Course Code: CA LAB-XI (B) LAB on AI Practice using Python

**Ass 2 Data Preparation using techniques like Data Filtration.**

Filtering data from a data frame is one of the most common operations when cleaning the data.

|  |
| --- |
|  |
|  | Import pandas as pd |
|  | # information about employees |
|  | id\_number = ['128', '478', '257', '299', '175', '328', '099', '457', '144', '222'] |
|  | name = ['Patrick', 'Amanda', 'Antonella', 'Eduard', 'John', 'Alejandra', 'Layton', 'Melanie', 'David', 'Lewis'] |
|  | surname = ['Miller', 'Torres', 'Brown', 'Iglesias', 'Wright', 'Campos', 'Platt', 'Cavill', 'Lange', 'Bellow'] |
|  | division = ['Sales', 'IT', 'IT', 'Sales', 'Marketing', 'Engineering', 'Engineering', 'Sales', 'Engineering', 'Sales'] |
|  | salary = [30000, 54000, 80000, 79000, 15000, 18000, 30000, 35000, 45000, 30500] |
|  | telephone = ['7366578', '7366444', '7366120', '7366574', '7366113', '7366117', '7366777', '7366579', '7366441', '7366440'] |
|  | type\_contract = ['permanent', 'temporary', 'temporary', 'permanent', 'internship', 'internship', 'permanent', 'temporary', 'permanent', 'permanent'] |
|  |  |
|  | # data frame containing information about employees |
|  | df\_employees = pd.DataFrame({'name': name, 'surname': surname, 'division': division, |
|  | 'salary': salary, 'telephone': telephone, 'type\_contract': type\_contract}, index=id\_number) |
|  | Input: |
|  | print(df\_employees) |

**Output:**

**name surname division salary telephone type\_contract**

**128 Patrick Miller Sales 30000 7366578 permanent**

**478 Amanda Torres IT 54000 7366444 temporary**

**257 Antonella Brown IT 80000 7366120 temporary**

**299 Eduard Iglesias Sales 79000 7366574 permanent**

**175 John Wright Marketing 15000 7366113 internship**

**328 Alejandra Campos Engineering 18000 7366117 internship**

**099 Layton Platt Engineering 30000 7366777 permanent**

**457 Melanie Cavill Sales 35000 7366579 temporary**

**144 David Lange Engineering 45000 7366441 permanent**

**222 Lewis Bellow Sales 30500 7366440 permanent**

**1 Selecting a single column by label**

df[string]

|  |  |  |
| --- | --- | --- |
| salary = df\_employees.salary | | |
|  |  |
| # select the column (salary) using square brackets |  |
| salary\_2 = df\_employees['salary'] |  |
|  |  |
| # we obtain a Series object, when a single column is selected |  |
| print(type(salary)) |  |
| # <class 'pandas.core.series.Series'> |  |
|  |  |
| print(type(salary\_2)) |  |
| # <class 'pandas.core.series.Series'> |  |
|  |  |
| print(salary)  output  128 30000  478 54000  257 80000  299 79000  175 15000  328 18000  099 30000  457 35000  144 45000  222 30500  Name: salary, dtype: int64 |  |
|  |  |

# 2. Selecting multiple columns by label

df[list\_of\_strings]

print(df\_employees[['division', 'salary']])

**output**

**division salary**

**128 Sales 30000**

**478 IT 54000**

**257 IT 80000**

**299 Sales 79000**

**175 Marketing 15000**

**328 Engineering 18000**

**099 Engineering 30000**

**457 Sales 35000**

**144 Engineering 45000**

**222 Sales 30500**

# 3. Selecting columns by data type

df.select\_dtypes(include=None, exclude=None)

|  |
| --- |
| # select numeric columns - numpy object |
| numeric\_inputs = df\_employees.select\_dtypes(include=np.number) |  |
| OR  # select numeric columns - string |  |
| numeric\_inputs\_2 = df\_employees.select\_dtypes(include='number') |  |
|  |  |
| # check selected columns with the .columns attribute |  |
| print(numeric\_inputs\_2.columns) |  |
| # Index(['salary'], dtype='object') |  |
|  |  |
| # the method returns a DataFrame object |  |
| print(type(numeric\_inputs\_2)) |  |
| # <class 'pandas.core.frame.DataFrame'> |  |
|  |  |
| # visualize the data frame |  |
| print(numeric\_inputs)  output  30000  478 54000  257 80000  299 79000  175 15000  328 18000  099 30000  457 35000  144 45000  222 30500   |  | | --- | | # concise summary of the data frame, including the column names and their data types | | print(df\_employees.info())  output:  Index: 10 entries, 128 to 222  Data columns (total 6 columns):  # Column Non-Null Count Dtype  --- ------ -------------- -----  0 name 10 non-null object  1 surname 10 non-null object  2 division 10 non-null object  3 salary 10 non-null int64  4 telephone 10 non-null object  5 type\_contract 10 non-null object  dtypes: int64(1), object(5)  memory usage: 560.0+ bytes  None  print(df\_employees.dtypes)  Output  name object  surname object  division object  salary int64  telephone object  type\_contract object  dtype: objectmemory usage: 560.0+ bytes  None 4. Selecting a single row by label **DataFrames** and **Series** do not necessarily have numerical indexes. By default, the index is an integer indicating the row position; however, it can also be an alphanumeric string. In our current example, the index is the id number of the employee.  To select a single row by id number, we can use the **.loc[] indexer** providing as input a **single string** (index name).  Selecting a single row by label  → df.loc[string]  df\_employees.loc['478']  The code below shows how to select the employee with id number 478.    As shown above, when a single row is selected, the**.loc[] indexer** returns a **Series object**. However, we can also obtain a single-row **DataFrame** by passing a single-element list to the **.loc[]** method as follows.   5. Selecting multiple rows by label We can select **multiple rows** with the**.loc[] indexer**. Besides a single label, the indexer also accepts as input**a list** or **a slice** of labels.  Selecting multiple rows by label  → df.loc[list\_of\_strings]  → df.loc[slice\_of\_strings]  Next, we obtain a subset of our data frame containing the employees with id number 478 and 222 as follows.    Notice that, the end index of **.loc[] method**is always included, meaning the selection includes the last label. 6. Selecting a single row by position The **.iloc[] indexer** is used to index a data frame by position. To select a single row with the .iloc[] attribute, we pass in the row position (a single integer) to the indexer.  Selecting a single row by position  → df.iloc[integer]  In the following block of code, we select the row with index 0. In this case, the first row of the DataFrame is returned because in Pandas indexing starts at 0.    Additionally, the**.iloc[] indexer** also supports **negative integers** (starting at -1) as relative positions to the end of the data frame.    As shown above, when a single row is selected, the **.iloc[] indexer**returns a **Series** object that has the column names as indexes. However, as we did with the **.loc[] indexer**, we can also obtain a **DataFrame** by passing a single-integer list to the indexer in the following way.    Lastly, keep in mind that an **IndexError** is raised when trying to access an index that is out-of-bounds. 7. Selecting multiple rows by position To extract multiple rows by position, we pass either a list or a slice object to the **.iloc[]** indexer.  Selecting multiple rows by position  → df.iloc[list\_of\_integers]  → df.iloc[slice\_of\_integers]  The following block of code shows how to select the first five rows of the data frame using a list of integers.    Alternatively, we can obtain the same results using slice notation.    As shown above**,**Python slicing rules (half-open interval) apply to the **.iloc[] attribute**, meaning the first index is included, but not the end index. 8. Selecting rows and columns simultaneously So far, we have learnt how to select rows in a data frame by label or position using the **.loc[]**and**.iloc[] indexers**. However, both indexers are not only capable of selecting rows, but also rows and columns simultaneously.  To do so, we have to provide the row and column labels/positions separated by a comma as follows:  Selecting rows and columns simultaneously  → df.loc[row\_labels, column\_labels]  → df.iloc[row\_positions, column\_positions]  where **row\_labels** and **column\_labels** can be a single string, a list of strings, or a slice of strings. Likewise, **row\_positions** and **column\_positions** can be a single integer, a list of integers, or a slice of integers.  The following examples show how to extract rows and columns at once using the **.loc[]** and **.iloc[]**indexers.   * **Selecting a scalar value**   We select the salary of the employee with the id number 478 by position and label in the following manner.  In this case, the output of both indexers is an integer.   * **Selecting a single row and multiple columns**   We select the name, surname, and salary of the employee with id number 478 by passing a single value as the first argument and a list of values as the second argument, obtaining as a result a Series object.     * **Selecting disjointed rows and columns**   To select multiple rows and columns, we need to pass two list of values to both indexers. The code below shows how to extract the name, surname, and salary of employees with id number 478 and 222.    Unlike before, the output of both indexers is a DataFrame object.   * **Selecting continuous rows and columns**   We can extract continuous rows and columns of the data frame by using slice notation. The following code snippet shows how to select the name, surname, and salary of employees with id number 128, 478, 257, and 299.    As shown above, we only employ slice notation to extract the rows of the data frame since the id numbers we want to select are continuous (indexes from 0 to 3).  It is important to remember that the **.loc[] indexer** uses a **closed interval**, extracting both the start label and the stop label. On the contrary, the **.iloc[] indexer**employs a **half-open interval**, so the value at the stop index is not included. ****9. Selecting a scalar value using the .at[] and .iat[]****indexers As mentioned above, we can select a scalar value by passing two strings/integers separated by a comma to the **.loc[]** and**.iloc[]** **indexers.**Additionally, Pandas provides two optimized functions to extract a scalar value from a data frame object: the **.at[]** and **.iat[] operators**. The former extracts a single value by label, while the latter access a single value by position.  Selecting a scalar value by label and position  → df.at[string, string]  → df.iat[integer, integer]  The code below shows how to select the salary of the employee with the id number 478 by label and position with the**.at[]**and **.iat[] indexers**.  We can use the **%timeit magic function** to calculate the execution time of both Python statements. As shown below, the **.at[]**and **.iat[] operators**are much faster than the **.loc[]** and **.iloc[] indexers**.      Lastly, it is important to remember that the **.at[]** and **.iat[]** indexers can only be used to access a single value, raising a type error when trying to select multiple elements of the data frame. 10. Selecting rows using Boolean selection So far, we have filtered rows and columns in a data frame by label and position. Alternatively, we can also select a subset in Pandas with boolean indexing. Boolean selection consists of selecting rows of a data frame by providing a boolean value (True or False) for each row.  In most cases, this array of booleans is calculated by applying to the values of a single or multiple columns a condition that evaluates to True or False, depending on whether or not the values meet the condition. However, it is also possible to manually create an array of booleans using among other sequences, Numpy arrays, lists, or Pandas Series.  Then, the sequence of booleans is placed inside square brackets [], returning the rows associated with a True value.  Selecting rows using Boolean selection  → df[sequence\_of\_booleans] Boolean selection according to the values of a single column The most common way to filter a data frame according to the values of a single column is by using a comparison operator.  A comparison operator evaluates the relationship between two operands (a and b) and returns True or False depending on whether or not the condition is met. The following table contains the comparison operators available in Python.    These comparison operators can be used on a single column of the data frame to obtain a sequence of booleans. For instance, we determine whether the salary of the employee is greater than 45000 euros by using the greater than operator as follows.    The output is a Series of booleans where salaries higher than 45000 are True and those less than or equal to 45000 are False. As you may notice, the Series of booleans has the same indexes (id number) as the original data frame.  This Series can be passed to the indexing operator [] to return only the rows where the result is True.    As shown above, we obtain a data frame object containing only the employees with a salary higher than 45000 euros. Boolean selection according to the values of multiple columns Previously, we have filtered a data frame according to a single condition. However, we can also combine multiple boolean expression together using logical operators. In Python, there are three logical operators: and, or, and not. However, these keywords are not available in Pandas for combining multiple boolean conditions. Instead, the following operators are used.    The code below shows how to select employees with a salary greater than 45000 and a permanent contract combining two boolean expressions with the logical operator &.    As you may know, in Python, the comparison operators have a higher precedence than the logical operators. However, it does not apply to Pandas where logical operators have higher precedence than comparison operators. Therefore, we need to wrap each boolean expression in parenthesis to avoid an error. Boolean selection using Pandas methods Pandas provides a wide range of built-in functions that return a sequence of booleans, being an appealing alternative to more complex boolean expressions that combine comparison and logical operators.   * **The isin method**   The **[pandas.Series.isin](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.isin.html" \t "_blank)**method takes a sequence of values and returns True at the positions within the Series that match the values in the list.  This method allows us to check for the presence of one or more elements within a column without using the logical operator or. The code below shows how to select employees with a permanent or temporary contract using both the logical operator or and the isin method.    As you can see, the isin method comes in handy for checking multiple or conditions in the same column. Additionally, it is faster!     * **The between method**   The**[pandas.Series.between](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.between.html" \t "_blank)**method takes two scalars separated by a comma which represent the lower and upper boundaries of a range of values and returns True at the positions that lie within that range.  The following code selects employees with a salary higher than or equal to 30000 and less than or equal to 80000 euros.    As you can observe, both boundaries (30000 and 80000) are included. To exclude them, we have to pass the argument **inclusive=False**in the following manner.    As you may noticed, the above code is equivalent to writing two boolean expressions and evaluate them using the logical operator and.   * **String methods**   Additionally, we can also use boolean indexing with string methods as long as they return a sequence of booleans.  For instance, the **[pandas.Series.str.contains](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.contains.html" \t "_blank)**method checks for the presence of a substring in all the elements of a column and returns a sequence of booleans that we can pass to the indexing operator to filter a data frame.  The code below shows how to select all telephone numbers that contain 57.    While the **contains**method evaluates whether or not a substring is contained in each element of a Series, the **[pandas.Series.str.startswith](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.startswith.html" \l "pandas.Series.str.startswith" \t "_blank)**function checks for the presence of a substring at the beginning of a string. Likewise, the **[pandas.Series.str.endswith](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.str.endswith.html" \l "pandas.Series.str.endswith" \t "_blank)**tests if a substring is present at the end of a string.  The following code shows how to select employees whose name starts with ‘A’.    https://towardsdatascience.com/filtering-data-frames-in-pandas-b570b1f834b9 |  | |  |