```
# R
libs <- c('dplyr',                # wrangling
          'palmerpenguins',       # data
          'knitr','kableExtra',   # table styling
          'reticulate')           # python support
invisible(lapply(libs, library, character.only = TRUE))

use_python("/usr/bin/python")

df <- penguins %>%
  mutate_if(is.integer, as.double)
# Python
import pandas as pd
pd.set_option('display.max_columns', None)
```

We will be using the Palmer Penguins dataset, as provided by the brilliant Allison Horst through the eponymous R package – complete with her trademark adorable artwork. I was looking for an excuse to work with this dataset. Therefore, this will be a genuine example for an exploratory analysis, as I'm encountering the data for the first time.

General overview

It's best to start your EDA by looking at the big picture: How large is the dataset? How many features are there? What are the data types? Here we have tabular data, but similar steps can be taken for text or images as well.

In R, I typically go with a combination of **head**, **glimpse**, and **summary** to get a broad idea of the data properties. (Beyond *dplyr*, there's also the skimr package for more sophisticated data overviews.)

With `head`, we see the first (by default) 6 rows of the dataset:

```
# R
head(df)
## # A tibble: 6 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_…
body_mass_g
##
## 1 Adelie  Torge…           39.1          18.7               181
3750
## 2 Adelie  Torge…           39.5          17.4               186
3800
## 3 Adelie  Torge…           40.3          18                 195
3250
## 4 Adelie  Torge…           NA            NA                  NA
NA
## 5 Adelie  Torge…           36.7          19.3               193
3450
## 6 Adelie  Torge…           39.3          20.6               190
3650
## # … with 2 more variables: sex , year
```

We find that our data is a mix of numerical and categorical columns. There are 8 columns in total. We get an idea of the order of magnitude of the numeric features, see that the categorical ones have text, and already spot a couple of missing values. (Note, that I converted all integer columns to double for easier back-and-forth with Python).

The *pandas* **head** command is essentially the same. One general difference here is that in *pandas* (and Python in general) everything is an object. Methods (and attributes) associated with the object, which is a *pandas* `DataFrame` here, are accessed via the dot "." operator. For passing an R object to Python we preface it with `r.` like such:

```
# Python
r.df.head()
##   species     island  bill_length_mm  bill_depth_mm
flipper_length_mm  \
## 0  Adelie  Torgersen            39.1           18.7
181.0
## 1  Adelie  Torgersen            39.5           17.4
186.0
## 2  Adelie  Torgersen            40.3           18.0
195.0
## 3  Adelie  Torgersen             NaN            NaN
NaN
## 4  Adelie  Torgersen            36.7           19.3
193.0
##
##    body_mass_g     sex    year
## 0       3750.0    male  2007.0
## 1       3800.0  female  2007.0
## 2       3250.0  female  2007.0
## 3          NaN     NaN  2007.0
## 4       3450.0  female  2007.0
```

The output is less pretty than for Rmarkdown, but the result is pretty much the same. We don't see column types, only their values. Another property of *pandas* data frames is that they come with a row index. By default this is a sequential number of rows, but anything can become an index.

With *dplyr*'s **glimpse** we can see a more compact, transposed display of column types and their values. Especially for datasets with many columns this can be a vital complement to `head`. We also see the number of rows and columns:

```
# R
glimpse(df)
## Observations: 344
## Variables: 8
## $ species           Adelie, Adelie, Adelie, Adelie, Adelie, Adelie…
## $ island            Torgersen, Torgersen, Torgersen, Torgersen, To…
## $ bill_length_mm    39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, …
## $ bill_depth_mm     18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, …
## $ flipper_length_mm 181, 186, 195, NA, 193, 190, 181, 195, 193, 19…
## $ body_mass_g       3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, …
## $ sex               male, female, female, NA, female, male, female…
## $ year              2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007…
```

Alongside the 8 columns, our dataset has 344 rows. That's pretty small.

In *pandas*, there's no identical equivalent to **glimpse**. Instead, we can use the **info** method to give us the feature types in vertical form. We also see the number of non-null features (the "sex" column has the fewest), together with the number of rows and columns.

```
# Python
r.df.info()
##
## RangeIndex: 344 entries, 0 to 343
## Data columns (total 8 columns):
## species              344 non-null category
## island               344 non-null category
## bill_length_mm       342 non-null float64
## bill_depth_mm        342 non-null float64
## flipper_length_mm    342 non-null float64
## body_mass_g          342 non-null float64
## sex                  333 non-null category
## year                 344 non-null float64
## dtypes: category(3), float64(5)
## memory usage: 14.8 KB
```

While several columns have missing values, the overall number of those null entries is small.

Additionally, *pandas* gives us a summary of data types (3 categorical, 5 float) and tells us how much memory our data takes up. In R, you can use the `utils` tool **object.size** for the latter:

```
# R
object.size(df)
## 21336 bytes
```

Next up is **summary**, which provides basic overview stats for each feature. Here in particular, we learn that there are 3 penguin species, 3 islands, and 11 missing values in the sex column. "Chinstrap" penguins are rarer than the other two species; and Torgersen is the least frequent island. We also see the fundamental characteristics of the numeric features (min, max, quartiles, median). Mean and median are pretty close for most of those, which suggests little skewness:

```
# R
summary(df)
##       species          island    bill_length_mm  bill_depth_mm
##  Adelie   :152   Biscoe   :168   Min.   :32.10   Min.   :13.10
##  Chinstrap: 68   Dream    :124   1st Qu.:39.23   1st Qu.:15.60
##  Gentoo   :124   Torgersen: 52   Median :44.45   Median :17.30
##                                  Mean   :43.92   Mean   :17.15
##                                  3rd Qu.:48.50   3rd Qu.:18.70
##                                  Max.   :59.60   Max.   :21.50
##                                  NA's   :2       NA's   :2
##  flipper_length_mm  body_mass_g       sex           year
##  Min.   :172.0     Min.   :2700   female:165   Min.   :2007
##  1st Qu.:190.0     1st Qu.:3550   male  :168   1st Qu.:2007
##  Median :197.0     Median :4050   NA's  : 11   Median :2008
##  Mean   :200.9     Mean   :4202                Mean   :2008
##  3rd Qu.:213.0     3rd Qu.:4750                3rd Qu.:2009
##  Max.   :231.0     Max.   :6300                Max.   :2009
```

```
##  NA's   :2         NA's   :2
```

The closest *pandas* equivalent to **summary** is **describe**. By default this only includes the numeric columns, but you can get around that by passing a list of features types that you want to include:

```
# Python
r.df.describe(include = ['float', 'category'])
##        species  island  bill_length_mm  bill_depth_mm
flipper_length_mm  \
## count      344     344      342.000000     342.000000
342.000000
## unique       3       3             NaN            NaN
NaN
## top     Adelie  Biscoe             NaN            NaN
NaN
## freq       152     168             NaN            NaN
NaN
## mean       NaN     NaN       43.921930      17.151170
200.915205
## std        NaN     NaN        5.459584       1.974793
14.061714
## min        NaN     NaN       32.100000      13.100000
172.000000
## 25%        NaN     NaN       39.225000      15.600000
190.000000
## 50%        NaN     NaN       44.450000      17.300000
197.000000
## 75%        NaN     NaN       48.500000      18.700000
213.000000
## max        NaN     NaN       59.600000      21.500000
231.000000
##
##         body_mass_g   sex         year
## count    342.000000   333   344.000000
## unique          NaN     2          NaN
## top             NaN  male          NaN
## freq            NaN   168          NaN
## mean    4201.754386   NaN  2008.029070
## std      801.954536   NaN     0.818356
## min     2700.000000   NaN  2007.000000
## 25%     3550.000000   NaN  2007.000000
## 50%     4050.000000   NaN  2008.000000
## 75%     4750.000000   NaN  2009.000000
## max     6300.000000   NaN  2009.000000
```

You see that the formatting is less clever, since categorical indicators like “unique” or “top” are shown (with NAs) for the numeric columns and vice versa. Also, we only get told that there are 3 species and the most frequent one is “Adelie”; not the full counts per species.

We have already learned a lot about our data from those basic overview tools. Typically, at this point in the EDA I would now start plotting feature distributions and interactions. Since we’re only focussing on *dplyr* here, this part will have to wait for a future “ggplot2 vs seaborn” episode.

For now, let's look at the most simple overview before moving on to *dplyr* verbs: number of rows and columns.

In R, there is **dim** while pandas has **shape**:

```
# R
dim(df)
## [1] 344   8
# Python
r.df.shape
## (344, 8)
```

Subsetting rows and columns

For extracting subsets of rows and columns, *dplyr* has the verbs **filter** and **select**, respectively. For instance, let's look at the species and sex of the penguins with the shortest bills:

```
# R
df %>%
  filter(bill_length_mm < 34) %>%
  select(species, sex, bill_length_mm)
## # A tibble: 3 x 3
##   species sex    bill_length_mm
##
## 1 Adelie  female           33.5
## 2 Adelie  female           33.1
## 3 Adelie  female           32.1
```

All of those are female Adelie penguins. This is good news for a potential species classifier.

In pandas, there are several ways to achieve the same result. All of them are a little more complicated than our two *dplyr* verbs. The most pythonic way is probably to use the **loc** operator like such:

```
# Python
r.df.loc[r.df.bill_length_mm < 34, ['species', 'sex',
'bill_length_mm']]
##     species     sex  bill_length_mm
## 70   Adelie  female            33.5
## 98   Adelie  female            33.1
## 142  Adelie  female            32.1
```

This indexing via the "[]" brackets is essentially the same as in base R:

```
# R
na.omit(df[df$bill_length_mm < 34, c('species', 'sex',
'bill_length_mm')])
## # A tibble: 3 x 3
##   species sex    bill_length_mm
##
## 1 Adelie  female           33.5
## 2 Adelie  female           33.1
## 3 Adelie  female           32.1
```

Another way is to use the pandas method **query** as an equivalent for **filter**. This allows us to

essentially use *dplyr*-style evaluation syntax. I found that in practice, **query** is often notably slower than the other indexing tools. This can be important if your're dealing with large datasets:

```
# Python
r.df.query("bill_length_mm < 34").loc[:, ['species', 'sex',
'bill_length_mm']]
##     species     sex  bill_length_mm
## 70   Adelie  female            33.5
## 98   Adelie  female            33.1
## 142  Adelie  female            32.1
```

In tidy R, the *dplyr* verb **select** can extract by value as well as by position. And for extracting rows by position there is **slice**. This is just an arbitrary subset:

```
# R
df %>%
  slice(c(1,2,3)) %>%
  select(c(4,5,6))
## # A tibble: 3 x 3
##   bill_depth_mm flipper_length_mm body_mass_g
##
## 1          18.7               181        3750
## 2          17.4               186        3800
## 3          18                 195        3250
```

In pandas, we would use bracket indexing again, but instead of **loc** we now need **iloc** (i.e. locating by index). This might be a good point to remind ourselves that Python starts counting from 0 and R starts from 1:

```
# Python
r.df.iloc[[0, 1, 2], [3, 4, 5]]
##    bill_depth_mm  flipper_length_mm  body_mass_g
## 0           18.7              181.0       3750.0
## 1           17.4              186.0       3800.0
## 2           18.0              195.0       3250.0
```

To emphasise again: pandas uses **loc** for conditional indexing and **iloc** for positional indexing. This takes some getting uses to, but this simple rule covers most of *pandas'* subsetting operations.

Retaining unique rows and removing duplicates can be thought of as another way of subsetting a data frame. In *dplyr*, there's the **distinct** function which takes as arguments the columns that are considered for identifying duplicated rows:

```
# R
df %>%
  slice(c(2, 4, 186)) %>%
  distinct(species, island)
## # A tibble: 2 x 2
##   species island
##
## 1 Adelie  Torgersen
## 2 Gentoo  Biscoe
```

In *pandas* we can achieve the same result via the **drop_duplicates** method:

```
r.df.iloc[[2, 4, 186], :].drop_duplicates(['species', 'island'])
##     species     island  bill_length_mm  bill_depth_mm
flipper_length_mm  \
## 2    Adelie  Torgersen            40.3           18.0
195.0
## 186  Gentoo     Biscoe            49.1           14.8
220.0
##
##      body_mass_g     sex    year
## 2         3250.0  female  2007.0
## 186       5150.0  female  2008.0
```

In order to retain only the two affected rows, like in the dplyr example above, you would have to select them using `.loc`.

Arrange and Sample

Let's deal with the arranging and sampling of rows in one fell swoop. In *dplyr*, we use **sample_n** (or **sample_frac**) to choose a random subset of `n` rows (or a fraction `frac` of rows). Ordering a row by its values uses the verb **arrange**, optionally with the **desc** tool to specific descending order:

```
# R
set.seed(4321)
df %>%
  select(species, bill_length_mm) %>%
  sample_n(4) %>%
  arrange(desc(bill_length_mm))
## # A tibble: 4 x 2
##   species   bill_length_mm
##
## 1 Chinstrap           47.5
## 2 Adelie              42.7
## 3 Adelie              40.2
## 4 Adelie              34.6
```

In *pandas*, random sampling is done through the **sample** function, which can take either a fraction or a number of rows. Here, we can also pass directly a random seed (called *random_state*), instead of defining it outside the function via R's **set.seed**. Arranging is called **sort_values** and we have to specify an *ascending* flag because *pandas* wants to be different:

```
# Python
r.df.loc[:, ['species', 'bill_length_mm']].\
  sample(n = 4, random_state = 4321).\
  sort_values('bill_length_mm', ascending = False)
##        species  bill_length_mm
## 305  Chinstrap            52.8
## 301  Chinstrap            52.0
## 291  Chinstrap            50.5
## 165     Gentoo            48.4
```

As you see, the *pandas* dot methods can be chained in a way reminiscent of the *dplyr* pipe

(`%>%`). The scope for this style is much narrower than the pipe, though. It only works for methods and attributes associated with the specific *pandas* data frame and its transformation results. Nevertheless, it has a certain power and elegance for pipe aficionados. When written accross multiple lines it requires the Python line continuation operator `\`.

Group By and Summarise

Here is where *dplyr* really shines. Grouping and summarising transformations fit seemlessly into any wrangling workflow by preserving the tidy `tibble` data format. The verbs are **group_by** and **summarise**. Let's compare average bill lengths among species to find that "Adelie" penguins have significantly shorter bills:

```
# R
df %>%
  group_by(species) %>%
  summarise(mean_bill_length = mean(bill_length_mm, na.rm = TRUE),
            sd_bill_length = sd(bill_length_mm, na.rm = TRUE))
## # A tibble: 3 x 3
##   species   mean_bill_length sd_bill_length
##
## 1 Adelie                38.8           2.66
## 2 Chinstrap             48.8           3.34
## 3 Gentoo                47.5           3.08
```

This is consistent to what we had seen before for the 3 rows with shortes bills. The results also indicate that it would be much harder to try to separate "Chinstrap" vs "Gentoo" by `bill_length`.

In terms of usage, *pandas* is similar: we have **groupby** (without the "_") to define the grouping, and **agg** to aggregate (i.e. summarise) according to specific measures for certain features:

```
# Python
r.df.groupby('species').agg({'bill_length_mm': ['mean', 'std']})
##           bill_length_mm
##                     mean       std
## species
## Adelie         38.791391  2.663405
## Chinstrap      48.833824  3.339256
## Gentoo         47.504878  3.081857
```

This is pretty much the default way in *pandas*. It is worth noting that *pandas*, and Python in general, gets a lot of milage out of 2 data structures: **lists** like `[1, 2, 3]` which are the equivalent of `c(1, 2, 3)`, and **dictionaries** `{'a': 1, 'b': 2}` which are sets of key-value pairs. Values in a dictionary can be pretty much anything, including lists or dictionaries. (Most uses cases for dictionaries are probably served by *dplyr* tibbles.) Here we use a dictionary to define which column we want to summarise and how. Note again the chaining via dots.

The outcome of this operation, however, is slightly different from *dplyr* in that we get a hierarchical data frame with a categorical index (this is no longer a "normal" column") and 2 data columns that both have the designation `bill_length_mm`:

```
# Python
r.df.groupby('species').agg({'bill_length_mm': ['mean', 'std']}).info()
```

```
##
## CategoricalIndex: 3 entries, Adelie to Gentoo
## Data columns (total 2 columns):
## (bill_length_mm, mean)    3 non-null float64
## (bill_length_mm, std)     3 non-null float64
## dtypes: float64(2)
## memory usage: 155.0 bytes
```

You will see the limitations of that format if you try to chain another method. To get something instead that's more closely resembling our *dplyr* output, here is a different way: we forego the dictionary in favour of a simple list, then add a suffix later, and finally reset the index to a normal column:

```
# Python
r.df.groupby('species').bill_length_mm.agg(['mean',
'std']).add_suffix("_bill_length").reset_index()
##      species  mean_bill_length  std_bill_length
## 0     Adelie         38.791391         2.663405
## 1  Chinstrap         48.833824         3.339256
## 2     Gentoo         47.504878         3.081857
```

Now we can treat the result like a typical data frame. Note, that here we use another form of column indexing to select `bill_length_mm` after the `groupby`. This shorthand, which treats a feature as an object attribute, only works for single columns. Told you that *pandas* indexing can be a little confusing.

Joining and binding

There's another source of *dplyr* vs *pandas* confusion when it comes to SQL-style joins and to binding rows and columns. To demonstrate, we'll create an additional data frame which holds the mean bill length by species. And we pretend that it's a separate table. In terms of analysis steps, this crosses over from EDA into feature engineering territory, but that line can get blurry if you're exploring a dataset's hidden depths.

```
# R
df2 <- df %>%
  group_by(species) %>%
  summarise(mean_bill = mean(bill_length_mm, na.rm = TRUE))

df2
## # A tibble: 3 x 2
##   species   mean_bill
##
## 1 Adelie         38.8
## 2 Chinstrap      48.8
## 3 Gentoo         47.5
```

The *dplyr* verbs for SQL-like joins are very similar to the various SQL flavours. We have **left_join**, **right_join**, **inner_join**, **outer_join**; as well as the very useful filtering joins **semi_join** and **anti_join** (keep and discard what matches, respectively):

```
# R
set.seed(4321)
df %>%
```

```
  left_join(df2, by = "species") %>%
  select(species, bill_length_mm, mean_bill) %>%
  sample_n(5)
## # A tibble: 5 x 3
##   species   bill_length_mm mean_bill
##
## 1 Adelie              42.7      38.8
## 2 Chinstrap           47.5      48.8
## 3 Adelie              40.2      38.8
## 4 Adelie              34.6      38.8
## 5 Gentoo              53.4      47.5
```

Now we can for instance subtract the mean bill length from the individual values to get the residuals or to standardise our features.

In *pandas*, joining uses the **merge** operator. You have to specify the type of join via the *how* parameter and the join key via *on*, like such:

```
# Python
r.df.\
  sample(n = 5, random_state = 24).\
  merge(r.df2, how = "left", on = "species").\
  loc[:, ['species', 'bill_length_mm', 'mean_bill']]
##      species  bill_length_mm  mean_bill
## 0     Gentoo            43.4  47.504878
## 1  Chinstrap            51.7  48.833824
## 2     Gentoo            46.2  47.504878
## 3     Adelie            43.1  38.791391
## 4     Adelie            41.3  38.791391
```

Now, to bring in the aforementioned source of confusion: *pandas* also has an operator called **join** which joins by matching indeces, instead of columns, between two tables. This is a pretty pandas-specific convenience shortcut, since it relies on the data frame index. In practice I recommend using **merge** instead. The little conveniece provided by **join** is not worth the additional confusion.

Binding rows and columns in *dplyr* uses **bind_rows** and **bind_cols**. For the rows, there's really not much of a practical requirement other than that some of the column names match and that those ideally have the same type (which is not strictly necessary):

```
# R
head(df, 2) %>%
  bind_rows(tail(df, 2) %>% select(year, everything(), -sex))
## # A tibble: 4 x 8
##   species island bill_length_mm bill_depth_mm flipper_length_…
body_mass_g
##
## 1 Adelie  Torge…           39.1          18.7              181
3750
## 2 Adelie  Torge…           39.5          17.4              186
3800
## 3 Chinst… Dream            50.8          19                210
4100
```

```
## 4 Chinst… Dream            50.2          18.7              198
3775
## # … with 2 more variables: sex , year
```

For columns, it is necessary that the features we're binding to a tibble have the same number of rows. The big assumption here is that the row orders match (but that can be set beforehand):

```
# R
select(df, species) %>%
  bind_cols(df %>% select(year)) %>%
  head(5)
## # A tibble: 5 x 2
##   species  year
##
## 1 Adelie   2007
## 2 Adelie   2007
## 3 Adelie   2007
## 4 Adelie   2007
## 5 Adelie   2007
```

The *pandas* equivalent to **bind_rows** is **append**. This nomenclature is borrowed from Python's overall reliance on list operations.

```
# Python
r.df.head(2).append(r.df.tail(2).drop("sex", axis = "columns"), sort =
False)
##        species     island  bill_length_mm  bill_depth_mm
flipper_length_mm  \
## 0       Adelie  Torgersen            39.1           18.7
181.0
## 1       Adelie  Torgersen            39.5           17.4
186.0
## 342  Chinstrap      Dream            50.8           19.0
210.0
## 343  Chinstrap      Dream            50.2           18.7
198.0
##
##      body_mass_g     sex    year
## 0         3750.0    male  2007.0
## 1         3800.0  female  2007.0
## 342       4100.0     NaN  2009.0
## 343       3775.0     NaN  2009.0
```

Here we also demonstrate how to **drop** a column from a data frame (i.e. the equivalent to *dplyr* **select** with a minus).

Binding *pandas* columns uses **concat**, which takes a Python list as its parameter:

```
# Python
pd.concat([r.df.loc[:, 'species'], r.df.loc[:, 'year']], axis =
"columns").head(5)
##   species    year
## 0  Adelie  2007.0
## 1  Adelie  2007.0
```

```
## 2  Adelie  2007.0
## 3  Adelie  2007.0
## 4  Adelie  2007.0
```

And that's it for the basics!

Of course, there are a number of intricacies and special cases here, but by and large this should serve as a useful list of examples to get you started in *pandas* coming from *dplyr*, and perhaps also vice versa. Future installments in this series will go beyond *dplyr*, but likely also touch back on some aspect of it that are easy to get lost in translation.