Toolbox

Bhaswar Chakma

2021-09-23

Contents

Chapter 1

Grammar: dplyr vs pandas & numpy

We will use the five dplyr verbs (also pandas' guide) 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()

1.1.1 *Example: 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 = ['gender'])
```

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

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

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

Using positions of columns:

```
df[df.columns[[0,2]]]
```

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

```
df.iloc[:, [0,2]]
```

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

1.2 `mutate()`

1.2.1 *Example: 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()

1.3.1 *Example: Keep if the student is Barney or female.*

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()

1.4.1 *Example: 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
```

Option 1:

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

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

Option 2:

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

```
##             grade
```

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

Option 3:

```
# useful for multiple groups and stat
df.groupby(['gender']).agg(
    {'grade': ['mean']}
    # Here key: variable name; value: stat function
)
# here [] is unnecessary but
# required for multiple groups and stats
```

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

1.4.2 *Example: 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"]).agg(
  # provide a dictionary
  #   key: variable name
  #   value: stat function
   {'grade':['mean',
             'median',
             'min',
             'max']}
```

```
)
```

```
##             grade
##              mean median min max
## gender
## Female  13.000000     13  12  14
## Male    11.333333     11  10  13
```

1.5 arrange()

1.5.1 *Example: 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
```

1.5.2 *Example: Arrange grade in descending 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
```

1.5.3 *Example: 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

Helper Functions

2.1 `case_when()` vs `pd.cut()` and `np.select()`

Suppose we have a data frame with a variable called ***`age`****. We want to create a variable* ***`age_cat`*** *with the following conditions:*

- `age` $<$ 18: "Kids"
- 18 $\leq$ `age` $<$ 31: "18-30"
- 31 $\leq$ `age`: "31 and above"

Example:

age	age_cat
9	Kids
10	Kids
18	18-30
21	18-30
29	18-30
31	31 and above
45	31 and above

dplyr::case_when()

With `dplyr::case_when()` we can do it in the following way:

```
# example data
df <- tibble(age = c(9, 10, 18, 21, 29, 31, 45))
# case_when() in action
df %>%
  mutate(age_cat = case_when(
```

```
    age < 18 ~ "Kids",
    age >= 18 & age < 31 ~ "18-30",
    age >= 31 ~ "31 and above"
  ))
```

```
## # A tibble: 7 x 2
##     age age_cat
##   <dbl> <chr>
## 1     9 Kids
## 2    10 Kids
## 3    18 18-30
## 4    21 18-30
## 5    29 18-30
## 6    31 31 and above
## 7    45 31 and above
```

We can achieve the same result in Python using

- `np.select()`
- `pd.cut()`

```
df = pd.DataFrame({'age': [9, 10, 18, 21, 29, 31, 45]})
```

np.select()

pd.cut()

```
# Step 1: Create conditions
cond = [
  (df['age'].lt(18)),
  (df['age'].ge(18) & (df['age'].lt(31))),
  (df['age'].ge(31))
]

# Step 2: Assign labels
cond_labs = [
  'Kids', '18-30', '30 and above'
]

# Step 3: apply np.select()
df['age_cat'] = np.select(cond, cond_labs)
df
```

```
##    age       age_cat
## 0    9          Kids
## 1   10          Kids
## 2   18         18-30
## 3   21         18-30
```

```
## 4   29         18-30
## 5   31  30 and above
## 6   45  30 and above
```

```
# Step 1: Create bin condition
bin_cond = [0, 17, 30, np.inf]
# note: instead of 0,
#       -np.inf will also work
#
# 0: greater than 0
# 17: upper limit is 17

# Step 2: Assign bin labs
bin_labs = [
  'Kids',
  '18-30',
  '30 and above'
]

# Step 3: apply pd.cut()
df["age_cat"] = pd.cut(
  df["age"],
  bins = bin_cond,
  labels = bin_labs
)
df
```

```
##    age       age_cat
## 0    9          Kids
## 1   10          Kids
## 2   18         18-30
## 3   21         18-30
## 4   29         18-30
## 5   31  30 and above
## 6   45  30 and above
```

2.2 if_else() vs np.where()

Given prices of shirts `price`*, how do we create a variable* `price_cat` *with the following conditions?*

- when `price` is less than 50, we label it as "Cheap"
- when `price` is 50 or more, we label it as "Expensive"

dplyr::if_else()

np.where()

```r
# toy data
prices <- c(25, 30, 45, 80,100, 125)
df <- tibble(price = prices)
# if_else in action
df %>%
  mutate(price_cat = if_else(
    price <50, "Cheap", "Expenseive"
  ))
```

```
## # A tibble: 6 x 2
##   price price_cat
##   <dbl> <chr>
## 1    25 Cheap
## 2    30 Cheap
## 3    45 Cheap
## 4    80 Expenseive
## 5   100 Expenseive
## 6   125 Expenseive
```

```python
# toy data
prices = {
  'price': [25, 30, 45, 80,100, 125]
}
df = pd.DataFrame(prices)
# np.where() in action
df['price_cat'] = np.where(
  df.price < 50, "Cheap", "Expenseive"
)
df
```

```
##    price   price_cat
## 0     25       Cheap
## 1     30       Cheap
## 2     45       Cheap
## 3     80  Expenseive
## 4    100  Expenseive
## 5    125  Expenseive
```

2.3 %in% vs isin, @, and in

code	capital
BD	Dhaka
PT	Lisbon
ES	Madrid

code	capital
FR	Paris

How to keep observations that belong to BD or DE (without using the | operator)?

R

```r
# toy data
df <- tibble(
  # country code
  code = c(
    "BD", "PT",
    "ES", "FR"
  ),
  # capital
  capital = c(
    "Dhaka", "Lisbon",
    "Madrid", "Paris"
  )
)
df
```

```
## # A tibble: 4 x 2
##   code  capital
##   <chr> <chr>
## 1 BD    Dhaka
## 2 PT    Lisbon
## 3 ES    Madrid
## 4 FR    Paris
```

```r
# %in% in action
df %>%
  filter(code %in% c("BD", "PT"))
```

```
## # A tibble: 2 x 2
##   code  capital
##   <chr> <chr>
## 1 BD    Dhaka
## 2 PT    Lisbon
```

Python

```python
# toy data
data = {
    # country code
    'code':[
```

```
        "BD", "PT",
        "ES", "FR"
    ],
    # capital
    'capital':[
        "Dhaka", "Lisbon",
        "Madrid", "Paris"
    ]
}
df = pd.DataFrame(data)
df
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
## 2   ES  Madrid
## 3   FR   Paris
```

isin

```
# isin in action
df[df["code"].isin(["BD", "PT"])]
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

@

```
country_list = ["BD", "PT"]
# @ in action
df.query('code == @country_list')
# note: you must create a list first
#   @["BD", "PT"] doesn't work
#   but, @list(["BD", "PT"] works
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

in

```
# in in action
df.query('code in ["BD", "PT"]')
```

```
##   code capital
## 0   BD   Dhaka
## 1   PT  Lisbon
```

2.4 stringr::str_detect() vs str.contains()

Example:

info	amount
XYZ Deposit 2020	0
Cash Deposit	1
ATM	2
XYZ Fee 2021	3
XYZ Deposit 2021	4

How to keep or drop only those observations where `info` *is about* `XYZ`*?*

R `stringr::str_detect()`

Python `str.contains()`

```
# toy data
df <- tibble(
  info = c(
    "XYZ Deposit 2020",
    "Cash Deposit",
    "ATM",
    "XYZ Fee 2021",
    "XYZ Deposit 2021"
  ),
  amount = seq(1,5) - 1
)

df
```

```
## # A tibble: 5 x 2
##   info              amount
##   <chr>              <dbl>
## 1 XYZ Deposit 2020       0
## 2 Cash Deposit           1
## 3 ATM                    2
## 4 XYZ Fee 2021           3
## 5 XYZ Deposit 2021       4
```

Keep:

```
# str_detect() in action
df %>%
  filter( # keeps
    stringr::str_detect(
      info, "XYZ"
```

```
    )
  )
```

```
## # A tibble: 3 x 2
##   info             amount
##   <chr>             <dbl>
## 1 XYZ Deposit 2020      0
## 2 XYZ Fee 2021          3
## 3 XYZ Deposit 2021      4
```

Drop:

```
# str_detect() in action
df %>%
  filter( # drops
    ! stringr::str_detect(
      info, "XYZ"
    )
  )
```

```
## # A tibble: 2 x 2
##   info         amount
##   <chr>         <dbl>
## 1 Cash Deposit      1
## 2 ATM               2
```

```
# toy data
df = pd.DataFrame({
    'info':
    [
    "XYZ Deposit 2020",
    "Cash Deposit",
    "ATM",
    "XYZ Fee 2021",
    "XYZ Deposit 2021"
],
    'amount': np.arange(5)
})
df
```

```
##                info  amount
## 0  XYZ Deposit 2020       0
## 1       Cash Deposit       1
## 2                ATM       2
## 3       XYZ Fee 2021       3
## 4  XYZ Deposit 2021       4
```

Keep:

```
# str.contains in action: keep
df[df['info'].str.contains("XYZ")]
```

```
##                info  amount
## 0  XYZ Deposit 2020       0
## 3      XYZ Fee 2021       3
## 4  XYZ Deposit 2021       4
```

Drop:

```
# str.contains in action: drop
df[~ df['info'].str.contains("XYZ")]
```

```
##            info  amount
## 1  Cash Deposit       1
## 2           ATM       2
```

2.5 distinct() vs unique()

name	year
A	2020
B	2020
C	2020
A	2021
B	2021
D	2021

What are the distinct names?

dplyr::distinct()

pd.unique()

```
# toy data
df <- tibble(
    name = c(
        'A', 'B', 'C',
        'A', 'B', 'D'
    ),
    year = c(
        2020, 2020, 2020,
        2021, 2021, 2021
    )
)
```

```r
# dplyr::distinct in action
df %>% distinct(name)
```

```
## # A tibble: 4 x 1
##   name
##   <chr>
## 1 A
## 2 B
## 3 C
## 4 D
```

```r
# if you want to count
df %>% distinct(name) %>% count()
```

```
## # A tibble: 1 x 1
##       n
##   <int>
## 1     4
```

```python
# toy data
df = pd.DataFrame({
    'name':[
        'A', 'B', 'C',
        'A', 'B', 'D'
    ],
    'year':[
        2020, 2020, 2020,
        2021, 2021, 2021
    ]
})
```

```python
# pd.unique() in action
df['name'].unique()
```

```
## array(['A', 'B', 'C', 'D'], dtype=object)
```

```python
# if you want to count
len(df['name'].unique())
```

```
## 4
```

2.6 slice() vs iloc()

age	id
20	1
21	1
22	1

age	id
23	1
24	1

How to slice single or multiple rows?

dplyr::slice()

pd.iloc()

```
# toy data
df <- tibble(
  age = seq(20,24),
  id = seq(1,5)
)
df
```

```
## # A tibble: 5 x 2
##     age    id
##   <int> <int>
## 1    20     1
## 2    21     2
## 3    22     3
## 4    23     4
## 5    24     5
```

```
# slice row 1
df %>% slice(1)
```

```
## # A tibble: 1 x 2
##     age    id
##   <int> <int>
## 1    20     1
```

```
# slice rows 3 and 4
df %>% slice(3:4)
```

```
## # A tibble: 2 x 2
##     age    id
##   <int> <int>
## 1    22     3
## 2    23     4
```

```
# toy data
df = pd.DataFrame({
    'age' : np.arange(20, 25),
    'id': np.arange(1, 6)
})
```

```
df
```

```
##    age  id
## 0   20   1
## 1   21   2
## 2   22   3
## 3   23   4
## 4   24   5
```

```
# slice row 1
df.iloc[0:1]
```

```
##    age  id
## 0   20   1
```

```
# slice row 3 and 4
df.iloc[2:4]
```

```
##    age  id
## 2   22   3
## 3   23   4
```

Chapter 3

Join and Bind

3.1 Join Data Frames: `*_join` vs `merge()`

Toy Data

`df1`:

id	first_name
hiRS	Robin
hiTM	Ted
bbP	Penny
bbSC	Sheldon

`df2`:

id	last_name
hiRS	Robin
hiTM	Ted
bbSC	Cooper
bbLH	Hofstadter

R

Python

```
# toy data
df1 = tibble(
    'id'= c("hiRS", "hiTM",
            "bbP", "bbSC"),
```

```
    'first_name'= c(
        'Robin',
        'Ted',
        'Penny',
        'Sheldon'
    )
)



df2 = tibble(
    'id'= c("hiRS", "hiTM",
            "bbSC", "bbLH"),
    'last_name'= c(
        'Robin',
        'Ted',
        'Cooper',
        'Hofstadter'
    )
)
```

```
# toy data
df1 = pd.DataFrame({
    'id': ["hiRS", "hiTM",
           "bbP", "bbSC"],
    'first_name':[
        'Robin',
        'Ted',
        'Penny',
        'Sheldon'
    ]
})



df2 = pd.DataFrame({
    'id': ["hiRS", "hiTM",
           "bbSC", "bbLH"],
    'last_name':[
        'Scherbatsky',
        'Mosby',
        'Cooper',
        'Hofstadter'
    ]
})
```

3.1.1 Inner Join

dplyr::inner_join()

pandas::merge(how = "inner")

```
# inner_join in action
# note: by argument is not needed here
df1 %>% inner_join(df2, by = "id")
```

```
## # A tibble: 3 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbSC  Sheldon    Cooper
```

```
# merge(how = "inner") in action
df1.merge(df2, how = 'inner', on = 'id')
# note: default is inner join
# i.e. df1.merge(df2) does inner join
```

```
##      id first_name    last_name
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2  bbSC    Sheldon       Cooper
```

3.1.2 Left Join

dplyr::inner_join()

pandas::merge(how = "left")

```
# left_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% left_join(df2, by = "id")
```

```
## # A tibble: 4 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbP   Penny      <NA>
## 4 bbSC  Sheldon    Cooper
```

```
# merge(how = "left") in action
df1.merge(df2, how = 'left', on = 'id')
```

```
##      id first_name    last_name
```

```
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2   bbP      Penny          NaN
## 3  bbSC    Sheldon       Cooper
```

3.1.3 Right Join

dplyr::right_join()

pandas::merge(how = "right")

```
# left_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% right_join(df2, by = "id")
```

```
## # A tibble: 4 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
## 3 bbSC  Sheldon    Cooper
## 4 bbLH  <NA>       Hofstadter
```

```
# merge(how = "right") in action
df1.merge(df2, how = 'right', on = 'id')
```

```
##      id first_name    last_name
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2  bbSC    Sheldon       Cooper
## 3  bbLH        NaN   Hofstadter
```

3.1.4 Full Join

dplyr::full_join()

pandas::merge(how = "outer")

```
# full_join in action
# note: by argument is not needed here
# needed when you have more keys
df1 %>% full_join(df2, by = "id")
```

```
## # A tibble: 5 x 3
##   id    first_name last_name
##   <chr> <chr>      <chr>
## 1 hiRS  Robin      Robin
## 2 hiTM  Ted        Ted
```

```
## 3 bbP   Penny      <NA>
## 4 bbSC  Sheldon    Cooper
## 5 bbLH  <NA>       Hofstadter
```

```
# merge(how = "outer") in action
df1.merge(df2, how = 'outer', on = 'id')
```

```
##      id first_name    last_name
## 0  hiRS      Robin  Scherbatsky
## 1  hiTM        Ted        Mosby
## 2   bbP      Penny          NaN
## 3  bbSC    Sheldon       Cooper
## 4  bbLH        NaN   Hofstadter
```

3.2 Bind Rows: `bind_rows()` vs `append()`

df1:

first_name	last_name
Steve	Vai
Joe	Satriani

df2:

first_name	last_name
Paul	Gilbert
Eric	Johnson

How do we combine the rows of `df1` and `df2`?

dplyr::bind_rows()

pandas::append()

```
# toy data
df1 <- tibble(
    first_name = c('Steve', 'Joe'),
    last_name = c('Vai', 'Satriani')
)

df2 <- tibble(
    first_name = c('Paul', 'Eric'),
    last_name = c('Gilbert', 'Johnson')
)
```

```
# bind_rows() in action
bind_rows(df1, df2)
```

```
## # A tibble: 4 x 2
##   first_name last_name
##   <chr>      <chr>
## 1 Steve      Vai
## 2 Joe        Satriani
## 3 Paul       Gilbert
## 4 Eric       Johnson
```

```
# toy data
df1 = pd.DataFrame({
    'first_name': ['Steve', 'Joe'],
    'last_name': ['Vai', 'Satriani']
})

df2 = pd.DataFrame({
    'first_name': ['Paul', 'Eric'],
    'last_name': ['Gilbert', 'Johnson']
})
```

```
# append in action: ignoring the index
df1.append(df2, ignore_index = True)
```

```
##   first_name last_name
## 0      Steve       Vai
## 1        Joe  Satriani
## 2       Paul   Gilbert
## 3       Eric   Johnson
```

```
# append in action: maintaining the index
df1.append(df2)
```

```
##   first_name last_name
## 0      Steve       Vai
## 1        Joe  Satriani
## 0       Paul   Gilbert
## 1       Eric   Johnson
```

Chapter 4

Reshape: tidyr vs pandas

4.1 `pivot_longer()` vs `melt()`

4.1.1 *Example: Life Expectancy data in "wide" format*

country	1997	2007
Bangladesh	59.4	64.1
Portugal	76.0	78.1

How do we make the table "long"?

Desired output:

country	year	life_exp
Bangladesh	1997	59.4
Bangladesh	2007	64.1
Portugal	1997	76.0
Portugal	2007	78.1

tidyr::pivot_longer

pandas::melt()

```
# toy data
df <- tibble(
  country = c("Bangladesh", "Portugal"),
  `1997` = c(59.4, 76.0),
  `2007` = c(64.1, 78.1)
```

```
  )
df
```

```
## # A tibble: 2 x 3
##   country    `1997` `2007`
##   <chr>       <dbl>  <dbl>
## 1 Bangladesh   59.4   64.1
## 2 Portugal     76     78.1
```

pivot_longer() in action!

```
# pivot_longer in action
df %>%
  pivot_longer(
    cols = c(`1997`, `2007`),
    names_to = "year",
    values_to = "life_exp"
  )
```

```
# toy data
data = {
  'country': ["Bangladesh", "Portugal"],
  '1997': [59.4, 76.0],
  '2007': [64.1, 78.1]
}
df = pd.DataFrame(data)
df
```

```
##       country  1997  2007
## 0  Bangladesh  59.4  64.1
## 1    Portugal  76.0  78.1
```

melt() in action!

```
# melt() in action
df.melt(
  id_vars = 'country',
  value_vars = ['1997', '2007'], # cols
  var_name = 'year', # names_to
  value_name = 'life_exp' # values_to
)
```

4.2 pivot_wider vs pivot

4.2.1 *Example: Life Expectancy data in "long" format*

country	year	life_exp
Bangladesh	1997	59.4
Bangladesh	2007	64.1
Portugal	1997	76.0
Portugal	2007	78.1

How do we make the table "wide"?

Desired output:

country	1997	2007
Bangladesh	59.4	64.1
Portugal	76.0	78.1

tidyr::pivot_wider()

pandas::pivot()

```
# toy data

country <- c(
  "Bangladesh", "Bangladesh",
  "Portugal", "Portugal"
)

year <- c(
  "1997", "2007",
  "1997", "2007"
)

life_exp <- c(
  59.4, 64.1,
  76, 78.1
)

df <- tibble(country, year, life_exp)
df
```

```
## # A tibble: 4 x 3
##   country    year  life_exp
##   <chr>      <chr>    <dbl>
## 1 Bangladesh 1997      59.4
## 2 Bangladesh 2007      64.1
## 3 Portugal   1997      76
## 4 Portugal   2007      78.1
```

pivot_wider() in action!

```
# pivot_wider in action
df %>%
  pivot_wider(
    names_from = "year",
    values_from = "life_exp"
  )
```

```
## # A tibble: 2 x 3
##   country    `1997` `2007`
##   <chr>       <dbl>  <dbl>
## 1 Bangladesh   59.4   64.1
## 2 Portugal     76     78.1
```

```
# toy data

country =[
    "Bangladesh", "Bangladesh",
    "Portugal", "Portugal"
]

year = [
    "1997", "2007",
    "1997", "2007"
]

life_exp = [
    59.4, 64.1,
    76, 78.1
]

df = pd.DataFrame(
   {'country': country,
    'year': year,
    'life_exp': life_exp}
)
df
```

```
##       country  year  life_exp
## 0  Bangladesh  1997      59.4
## 1  Bangladesh  2007      64.1
## 2    Portugal  1997      76.0
## 3    Portugal  2007      78.1
```

pivot() in action!

```
# pivot in action
df_wide = df.pivot(
    index = 'country',
    columns = 'year', # names from
    values = 'life_exp' # vasles from
)
df_wide
```

```
## year        1997  2007
## country
## Bangladesh  59.4  64.1
## Portugal    76.0  78.1
```

```
# Reset the names
df_wide.index.name = None
df_wide.columns.name = None
```

```
df_wide
```

```
##             1997  2007
## Bangladesh  59.4  64.1
## Portugal    76.0  78.1
```

4.3 pandas:: stack()

```
# toy data
df = pd.DataFrame({
    'year': np.arange(2020,2025),
    'Fall': np.linspace(10,15,5),
    'Spring': np.linspace(1, 5,5)
})
df
```

```
##    year   Fall  Spring
## 0  2020  10.00     1.0
## 1  2021  11.25     2.0
## 2  2022  12.50     3.0
## 3  2023  13.75     4.0
## 4  2024  15.00     5.0
```

How to create MultiIndex series?

```
# step 1: set year as index
df.set_index('year', inplace = True)
df
```

```
##        Fall  Spring
```

```
## year
## 2020  10.00     1.0
## 2021  11.25     2.0
## 2022  12.50     3.0
## 2023  13.75     4.0
## 2024  15.00     5.0
```

```
# step 2: apply stack()
df_stacked = df.stack()
df_stacked
```

```
## year
## 2020  Fall      10.00
##       Spring     1.00
## 2021  Fall      11.25
##       Spring     2.00
## 2022  Fall      12.50
##       Spring     3.00
## 2023  Fall      13.75
##       Spring     4.00
## 2024  Fall      15.00
##       Spring     5.00
## dtype: float64
```

```
# check type
type(df_stacked)
```

```
## <class 'pandas.core.series.Series'>
```

```
# check index
df_stacked.index
```

```
## MultiIndex([(2020,   'Fall'),
##             (2020, 'Spring'),
##             (2021,   'Fall'),
##             (2021, 'Spring'),
##             (2022,   'Fall'),
##             (2022, 'Spring'),
##             (2023,   'Fall'),
##             (2023, 'Spring'),
##             (2024,   'Fall'),
##             (2024, 'Spring')],
##            names=['year', None])
```

4.4 pandas:: unstack()

```
# toy data
year = [2010, 2010, 2010, 2020, 2020, 2020]
name = ["X", "Y", "Z", "X", "Y", "Z"]
gender = ["M", "F", "F", "M", "F", "F"]
grade = [10, 10, 20, 20, 12.5, 17.5]
df = pd.DataFrame(
    {
    'year': year,
    'name': name,
    'gender': gender,
    'grade': grade
    }
)
df
```

```
##    year name gender  grade
## 0  2010    X      M   10.0
## 1  2010    Y      F   10.0
## 2  2010    Z      F   20.0
## 3  2020    X      M   20.0
## 4  2020    Y      F   12.5
## 5  2020    Z      F   17.5
```

Suppose we want to find mean grade grouped by year and gender.

```
# grouped by year and gender:
# find mean grade
df_stat = df.groupby(
    ['year', 'gender']
).agg({
    'grade': ['mean']
})
df_stat
```

```
##             grade
##              mean
## year gender
## 2010 F       15.0
##      M       10.0
## 2020 F       15.0
##      M       20.0
```

How to get F and M as columns?

- Just apply unstack()

```
df_unstacked = df_stat.unstack()
df_unstacked
```

```
##        grade
##         mean
## gender     F     M
## year
## 2010    15.0  10.0
## 2020    15.0  20.0
```

```
# To change the names

# reset index
df_unstacked2 = df_unstacked.reset_index()
# rename
df_unstacked2.columns = ['year', 'M', 'F']
df_unstacked2
```

```
##    year     M     F
## 0  2010  15.0  10.0
## 1  2020  15.0  20.0
```

Chapter 5

Graphics

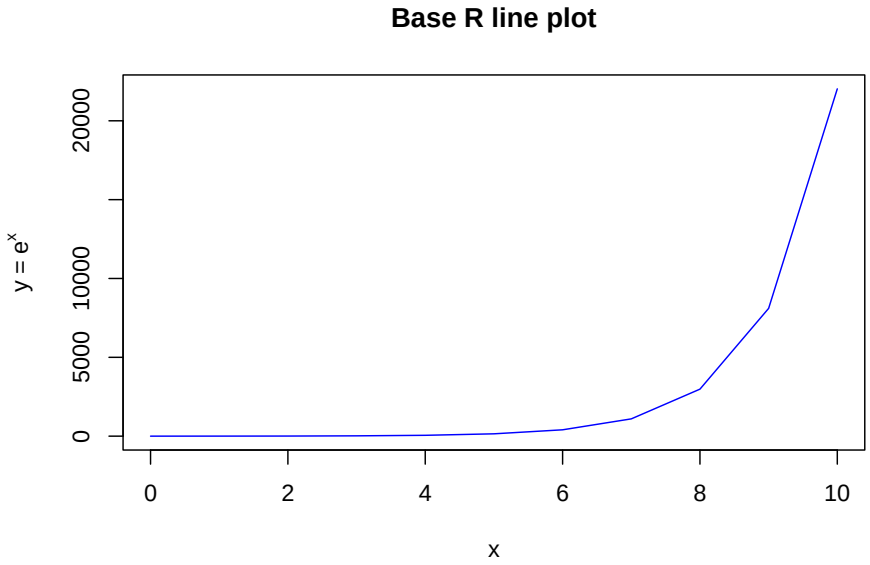
5.1 Base R vs Matplotlib

plot()

plt.plot()

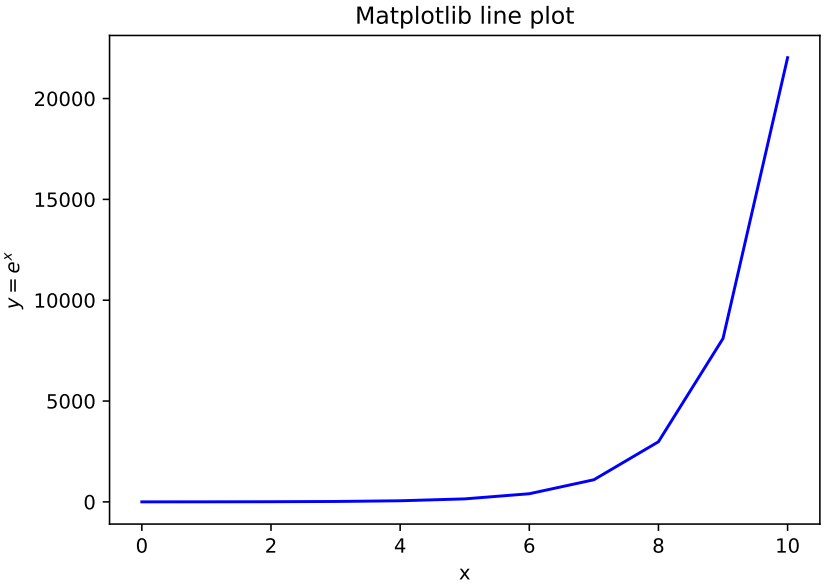
```r
# toy data
x <- seq(0, 10)
y <- exp(x)
```

```r
# default "type" is "p"
plot(
  x, y,
  type = "l",
  col = "blue",
  # title
  main = "Base R line plot",
  xlab = "x",
  ylab = expression("y = e"^"x")
  # latex2exp package offers more
)
```




```
# toy data
x = np.arange(11)
y = np.exp(np.arange(11))
```

```
# default 'kind' is 'line'
plt.plot(
  x, y,
  color = 'blue'
  # titles are added separately
);
plt.title('Matplotlib line plot');
plt.xlabel('x');
plt.ylabel('$y = e^x$');
plt.show()
```


Matplotlib line plot
20000
15000
10000
5000
0
$y = e^x$
0
2
4
6
8
10
x

Chapter 6

Missing Values and Duplicates

6.1 Missing Values

6.1.1 `tidyr::drop_na()` vs `pd.dropna()` and `pd.notpna()`

name	age
A	21
B	NA
NA	NA
D	40

How to drop missing rows or columns?

tidyr::drop_na()

pd.dropna() and pd.notna()

```
# toy data
df <- tibble(
  name = c('A', 'B',
           NA, 'D'),
  age  =  c(21, NA,
           NA, 40)
)
df
```

```
## # A tibble: 4 x 2
##   name    age
```

```
##   <chr> <dbl>
## 1 A        21
## 2 B        NA
## 3 <NA>     NA
## 4 D        40
```

```
# drop missing rows
df %>% drop_na()
```

```
## # A tibble: 2 x 2
##   name    age
##   <chr> <dbl>
## 1 A        21
## 2 D        40
```

```
# drop specific variable
# with NaN
df %>% drop_na(name)
```

```
## # A tibble: 3 x 2
##   name    age
##   <chr> <dbl>
## 1 A        21
## 2 B        NA
## 3 D        40
```

```
# toy data
df = pd.DataFrame({
    'name': [
        'A', 'B', np.nan, 'D'
    ],
    'age': [
        21, np.nan, np.nan, 40
    ]
})
df
```

```
##   name   age
## 0    A  21.0
## 1    B   NaN
## 2  NaN   NaN
## 3    D  40.0
```

```
# drop missing rows
df.dropna()
```

```
##   name   age
## 0    A  21.0
## 3    D  40.0
```

```
# drop specific variable
# with NaN
df[df['name'].notna()]
```

```
##   name   age
## 0    A  21.0
## 1    B   NaN
## 3    D  40.0
```

Chapter 7

SQL

7.1 Create Table

```
DROP TABLE students;
--- create toy table: students

CREATE TABLE students
(student_id CHAR(5) NOT NULL,
name VARCHAR(10),
gender VARCHAR(6),
grade INT,
PRIMARY KEY (student_id));

-- insert data

INSERT INTO students
VALUES
('SQ001', 'Barney', 'Male', 18),
('SQ002', 'Robin', 'Female', 17),
('SQ007', 'Sheldon', 'Male', 20),
('SQ008', 'Penny', 'Female', 13);
```