent algorithm in Python.

We'll cover the following

* + [Minimization with Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#minimization-with-gradient-descent)
    - [gradient](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#gradient)
    - [Stopping condition](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#stopping-condition)
    - [Learning rate](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#learning-rate)
    - [Implementation](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#implementation)
    - [Results](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#results)
  + [Types of Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#types-of-gradient-descent)
    - [Batch Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#batch-gradient-descent)
    - [Stochastic Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#stochastic-gradient-descent)
    - [Mini-batch Gradient Descent](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#mini-batch-gradient-descent)
  + [Scratching the surface](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#scratching-the-surface)

In the previous lesson, we looked at the intuition behind the gradient descent algorithm and the update equation. In this lesson, we are going to implement it in Python. We are going to predict the tips paid by a customer at a restaurant. We will choose the best model using gradient descent.

**Minimization with Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#minimization-with-gradient-descent)

Recall that the gradient descent algorithm is

* Start with a random initial value of θ\thetaθ.
* Compute θt−α∂∂θL(θt,Y)\theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta\_t,Y)θ​t​​−α​∂θ​​∂​​L(θ​t​​,Y) to update the value of θ\thetaθ.
* Keep updating the value of θ\thetaθ until it stops changing values. This can be the point where we have reached the minimum of the error function.

We will be using the *tips* dataset that has the following data.

1

2

3

4

5

6

7

8





Our simple model said that for predicting tips we only need the amount of the total bill paid by the customer. Therefore, our prediction (y^\hat{y}​y​^​​) depends on the total bill (xxx) and the model parameter (θ\thetaθ). We have:

y^=θx\hat{y} = \theta x ​y​^​​=θx

We will need a function that gives us the derivative of the loss function.

**gradient** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#gradient)

Before we can implement this in code on our predicting tips example, we need to evaluate the gradient term in the update expression

θt+1=θt−α∂∂θL(θt,Y)\theta\_{t+1} = \theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta\_t,Y) θ​t+1​​=θ​t​​−α​∂θ​​∂​​L(θ​t​​,Y)

We know that if our Loss function is *mean squared error* then:

∂∂θL(θt,Y)=∂∂θt[1n∑i=1n(yi−y^i)2]\frac{\partial}{\partial \theta} L(\theta\_t,Y) = \frac{\partial}{\partial \theta\_t}[ \frac{1}{n}\sum\_{i=1}^{n} (y\_i - \hat{y}\_i )^2 ] ​∂θ​​∂​​L(θ​t​​,Y)=​∂θ​t​​​​∂​​[​n​​1​​​i=1​∑​n​​(y​i​​−​y​^​​​i​​)​2​​]

since we chose a simple model which said that for predicting tips we only need the amount of the total bill paid by the customer. Therefore, our prediction (y^\hat{y}​y​^​​) depends on the total bill (xxx) and the model parameter (θ\thetaθ). We have

y^=θx\hat{y} = \theta x ​y​^​​=θx

∂∂θL(θt,Y)=∂∂θt[1n∑i=1n(yi−θtxi)2]\frac{\partial}{\partial \theta} L(\theta\_t,Y) = \frac{\partial}{\partial \theta\_t}[ \frac{1}{n}\sum\_{i=1}^{n} (y\_i - \theta\_t x\_i )^2 ] ​∂θ​​∂​​L(θ​t​​,Y)=​∂θ​t​​​​∂​​[​n​​1​​​i=1​∑​n​​(y​i​​−θ​t​​x​i​​)​2​​]

Evaluating this expression gives us:

∂∂θL(θt,Y)=−2n∑i=1n(yi−θtxi)(xi)\frac{\partial}{\partial \theta} L(\theta\_t,Y) = \frac{-2}{n} \sum\_{i=1}^{n} (y\_i - \theta\_t x\_i)(x\_i ) ​∂θ​​∂​​L(θ​t​​,Y)=​n​​−2​​​i=1​∑​n​​(y​i​​−θ​t​​x​i​​)(x​i​​)

Therefore, we will be coding this expression to compute the gradient. gradient function will need the same arguments as the mse\_loss\_func which are:

* theta: The model parameters θ\thetaθ
* x: The dataset needed to make predictions
* y: The actual values with which to compare

1

2

3

4

5

6





In **line 3**, we evaluate the expression inside the summation and take its sum in the next line. **In line 5**, we multiply it by −2n\frac{-2}{n}​n​​−2​​ to compute the gradient. We return the gradient in the last line.

**Stopping condition** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#stopping-condition)

Now we need to decide a stopping condition for gradient descent. We will define a number epsilon and say that we will stop updating our model parameter when the change in the model parameter is below epsilon. In our case, we can set it to 0.0010.0010.001.

**Learning rate** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#learning-rate)

We will call this alpha. Its value should be between 000 and 111.

**Implementation** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#implementation)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





In **lines 4-8**, we write the loss function that we discussed above. In **lines 11-16**, we write the gradient function. We read the data and then give values to epsilon (**line 22**) and the learning rate alpha (**line 23**). We chose our initial θ=0.0\theta=0.0θ=0.0 on **line26**. We make a variable, iterations\_completed, which will keep track of the number of iterations of gradient descent at all times.

We start a while loop on **line 30** which will keep running until we break it with a break statement. Then we start the gradient descent algorithm and compute the gradients in **line 32**. We find the new value of theta (new\_theta) in **line 35**. One iteration of gradient descent is complete until here. Therefore, we increase the iterations\_completed by 111.

We then find the loss with new\_theta on **line 39** and print our findings. To test for the stopping condition, we compute the absolute difference between new\_theta and theta on **line 47** and test the condition on the next line. If the stopping condition is satisfied, we exit the loop with a break statement in **line 49**. But if the condition is not satisfied, we move to **line 51** where the variable theta is given the value of new\_theta, so that the same variables can be used in the next iteration of the loop.

**Results** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#results)

From the output, we can see that it takes only 333 iterations for gradient descent to reach the best model. It chooses θ≈0.143\theta \approx 0.143θ≈0.143. The mean squared loss for θ≈0.143\theta \approx 0.143θ≈0.143 is approximately 1.171.171.17 which is better than what we chose manually a few lessons back. This is the best model parameter that can be chosen for this model and gives the minimum error.

**Types of Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#types-of-gradient-descent)

Gradient descent is normally categorized into three types:

* Batch Gradient Descent
* Stochastic Gradient Descent
* Mini-batch Gradient Descent

We will study them one by one.

**Batch Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#batch-gradient-descent)

This is the gradient descent that we implemented above. In **batch gradient descent**, we calculate the gradient of the loss function on the entire dataset. We update model parameters according to the errors introduced on the whole data set. However, this approach is not scalable with large datasets. A single iteration may take a lot of time if the dataset is very large.

**Stochastic Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#stochastic-gradient-descent)

In **stochastic gradient descent**, we compute the gradients of the loss function on a single randomly chosen example form the dataset. Instead of using the whole dataset in every iteration we use a single data point from the dataset to compute the gradient of the loss function.

Even though batch gradient descent takes big steps towards the minimum, stochastic gradient descent is faster for a single iteration and it takes steps towards the minimum very rapidly. It may reach the minimum faster for large datasets than batch gradient descent, as batch gradient descent takes steps after a long time.

**Mini-batch Gradient Descent** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#mini-batch-gradient-descent)

Mini-batch gradient descent is in between the two approaches explained above. In **mini-batch gradient descent**, a set of random data points are chosen to calculate the gradient of the loss function at each iteration. It was named **mini-batch** because the chosen set is a very small subset of the entire dataset. This is used widely for optimizing functions as it provides a balance between the speed and size of steps taken towards the minimum.

**Scratching the surface** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7AVMoXMlm0r#scratching-the-surface)

So far in this chapter, we have learned how to use gradient descent to minimize a function. Gradient Descent is a very useful algorithm that we will keep coming back to as our problems and models become more and more complex. The current example used a very basic model for an easy problem. We can perform better at predicting tips from this dataset by using some other models with gradient descent.

In the next lesson, we will look at a fundamental concept for predictive analysis which is *linear regression*.

**Simple Linear Regression**

This lesson will focus on what linear regression is and why we need it.

We'll cover the following

* + [Optimizing the Simple linear model](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#optimizing-the-simple-linear-model)
  + [Simple linear model fitting in Python](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#simple-linear-model-fitting-in-python)
    - [gradient](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#gradient)
    - [Stopping condition](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#stopping-condition)
    - [Learning rate](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#learning-rate)
    - [Implementation](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#implementation)
    - [Results](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#results)
  + [Best fit line](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#best-fit-line)

During the last stage of the data science lifecycle, we are faced with the question of which model to choose to make predictions. We can decide what kind of model to use by looking at the relationship between the data variables that we have.

Let’s take the example of predicting tips paid to waiters.

1

2

3

4

5

6

7

8

9





From the plot, we can see that there is a direct linear relationship between total\_bill and tips where increasing/decreasing total\_bill results in an increase/decrease in tips as well. Looking at the linear relationship, we need a linear model for this problem. If we represent total\_bill values by xxx then our prediction (y^\hat{y}​y​^​​) becomes

y^=θ0x+θ1\hat{y} = \theta\_0x + \theta\_1 ​y​^​​=θ​0​​x+θ​1​​

This is the form of a linear equation. The term θ1x\theta\_1xθ​1​​x implies that increasing/decreasing total bill(x) will increase/decrease the tip(y^\hat{y}​y​^​​). Now that we have our model, we need to fit it to the data using gradient descent optimization. We will be using the *mean squared error* loss function.

**Optimizing the Simple linear model** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#optimizing-the-simple-linear-model)

If we denote our predictions by y^\hat{y}​y​^​​ and the actual values by yyy, then loss function will be calculated as:

L(θ,X,Y)=1n∑i=1n(yi−yi^)2L(\theta,X,Y) = \frac{1}{n}\sum\_{i=1}^{n}{( y\_i - \hat{y\_i})^2} L(θ,X,Y)=​n​​1​​​i=1​∑​n​​(y​i​​−​y​i​​​^​​)​2​​

L(θ,X,Y)=1n∑i=1n(yi−(θ0x+θ1))2L(\theta,X,Y) = \frac{1}{n}\sum\_{i=1}^{n}{( y\_i - (\theta\_0x + \theta\_1 ))^2} L(θ,X,Y)=​n​​1​​​i=1​∑​n​​(y​i​​−(θ​0​​x+θ​1​​))​2​​

We will be minimizing this loss function with gradient descent. Recall that in gradient descent we:

* Start with random initial value of θ\thetaθ.
* Compute θt−α∂∂θL(θ,X,Y)\theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta,X,Y)θ​t​​−α​∂θ​​∂​​L(θ,X,Y) to update the value of θ\thetaθ.
* Keep updating the value of θ\thetaθ until it stops changing values. This can be the point where we have reached the minimum of the error function.

**Simple linear model fitting in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#simple-linear-model-fitting-in-python)

We will need a function that gives us the partial derivative of the loss function.

**gradient** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#gradient)

Before we can implement this in code on our predicting weights example, we need to evaluate the gradient term in the update expression

θt+1=θt−α∂∂θL(θt,X,Y)\theta\_{t+1} = \theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta\_t,X,Y) θ​t+1​​=θ​t​​−α​∂θ​​∂​​L(θ​t​​,X,Y)

We know that if our Loss function is *mean squared error* then

∂∂θL(θ,X,Y)=∂∂θt[1n∑i=1n(yi−y^i)2]\frac{\partial}{\partial \theta} L(\theta,X,Y) = \frac{\partial}{\partial \theta\_t}[ \frac{1}{n}\sum\_{i=1}^{n} (y\_i - \hat{y}\_i )^2 ] ​∂θ​​∂​​L(θ,X,Y)=​∂θ​t​​​​∂​​[​n​​1​​​i=1​∑​n​​(y​i​​−​y​^​​​i​​)​2​​]

=1n∑i=1n−2(yi−yi^)∂∂θt(yi^)= \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i}) \frac{\partial}{\partial \theta\_t}(\hat{y\_i}) =​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)​∂θ​t​​​​∂​​(​y​i​​​^​​)

We know that yi^=θ1xi+θ2\hat{y\_i} = \theta\_1x\_i + \theta\_2​y​i​​​^​​=θ​1​​x​i​​+θ​2​​

=1n∑i=1n−2(yi−yi^)∂∂θt(θ0xi+θ1)= \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i}) \frac{\partial}{\partial \theta\_t}(\theta\_0x\_i + \theta\_1) =​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)​∂θ​t​​​​∂​​(θ​0​​x​i​​+θ​1​​)

Now since we have two parameters (θ0\theta\_0θ​0​​ and θ1\theta\_1θ​1​​), we need to differentiate with respect to both. So

Derivating w.r.t θ0\theta\_0θ​0​​

∂∂θL(θ,X,Y)=1n∑i=1n−2(yi−yi^)∂∂θ0(θ0xi+θ1)\frac{\partial}{\partial \theta} L(\theta,X,Y) = \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i}) \frac{\partial}{\partial \theta\_0}(\theta\_0x\_i + \theta\_1) ​∂θ​​∂​​L(θ,X,Y)=​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)​∂θ​0​​​​∂​​(θ​0​​x​i​​+θ​1​​)

=1n∑i=1n−2(yi−yi^)xi= \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i})x\_i =​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)x​i​​

Derivating w.r.t θ1\theta\_1θ​1​​

∂∂θL(θ,X,Y)=1n∑i=1n−2(yi−yi^)∂∂θ1(θ0xi+θ1)\frac{\partial}{\partial \theta} L(\theta,X,Y) = \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i}) \frac{\partial}{\partial \theta\_1}(\theta\_0x\_i + \theta\_1) ​∂θ​​∂​​L(θ,X,Y)=​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)​∂θ​1​​​​∂​​(θ​0​​x​i​​+θ​1​​)

=1n∑i=1n−2(yi−yi^)= \frac{1}{n} \sum\_{i=1}^{n} -2(y\_i - \hat{y\_i}) =​n​​1​​​i=1​∑​n​​−2(y​i​​−​y​i​​​^​​)

Therefore, we will be coding this expression to compute the gradient. gradient function will need the same arguments as the mse\_loss\_func which are:

* thetas: The model parameters θ\thetaθ
* x: The dataset needed to make predictions
* y: The actual values with which to compare

1

2

3

4

5

6

7





In **line 3**, we evaluate the expression inside the summation for θ0\theta\_0θ​0​​ and take its sum. We do the same in the next line for θ1\theta\_1θ​1​​. Then in **line 5** we make a numpy array to store both gradients. In **line 6**, we multiply it by −2n\frac{-2}{n}​n​​−2​​ to compute the gradient. We return the gradient in the last line. Note that the returned variable grad is also a numpy array of size 222.

**Stopping condition** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#stopping-condition)

Now we need to decide a stopping condition for gradient descent. We will define a number epsilon and say that we will stop updating our model parameter when the change in the model parameter is below epsilon. In our case, we can set it to 0.0010.0010.001.

**Learning rate** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#learning-rate)

We will call this alpha. Its value should be between 000 and 111.

**Implementation** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#implementation)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





In **lines 5-9**, we write the loss function that we discussed above. In **lines 12-18**, we write the gradient function. We read the data and then give values to epsilon (**line 24**) and the learning rate alpha (**line 25**). We chose our initial θ0=0.0\theta\_0=0.0θ​0​​=0.0 and θ1=0.1\theta\_1=0.1θ​1​​=0.1 on **line 28**. We make a variable, iterations\_completed, which will keep track of the number of iterations of gradient descent at all times.

We start a while loop on **line 32** which will keep running until we break it with a break statement. Then we start the gradient descent algorithm and compute the gradients in **line 34**. We find the new value of theta (new\_thetas) in **line 37**. One iteration of gradient descent is complete here. Therefore, we increase the iterations\_completed by 111.

We then find the loss with new\_thetas on **line 41** and print our findings. To test for the stopping condition, we compute the absolute difference between new\_thetas and thetas on **line 49** and test the condition on the next line. If the stopping condition is satisfied, we exit the loop with the break statement in **line 51**. But if the condition is not satisfied, we move to **line 53** where the variable thetas is given the value of new\_thetas, so that the same variables can be used in the next iteration of the loop.

**Results** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#results)

From the output, we can see that it takes only 333 iterations for gradient descent to reach the best model. It chooses θ0≈0.139\theta\_0 \approx 0.139θ​0​​≈0.139 and θ1≈0.107\theta\_1 \approx 0.107θ​1​​≈0.107. The mean squared loss is approximately 1.141.141.14, which is better than what we had gotten in the previous lesson. These are the best model parameters that can be chosen for this model and give the minimum error.

**Best fit line** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7XE0ZgKG11y#best-fit-line)

The equation of our model y^=θ0x+θ1\hat{y} = \theta\_0x + \theta\_1​y​^​​=θ​0​​x+θ​1​​ is the equation of a line with θ0\theta\_0θ​0​​ as the slope of the line and θ1\theta\_1θ​1​​ as the intercept. The line produced by this equation is called the **best fit** line. We can plot this line alongside the scatter plot we plotted above.

1

2

3

4

5

6

7

8

9

10

11

12

13

14





In **line 8** we plot the scatter plot between total\_bill and tip. In **line 11**, we initialize our thetas to the best fit thetas we found above by gradient optimization. We retrieve predictions using these thetas in the next line. Then we plot the predictions at the y-axis and total\_bill at x-axis on the same scatter plot.

From the plot, we can see that line fits the data quite well. The whole process that we did above for finding this best fit line is called **linear regression**. It gives us the best fit line, i.e., the line with the minimum error.

This is a very fundamental concept that we will extend in the next lesson. Can you think of a way to improve or extend this model?

# Multiple Linear Regression

This lesson will introduce multiple linear regression and focus on how to perform it in Python.

We'll cover the following

* + [Multiple Linear Regression](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#multiple-linear-regression)
  + [Minimizing the Loss function](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#minimizing-the-loss-function)
  + [Model fitting in Python](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#model-fitting-in-python)
    - [sklearn.metrics.mean\_squared\_error](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#sklearnmetricsmean_squared_error)
      * [mean\_squared\_error](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#mean_squared_error)
    - [sklearn.linear\_model.LinearRegression](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#sklearnlinear_modellinearregression)
      * [fit](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#fit)
      * [predict](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#predict)

In the last lesson, we performed simple linear regression. But it was a limited model because we only used one variable from our dataset in our predictions. Our linear model was based on the fact that there was some relationship between two variables. But the variable that we are trying to predict can have relationships with other variables too. Including other variables in our model may help us make better predictions.

## Multiple Linear Regression [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#multiple-linear-regression)

We can extend the simple linear regression model to a multiple linear regression model by adding more variables and parameters to the model equation. Then the prediction (y^\hat{y}​y​^​​) becomes:

y^=θ0+θ1x1+θ2x2+θ3x3+....+θmxm\hat{y} = \theta\_0 + \theta\_1x\_1 + \theta\_2x\_2 + \theta\_3x\_3 + .... + \theta\_mx\_m ​y​^​​=θ​0​​+θ​1​​x​1​​+θ​2​​x​2​​+θ​3​​x​3​​+....+θ​m​​x​m​​

where x1,x2,x3,...,xnx\_1,x\_2,x\_3,...,x\_nx​1​​,x​2​​,x​3​​,...,x​n​​ are different attributes in our data. The above linear equation takes multiples attributes, multiplies them with some weights to get a prediction. Weights can be obtained by minimizing a loss function using gradient descent as we did for simple linear regression.

We will be using the [USA Housing Dataset](https://www.kaggle.com/dmvreddy91/usahousing). Our goal is to predict the **price** of a house. Other variables in the dataset are listed below.

USA\_Housing.csv

1

2

3

4

5

6

7

8





Now we need to decide what variables we need to include in our model. For that we can look at the relationships of all variables with Price.

1

2

3

4

5

6





We have plotted scatter plots of all variables with each other using the function scatter\_matrix. Note that this function was not called on the dataframe df, rather it was used with pd since it is a function available in Pandas. We gave it our dataframe df and the figure size.

We only need to focus on the last row of these scatter plots as Price is on the y-axis here. We can see that all five variables show some kind of pattern with Price which means we should include these variables in our model. So, the model equation will become:

y^=θ0+θ1∗AvgAreaIncome+θ2∗AvgAreaHouseAge+θ3∗AvgAreaNumberRooms \hat{y} = \theta\_0 + \theta\_1\*AvgAreaIncome + \theta\_2\*AvgAreaHouseAge + \theta\_3\*AvgAreaNumberRooms ​y​^​​=θ​0​​+θ​1​​∗AvgAreaIncome+θ​2​​∗AvgAreaHouseAge+θ​3​​∗AvgAreaNumberRooms

+ θ4∗AvgAreaNumberBedrooms+θ5∗AreaPopulation+ \ \theta\_4\*AvgAreaNumberBedrooms + \theta\_5\*AreaPopulation + θ​4​​∗AvgAreaNumberBedrooms+θ​5​​∗AreaPopulation

Here θ0\theta\_0θ​0​​ is called the **intercept** and it is not multiplied with any variable. Now that we have our model decided we need to minimize the Mean Squared Error function.

## Minimizing the Loss function [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#minimizing-the-loss-function)

If we denote our predictions by y^\hat{y}​y​^​​ and the actual values by yyy, then loss function will be calculated as

L(θ,X,Y)=1n∑i=1n(yi−yi^)2L(\theta,X,Y) = \frac{1}{n}\sum\_{i=1}^{n}{( y\_i - \hat{y\_i})^2} L(θ,X,Y)=​n​​1​​​i=1​∑​n​​(y​i​​−​y​i​​​^​​)​2​​

L(θ,X,Y)=1n∑i=1n(yi−(θ0+θ1x1+θ2x2+θ3x3+....+θmxm))2L(\theta,X,Y) = \frac{1}{n}\sum\_{i=1}^{n}{( y\_i - (\theta\_0 + \theta\_1x\_1 + \theta\_2x\_2 + \theta\_3x\_3 + .... + \theta\_mx\_m ))^2} L(θ,X,Y)=​n​​1​​​i=1​∑​n​​(y​i​​−(θ​0​​+θ​1​​x​1​​+θ​2​​x​2​​+θ​3​​x​3​​+....+θ​m​​x​m​​))​2​​

We will be minimizing this loss function with gradient descent. Recall that in gradient descent we:

* Start with random initial values of θ\thetaθ.
* Compute θt−α∂∂θL(θ,X,Y)\theta\_t - \alpha \frac{\partial}{\partial \theta} L(\theta,X,Y)θ​t​​−α​∂θ​​∂​​L(θ,X,Y) to update the value of all θ\thetaθ.
* Keep updating the value of all θ\thetaθ until they stop changing values. This can be the point where we have reached the minimum of the error function.

We need to do step 2 for all the parameters as we did for 2 parameters in simple linear regression in the previous lesson.

## Model fitting in Python [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#model-fitting-in-python)

In the previous lesson, we coded the entire gradient descent algorithm for simple linear regression ourselves. But extending that to multiple linear models can be messy. We have libraries in Python that will do that for us. We do not need to get into the messy details of mathematics. However, we do need to know what is going on behind the scenes when we call a library function, that is why we coded gradient descent ourselves in the previous lessons.

Python has a library called **scikit-learn** that has all kinds of functions that help make different kinds of predictive models and optimize them. It also has a lot of loss functions that we can use directly. Scikit-learn is written in code as sklearn. We will be importing two things from it.

### sklearn.metrics.mean\_squared\_error [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#sklearnmetricsmean_squared_error)

The metrics module, which can be imported as sklearn.metrics, has different loss functions that we can use. We will import the mean squared error function from this.

#### mean\_squared\_error [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#mean_squared_error)

It gives us the mean squared error loss. It expects the following arguments:

* y\_true: Actual values of target variable
* y\_pred: Predicted values of target variable

### sklearn.linear\_model.LinearRegression [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#sklearnlinear_modellinearregression)

The linear\_model module has functions and classes to use different kinds of predictive models. A model like a linear regression model is implemented as a class. It has different attributes and functions that we can use. The two main functions are:

#### fit [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#fit)

It makes a model using the data provided and fits it to find the best model parameters. It expects two arguments:

* X: The data that will be used to predict
* Y: The actual values of the target variable

#### predict [#](https://www.educative.io/courses/data-science-for-non-programmers/gkXnnYZ1Xl3#predict)

It gives us the predicted values using the equation y^=θ0+θ1x1+θ2x2+θ3x3+....+θmxm\hat{y} = \theta\_0 + \theta\_1x\_1 + \theta\_2x\_2 + \theta\_3x\_3 + .... + \theta\_mx\_m​y​^​​=θ​0​​+θ​1​​x​1​​+θ​2​​x​2​​+θ​3​​x​3​​+....+θ​m​​x​m​​. It expects the following arguments

* X: The data that will be used to predict. For a dataframe, it gives us a predicted value for each row.

Now that we know what functionalities we need from the library, let’s use them below to predict house prices.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





In **lines 3 and 4**, we have imported the LinearRegression class and mean\_squared\_error function as discussed above. We read the data into a dataframe in **line 7**. Since we decided above that we will not be using any non-numeric variables for prediction, we drop Price and Address and form a new dataframe X in **line 10**. It has all the variables that we can use in prediction. In **line 11**, we separate the actual values of Price in a dataframe called Y.

In **line 14**, we initialize the LinearRegression class. We call the class object lr. We then use the fit function to fit our model in the next line. The fit function will find the best model for us and store the model parameters internally.

Now we get predictions using our fitted model in **line 18** using the predict function. We save them in predictions. Then, we find the mean squared error in the predictions using the mean\_squared\_error function. We print the loss in the next line. In the end, we print our model parameters which are stored as coef\_. The intercept (θ0\theta\_0θ​0​​) is stored as intercept\_.

So, this is how easily we can make linear regression models in Python. In the next lesson, we will focus on the performance of these models.

**Evaluating Regression Models**

This lesson will focus on ways to evaluate the performance of regression Models.

We'll cover the following

* + [Losses](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#losses)
    - [Interpreting losses](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-losses)
  + [Plotting absolute error percentages](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#plotting-absolute-error-percentages)
    - [Interpreting absolute error percentages](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-absolute-error-percentages)
  + [R2R^2R​2​​ score](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#r2-score)
    - [Interpreting R2R^2R​2​​](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-r2)
    - [Contribution of each variable](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#contribution-of-each-variable)
  + [Takeaway](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#takeaway)

In the previous lessons, we learned how to make and fit linear regression models in Python. But we did not discuss ways to judge the performance of the models. In this lesson, we will focus on techniques used to evaluate the performance of linear regression models.

We will be using the same model that we used in the last lesson where we tried to predict house prices using the [USA Housing Dataset](https://www.kaggle.com/dmvreddy91/usahousing).

**Losses** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#losses)

We can evaluate the model performance by looking at different losses. We have already looked at mean squared loss.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





In **lines 3 and 4**, we have imported the LinearRegression class and mean\_squared\_error function. We read the data into a dataframe in **line 7**. Since we will not be using any non-numeric variables for prediction, we drop Price and Address and form a new dataframe X in **line 10**. It has all the variables that we can use in prediction. In **line 11**, we separate the actual values of Price in a dataframe called Y.

In **line 14**, we initialize the LinearRegression class and call the class object lr. We then use the fit function to fit our model in the next line. The fit function will find the best model for us and store the model parameters internally.

Now we get predictions using our fitted model in **line 18** using the predict function. Then we add a column Predictions in the dataframe in **line 20**. The next line will show us the actual values and predicted values of the top 5 rows side by side.

In **line 23**, we take the mean squared error and save it as mse\_loss using the mean\_squared\_error function. In the next line, we take the mean absolute error using the mean\_absolute\_error function and save it as mae\_loss. Both functions expect the same arguments, actual values (y\_true) and predicted values(y\_pred). We print these losses in the next two lines.

**Interpreting losses** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-losses)

Losses are a good indication of the performance of the model. Now by looking at the losses, we can see that the model does not perform great. The more intuitive Mean Absolute Error of almost 810008100081000 does not seem very good performance by the model on average.

However, some loss metrics, such as MSE and MAE, are greatly affected by outliers. For instance, there might be some outliers in the data that push the mean loss value up. Therefore, we also calculate the median absolute loss in **line 28**. We can see that the median absolute loss is almost 690006900069000, which is a noticeable drop from the mean absolute error. This shows that we cannot always rely on loss functions to evaluate the performance, so we might need some other ways to look at the model’s performance.

**Plotting absolute error percentages** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#plotting-absolute-error-percentages)

To check how well our model performed let’s plot the absolute percentage error in each prediction

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





After performing the regression, we compute absolute percentage errors in **line 21**. We use the following formula:

%error=abs(actual−predicted)actual∗100\% error = \frac{abs(actual - predicted)}{actual} \* 100 %error=​actual​​abs(actual−predicted)​​∗100

We take the absolute error because an error can be both negative and positive. Then in **line 22**, we plot these errors. To plot, we use the function plot with plt. On the x-axis, we give a list of values using range(errors.shape[0]). The errors.shape[0] is the number of errors that we have. It is 500050005000 in this example. range(5000) gives us a consecutive list that starts from 000 and ends at (4999). We give the errors on the y-axis. Note that using errors.plot(kind='bar') would have given us the same plot as well but unlike this example, we do not always have our errors in a series or dataframe object, so we use the plot function from matplotlib.

**Interpreting absolute error percentages** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-absolute-error-percentages)

Looking at the absolute error percentages, we can see that most of the errors are within 15%. This error may be acceptable in this example.

We can also see some outliers that we were discussing in the above section. The model performs poorly on these examples. There might be some other factors that were responsible for the different prices of these houses which our model does not consider.

**R2R^2R​2​​ score** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#r2-score)

**R-squared** also known as **coefficient of determination** is a statistical measure that tells us the strength of the relationship between a model and the *dependent (predicted)* variable. It measures the proportion of the variation in the dependent variable that can be explained by the model. It is also known as the **goodness of fit** of a model.

Mathematically it is:

R2=1−ExplainedVariationTotalVariationR^2 = 1 - \frac{Explained \: \: Variation}{Total \: \:Variation} R​2​​=1−​TotalVariation​​ExplainedVariation​​

Its value ranges from 0 to 1. A value of 1 means 100% variation of the target variable can be explained by the model.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





We will extend the code that we wrote in the above. After performing regression, in **line 21**, we calculate R2R^2R​2​​ using the r2\_score function that we imported in **line 3** from sklearn.metrics. It comes out to be approximately 0.910.910.91

**Interpreting R2R^2R​2​​** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#interpreting-r2)

A value of 0.910.910.91 means that the variables in our model can account for 91% of the variation in the price of houses.

However, this does not mean that our model is 91% accurate. Higher R2R^2R​2​​ values do not guarantee reliable and accurate models. We can see that in this example, we have a very high R2R^2R​2​​ value but out model is not very accurate.

**Contribution of each variable** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#contribution-of-each-variable)

Since our regression models are linear of the form:

y^=θ0+θ1x1+θ2x2+θ3x3+....+θmxm\hat{y} = \theta\_0 + \theta\_1x\_1 + \theta\_2x\_2 + \theta\_3x\_3 + .... + \theta\_mx\_m ​y​^​​=θ​0​​+θ​1​​x​1​​+θ​2​​x​2​​+θ​3​​x​3​​+....+θ​m​​x​m​​

we can see that each independent variable is multiplied by a model parameter θ\thetaθ. These model parameters can be thought of as weights of each variable. A variable with a higher value of weight contributes more to predicting the target variable, and a variable with a small weight does not contribute much. Thus, we can interpret the importance of independent variables to the target variable by their weights.

**Takeaway** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/xllZ5w47l8n#takeaway)

The big takeaway from our discussion in this lesson is that we cannot rely on a single metric to evaluate the performance of our model. Being better in one metric does not mean we will get good values for other metrics. However, if a model is poor in one metric such as losses, then we can disregard the model.

So far, we have learned how to make and evaluate a linear regression model for predicting numerical quantities. In the next lesson, we will look at regression for categorical variables.

**Logistic Regression**

This lesson will focus on logistic regression in Python.

We'll cover the following

* + [Logistic function](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#logistic-function)
  + [Cost function](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#cost-function)
  + [Logistic Regression in Python](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#logistic-regression-in-python)

Until now, we have been predicting numerical quantities. But what if we want a model to predict a categorical variable? Categorical data is divided into distinct classes. The task of predicting a categorical variable is known as **classification**. We can perform classification using logistic regression.

**Logistic function** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#logistic-function)

**Logistic Regression** is a classification method that is built on the same concept as linear regression. In linear regression, we take a linear combination of different variables plus an intercept term to predict the output. But in classification problems, the predicted variable is categorical. The simplest case of classification is when the predicted variable is binary, i.e., it has only two classes, e.g., yes/no, male/female, etc. Logistic regression also takes the linear combination of different variables plus the intercept term, but afterward, it takes the result and passes it through a **logistic** function. The logistic function also known as **sigmoid** is defined as:

sigmoid(t)=11+e−tsigmoid(t) = \frac{1}{1+ e^{-t} } sigmoid(t)=​1+e​−t​​​​1​​

where t is the output of the linear regression equation, i.e., linear combination of variables plus the intercept term. Let’s look at the plot of the logistic function below.

The logistic function has a fixed range. It will always output numbers in the range of 000 to 111. This means that we will always get an output in the range of 0 to 1. Therefore, our prediction function h(x)h(x)h(x) becomes:

h(x)=g(f(x))h(x) = g(f(x)) h(x)=g(f(x))

where g is the sigmoid function and f(x)=θ0+θ1x1+θ2x2+...+θmxmf(x)=\theta\_0 + \theta\_1x\_1 + \theta\_2x\_2+ ... + \theta\_mx\_mf(x)=θ​0​​+θ​1​​x​1​​+θ​2​​x​2​​+...+θ​m​​x​m​​

The illustration below explains this concept.

In logistic regression, this output is interpreted as the probability of the observation belonging to the second class. In binary classification, if the result is greater than 0.5, we say that the observation belongs to the second class, and if it is less than 0.5, it belongs to the first class.

**Cost function** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#cost-function)

The cost function used instead of mean squared error is the **cross-entropy** function.

J(y)=1m∑i=1m[ −yi log(h(xi))−(1−yi)log(h(xi)) ]J(y) = \frac{1}{m} \sum\_{i=1}^{m}[\ -y\_i \ log( h(x\_i)) - (1 - y\_i)log(h(x\_i)) \ ] J(y)=​m​​1​​​i=1​∑​m​​[ −y​i​​ log(h(x​i​​))−(1−y​i​​)log(h(x​i​​)) ]

where yiy\_iy​i​​ denotes labels of our classes. It can be 1 or 0 for binary classification. The expression inside the square brackets is the loss for one observation. The error is summed for all observations.

This function will be minimized using gradient descent, as we did earlier for linear regression. However, we will not go further into the math of how gradient descent would optimize this function at this point and move straight to using it in Python.

**Logistic Regression in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JYlQoqNYRkv#logistic-regression-in-python)

Now it is time for us to perform logistic regression in Python. Fortunately, sklearn has us covered. We will use the LogisticRegression class available in sklearn.linear\_model.

To evaluate the performance, we will be using the function accuracy\_score from sklearn.metrics, which tells us the percentage of accurate results.

We will be predicting whether a credit card client defaults or not by using the [Credit Card Clients Default Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). The binary prediction variable is default.payment.next.month.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20





We load the data in **line 5**. Then we separate our training data and the predictions in **lines 7 and 8**. We initialize the class in **line 10**. We use the fit function to fit the model and obtain the best parameters in **line 11**. Then we print the model parameters and the intercept parameter in **lines 13 and 14**. Afterward, we obtain predictions using the predict function. We calculate the accuracy in **line 17**. accuracy\_score expects the actual values and the predicted values. Then we print the accuracy in the last line.

From the output, we can see that our model gives the correct prediction 77% of the time.

In the next lesson, we will look at how we can evaluate logistic regression models.

# Evaluating Logistic Regression Models

This lesson will focus on how to evaluate logistic regression models.

We'll cover the following

* + [Evaluating classification models](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#evaluating-classification-models)
    - [Classification report](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#classification-report)
      * [Precision](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#precision)
      * [Recall](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#recall)
      * [F1-Score](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#f1-score)
      * [Support](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#support)
  + [Multiclass classification](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#multiclass-classification)

## Evaluating classification models [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#evaluating-classification-models)

Just like there were many ways to evaluate linear regression models, there are many ways to evaluate the performance of classification models. Accuracy is one of the techniques. But it is not a sufficient metric alone. Why?

Think about a scenario where our model predicts a rare disease that is present only in 0.01% of the data. If our model always predicts that no disease is present, it will still be accurate 99% of the time but it would not diagnose correctly when it matters the most.

### Classification report [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#classification-report)

A classification report is a table that calculates different metrics to evaluate our model. We can obtain the table using the function classification\_report in sklearn.metrics

We will make the same model that we made in the last lesson.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17





By printing the classification report we can see different metrics listed for both classes. Let’s look at these one by one. But before that, we need to know the concept of positive class and a negative class. A **positive class** is the one in which we are interested. If we are interested in predicting customers that will not default, then no is the positive class and yes is the negative class. However, we can invert this and call yes a positive class and no the negative class. In our example above, no is the positive class. Some of the concepts associated with positive and negative classes are:

* **True Positive**: A true positive is an outcome where the model correctly predicts the positive class.
* **False Positive**: A false positive is an outcome where the actual class was the negative class, but the model predicts the positive class.
* **True Negative**: A true negative is an outcome where the model correctly predicts the negative class.
* **False Negative**: A false negative is an outcome where the actual class was the positive class but the model predicts the negative class.

#### Precision [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#precision)

Precision is a measure of how well the model performs when it predicts the positive class. It tells us that for all instances classified positive, what percent was correct. It is defined as:

precision=TPTP+FPprecision = \frac{TP}{TP + FP} precision=​TP+FP​​TP​​

#### Recall [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#recall)

Recall is a measure of how well the model predicts the positive class. It tells us that for all instances that were actually positive, what percent was classified correctly.

recall=TPTP+FNrecall = \frac{TP}{TP + FN} recall=​TP+FN​​TP​​

#### F1-Score [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#f1-score)

If precision increases, recall decreases and vice versa. Therefore, a new measure was introduced called F1-score. It is the harmonic mean of precision and recall. The best score is 1.0, whereas the worst score is 0.0.

F1=2precision∗recallprecision+recallF1 = 2 \frac{precision \* recall}{precision + recall} F1=2​precision+recall​​precision∗recall​​

#### Support [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#support)

Support is the number of instances predicted in each class.

We need to look at all of these measures when evaluating our logistic regression models. The above model that we created using logistic regression does not do a very good job of predicting. The accuracy is 77%. Precision is on the lower side. Also, the F1-score is on the lower side, as well. A reason could be **imbalanced classes**. In our dataset, there are more negative examples than positive ones. This affects the outcome of the model, as well.

## Multiclass classification [#](https://www.educative.io/courses/data-science-for-non-programmers/39gxvwzBmgO#multiclass-classification)

We can use the same code with different data for multi-class classification as well. The fit function will handle the details itself, and we would not need to do anything extra.

This brings an end to this lesson. In the next lesson, you will be given a challenge that you need to complete.

**Exercise: Churn Prediction**

This lesson gives an exercise on churn prediction using logistic regression in python.

We'll cover the following

* + [Churn prediction](https://www.educative.io/courses/data-science-for-non-programmers/gx7n65155WG#churn-prediction)
    - [Prediction challenge](https://www.educative.io/courses/data-science-for-non-programmers/gx7n65155WG#prediction-challenge)

**Churn prediction** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gx7n65155WG#churn-prediction)

In this lesson, you are required to make a predictive model using logistic regression that predicts churn. The dataset given to you is from a telecom operator for August and September 2015. It has the following variables:

1

2

3

4

5

6

7

8

9

10

11

12

13





You will be writing the function churn\_prediction\_acc. You will be given four dataframes:

* X: It has all the inputs that you will need to fit your model.
* Y: It has the target variable, i.e., the Class for every input in X.
* test\_inputs: Input dataframe on which to make predictions.
* test\_outputs: The target variable, i.e., the Class for every input in test\_inputs.

Steps you are required to do are:

1. Fit a logistic regression model using X and Y.
2. Obtain predictions on test\_inputs.
3. Find the accuracy of the model on the predictions obtained in step 2 and return the accuracy.

**Prediction challenge** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gx7n65155WG#prediction-challenge)

1

2

3

4

5

6

7

8





If you feel stuck, feel free to check out the solution review in the next lesson. Good luck!

**Solution Review: Churn Prediction**

This lesson will present the solution to the exercise of churn prediction in the previous lesson.

We'll cover the following

* + [Solution](https://www.educative.io/courses/data-science-for-non-programmers/YMqvvXjOY7O#solution)

**Solution** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YMqvvXjOY7O#solution)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15





The solution is simple. We make a logistic regression model in **line 3**. We fit the model using the X and Y provided in the next line. We take the predictions by using the predict function in preds in **line 6**. Then we use the accuracy\_score function and provide both the actual values and the predicted values. The function gives us the accuracy of the predictions, which we then return in **line 8**.

With this exercise, we conclude this chapter. In the next chapter, we will continue our discussion of predictive models, and look at some other models that are used.

**Why Machine Learning**

This lesson will focus on why we need machine learning models for predictions.

We'll cover the following

* + [Issues with Regression](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#issues-with-regression)
    - [Non-linear relationships](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#non-linear-relationships)
    - [Linear Regression parameters converge on the mean of the predicted variable](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#linear-regression-parameters-converge-on-the-mean-of-the-predicted-variable)
    - [Sensitivity to outliers](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#sensitivity-to-outliers)
    - [Independent data assumption](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#independent-data-assumption)
  + [What is Machine Learning?](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#what-is-machine-learning)
    - [Supervised Learning](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#supervised-learning)
    - [Unsupervised Learning](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#unsupervised-learning)
  + [Machine Learning vs. Artificial Intelligence vs. Data Science](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#machine-learning-vs-artificial-intelligence-vs-data-science)

**Issues with Regression** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#issues-with-regression)

In the previous chapter, we learned how to use linear and logistic regression models for making predictions from data. But there are some issues with the regression framework. We will look at these issues one by one.

**Non-linear relationships** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#non-linear-relationships)

By design, linear regression explores linear relationships between the dependent and independent variables. It assumes that there is a straight-line relationship between the variables and tries to find the line that best fits the data. Sometimes, it is not the case that variables follow a linear relationship. For instance, the relationship between age and income is not linear. Income rises exponentially during the early years and then grows almost linearly at the later stages.

**Linear Regression parameters converge on the mean of the predicted variable** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#linear-regression-parameters-converge-on-the-mean-of-the-predicted-variable)

Linear regression finds the mean of the dependent variable since the error at the mean is relatively less to all points as compared to some other value. That is why in our **tips** example, the parameter chosen was close to the mean of the percentage tip. However, looking at extremes is also important.

**Sensitivity to outliers** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#sensitivity-to-outliers)

Linear regression is sensitive to outliers since it looks at the mean of the data. The best fit line can change direction to try to fit outliers.

**Independent data assumption** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#independent-data-assumption)

Linear regression assumes that there is no significant relationship between the dependent variables. However, that is not always the case. Correlated independent variables affect the performance of linear regression models. This problem is also known as **multi collinearity** in statistics. Although there are ways to handle this, the performance of linear regression is not satisfactory when the dependent variables have relationships among themselves.

Because of these issues present in typical data, the performance of linear regression is often not very good. Linear regression works best when there are linear relationships between the dependent and independent variables, and the data contains no outliers. Therefore, we need another framework of predictive models that perform better. This is where *machine learning* comes in.

**What is Machine Learning?** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#what-is-machine-learning)

**Machine Learning** is the branch of computer science that deals with algorithms and systems performing specific tasks using patterns and inference, rather than explicitly programmed instructions. One use case of machine learning is predictive models. Most machine learning tasks can be categorized in the following two types:

* Supervised Learning
* Unsupervised Learning

**Supervised Learning** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#supervised-learning)

In **supervised learning**, we make predictive models using data that has *labels*. When we have the target variable available in our data, we call the values of target variable **labels**. Until now, when we made linear regression models, we had labels available. Some supervised machine learning algorithms to make predictive models are:

* Decision Trees
* Neural Networks
* Random Forests
* Support Vector Machines (SVM)

We will look at some of these later in this chapter.

**Unsupervised Learning** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#unsupervised-learning)

**Unsupervised learning** is when we use unlabeled data to allow a model to learn relationships between data observations and pick up on underlying patterns. Most data in the world is unlabeled, which makes unsupervised learning a very useful method of machine learning. The most common algorithms for unsupervised learning are *clustering algorithms*. We will look at some of these later in this chapter.

**Machine Learning vs. Artificial Intelligence vs. Data Science** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/7D8mq8vqo3Q#machine-learning-vs-artificial-intelligence-vs-data-science)

People often use the terms *machine learning*, *artificial intelligence*, and *data science* interchangeably. In reality, machine learning is a subset of artificial intelligence and overlaps heavily with data science. Artificial intelligence deals with any technique that allows machines to display *intelligence*, similar to humans. Machine learning is one of the main techniques used to create artificial intelligence, but other non-ML techniques are also widely used in AI.

On the other hand, data science deals with gathering insights from datasets. Traditionally, data scientists have used statistical methods, such as regression, for gathering these insights. However, as machine learning continues to grow, it has also penetrated into the field of data science.

In the next lesson, we will look at the Machine Learning pipeline.

**Machine Learning Pipeline**

This lesson will focus on the process of training machine learning models.

We'll cover the following

* + [Training process](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#training-process)
    - [Feature engineering](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#feature-engineering)
    - [Selecting the model](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#selecting-the-model)
    - [Train and test set](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#train-and-test-set)
    - [Training the model](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#training-the-model)
    - [Evaluating model performance](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#evaluating-model-performance)
    - [Model tuning](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#model-tuning)

In the last lesson, we talked about why we need machine learning models. This leads us to our first model, which is a *decision tree*. But before we dive into the details of decision trees, we need to look at the training process of these models.

**Training process** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#training-process)

To use a machine learning model, we go through the following different stages:

* Data collection
* Data preprocessing
* Feature engineering
* Selecting the model
* Splitting the data set into train and test sets
* Use the train set to train the model
* Evaluate performance on the test set
* Tune the model

Machine Learning Pipeline

We are already familiar with the first two steps as we have done these extensively in this course.

**Feature engineering** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#feature-engineering)

This includes combining and manipulating existing variables, aka *features* in ML, to make new features that might help us capture the relationships better. This also includes transforming the features to a different scale. For instance, normalizing skewed data.

**Selecting the model** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#selecting-the-model)

This is a very important task in the whole process. Choosing which model to use requires looking at the data and deciding which model will work better with the data.

**Train and test set** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#train-and-test-set)

We divide our data into two sets, a *training* set and a *testing* set. We use the training set repeatedly to train the model.

**Training the model** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#training-the-model)

The model is given the training set. It learns the hidden patterns and relationships in the data. It is not given anything from the testing set during the training process. Most of the models are trained by algorithms similar to gradient descent optimization. The models update their parameters based on how they perform on the training set, repeating this process several times.

**Evaluating model performance** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#evaluating-model-performance)

After training, models are evaluated by giving them the testing set as input, and the predictions they make on the testing set are evaluated using different metrics and loss functions.

**Model tuning** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/gxpjjXPlLor#model-tuning)

If the models do not give satisfactory performance, we change the *hyperparameters* of the models. We will look at hyperparameters later in this chapter.

With this, we can now start using our first ML model. We will look at *decision trees* in the next lesson.

**Decision Trees**

This lesson will focus on training decision tree models in Python.

We'll cover the following

* + [Decision trees](https://www.educative.io/courses/data-science-for-non-programmers/YV5vxQPVorn#decision-trees)
    - [Decision trees in Python](https://www.educative.io/courses/data-science-for-non-programmers/YV5vxQPVorn#decision-trees-in-python)

**Decision trees** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YV5vxQPVorn#decision-trees)

In the first lesson of this chapter, we talked about how linear regression models focus only on linear relationships between the dependent and independent variables; they fail to capture nonlinear relationships. Decision trees are made to capture nonlinear relationships.

**Decision trees** model data as a **tree** of hierarchical branches. It is a flowchart-like structure in which each internal node represents a *test* on an attribute (e.g. whether a coin flip comes up heads or tails), each branch represents the outcome of the test, and each leaf node represents a class label (decision taken after computing all attributes). The paths from the root to the leaf represent classification rules. Decision Trees can adapt to both regression and classification tasks.

Common terms used with Decision trees:

* **Root node**: It represents the entire population or sample, and this further gets divided into two or more homogeneous sets.
* **Splitting**: It is a process of dividing a node into two or more sub-nodes.
* **Decision node**: When a sub-node splits into further sub-nodes, then it is called a decision node.
* **Leaf/Terminal node**: Nodes that do not split are called Leaf or Terminal node.
* **Pruning**: When we remove sub-nodes of a decision node, this process is called pruning. It is the opposite process of splitting.
* **Branch/Sub-tree**: A subsection of the entire tree is called branch or sub-tree.
* **Parent** and **Child** node: A node, which is divided into sub-nodes is called a parent node of sub-nodes, whereas sub-nodes are the children of the parent node.

A very common example that is given in the context of decision trees is that we want to classify a person as unfit or fit based on the person’s age, whether he/she eats pizza, and whether he/she exercises in the morning. A decision tree of this could be:

From the diagram, we can see that at every node there is a yes/no decision. We keep moving in the tree until we reach the leaf nodes, where the observation is classified into a class.

**Decision trees in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YV5vxQPVorn#decision-trees-in-python)

We will be using the [Audit Risk Dataset](https://www.kaggle.com/sid321axn/audit-data) of different firms. We will be performing the binary classification task of predicting whether a company is fraudulent or not. The dataset has the following attributes:

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19





In python, the sklearn package has a lot of machine learning models implemented as classes. We will be importing the DecisionTreeClassifier class from sklearn.tree. We will also be importing a function named train\_test\_split from sklearn.model\_selection that divides our dataset into train and test sets.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





After we read the data in **line 6**, we separate our target variable as Y. We drop the LOCATION\_ID column since it would not provide any useful information to the model. To split the data into training and test sets, we use the function train\_test\_split. We provide our inputs, X, and the labels, Y, to the function in **line 12**. We also provide the test set size as test\_size. 0.20.20.2 implies that 20% data will be included in the testing set, while the rest 80% will form the training set. The function outputs 444 items that we can retrieve directly into 444 variables. These are:

1. Inputs for the training data that we store in X\_train
2. Inputs for the testing data that we store in X\_test
3. Labels for the training data that we store in Y\_train
4. Labels for the testing data that we store in Y\_test

In **line 15**, we make our Decision Tree model just like we did in the last lesson. We call DecisionTreeClassifier without any arguments. Then in the next line, we call the fit function of the model. We provide the training examples and labels to the function. After this, we need to evaluate our model. Therefore, we use the predict function of the model in **line 19** to store predictions in preds. We give the testing inputs X\_test to predict as an argument. We use the accuracy\_score function to measure the accuracy of the predictions. We print the accuracy with the classification report, which we obtained by using the classification\_report function, in the last two lines.

From the outputs, we can see that the model performs excellent on the testing data. The model has learned all patterns and relationships in the dataset and gives correct results 100% of the time.

Note that this was a relatively easier task for the model. For bigger and more complex datasets, we might not get 100% accuracy.

This brings an end to this lesson where we looked at Decision Trees. In the next lesson, we will look at another machine learning model called *Random Forests*.

# Random Forests

This lesson will focus on training random forest models in Python.

We'll cover the following

* + [Random Forests](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#random-forests)
  + [How do Random Forests work?](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#how-do-random-forests-work)
    - [Uncorrelated trees](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#uncorrelated-trees)
      * [Bagging](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#bagging)
      * [Feature randomness](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#feature-randomness)
    - [Performing as a committee](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#performing-as-a-committee)
  + [Random Forests in Python](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#random-forests-in-python)
    - [Hyper parameters tuning](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#hyper-parameters-tuning)

## Random Forests [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#random-forests)

**Random Forest**, as the name implies, consists of a large number of [decision trees](https://www.educative.io/collection/page/10370001/4733468011397120/6280646001426432) that operate as an ensemble for classification. An **ensemble** is a collection of different predictive models that collectively decide the predicted output. In random forests, each individual tree gives a class as an output. The class with the most votes gets chosen as the final output of the model.

Picture courtesy of towardsdatascience.com

In the above example, we can see that six trees predict class 1 while three predict 0. Class 1 has more votes; hence the final prediction is 1.

## How do Random Forests work? [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#how-do-random-forests-work)

The idea behind random forests is that a large number of uncorrelated decision trees working individually will perform better as a committee than any individual tree.

There are two important words in the above statement.

* Uncorrelated trees
* Performing as a committee

### Uncorrelated trees [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#uncorrelated-trees)

For a random forest model to perform nicely, the individual decision tree models need to have low correlation amongst themselves. Just like investments with low correlations, such as stocks and bonds, combine to form a portfolio greater than the sum of its parts, a random forest can produce predictions better than its individual trees. The reason is that all trees do not have the same error. Rather each tree has a different kind of error. They collectively move in a direction to reduce the total error. Therefore, the predictions made by individual trees need to have a low correlation.

To ensure that the outcomes of the individual decision trees are uncorrelated, random forests use two techniques:

* Bagging
* Feature randomness

#### Bagging [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#bagging)

Decision trees are very sensitive to training data, and any change in training data can significantly change the outcome of the model. Random forests make use of this observation. Each individual decision tree randomly samples from the training data with replacement. This means each tree has a different training set, which leads to different decision tree models. This technique of using random samples with replacement is known as **bagging**.

Note that we do not train individual trees on random subsets of the data, rather they are trained on the whole data set where each training example is randomly sampled with replacement. For instance, if our training data has 6 observations such as [1,2,3,4,5,6] and we sample 6 times with replacement, we might get [1,2,2,4,5,5]. Therefore, each individual tree will have a different training set with some overlap.

Therefore, each individual decision tree is different.

#### Feature randomness [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#feature-randomness)

In decision trees, when we split a node, we consider all features and then decide on the feature that gives us the most separation between the left node and the right node. But in random forests, individual trees can only select from a subset of features. This introduces more variability among the trees. Therefore, the trees made have low correlation amongst themselves.

So, in random forest models, we produce individual decision trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions.

### Performing as a committee [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#performing-as-a-committee)

In the end, a random forest model chooses the class which had the most votes from individual decision trees.

## Random Forests in Python [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#random-forests-in-python)

We can easily use random forests for our predictions in Python. The model is available in sklearn.ensemble as RandomForestClassifier. We will be using the [Default of Credit Card Clients](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset) dataset to make our predictions.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





We import the RandomForestClassifier in **line 2**. After we read the data in **line 6**, we separate our target variable as Y. To split the data into training and test sets, we use the function train\_test\_split. We provide our inputs X and the labels Y to the function in **line 12**. We also provide the test set size as test\_size. 0.20.20.2 implies that 20% data will be included in the testing set, while the remaining 80% will form the training set. The function outputs 444 items that we can retrieve directly into 444 variables. These are:

1. Inputs for the training data that we store in X\_train
2. Inputs for the testing data that we store in X\_test
3. Labels for the training data that we store in Y\_train
4. Labels for the testing data that we store in Y\_test

In **line 15**, we make our Random Forest classifier object just like we did in the last lesson. We call RandomForestClassifier without any arguments. Then in the next line, we call the fit function of the model. We provide the training examples and labels to the function. After this, we need to evaluate our model. Therefore, we use the predict function of the model in **line 19** to store predictions in preds. We give the testing inputs X\_test to predict as an argument. We use accuracy\_score function to measure the accuracy of the predictions. We print the accuracy with the classification report, which we obtained by using the classification\_report function, in the last two lines.

We see that the model is approximately 80% accurate.

### Hyper parameters tuning [#](https://www.educative.io/courses/data-science-for-non-programmers/RLKvpwEM1vO#hyper-parameters-tuning)

When we made the random forest model above, we did not mention how many trees to include in the model or how many features each tree should use. Parameters like these are called **hyperparameters** of a model.

Let’s make a model by including 202020 trees in the model. By default, it used 10 individual decision trees.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





All of the code is the same except **line 15** in which we give n\_estimator = 20. This implies that the model should use 20 decision trees. We can see that the accuracy changes a bit. For a list of all hyperparameters, refer to the documentation of [Random Forest Classifier](https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html).

In the next lesson, we will learn another machine learning model known as Support Vector Machine.

**Support Vector Machines**

This lesson will focus on training Support Vector Machines.

We'll cover the following

* + [Support Vector Machine](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#support-vector-machine)
    - [Kernels](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#kernels)
  + [SVM in Python](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#svm-in-python)

**Support Vector Machine** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#support-vector-machine)

A **Support Vector Machine (SVM)** is a supervised machine learning algorithm that can be used for both classification and regression problems. In this algorithm, we plot each data item as a point in n-dimensional space (where n is the number of features we have) with the value of each feature being the value of a particular coordinate. Then, we perform classification by finding the hyper-plane/line that differentiates the two classes. A **hyper-plane** is the higher dimensional version of a line. As a line can separate classes on two-dimensional data, a hyper-plane can separate on higher dimensions.

Image from: https://www.kdnuggets.com/2016/07/support-vector-machines-simple-explanation.html

We can see that the line separates the two classes (green and blue) very well. These data points are called **support vectors**. Altering these points can alter the line. The black line is the **support vector machine** here.

When predicting classes, an observation is plotted, and if it falls on the left side of the line, it is classified as the green class, if it falls on the right side of the line, it is classified as the blue class.

**Kernels** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#kernels)

The example we saw above is too simple. Usually, classes are not easy to separate. Sometimes we cannot separate data easily by a line or a hyperplane. Therefore, SVMs are found by converting data into higher dimensions and finding hyper-planes on the higher dimensional data. This is known as the **kernel trick**. **Kernels** are functions that take low dimensional data and transform it into a higher dimensional space, i.e., they convert an inseparable problem to a separable problem. Simply put, they do some extremely complex data transformations, then find out the process to separate the data based on the labels or outputs we have defined.

Image from https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

In the example above, the data is two-dimensional (x and y). The points are plotted using the x and y values. We cannot draw a line to separate the two classes. Therefore, another dimension/feature is introduced. We call it z. It is defined as z=x2+y2z = x^2 + y^2z=x​2​​+y​2​​. This function that computes z is the kernel in this case. Now plotting the x and z values would give us:

Image from https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

We can easily draw a line to separate the two classes. In this way, using a *kernel*, we transformed 2-dimensional data to 3-dimensional data and separated the classes.

**SVM in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/YQl7rOMGlpY#svm-in-python)

SVMs are available in sklearn.svm module. This module has many different SVMs based on different kernels. We will be using the SVC model. SVC stands for Support Vector Classification. We will be using the [Audit Risk Dataset](https://www.kaggle.com/sid321axn/audit-data) of different firms. We will be performing the binary classification task of predicting whether a company is fraudulent or not.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





We import SVC in **line 2**. After we read the data in **line 6**, we separate our target variable as Y. To split the data into training and test sets, we use the function train\_test\_split. We provide our inputs X and the labels Y to the function in **line 12**. We also provide the test set size as test\_size. 0.20.20.2 implies that 20% data will be included in the testing set, while the remaining 80% will form the training set. The function outputs 444 items that we store in X\_train, X\_test, Y\_train, Y\_test.

In **line 15**, we initialize our SVM classifier object just like we did in the last lesson. We call SVC without any arguments. Then in the next line, we call the fit function of the model. We provide the training examples and labels to the function. After this, we need to evaluate our model. Therefore, we use the predict function of the model in **line 19** to store predictions in preds. We give the testing inputs X\_test to predict as an argument. We use accuracy\_score function to measure the accuracy of the predictions. We print the accuracy with the classification report, which we obtained by using the classification\_report function, in the last two lines.

We see that the model is approximately 98% accurate which is a greater prediction accuracy.

In the next lesson, we will look at an *ensemble* machine learning method which is known as *boosting*.

**Ensembles: Bagging vs Boosting**

This lesson will focus on how to use boosting and bagging machine learning algorithms in Python.

We'll cover the following

* + [Ensembles](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#ensembles)
  + [Bagging](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#bagging)
  + [Boosting](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#boosting)
  + [Bagging classifier in Python](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#bagging-classifier-in-python)
  + [AdaBoost classifier in Python](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#adaboost-classifier-in-python)
  + [Gradient Boosting classifier in Python](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#gradient-boosting-classifier-in-python)

**Ensembles** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#ensembles)

Recall that in the lesson [Random Forests](https://www.educative.io/collection/page/10370001/4733468011397120/5178006093955072/), we learned that an **ensemble** is a collection of different predictive models that collectively decide the predicted output. Ensemble methods are divided into two categories:

* Bagging
* Boosting

**Bagging** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#bagging)

In **bagging**, each individual model randomly samples from the training data with replacement. This means each model is different.

Note that we do not train individual models on random subsets of the data; rather, they are trained on the whole data set, but each training example is randomly sampled with replacement. For instance if our training data has 6 numbers such as [1,2,3,4,5,6] and we sample 6 times with replacement, we might get [1,2,2,4,5,5]. Therefore each individual model is different.

In Bagging, the result is obtained by averaging the responses of the N models or majority vote.

**Boosting** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#boosting)

In **boosting**, each individual model samples from the data, but this time, the sampling is not random, rather it is weighted. This means that in contrast to bagging, where each example in the training set has an equal chance of being chosen for training, in boosting, different samples have different chances of being chosen.

Therefore, individual models are trained on different training sets, and hence, different models are obtained.

In Boosting algorithms, each model is trained on data, considering the previous model’s success. After each training step, the weights are redistributed. Misclassified data increases its weights to emphasize the most difficult examples. In this way, subsequent models will focus on them during their training.

Once all of the models give their output, the second stage is to decide the final output. the final output is again based on weights. Boosting assigns a second set of weights, this time for the N models, in order to take a weighted average of their predictions. In the Boosting training stage, the algorithm allocates weights to each resulting model. A model with good classification results on the training data will be assigned a higher weight than a model that performs poorly.

Two common boosting algorithms are:

* Gradient Boosting
* AdaBoost

These differ in their loss functions. AdaBoost can be considered a special case of Gradient Boosting with *exponential loss*. Whereas Gradient Boosting is more flexible as it uses the entropy loss we used with logistic regression. We will use both of these later in this lesson.

**Bagging classifier in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#bagging-classifier-in-python)

We can easily use bagging methods for our predictions in Python. The model is available in sklearn.ensemble as BaggingClassifier. We will be using the [Default of Credit Card Clients](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset) dataset to make our predictions.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





We import BaggingClassifier in **line 2**. After we read the data in **line 6**, we separate our target variable as Y. To split the data into training and test sets, we use the function train\_test\_split. We provide our inputs X and the labels Y to the function in **line 12**. We also provide the test set size as test\_size. 0.20.20.2 implies that 20% data will be included in the testing set, while the remaining 80% will form the training set. The function outputs 444 items that we store in X\_train, X\_test, Y\_train, Y\_test.

In **line 15**, we initialize our Bagging classifier object just like we did in the last lesson. We call BaggingClassifier with n\_estimators=30, which means 30 individual models will be used. Then in the next line, we call the fit function of the model. We provide the training examples and labels to the function. After this, we need to evaluate our model. Therefore, we use the predict function of the model in **line 19** to store predictions in preds. We give the testing inputs X\_test to predict as an argument. We use accuracy\_score function to measure the accuracy of the predictions. We print the accuracy with the classification report, which we obtained by using the classification\_report function, in the last two lines.

We see that the model is approximately 81% accurate and the F1-score is 0.790.790.79 which is considered decent performance.

**AdaBoost classifier in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#adaboost-classifier-in-python)

The model is available in sklearn.ensemble as AdaBoostClassifier. We will be using the [Default of Credit Card Clients](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset) dataset to make our predictions.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24





The code is the same as the one we used for bagging classifier. The only differences are in **line 2**, where we import AdaBoostClassifier and in **line 15**, where we call AdaBoostClassifier with n\_estimators = 30.

The performance is almost the same as the bagging classifier.

**Gradient Boosting classifier in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/mE7w3AZ6vO3#gradient-boosting-classifier-in-python)

The model is available in sklearn.ensemble as GradientBoostingClassifier. We will be using the [Default of Credit Card Clients](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset) dataset to make our predictions.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23





The code is the same as the one we used for the bagging classifier. The only differences are in **line 2**, where we import GradientBoostingClassifier and in **line 15**, where we call GradientBoostingClassifier with n\_estimators = 30.

The performance is almost the same as the bagging and adaboost classifier.

A lot of other ensemble methods are also available in sklearn. Refer to the [documentation](https://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble) to look at all of these methods.

So, until now, we have only seen supervised machine learning algorithms. In the next lesson, we will see some examples of unsupervised learning.

**Clustering for Unsupervised Learning**

This lesson will introduce clustering techniques in Python for unsupervised machine learning.

We'll cover the following

* + [Unsupervised learning](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#unsupervised-learning)
  + [Clustering](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#clustering)
    - [Applications of Clustering](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#applications-of-clustering)
    - [Common clustering algorithms](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#common-clustering-algorithms)
  + [K-means Clustering](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#k-means-clustering)
    - [How k-means works?](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#how-k-means-works)

**Unsupervised learning** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#unsupervised-learning)

**Unsupervised learning** is when we use unlabeled data to allow a model to learn relationships between data observations and pick up on underlying patterns. Most data in the world is unlabeled, which makes unsupervised learning a very useful method of machine learning. The most common algorithms for unsupervised learning are *clustering* algorithms. We will look at some of these later in this lesson.

**Clustering** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#clustering)

**Clustering** is the process of dividing the data points into groups such that the data points in a group are similar to each other, but they are dissimilar to other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

Data points

Data points divided into clusters

In the above illustration, we can see the data was divided into three clusters based on the similarity of the data points.

**Applications of Clustering** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#applications-of-clustering)

Clustering is widely used in different fields for different purposes such as in:

* **Marketing**: for discovering customer segments
* **Insurance**: categorizing potential customers or differentiating between potential fraudulent customers and safe customers
* **Biology**: for grouping together different species of animals or plants
* **Libraries:** for placing books on similar topics together
* **City Planning**: for grouping housing areas

**Common clustering algorithms** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#common-clustering-algorithms)

Some of the common clustering algorithms are:

* K-means clustering
* Agglomerative hierarchical clustering
* Mean shift Clustering
* Density-based spectral

We will only look at K-means clustering algorithm.

**K-means Clustering** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#k-means-clustering)

The objective of a clustering algorithm is to group similar points and find underlying patterns in the data. K-means finds a fixed number (**k**) of clusters.

To understand this algorithm, we need to understand what a *centroid* is. A **centroid** is the location of the center of the cluster. Every data point is allocated to its nearest cluster. The nearest cluster is found by calculating the distance to the centroids of all clusters and then choosing the cluster whose centroid was the closest.

The centroid is found by averaging the data points in the cluster. That is why there is the term means in the name of the algorithm.

**How k-means works?** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/BnRB3vn0rGk#how-k-means-works)

The algorithm is as follows:

1. Choose k random centroids. Each centroid forms a cluster.
2. Assign every point to its closest cluster.
3. Compute the new centroid of all clusters.
4. Repeat steps 2 and 3 until the specified number of iterations has been reached or the centroids have stopped changing.

In the next lesson, we will implement K-Means in Python.

**K-Means on Two-Dimensional Data**

This lesson will focus on K-Means on two-dimensional data in Python.

We'll cover the following

* + - [K-means in Python](https://www.educative.io/courses/data-science-for-non-programmers/B8EomEn3YB2#k-means-in-python)

**K-means in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/B8EomEn3YB2#k-means-in-python)

We do not need to code the above algorithm because it is available in sklearn.cluster. We will be clustering on a dummy dataset first. The dummy dataset has three columns, feature\_1, feature\_2, and label. The dataset has 4 classes, which mean each row of the data set can have a label from 0, 1, 2, or 3. First, we will plot a scatter plot of the two features.

dummy.csv

1

2

3

4

5

6

7

8

9





We drop the label column in **line 6** because in reality, we do not have labels for unsupervised learning. We plot a scatter plot between the two columns in **line 9**. By looking at the plot we observe that this data can be categorized into clusters.

Let’s make these clusters below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23





We import the KMeans class in **line 3**. Then we read and set the data in **lines 5-7**. We initialize the KMeans class with n\_clusters = 4 in **line 10** which means we want 4 clusters to be made. Then we use the fit function to make our cluster centers in **line 11**. To predict the clusters for our data points we use the function predict in **line 14** and store the result in preds. We obtain the cluster centers in **line 15**. We can see from the output of **line 16** that preds is a list which contains the cluster number to which each individual data point belongs to.

To visualize our results, we plot the predictions in **line 20**. We use the plt.scatter function. We color the data points by the cluster number to which they belong to. For this, we give the predicted cluster numbers preds as the color by giving the argument c=preds. All of the points in a cluster are colored with the same color. We then again use plt.scatter to plot the cluster centers in **line 21**. centers is a two-dimensional list which has all the x-coordinates and the y-coordinates of the cluster centers. To give the x-coordinates we write centers[:,0], which means the values in all rows and the first column. In the same way, we provide the y-coordinates as centers[:,1]. We color the centers black, and we specify the size in *points*, aka *pt*, of the cluster centers as s = 300.

From the plot, we can see that the data points have been divided into 4 clusters. Note that we did not use the original labels anywhere for clustering. This is unsupervised learning!

The above example of dummy data was a very simple one. The dataset was two dimensional, i.e., it had only two features. But we have been using datasets that have many features. So, how do we cluster for n-dimensions? Also, how do we visualize n-dimensional data? We will look at that in the next lesson.

**K-Means on n-Dimensional Data**

This lesson will focus on K-Means on n-dimensional data in Python.

We'll cover the following

* + - [K-means on n-dimensional data](https://www.educative.io/courses/data-science-for-non-programmers/JY7p0go4lyK#k-means-on-n-dimensional-data)

**K-means on n-dimensional data** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/JY7p0go4lyK#k-means-on-n-dimensional-data)

For clustering n-dimensional data, we can use the same procedure. The only difference is that we first need to reduce the number of dimensions to 2. For reducing dimensions, we will use a technique called **Principal Component Analysis (PCA)**. We do not need to go into the details of how this technique works since the details are out of the scope of this course. We can reduce data to 2 dimensions by using the PCA class available in sklearn.decomposition. We will cluster the reduced data.

We will be using the [Default of Credit Card Dataset](https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset). This dataset originally had two classes. Naturally, if we do not have the class labels, we would make two clusters for this example.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19





We import the PCA class from sklearn.decomposition in **line 3**. After reading the data in **line 6**, we separate the labels in Y in **line 7**. We drop the categorical columns in the next line. We initialize the PCA class in **line 12** with n\_components=2, which means we need to reduce dimensions to 2. The PCA class has the function fit\_transform, which gives us the transformed data. We use fit\_transform in **line 13** and store the reduced data in X. At this time, the returned data is not a dataframe, rather it is a 2-dimensional list. So we convert it into a dataframe in **line 14** by giving the list to pd.DataFrame. We name the columns feature\_1 and feature\_2 in **line 15**, so they are easy to access later. From the outputs of **lines 9 and 16**, we can see that the data has been reduced to 2 columns.

In the end, we just plot the reduced data as we did for the above simple example.

Now we have this data that we need to cluster. Just by looking at the data, we cannot easily separate the data into two clusters. Let’s see how K-means clustering does that for us below.

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31





In **line 21**, we initialize the KMeans class object with n\_clusters = 2, which means that we want 2 clusters. Then we use the function fit and give it the reduced data X. We obtain the predicted classes by using the predict function in **line 25**. In **line 26**, we retrieve the cluster centers. We plot the data and color the data by the predicted classes in **line 29**. We plot the cluster centers in **line 30**.

From the produced plot, we can see that the data has been divided into two clusters. We can see a boundary between the two clusters.

To compare the results of unsupervised learning to supervised learning, we can compare the predicted clusters with the actual labels that we have. For that, we need to set the class label no to 0 and yes to 1 since clustering predicted classes as 0 and 1. We do that in **lines 33 and 34**. Then we print the accuracy with the predicted labels and actual labels. The accuracy comes out to be approximately 70%, which is decent because we got approximately 75-80% accuracy with supervised learning models on this dataset.

This concludes the lesson on unsupervised learning. In the next lesson, you will be tested on the concepts that you have learned in this lesson.

**Test your Knowledge**

This lesson has a quiz that tests the learners on the concepts learned in this lesson.

We'll cover the following

* + [Quiz on Machine Learning in Python](https://www.educative.io/courses/data-science-for-non-programmers/3YRQmqQWW9R#quiz-on-machine-learning-in-python)

**Quiz on Machine Learning in Python** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/3YRQmqQWW9R#quiz-on-machine-learning-in-python)

1

Artificial Intelligence is a sub domain of Machine Learning.

A)

True

B)

False

2

Decision Trees capture non linear relationships between variables.

A)

True

B)

False

3

Linear Regression models can NOT capture non linear relationships.

A)

True

B)

False

4

Out of the following algorithms:

* Decision Trees
* Support Vector Machines

Which performs better?

A)

Decision Trees

B)

Support Vector Machines

C)

Depends on the problem and the dataset

5

Random Forest is a boosting algorithm.

A)

True

B)

False

6

In bagging, individual models train on data that is sampled \_\_\_\_\_.

A)

without replacement

B)

with replacement

7

Which of the following algorithms can be used for unsupervised learning? Check all answer that you think are correct.

A)

SVMs

B)

KMeans

C)

Mean Shift

D)

Random Forests

E)

AdaBoost

8

PCA is used for

A)

clustering

B)

dimensionality reduction

C)

none of these

9

km = KMeans(n\_clusters = 2)  
km.fit(data)  
result = km.predict(data)

In the above code, what is being stored in result?

A)

The cluster centers

B)

The cluster numbers to which each observation in data belongs to

C)

None of these

10

Clustering can NOT be used to segment customer groups.

A)

True

B)

False

**Conclusion**

This lesson gives an ending note.

We'll cover the following

* + [Where to go from here?](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#where-to-go-from-here)
    - [Resources for datasets](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#resources-for-datasets)
    - [Python resources](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#python-resources)
    - [Machine Learning resources](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#machine-learning-resources)

Congratulations! You have completed this introductory course on Data Science. You are now familiar with the Data Science lifecycle and are ready to enter into the world of Data Scientists.

**Where to go from here?** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#where-to-go-from-here)

Since you have now completed the introductory course, you need to start putting this knowledge to use and go towards learning advanced concepts. If you are a student, then you can find interesting datasets and apply the learned techniques on them to build your profile. If you are a professional, you can start applying the learned techniques in your day to day work.

Some advanced concepts and tools that are becoming essential for data scientists these days are:

* SQL
* Deep Learning in Python
* Matplotlib and Seaborn

**Resources for datasets** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#resources-for-datasets)

The following are good resources for finding datasets:

* [Kaggle](http://www.kaggle.com)
* [UCI Machine Learning Repository](https://archive.ics.uci.edu/ml/index.php)
* [Google Dataset Search Engine](https://toolbox.google.com/datasetsearch)
* Public datasets from Governments and International Organizations such as WHO, US Census Bureau, etc.

**Python resources** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#python-resources)

* [Educative’s Python courses](https://www.educative.io/grow-my-skillset/python)
* [Python’s Documentation](https://docs.python.org/3/)
* [Towards Data Science](https://towardsdatascience.com/)

**Machine Learning resources** [**#**](https://www.educative.io/courses/data-science-for-non-programmers/m7ownylxr8O#machine-learning-resources)

* [AdaptiLab’s Machine Learning Track on Educative](https://www.educative.io/track/become-ml-engineer)
* [Medium](https://medium.com/topic/data-science)
* [Towards Data Science](https://towardsdatascience.com/)

Thank you for sticking with us until the end. We hope this course met your expectations, and you had fun learning. We hope to see you soon in another course.

Did you complete this lesson?