Toolbox

Bhaswar Chakma

2021-07-17

Contents

Chapter 1

dplyr vs pandas

We will use the five dplyr verbs (also pandas' guide is also helpful) for comparison

- `select()` picks variables based on their names.
- `mutate()` adds new variables that are functions of existing variables
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

and use the following toy data to apply the verbs.

name	gender	grade
Barney	Male	10
Ted	Male	11
Marshall	Male	13
Lilly	Female	12
Robin	Female	14

Create Toy Data

dplyr

pandas

```
df <- tibble(
  name = c("Barney", "Ted", "Marshall",
           "Lilly","Robin"),
  gender = c("Male", "Male","Male",
             "Female", "Female"),
  grade = c(10, 11, 13, 12, 14)
)
df
```

```
## # A tibble: 5 x 3
##   name     gender grade
##   <chr>    <chr>  <dbl>
## 1 Barney   Male      10
## 2 Ted      Male      11
## 3 Marshall Male      13
## 4 Lilly    Female    12
## 5 Robin    Female    14
```

```
df = pd.DataFrame({
  'name':["Barney", "Ted", "Marshall",
          "Lilly", "Robin"],
  'gender':["Male", "Male","Male",
            "Female", "Female"],
  'grade':[10, 11, 13, 12, 14]
})
df
```

```
##        name  gender  grade
## 0    Barney    Male     10
## 1       Ted    Male     11
## 2  Marshall    Male     13
## 3     Lilly  Female     12
## 4     Robin  Female     14
```

Check Data Structure

dplyr

pandas

```
glimpse(df)
```

```
## Rows: 5
## Columns: 3
## $ name   <chr> "Barney", "Ted", "Marshall", "Lilly", "Robin"
## $ gender <chr> "Male", "Male", "Male", "Female", "Female"
## $ grade  <dbl> 10, 11, 13, 12, 14
```

```
df.dtypes
```

```
## name      object
## gender    object
## grade      int64
## dtype: object
```

```
df.shape
```

```
## (5, 3)
```

```
df.info()
```

```
## <class 'pandas.core.frame.DataFrame'>
## RangeIndex: 5 entries, 0 to 4
## Data columns (total 3 columns):
## name      5 non-null object
## gender    5 non-null object
## grade     5 non-null int64
## dtypes: int64(1), object(2)
## memory usage: 248.0+ bytes
```

1.1 select()

Task: Pick the variables `name` and `grade`.

dplyr

pandas

```
df %>%
  select(name, grade)
```

```
## # A tibble: 5 x 2
##   name     grade
##   <chr>    <dbl>
## 1 Barney      10
## 2 Ted         11
## 3 Marshall    13
## 4 Lilly       12
## 5 Robin       14
```

```
df[['name', 'grade']]
```

```
##        name  grade
## 0    Barney     10
## 1       Ted     11
## 2  Marshall     13
## 3     Lilly     12
## 4     Robin     14
```

```
# or
df.drop(columns = ['grade'])
```

```
##        name  gender
## 0    Barney    Male
## 1       Ted    Male
## 2  Marshall    Male
## 3     Lilly  Female
## 4     Robin  Female
```

```
# or
df.drop(['grade'], axis = 1)
```

```
##        name  gender
## 0    Barney    Male
## 1       Ted    Male
## 2  Marshall    Male
## 3     Lilly  Female
## 4     Robin  Female
```

1.2 mutate()

Task: Generate a variable grade_p, expressing grade out of 100.

dplyr

pandas

```
df %>%
  mutate(grade_p = grade/20*100)
```

```
## # A tibble: 5 x 4
##   name     gender grade grade_p
##   <chr>    <chr>  <dbl>   <dbl>
## 1 Barney   Male      10      50
## 2 Ted      Male      11      55
## 3 Marshall Male      13      65
## 4 Lilly    Female    12      60
## 5 Robin    Female    14      70
```

```
df['grade_p'] = df['grade']/20*100
df
```

```
##        name  gender  grade  grade_p
## 0    Barney    Male     10     50.0
## 1       Ted    Male     11     55.0
## 2  Marshall    Male     13     65.0
## 3     Lilly  Female     12     60.0
## 4     Robin  Female     14     70.0
```

```
# now drop the newly created variable
df.drop(columns = 'grade_p', inplace = True)
```

1.3 filter()

Task: Keep Barney or females.

dplyr

pandas

```
df %>%
  filter(name == "Barney"|
         gender == "Female")
```

```
## # A tibble: 3 x 3
##   name   gender grade
##   <chr>  <chr>  <dbl>
## 1 Barney Male      10
## 2 Lilly  Female    12
## 3 Robin  Female    14
```

```
# similar to base R
df[(df["name"] == "Barney") |
   (df["gender"] == "Female")]
```

```
##      name  gender  grade
## 0  Barney    Male     10
## 3   Lilly  Female     12
## 4   Robin  Female     14
```

```
# query with ''; need to use "" for conditions
df.query('name == "Barney"|gender == "Female"')
```

```
##      name  gender  grade
## 0  Barney    Male     10
## 3   Lilly  Female     12
## 4   Robin  Female     14
```

```
# query with ""; need to use '' for conditions
df.query("name == 'Barney'| gender == 'Female'")
```

```
##      name  gender  grade
## 0  Barney    Male     10
## 3   Lilly  Female     12
## 4   Robin  Female     14
```

1.4 group_by() and summarize()

Task: Grouped by gender, find mean grade.

dplyr

pandas

```
df %>%
  group_by(gender) %>%
  summarize(avg_grade = mean(grade))
```

```
## # A tibble: 2 x 2
##   gender avg_grade
##   <chr>      <dbl>
## 1 Female      13
## 2 Male        11.3
```

```
# returns a series
df.groupby("gender")['grade'].mean()
```

```
## gender
## Female    13.000000
## Male      11.333333
## Name: grade, dtype: float64
```

```
# returns a data frame
df[['gender', 'grade']].groupby("gender").mean()
```

```
##             grade
## gender
## Female  13.000000
## Male    11.333333
```

Task: Grouped by gender, find mean, median, minimum, and maximum grade.

dplyr

pandas

```
df %>%
  group_by(gender) %>%
  summarize(mean = mean(grade),
            median = median(grade),
            min = min(grade),
            max = max(grade))
```

```
## # A tibble: 2 x 5
##   gender  mean median   min   max
##   <chr>  <dbl>  <dbl> <dbl> <dbl>
## 1 Female  13        13    12    14
## 2 Male    11.3      11    10    13
```

```
df.groupby("gender")['grade'].agg(
  # provide a dictionary
  {# variable name followed by function
    'mean': 'mean',
    'median': 'median',
    'min': 'min',
    'max': 'max'
  }
)
```

```
##              mean  median  min  max
## gender
## Female  13.000000      13   12   14
## Male    11.333333      11   10   13
##
## C:/Users/Bhaswar/AppData/Local/r-miniconda/envs/r-reticulate/python.exe:7: FutureWa
## is deprecated and will be removed in a future version. Use                named ag
##
##     >>> grouper.agg(name_1=func_1, name_2=func_2)
```

1.5 arrange()

Task: Arrange grade in ascending order.

dplyr

pandas

```
df %>%
  arrange(grade)
```

```
## # A tibble: 5 x 3
##   name     gender grade
##   <chr>    <chr>  <dbl>
## 1 Barney   Male      10
## 2 Ted      Male      11
## 3 Lilly    Female    12
## 4 Marshall Male      13
## 5 Robin    Female    14
```

```
df.sort_values('grade')
```

```
##        name  gender  grade
## 0    Barney    Male     10
## 1       Ted    Male     11
## 3     Lilly  Female     12
## 2  Marshall    Male     13
## 4     Robin  Female     14
```

Task: Arrange grade in ascending order.

dplyr

pandas

```
df %>% arrange(desc(grade))
```

```
## # A tibble: 5 x 3
##   name     gender grade
##   <chr>    <chr>  <dbl>
## 1 Robin    Female    14
## 2 Marshall Male      13
## 3 Lilly    Female    12
## 4 Ted      Male      11
## 5 Barney   Male      10
```

```
df.sort_values('grade', ascending = False)
```

```
##        name  gender  grade
## 4     Robin  Female     14
## 2  Marshall    Male     13
## 3     Lilly  Female     12
## 1       Ted    Male     11
## 0    Barney    Male     10
```

Task: Arrange gender in ascending order then arrange grade in descending order.

dplyr

pandas

```
df %>%
  arrange(gender, desc(grade))
```

```
## # A tibble: 5 x 3
##   name     gender grade
##   <chr>    <chr>  <dbl>
## 1 Robin    Female    14
## 2 Lilly    Female    12
## 3 Marshall Male      13
## 4 Ted      Male      11
## 5 Barney   Male      10
```

```
df.sort_values(['gender','grade'],
                ascending = [True, False])
```

```
##        name  gender  grade
## 4     Robin  Female     14
## 3     Lilly  Female     12
## 2  Marshall    Male     13
## 1       Ted    Male     11
## 0    Barney    Male     10
```

Chapter 2

Base Python

2.1 map()

map() lets you apply a function to each element of a list.

```
# toy list
toy_list = [1, 200, 3, 400]

# Create toy function
def smaller_than_100(k):
  if k < 100:
    return True
  else:
    False
# test the function
smaller_than_100(2)
```

```
## True
```

```
# apply it to toy_list
mapped = map(smaller_than_100, toy_list)
print(mapped) # doesn't provide the desired output; use loop
```

```
## <map object at 0x000000003082B5F8>
```

```
for i in mapped:
    print(i)
```

```
## True
## None
## True
## None
```

Chapter 3

R Strings

3.1 String Manupulation with Base R Functions

There are many functions in base R for basic string manipulation.

Function

Example

`nchar()`

```r
y <- c("Hello", "World", "Hello", "Universe")
nchar(y) # Returns number of characters
```

```
## [1] 5 5 5 8
```

`tolower()`

```r
tolower(y)
```

```
## [1] "hello"    "world"    "hello"    "universe"
```

`toupper()`

```r
toupper(y)
```

```
## [1] "HELLO"    "WORLD"    "HELLO"    "UNIVERSE"
```

`chartr()`

```
chartr("oe", "$#", y)#o becomes $; e becomes #
```

```
## [1] "H#ll$"    "W$rld"    "H#ll$"    "Univ#rs#"
```

`substr()`

```
x <- "1t345s?"
substr(x, 2, 6) # provides strings from 2 to 6
```

```
## [1] "t345s"
```

`strsplit()`

```
x <- "R#Rocks#!"
strsplit(x, split = "#")
```

```
## [[1]]
## [1] "R"     "Rocks" "!"
```

3.2 `stringr`

```
library(stringr)
```

Job	stringr	Base R
String concatenation	`str_c()`	`paste()`
Number of characters	`str_length()`	`nchar()`
Extracts substrings	`str_sub()`	`substr()`
Duplicates characters	`str_dup()`	
Removes leading and trailing whitespace	`str_trim()`	
Pads a string	`str_pad()`	
Wraps a string paragraph	`str_wrap()`	

3.3 Regular Expressions

A **regular expression** (or **regex**) is a set of symbols that describes a text pattern. More formally, a regular expression is a pattern that describes a set of strings.

Regular expressions are a formal language in the sense that the symbols have a defined set of rules to specify the desired patterns.

3.3.1 `stringr` Functions for Regular Expressions

Function	Job
`str_detect`(str, pattern)	Detects the presence of a pattern and returns TRUE if it is found
`str_locate`(str, pattern)	Locate the 1st position of a pattern and return a matrix with start & end.s
`str_extract`(str, pattern)	Extracts text corresponding to the first match.
`str_match`(str, pattern)	Extracts capture groups formed by () from the first match.
`str_split`(str, pattern)	Splits string into pieces and returns a list of character vectors.

Chapter 4

SQL

4.1 CREATE

The general syntax to create a table:

```sql
create table TABLENAME (
  COLUMN1 datatype,
  COLUMN2 datatype,
  COLUMN3 datatype,
  ... );
```

To create a table called `TEST` with two columns - `ID` of type integer, and `NAME` of type varchar, we could create it using the following SQL statement:

```sql
create table TEST(
  ID int
  NAME varchar(30)
);
```

To create a table called `COUNTRY` with an `ID` column, a two letter country code column `CCODE`, and a variable length country name column `NAME`:

```sql
create table COUNTRY(
    ID int,
    CCODE char(2),
    NAME varchar(60)
);
```

Sometimes you may see additional keywords in a create table statement:

```
create table COUNTRY(
    ID int NOT NULL,
    CCODE char(2),
    NAME varchar(60),
    PRIMARY KEY(ID)
);
```

- In the above example the `ID` column has the **`NOT NULL`** constraint added after the datatype - meaning that *it cannot contain a NULL or an empty value.*
- If you look at the last row in the create table statement above you will note that we are using `ID` as a **Primary Key** and the database **does not allow** Primary Keys to have **`NULL`** values. *A Primary Key is a unique identifier in a table, and using Primary Keys can help speed up your queries significantly.*
- If the table you are trying to create already exists in the database, you will get an error indicating table `XXX.YYY` already exists. To circumvent this error, either create a table with a different name or first `DROP` the existing table. It is quite common to issue a `DROP` before doing a `CREATE` in test and development scenarios.

4.2 DROP

The general syntax to drop a table:

```
drop table TABLENAME;
```

For example, to drop the table COUNTRY, we can use the following code:

```
drop table COUNTRY;
```

4.3 ALTER

```
ALTER TABLE table_name
ADD COLUMN column_name data_type column_constraint;

ALTER TABLE table_name
DROP COLUMN column_name;
```

```sql
ALTER TABLE table_name
ALTER COLUMN column_name SET DATA TYPE data_type;

ALTER TABLE table_name
RENAME COLUMN current_column_name TO new_column_name;
```

4.4 TRUNCATE

```sql
TRUNCATE TABLE table_name;
```

4.5 Guided Exercise: Create table and insert data

You will to create two tables

1. PETSALE
2. PET.

```sql
CREATE TABLE PETSALE (
    ID INTEGER NOT NULL,
    PET CHAR(20),
    SALEPRICE DECIMAL(6,2),
    PROFIT DECIMAL(6,2),
    SALEDATE DATE
    );

CREATE TABLE PET (
    ID INTEGER NOT NULL,
    ANIMAL VARCHAR(20),
    QUANTITY INTEGER
    );
```

Now insert some records into the two newly created tables and show all the records of the two tables.

```sql
INSERT INTO PETSALE VALUES
    (1,'Cat',450.09,100.47,'2018-05-29'),
    (2,'Dog',666.66,150.76,'2018-06-01'),
    (3,'Parrot',50.00,8.9,'2018-06-04'),
    (4,'Hamster',60.60,12,'2018-06-11'),
    (5,'Goldfish',48.48,3.5,'2018-06-14');

INSERT INTO PET VALUES
    (1,'Cat',3),
    (2,'Dog',4),
    (3,'Hamster',2);

SELECT * FROM PETSALE;
SELECT * FROM PET;
```

4.6 Guided Exercise: Use the `ALTER` statement to add, delete, or modify columns in two of the existing tables created in the previous exercise.

*Add a new **`QUANTITY`** column to the **`PETSALE`** table and show the altered table.*

```
ALTER TABLE PETSALE
ADD COLUMN QUANTITY INTEGER;

SELECT * FROM PETSALE;
```

*Now update the newly added **`QUANTITY`** column of the **`PETSALE`** table with some values and show all the records of the table.*

```
UPDATE PETSALE SET QUANTITY = 9 WHERE ID = 1;
UPDATE PETSALE SET QUANTITY = 3 WHERE ID = 2;
UPDATE PETSALE SET QUANTITY = 2 WHERE ID = 3;
UPDATE PETSALE SET QUANTITY = 6 WHERE ID = 4;
UPDATE PETSALE SET QUANTITY = 24 WHERE ID = 5;

SELECT * FROM PETSALE;
```

*Delete the **`PROFIT`** column from the **`PETSALE`** table and show the altered table.*

```
ALTER TABLE PETSALE
DROP COLUMN PROFIT;

SELECT * FROM PETSALE;
```

Change the data type to `VARCHAR(20)` *type of the column* `PET` *of the table* `PETSALE` *and show the altered table.*

```
ALTER TABLE PETSALE
ALTER COLUMN PET SET DATA TYPE VARCHAR(20);

SELECT * FROM PETSALE;
```

If you are using IBM db2: Now verify if the data type of the column PET of the table PETSALE changed to `VARCHAR(20)` type or not. Click on the 3 bar menu icon in the top left corner and click Explore > Tables. Find the `PETSALE` table from Schemas by clicking Select All. Click on the `PETSALE` table to open the Table Definition page of the table. Here, you can see all the current data type of the columns of the `PETSALE` table.

Rename the column PET to ANIMAL of the PETSALE table and show the altered table.

```
ALTER TABLE PETSALE
RENAME COLUMN PET TO ANIMAL;

SELECT * FROM PETSALE;
```

4.7 Guided Exercise: TRUNCATE

In this exercise, you will use the `TRUNCATE` statement to remove all rows from an existing table created in exercise 1 without deleting the table itself.

Remove all rows from the PET table and show the empty table.

```
TRUNCATE TABLE PET IMMEDIATE;
SELECT * FROM PET;
```

4.8 Guided Exercise: DROP

In this exercise, you will use the `DROP` statement to delete an existing table created in the previous exercise.

Delete the PET table and verify if the table still exists or not (SELECT statement won't work if a table doesn't exist).

```
DROP TABLE PET;
SELECT * FROM PET;
```

4.9 Exercise: String Patterns

In this exercise, you will go through some SQL problems on String Patterns.

Here is `EMPLOYEES` table.

EMP_ID	F_NAME	L_NAME	SSN	B_DATE
E1001	John	Thomas	123456	1976-01-09
E1002	Alice	James	123457	1972-07-31
E1003	Steve	Wells	123458	1980-08-10
E1004	Santosh	Kumar	123459	1985-07-20
E1005	Ahmed	Hussain	123410	1981-01-04
E1006	Nancy	Allen	123411	1978-02-06
E1007	Mary	Thomas	123412	1975-05-05
E1008	Bharath	Gupta	123413	1985-05-06
E1009	Andrea	Jones	123414	1990-07-09
E1010	Ann	Jacob	123415	1982-03-30

SEX	ADDRESS	JOB_ID	SALARY	MANAGER_ID	DEP_ID
M	5631 Rice, OakPark,IL	100	100000	30001	2
F	980 Berry ln, Elgin,IL	200	80000	30002	5
M	291 Springs, Gary,IL	300	50000	30002	5
M	511 Aurora Av, Aurora,IL	400	60000	30004	5
M	216 Oak Tree, Geneva,IL	500	70000	30001	2
F	111 Green Pl, Elgin,IL	600	90000	30001	2
F	100 Rose Pl, Gary,IL	650	65000	30003	7
M	145 Berry Ln, Naperville,IL	660	65000	30003	7
F	120 Fall Creek, Gary,IL	234	70000	30003	7
F	111 Britany Springs,Elgin,IL	220	70000	30004	5

4.9.1 *Retrieve all employees whose address is in Elgin,IL.*

Click here for the solution

```sql
SELECT F_NAME , L_NAME
FROM EMPLOYEES
WHERE ADDRESS LIKE '%Elgin,IL%';
```

4.9.2 *Retrieve all employees who were born during the 1970's..*

Click here for the solution

```
SELECT F_NAME , L_NAME
FROM EMPLOYEES
WHERE B_DATE LIKE '197%';
```

4.9.3 *Retrieve all employees in department 5 whose salary is between 60000 and 70000..*

Click here for the solution

```
SELECT F_NAME , L_NAME
FROM EMPLOYEES
WHERE DEP_ID = 5 and (SALARY BETWEEN 60000 AND 70000);
--Notice the "=" and "and"
```

4.10 Exercise: Sorting

4.10.1 *Retrieve a list of employees ordered by department ID..*

Click here for the solution

```
SELECT F_NAME, L_NAME, DEP_ID
FROM EMPLOYEES
ORDER BY DEP_ID;
```

4.10.2 *Retrieve a list of employees ordered in descending order by department ID and within each department ordered alphabetically in descending order by last name..*

Click here for the solution

```
SELECT F_NAME, L_NAME, DEP_ID
FROM EMPLOYEES
ORDER BY DEP_ID DESC, L_NAME DESC;
```

4.10.3 *In the previous problem, use department name instead of department ID. Retrieve a list of employees ordered by department name, and within each department ordered alphabetically in descending order by last name..*

Here is the `DEPARTMENTS` table.

DEPT_ID_DEP	DEP_NAME	MANAGER_ID	LOC_ID
2	Architect Group	30001	L0001
5	Software Group	30002	L0002
7	Design Team	30003	L0003

Click here for the solution

```
SELECT D.DEP_NAME , E.F_NAME, E.L_NAME
FROM EMPLOYEES as E, DEPARTMENTS as D
WHERE E.DEP_ID = D.DEPT_ID_DEP
ORDER BY D.DEP_NAME, E.L_NAME DESC;
```

In the SQL Query above, `D` and `E` are aliases for the table names. Once you define an alias like `D` in your query, you can simply write `D.COLUMN_NAME` rather than the full form `DEPARTMENTS.COLUMN_NAME`.

4.11 Exercise 3: Grouping

4.11.1 *For each department ID retrieve the number of employees in the department..*

Click here for the solution

```
SELECT DEP_ID, COUNT(*)
FROM EMPLOYEES
GROUP BY DEP_ID;
```

4.11.2 *For each department retrieve the number of employees in the department, and the average employee salary in the department..*

Click here for the solution

```
SELECT DEP_ID, COUNT(*), AVG(SALARY)
FROM EMPLOYEES
GROUP BY DEP_ID;
```

4.11.3 *Label the computed columns in the result set of the last SQL problem as NUM_EMPLOYEES and AVG_SALARY..*

Click here for the solution

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID;
```

4.11.4 *In the previous SQL problem , order the result set by Average Salary..*

Click here for the solution

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID
ORDER BY AVG_SALARY;
```

4.11.5 *In SQL problem 4 (Exercise 3 Problem 4), limit the result to departments with fewer than 4 employees..*

Click here for the solution

```
SELECT DEP_ID, COUNT(*) AS "NUM_EMPLOYEES", AVG(SALARY) AS "AVG_SALARY"
FROM EMPLOYEES
GROUP BY DEP_ID
HAVING count(*) < 4
ORDER BY AVG_SALARY;
```