# week3_lecture

July 13, 2020

## 1 Week 3 - Data handling: pandas and numpy

To obtain the course materials for today, please re-visit Sam's notes in week's 2 lecture.

The Python modules `pandas` and `numpy` are useful tools to handle datasets and apply basic operations on them.

Some of the things we learnt in weeks 1 and 2 using native Python (e.g. accessing, working with and writing data files) can often be easily achieved using `pandas` instead. This module offers data structures and operations for manipulating different types of datasets - see documentation.

Due to time reasons we will only cover `pandas` today, however feel free to explore `numpy` at your own pace e.g. following this tutorial and combining it with what you learn of `pandas` today. Feel free to ask for input/help from your tutor.

### 1.0.1 Learning objectives

- Learn key functions in `pandas` for manipulating your dataset
  - Create, read and write datasets
  - Selecting a subset of variables, i.e. selecting columns of your dataset
  - Selecting observations based on their values, i.e. selecting rows in your dataset
  - Sort observations in your dataset
  - Create new columns or modify existing ones
  - Summarise and collapse values in one or more columns to a single summary value
  - Handling missing data
  - Merge datasets

### 1.0.2 Installing the modules

The modules `pandas` and `numpy` do not come as part of the default Anaconda installation. In order to install them in your system, launch the "Anaconda Prompt (Anaconda3)" program and run the following command: `conda install pandas`.

The module `numpy` should install automatically with the `pandas` installation, otherwise run `conda install numpy` as well.

### 1.0.3 Loading modules

Once they are installed, we can import them using the aliases `pd` and `np` as follows:

[ ]:
```
import pandas as pd
import numpy as np
```

### 1.0.4 Reading datasets with `pandas`

We are going to use the METABRIC dataset `metabric_clinical_and_expression_data.csv` containing information about breast cancer patients as we did in weeks 1 and 2.

Pandas allows importing data from various file formats such as csv, xls, json, sql ...

To read a csv file, use the method `.read_csv()`:

[ ]:
```
metabric = pd.read_csv("../data/metabric_clinical_and_expression_data.csv")
metabric
```

If you forget to include `../data/` above, or if you include it but your copy of the file is saved somewhere else, you will get an error that ends with a line like this: `FileNotFoundError: File b'metabric_clinical_and_expression_data.csv' does not exist`

Generally, rows in a `DataFrame` are the **observations** (patients in the case of METABRIC) whereas columns are known as the observed **variables** (Cohort, Age_at_diagnosis ...).

Looking at the column on the far left, you can see the row names of the DataFrame `metabric` assigned using the known 0-based indexing used in Python.

Note that the `.read_csv()` method is not limited to reading csv files. For example, you can also read Tab Separated Value (TSV) files by adding the argument `sep='\t'`.

### 1.0.5 Exploring data

The pandas DataFrame object borrows many features from R's `data.frame` or SQL's `table`. They are 2-dimensional tables whose columns can contain different data types (e.g. boolean, integer, float, categorical/factor). Both the rows and columns are indexed, and can be referred to by number or name.

An index in a DataFrame refers to the position of an element in the data structure. Using the `.info()` method, we can view basic information about our DataFrame object:

[ ]:
```
metabric.info()
```

As expected, our object is a `DataFrame` (or, to use the full name that Python uses to refer to it internally, a `pandas.core.frame.DataFrame`).

[ ]:
```
type(metabric)
```

It has 1904 rows (the patients) and 32 columns. The columns consist of integer, floats and strings. It uses almost 500 KB of memory.

As mentioned, a DataFrame is a Python object or data structure, which means it can have **attributes** and **methods**.

**Attributes** contain information about the object. You can access them to learn more about the contents of your DataFrame. To do this, use the object variable name `metabric` followed by the attribute name, separated by a '.'. Do not use any () to access attributes.

For example, the types of data contained in the columns are stored in the `.dtypes` attribute:

[ ]:
```
metabric.dtypes
```

You can access the dimensions of your DataFrame using the `.shape` attribute. The first value is the number of rows, and the second the number of columns:

[ ]:
```
metabric.shape
```

The row and column names can be accessed using the attributes `.index` and `.columns` respectively:

[ ]:
```
metabric.index
```

[ ]:
```
metabric.columns.values
```

Transposing `metabric`:

[ ]:
```
metabric.T
```

**Methods** are functions that are associated with a DataFrame. Because they are functions, you do use () to call them, and can add arguments inside the parentheses to control their behaviour. For example, the `.info()` command we executed previously was a method.

The `.head()` method prints the first few rows of the table, while the `.tail()` method prints the last few rows:

[ ]:
```
metabric.head()
```

[ ]:
```
metabric.head(3)
```

[ ]:
```
metabric.tail()
```

The `.describe()` method computes summary statistics for the columns (including the count, mean, median, and std):

[ ]:
```
metabric.describe()
```

We often want to calculate summary statistics grouped by subsets or attributes within fields of our data. For example, we might want to calculate the average survival time for patients with an advanced tumour stage.

If you type the name of a DataFrame followed by e.g. `metabric.<TAB>` a display menu will allow you to find columns, attributes and methods for the DataFrame.

There are two ways to access columns in a DataFrame. The first is using a '.' followed by the name of the column. The second is using square brackets:

[ ]:
```
metabric.Survival_time
```

[ ]:
```
metabric['Survival_time']
```

We can also compute metrics on specific columns or on the entire DataFrame:

[ ]:
```
metabric['Survival_time'].mean()
```

[ ]:
```
metabric['Survival_time'].std()
```

[ ]:
```
metabric.mean()
```

### 1.0.6 Selecting columns and rows

The pandas cheat sheet can be very helpful for recalling basic pandas operations.

To select rows and columns in a DataFrame, we use square brackets [ ]. There are two ways to do this: with **positional** indexing, which uses index numbers, and **label-based** indexing which uses column or row names.

To select the first three rows using their numeric index:

[ ]:
```
metabric[:3]
```

To select one column using its name:

[ ]:
```
metabric['Mutation_count']
```

And we can combine the two like this:

[ ]:
```
metabric[:3]['Mutation_count']
```

However the following does not work:

[ ]:
```
metabric[:3,'Mutation_count']
```

To do **positional** indexing for both rows and columns, use `.iloc[]`. The first argument is the numeric index of the rows, and the second the numeric index of the columns:

[ ]:
```
metabric.iloc[:3,2]
```

The colon ':' defines a range

For **label-based** indexing, use `.loc[]` with the column and row names:

```
[ ]: metabric.loc[:3,"Age_at_diagnosis"]
```

Note: because the rows have numeric indices in this DataFrame, we may think that selecting rows with `.iloc[]` and `.loc[]` is same. As observed above, this is not the case.

```
[ ]: metabric.loc[:3, ['Cohort', 'Chemotherapy']]
```

```
[ ]: metabric.loc[:3, 'Cohort':'Chemotherapy']
```

### 1.0.7 Filtering rows

You can choose rows from a DataFrame that match some specified criteria. The criteria are based on values of variables and can make use of comparison operators such as `==`, `>`, `<` and `!=`.

For example, to filter `metabric` so that it only contains observations for those patients who died of breast cancer:

```
[ ]: metabric[metabric.Vital_status=="Died of Disease"]
```

To filter based on more than one condition, you can use the operators `&` (and), `|` (or).

```
[ ]: metabric[(metabric.Vital_status=="Died of Disease") & (metabric.
     ↪Age_at_diagnosis>70)]
```

For categorical variables e.g. `Vital_status` or `Cohort`, it may be useful to count how many occurrences there is for each category:

```
[ ]: metabric['Vital_status'].unique()
```

```
[ ]: metabric['Vital_status'].value_counts()
```

To filter by more than one category, use the `.isin()` method.

```
[ ]: metabric[metabric.Vital_status.isin(['Died of Disease', 'Died of Other␣
     ↪Causes'])]
```

```
[ ]: metabric['Cohort'].value_counts()
```

Two tabulate two categorical variables just like `table` in R, use the function `.crosstab()`:

```
[ ]: pd.crosstab(metabric['Vital_status'], metabric['Cohort'])
```

### 1.0.8 Define new columns

To obtain the age of the patient today `Age_today` (new column) based on the `Age_at_diagnosis` (years) and the `Survival_time` (days), you can do the following:

```
[ ]: metabric['Age_today'] = metabric['Age_at_diagnosis'] +␣
     ↪metabric['Survival_time']/365
     metabric
```

### 1.0.9 Sort data

To sort the entire DataFrame according to one of the columns, we can use the `.sort_values()` method. We can store the sorted DataFrame using a new variable name such as `metabric_sorted`:

```
[ ]: metabric_sorted = metabric.sort_values('Tumour_size')
     metabric_sorted
```

```
[ ]: metabric_sorted.iloc[0]
```

```
[ ]: metabric_sorted.loc[0]
```

We can also sort the DataFrame in descending order:

```
[ ]: metabric_sorted = metabric.sort_values('Tumour_size', ascending=False)
     metabric_sorted
```

### 1.0.10 Missing data

Pandas primarily uses `NaN` to represent missing data, which are by default not included in computations.

The `.info()` method shown above already gave us a way to find columns containing missing data:

```
[ ]: metabric.info()
```

To get the locations where values are missing:

```
[ ]: pd.isna(metabric)
```

```
[ ]: metabric.isnull()
```

To drop any rows containing at least one column with missing data:

```
[ ]: metabric.dropna()
```

Define in which columns to look for missing values before dropping the row:

```
[ ]: metabric.dropna(subset = ["Tumour_size"])
```

```
[ ]: metabric.dropna(subset = ["Tumour_size", "Tumour_stage"])
```

Filling missing data:

```
[ ]: metabric.fillna(value=0)
```

```
[ ]: metabric.fillna(value={'Tumour_size':0, 'Tumour_stage':5})
```

### 1.0.11 Grouping

Grouping patients by Cohort and then applying the `.mean()` function to the resulting groups:

```
[ ]: metabric.groupby('Cohort').mean()
```

Grouping by multiple columns forms a hierarchical index, and again we can apply the `.mean()` function:

```
[ ]: metabric.groupby(['Cohort', 'Vital_status']).mean()
```

### 1.0.12 Pivoting

In some cases, you may want to re-structure your existing DataFrame. The function `.pivot_table()` is useful for this:

```
[ ]: df = pd.DataFrame({'A': ['one', 'one', 'two', 'three'] * 3, 'B': ['A', 'B',␣
     ↪'C'] * 4, 'C': ['foo', 'foo', 'foo', 'bar', 'bar', 'bar'] * 2, 'D': np.
     ↪random.randn(12), 'E': np.random.randn(12)})
     df
```

```
[ ]: pd.pivot_table(df, values='D', index=['A', 'B'], columns=['C'])
```

### 1.0.13 Merge datasets

You can concatenate DataFrames using the function `concat()`:

```
[ ]: metabric_cohort1 = metabric[metabric["Cohort"]==1]
     metabric_cohort1
```

```
[ ]: metabric_cohort2 = metabric[metabric["Cohort"]==2]
     metabric_cohort2
```

```
[ ]: pd.concat([metabric_cohort1,metabric_cohort2])
```

Or join datasets using the function `.merge()`:

```
[ ]: left = pd.DataFrame({'key': ['foo', 'foo'], 'lval': [1, 2]})
     left
```

[ ]:
```
right = pd.DataFrame({'key': ['foo', 'foo'], 'rval': [4, 5]})
right
```

[ ]:
```
pd.merge(left, right, on='key')
```

A final example:

[ ]:
```
left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1, 2]})
left
```

[ ]:
```
right = pd.DataFrame({'key': ['foo', 'bar'], 'rval': [4, 5]})
right
```

[ ]:
```
pd.merge(left, right, on='key')
```

### 1.0.14 Assignment

**Exercise 1**

- Read the dataset `metabric_clinical_and_expression_data.csv` and store its summary statistics into a new variable called `metabric_summary`.
- Just like the `.read_csv()` method allows reading data from a file, `pandas` provides a `.to_csv()` method to write `DataFrames` to files. Write your summary statistics object into a file called `metabric_summary.csv`. You can use `help(metabric.to_csv)` to get information on how to use this function.
- Use the help information to modify the previous step so that you can generate a Tab Separated Value (TSV) file instead
- Similarly, explore the method `to_excel()` to produce an excel spreadsheet containing summary statistics

**Exercise 2**

- Read the dataset `metabric_clinical_and_expression_data.csv` into a variable e.g. `metabric`.
- Calculate mean tumour size of patients grouped by vital status and tumour stage
- Find the cohort of patients and tumour stage where the average expression of genes TP53 and FOXA1 is highest
- Do patients with greater tumour size live longer? How about patients with greater tumour stage? How about greater Nottingham_prognostic_index?

**Exercise 3 (bonus)** Review the section on missing data presented in the lecture. Consulting the user's guide section dedicated to missing data if necessary use the functionality provided by pandas to answer the following questions:

- Which variables (columns) of the metabric dataset have missing data?
- Find the patients ids who have missing tumour size and/or missing mutation count data. Which cohorts do they belong to?

- For the patients identified to have missing tumour size data for each cohort, calculate the average tumour size of the patients with tumour size data available within the same cohort to fill in the missing data