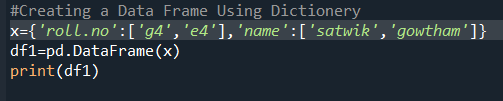
**Pandas**

There are two core objects in pandas: the **DataFrame** and the **Series**.

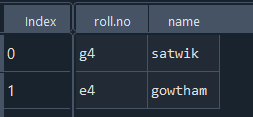
### **DataFrame**

A DataFrame is a table. It contains an array of individual entries, each of which has a certain value. Each entry corresponds to a row (or record) and a column.

Creating a Dataframe Using a Dictionery:



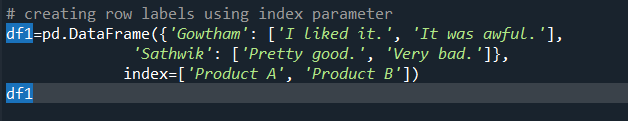
Output:

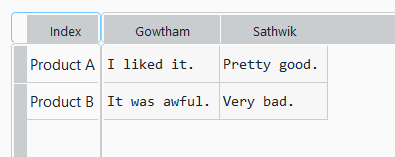


The list of row labels used in a DataFrame is known as an **Index**. By default row labels are taken from (0,1,2,3,4,5…..).

We can assign values to it by using an index parameter in our constructor:

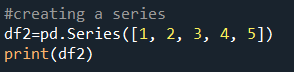
After using index parameter to name row labels:

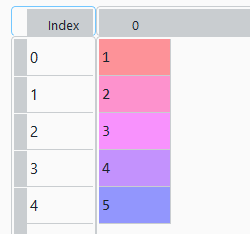




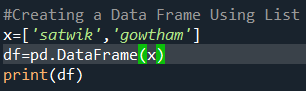
Series

A Series, by contrast, is a sequence of data values. If a DataFrame is a table, a Series is a list. And in fact you can create one with nothing more than a list:



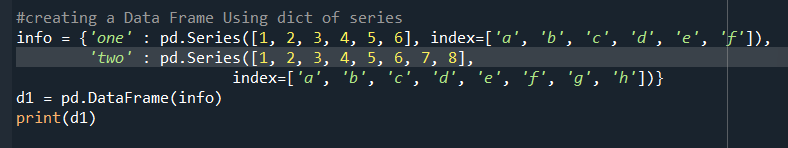


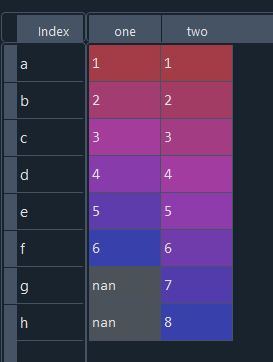
Creating a data frame using a list





creating a Data Frame Using dict of series:





Reading data files

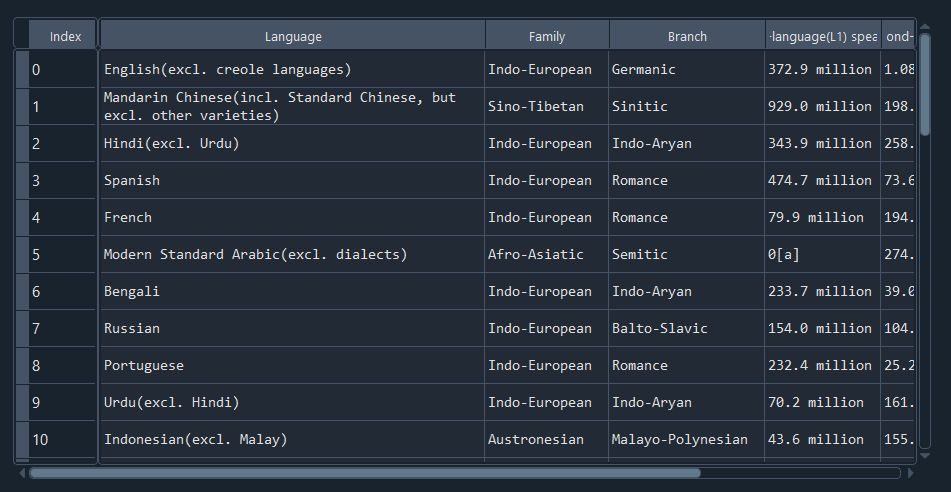
Being able to create a DataFrame or Series by hand is handy. But, most of the time, we won't actually be creating our own data by hand. Instead, we'll be working with data that already exists.

Data can be stored in any of a number of different forms and formats. By far the most basic of these is the humble CSV file. When you open a CSV file you get something that looks like this:

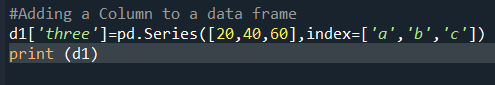
We'll use the pd.read\_csv() function to read the data into a DataFrame.

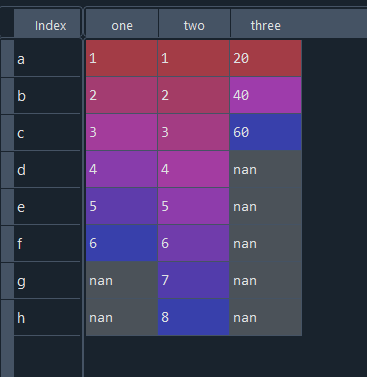


We can examine the contents of the resultant DataFrame using the head() command, which grabs the first five rows:



**Adding a Column to a data frame:**



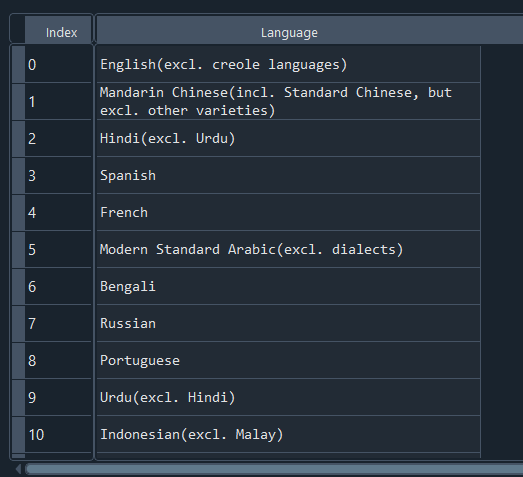


Native Accessors:

Accessing values of a column:

1) indexing operator





Accesing single data entry:





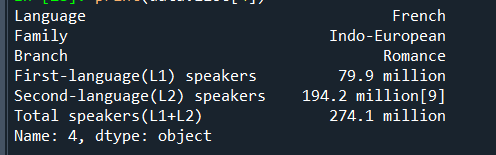
Accessors in pandas:

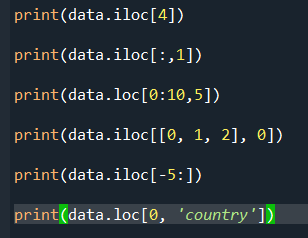
Indexing in Pandas:

The first is **index-based selection**: selecting data based on its numerical position in the data. iloc follows this paradigm.

To select the first row of data in a DataFrame, we use the following:







### **Label-based selection**

The second paradigm for attribute selection is the one followed by the loc operator: **label-based selection**. In this paradigm, it's the data index value, not its position, which matters.

iloc is conceptually simpler than loc because it ignores the dataset's indices. When we use iloc we treat the dataset like a big matrix (a list of lists), one that we have to index into by position. loc, by contrast, uses the information in the indices to do its work. Since your dataset usually has meaningful indices, it's usually easier to do things using loc instead.

# Manipulating the index

Label-based selection derives its power from the labels in the index. Critically, the index we use is not immutable. We can manipulate the index in any way we see fit.

The set\_index() method can be used to do the job. Here is what happens when we set\_index to the title field:



This is useful if you can come up with an index for the dataset which is better than the current one.

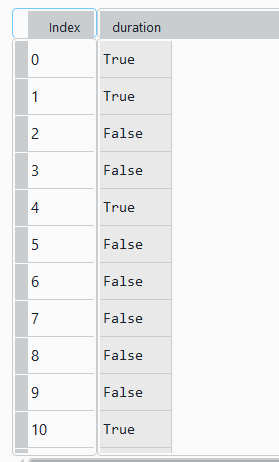
# Conditional selection

So far we've been indexing various strides of data, using structural properties of the DataFrame itself. To do interesting things with the data, however, we often need to ask questions based on conditions.

For example, suppose that we're interested specifically in song duration less than 5 min:

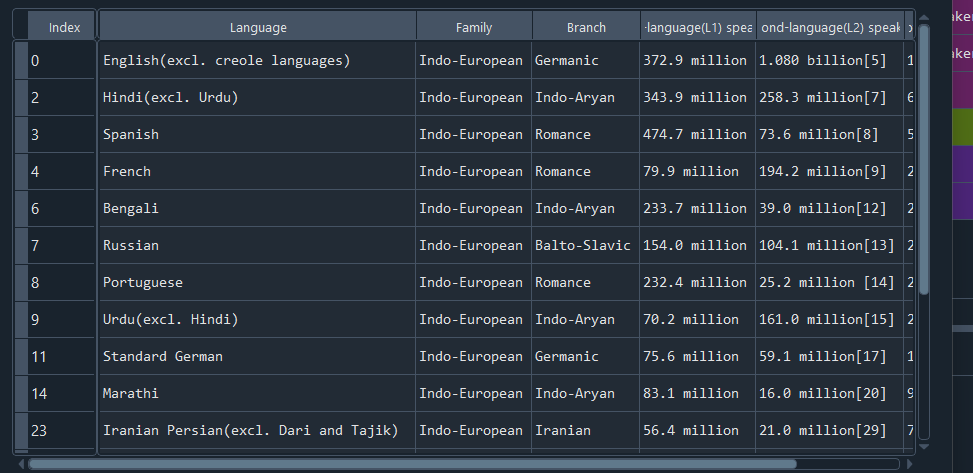
We can start by checking if each song is less than 3 mins:





This operation produced a Series of True/False booleans based on the duration of each record. This result can then be used inside of loc to select the relevant data:

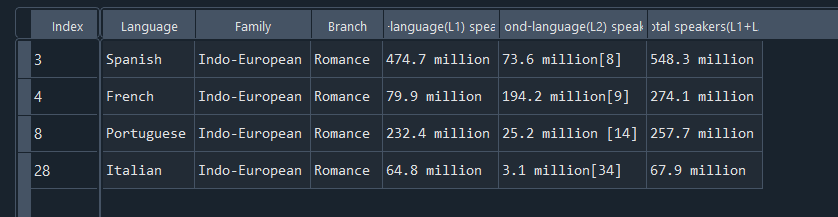




#mutiple conditions:



**Output:**



Pandas comes with a few built-in conditional selectors.

1)isin.

The first is isin. isin is lets you select data whose value "is in" a list of values.  For example, here's how we can use it to select wines only from Italy or France:





2)isnull:The second is isnull (and its companion notnull). These methods let you highlight values which are (or are not) empty (NaN). For example, to filter out wines lacking a price tag in the dataset, here's what we would do:

reviews.loc[reviews.price.notnull()]

# Summary functions

Pandas provides many simple "summary functions" (not an official name) which restructure the data in some useful way. For example, consider the describe() method:

 It is type-aware, meaning that its output changes based on the data type of the input.

1)Describe function:

Here id is

2) Mean function:





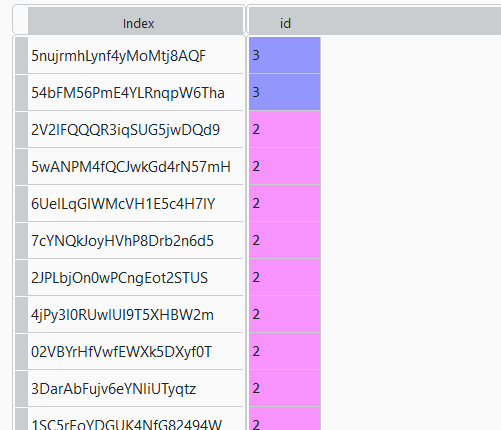
2)unique function:





3)value\_counts()





# Maps

A **map** is a term, borrowed from mathematics, for a function that takes one set of values and "maps" them to another set of values. In data science we often have a need for creating new representations from existing data, or for transforming data from the format it is in now to the format that we want it to be in later.

There are two mapping methods that you will use often.

1)Map





The function you pass to map() should expect a single value from the Series (a point value, in the above example), and return a transformed version of that value. map() returns a new Series where all the values have been transformed by your function.

2)Apply

[apply()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.apply.html) is the equivalent method if we want to transform a whole DataFrame by calling a custom method on each row.

Grouping