

# Scientific Programming

## Practical 8

---

### Introduction

Luca Bianco - Academic Year 2020-21  
luca.bianco@fmach.it

# Announcements

Two announcements:

**Gabriele Masina** is the new tutor for this course

**Midterm:** on Wednesday, November 4th - 11.30 - 13.30 online

(simulation of the midterm on Monday, November 2nd - 14.30 - 16.30 online)



→

MDC001115.177  
MDC013284.379  
MDC018185.243  
MDC018185.241  
MDC004527.213  
MDC012176.157  
MDC001204.810

```

"""exercises/filterFasta.py"""

import argparse

def readIDS(f):
    """reads a one column file in and stores
    the ids in a dictionary that is returned at the end"""
    ret = dict()
    with open(f, "r") as file:
        for line in file:
            line = line.strip()
            if (line not in ret):|
                ret[line] = 1
    return ret

def filterFasta(inF, outF, ids2keep):
    oF = open(outF, "w")

    outputME = False
    with open(inF, "r") as file:
        for line in file:
            line = line.strip()
            if line.startswith(">"):
                #this is the header
                if ids2keep.get(line[1:],False):
                    oF.write(line + "\n")
                    outputME = True
                    print("Writing contig ", line[1:])
                else:
                    outputME = False
            else:
                if outputME:
                    oF.write(line + "\n")

    oF.close()

parser = argparse.ArgumentParser(description="Filters a fasta file")
parser.add_argument("inputFasta", type = str, help = "The input fasta file")
parser.add_argument("inputIDS", type = str, help = "The IDS to keep")
parser.add_argument("outputFasta", type = str,
                    help = "The output fasta file with filtered entries")

args = parser.parse_args()
idsFile = args.inputIDS
inFasta = args.inputFasta
outFasta = args.outputFasta

ids = readIDS(idsFile)
filterFasta(inFasta,outFasta, ids)

```

# Exercise 1

```
biancol@bludell:~/work/courses/QCBsciprolab2020$ python3 exercises/filterFasta.py --help
usage: filterFasta.py [-h] inputFasta inputIDS outputFasta

Filters a fasta file

positional arguments:
  inputFasta  The input fasta file
  inputIDS    The IDS to keep
  outputFasta The output fasta file with filtered entries

optional arguments:
  -h, --help  show this help message and exit
biancol@bludell:~/work/courses/QCBsciprolab2020$ python3 exercises/filterFasta.py file_samples/contigs82.fasta file_samples/contig_ids.txt file_samples/filtered_contigs.fasta
Writing contig MDC001115.177
Writing contig MDC013284.379
Writing contig MDC018185.243
Writing contig MDC018185.241
Writing contig MDC004527.213
Writing contig MDC012176.157
Writing contig MDC001204.810
Writing contig MDC004389.256
Writing contig MDC018297.229
Writing contig MDC001802.364
```

```
biancol@bludell:~/work/courses/QCBsciprolab2020$ cat file_samples/filtered_contigs.fasta | grep ">"
>MDC001115.177
>MDC013284.379
>MDC018185.243
>MDC018185.241
>MDC004527.213
>MDC012176.157
>MDC001204.810
>MDC004389.256
>MDC018297.229
>MDC001802.364
>MDC014057.243
>MDC021015.302
>MDC017187.314
>MDC012865.410
```

# Pandas

Pandas (**panel-data**) is a very efficient library to deal with **numerical tables** and time series

## Two data structures:

**Series:** 1D tables

**DataFrames:** 2D tables

<https://pandas.pydata.org/>

# Series

Series are 1-dimensional structures (like lists) containing data. Series are characterized by two types of information: the **values** and the **index** (a list of labels associated to the data). A bit like **list** and a bit like **dictionary**!

```
Values and index explicitly defined
A    15
B     7
C    20
D     3
E    15
F     1
G     5
H    17
I    15
L    17
dtype: int64
The index: Index(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'L'], dtype='object')
The values: [15  7 20  3 15  1  5 17 15 17]
-----

From dictionary
forty    40
four     4
one       1
ten      10
three     3
two       2
dtype: int64
Index(['forty', 'four', 'one', 'ten', 'three', 'two'], dtype='object')
[40  4  1 10  3  2]
```

```
import pandas as pd
import random

print("Values and index explicitly defined")
#values and index explicitly defined
S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("The index:", S.index)
print("The values:", S.values)

print("-----\n")
print("From dictionary")
#from a dictionary
S1 = pd.Series({"one" : 1, "two" : 2, "ten": 10,
               "three" : 3, "four": 4, "forty" : 40})

print(S1)
print(S1.index)
print(S1.values)
```

# Series

Series are 1-dimensional structures (like lists) containing data. Series are characterized by two types of information: the **values** and the **index** (a list of labels associated to the data). A bit like **list** and a bit like **dictionary**!

If not specified, the index is added by default

Default index

```
0    8
1    2
2    8
3   10
4    1
5    5
6    3
7    8
8    9
9    5
```

dtype: int64

RangeIndex(start=0, stop=10, step=1)

```
[ 8  2  8 10  1  5  3  8  9  5]
```

Same value repeated

```
0    1.27
1    1.27
2    1.27
3    1.27
4    1.27
5    1.27
6    1.27
7    1.27
8    1.27
9    1.27
```

dtype: float64

RangeIndex(start=0, stop=10, step=1)

```
[1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27 1.27]
```

```
print("Default index")
#index added by default
myData = [random.randint(0,10) for x in range(10)]
S2 = pd.Series(myData)
```

```
print(S2)
print(S2.index)
print(S2.values)
```

```
print("-----\n")
print("Same value repeated")
S3 = pd.Series(1.27, range(10))
print(S3)
print(S3.index)
print(S3.values)
```

# Series

Data in a series can be accessed by using the **label** (i.e. the index) as in a dictionary or through its **position** as in a list. Slicing is also allowed both by **position** and **index**.

In the latter case, `Series[S:E]` with **S** and **E** **indexes**, both **S** and **E** **are included**.

```
import pandas as pd
import random

#values and index explicitly defined
S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("")

print("Value at label \"A\":", S["A"])
print("Value at index 1:", S[1])
print("")

print("Slicing from 1 to 3:") #note 3 excluded
print(S[1:3])
print("")
print("Slicing from C to H:") #note H included!
print(S["C":"H"])
print("")

print("Retrieving from list:")
print(S[[1,3,5,7,9]])
print(S[["A","C","E","G"]])
print("")

print("Top 3")
print(S.head(3))
print("")
print("Bottom 3")
print(S.tail(3))
```

```
A    15
B    11
C     4
D     7
E     1
F     4
G    15
H    14
I    14
L    17
dtype: int64
```

```
Value at label "A": 15
Value at index 1: 11
```

```
Slicing from 1 to 3:
B    11
C     4
dtype: int64
```

```
Slicing from C to H:
C     4
D     7
E     1
F     4
G    15
H    14
dtype: int64
```



# Series

Data in a series can be accessed by using the **label** (i.e. the index) as in a dictionary or through its **position** as in a list. Slicing is also allowed both by **position** and **index**.

In the latter case, `Series[S:E]` with **S** and **E** **labels**, both **S** and **E** are **included**.

```
import pandas as pd
import random

#values and index explicitly defined
S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("")

print("Value at label \"A\":", S["A"])
print("Value at index 1:", S[1])
print("")

print("Slicing from 1 to 3:") #note 3 excluded
print(S[1:3])
print("")
print("Slicing from C to H:") #note H included!
print(S["C":"H"])
print("")

print("Retrieving from list:")
print(S[[1,3,5,7,9]])
print(S[["A","C","E","G"]])
print("")

print("Top 3")
print(S.head(3))
print("")
print("Bottom 3")
print(S.tail(3))
```

Retrieving from list:

```
B    11
D     7
F     4
H    14
L    17
dtype: int64
A    15
C     4
E     1
G    15
dtype: int64
```

Top 3

```
A    15
B    11
C     4
dtype: int64
```

Bottom 3

```
H    14
I    14
L    17
dtype: int64
```

# Series

**Example:** Given a list of 10 integers, we want to divide them by 2.

Important operations on series:

## Operator broadcasting

Operations can automatically be broadcast to the entire Series. This is a quite cool feature and **saves us from looping through the elements of the Series.**

Without pandas:

```
import random

listS = [random.randint(0,20) for x in range(0,10)]

print(listS)

for el in range(0,len(listS)):
    listS[el] /=2  #compact of X = X / 2

print(listS)
```

[6, 4, 5, 19, 14, 16, 9, 3, 13, 11]  
[3.0, 2.0, 2.5, 9.5, 7.0, 8.0, 4.5, 1.5, 6.5, 5.5]

# Series

**Example:** Given a list of 10 integers, we want to divide them by 2.

Important operations on series:

## Operator broadcasting

Operations can automatically be broadcast to the entire Series. This is a quite cool feature and **saves us from looping through the elements of the Series.**

```
A    4
B   13
C   14
D    6
E   13
F    2
G   13
H   19
I   20
L    7
dtype: int64
```

```
A    2.0
B    6.5
C    7.0
D    3.0
E    6.5
F    1.0
G    6.5
H    9.5
I   10.0
L    3.5
dtype: float64
```

With pandas (operator broadcasting):

```
import pandas as pd
import random

S = pd.Series([random.randint(0,20) for x in range(0,10)],
               index = list("ABCDEFGHIL"))

print(S)
print("")
S1 = S / 2
print(S1)
```

# Series

Important operations on series:

## Operator broadcasting

## Filtering

We can also apply boolean operators to obtain only the **sub-Series** with all the values satisfying a specific condition. This allows us to **filter** the Series.

```
import pandas as pd
import random

S = pd.Series([random.randint(0,20) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print(S)
print("")
S1 = S>10
print(S1)
print("")
S2 = S[S > 10]
print(S2)
```

```
A    3
B    3
C   18
D    1
E   12
F   11
G    4
H   11
I    5
L   14
dtype: int64
```

```
A    False
B    False
C     True
D    False
E     True
F     True
G    False
H     True
I    False
L     True
dtype: bool
```

```
C    18
E    12
F    11
H    11
L    14
dtype: int64
```



series of True and False  
where condition is/is not  
met

# Series

Important operations on series:

**Operator broadcasting**

**Filtering**

**Computing stats**

```
import pandas as pd
import random

S = pd.Series([random.randint(0,10) for x in range(0,10)],
              index = list("ABCDEFGHIL"))

print("The data:")
print(S)
print("")
print("Its description")
print(S.describe())
print("")
print("Specifying different quantiles:")
print(S.quantile([0.1,0.2,0.8,0.9]))
print("")
print("Histogram:")
print(S.value_counts())
print("")
print("The type is a Series:")
print(type(S.value_counts()))
print("Summing the values:")
print(S.sum())
print("")
print("The cumulative sum:")
print(S.cumsum())
```

see notes for the complete  
results and other features  
like `Series.fillna(values)`

The data:

|   |    |
|---|----|
| A | 3  |
| B | 10 |
| C | 3  |
| D | 4  |
| E | 9  |
| F | 9  |
| G | 5  |
| H | 0  |
| I | 10 |
| L | 5  |

dtype: int64

Its description

|       |           |
|-------|-----------|
| count | 10.000000 |
| mean  | 5.800000  |
| std   | 3.489667  |
| min   | 0.000000  |
| 25%   | 3.250000  |
| 50%   | 5.000000  |
| 75%   | 9.000000  |
| max   | 10.000000 |

dtype: float64

Specifying different quantiles:

|     |      |
|-----|------|
| 0.1 | 2.7  |
| 0.2 | 3.0  |
| 0.8 | 9.2  |
| 0.9 | 10.0 |

dtype: float64

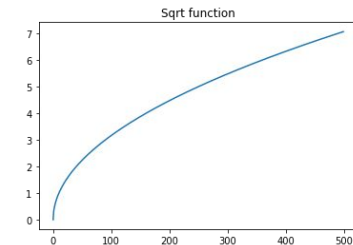
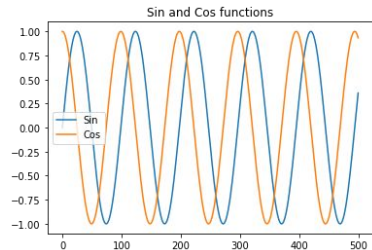
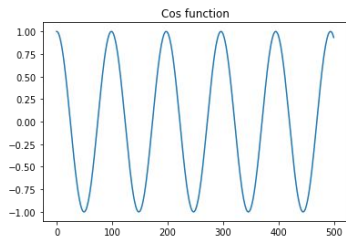
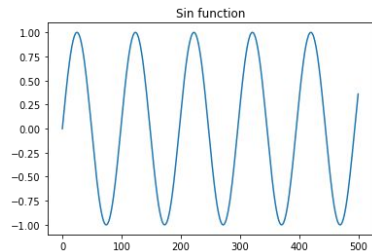
Histogram:

|    |   |
|----|---|
| 10 | 2 |
| 9  | 2 |
| 5  | 2 |
| 3  | 2 |
| 4  | 1 |
| 0  | 1 |

dtype: int64

# Plotting data

It is quite easy to plot data in Series and DataFrames thanks to matplotlib



```
import math
import matplotlib.pyplot as plt
import pandas as pd

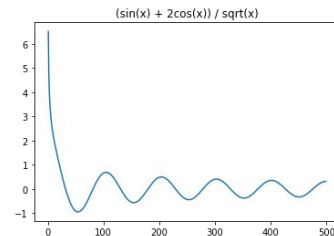
x = [i/10 for i in range(0,500)]

y = [math.sin(2*i/3.14) for i in x]
y1 = [math.cos(2*i/3.14) for i in x]
y2 = [math.sqrt(i) for i in x]
#print(x)

ySeries = pd.Series(y)
ySeries1 = pd.Series(y1)
ySeries2 = pd.Series(y2)

ySeries.plot()
plt.title("Sin function")
plt.show()
plt.close()
ySeries1.plot()
plt.title("Cos function")
plt.show()
plt.close()
plt.title("Sin and Cos functions")
ySeries.plot()
ySeries1.plot()
plt.legend(["Sin", "Cos"])
plt.show()
plt.close()

ySeries2.plot()
plt.title("Sqrt function")
plt.show()
plt.close()
ySeries2 = (ySeries + 2*ySeries1)/ySeries2
ySeries2.plot()
plt.title("(sin(x) + 2cos(x)) / sqrt(x)")
plt.show()
```



# DataFrames

2D analogous of Series.

They have an **index** and several **columns**.

**Data can be dishomogeneous.**

Most of the the things seen for Series apply to DataFrames

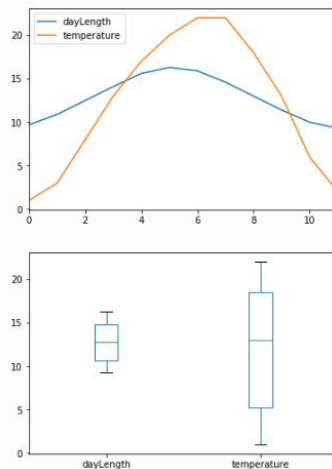
```
dayLength  temperature
Jan         9.7         1
Feb        10.9         3
Mar        12.5         8
Apr        14.1        13
May        15.6        17
Jun        16.3        20
Jul        15.9        22
Aug        14.6        22
Sep        13.0        18
Oct        11.4        13
Nov         9.0         6
Dec         9.3         2
Index(['dayLength', 'temperature'], dtype='object')
Index(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct',
       'Nov', 'Dec'],
      dtype='object')
```

```
import pandas as pd
import matplotlib.pyplot as plt

myData = {
    "temperature" : pd.Series([1, 3, 8, 13, 17, 20,
                              22, 22, 18, 13, 6, 2]),
    "dayLength" : pd.Series([9.7, 10.9, 12.5, 14.1, 15.6,
                            16.3, 15.9, 14.6, 13, 11.4, 10, 9.3])
}

DF = pd.DataFrame(myData)
print(DF)
print(DF.index)

DF.plot()
plt.show()
plt.close()
DF.plot(kind="box")
plt.show()
```





# DataFrames

We can **load external files**, extract info and apply operators, broadcasting and filtering...

Load from file



```
import pandas as pd
```

```
orders = pd.read_csv("file_samples/sampled_data_orders.csv", sep=";",  
index_col=0, header=0)
```

```
print("The Order Quantity column (top 5)")
```

```
print(orders["Order Quantity"].head(5))
```

```
print("")
```

```
print("The Sales column (top 10)")
```

```
print(orders.Sales.head(10))
```

```
print("")
```

```
print("The row with ID:50")
```

```
r50 = orders.loc[50]
```

```
print(r50)
```

```
print("")
```

```
print("The third row:")
```

```
print(orders.iloc[3])
```

```
print("The Order Quantity, Sales, Discount and Profit of the 2nd,  
4th, 6th and 8th row:")
```

```
print(orders[1:8:2][["Order Quantity", "Sales", "Discount", "Profit"]])
```

```
print("The Order Quantity, Sales, Discount and Profit of orders with  
discount > 10%:")
```

```
print(orders[orders["Discount"] > 0.1][["Order Quantity", "Sales",  
"Discount", "Profit"]])
```

1. Select by column `DataFrame[col]` returns a Series
2. Select by row label `DataFrame.loc[row_label]` returns a Series
3. Select row by integer location `DataFrame.iloc[row_position]` returns a Series
4. Slice rows `DataFrame[S:E]` (S and E are labels, both included) returns a DataFrame
5. Select rows by boolean vector `DataFrame[bool_vect]` returns a DataFrame

| Row ID | Sales      | Profit  | Product         | Category |
|--------|------------|---------|-----------------|----------|
| 1      | 261.5400   | -213.25 | Office Supplies |          |
| 49     | 10123.0200 | 457.81  | Office Supplies |          |
| 50     | 244.5700   | 46.71   | Office Supplies |          |
| 80     | 4965.7595  | 1198.97 | Technology      |          |
| 85     | 394.2700   | 30.94   | Office Supplies |          |
| 86     | 146.6900   | 4.43    | Furniture       |          |
| 97     | 93.5400    | -54.04  | Office Supplies |          |

see notes for results



# Merging DataFrames

```
pandas.merge(DataFrame1, DataFrame2, on="col_name", how="inner/outer/left/right")
```

## DFs1

|   |                | id  | type |
|---|----------------|-----|------|
| 0 | SNP_FB_0411211 | SNP |      |
| 1 | SNP_FB_0412425 | SNP |      |
| 2 | SNP_FB_0942385 | SNP |      |
| 3 | CH01f09        | SSR |      |
| 4 | Hi05f12x       | SSR |      |
| 5 | SNP_FB_0942712 | SNP |      |

## DFs2

|   | chr | id             |
|---|-----|----------------|
| 0 | 1   | SNP_FB_0411211 |
| 1 | 15  | SNP_FB_0412425 |
| 2 | 7   | SNP_FB_0942385 |
| 3 | 9   | CH01f09        |
| 4 | 1   | SNP_FB_0428218 |

1. how = inner : non-matching entries are discarded;
2. how = left : ids are taken from the first DataFrame;
3. how = right : ids are taken from the second DataFrame;
4. how = outer : ids from both are retained.

```
inJ = pd.merge(DFs1, DFs2, on = "id", how = "inner")  
print(inJ)
```

### Inner merge (only common in both)

|   | id             | type | chr |
|---|----------------|------|-----|
| 0 | SNP_FB_0411211 | SNP  | 1   |
| 1 | SNP_FB_0412425 | SNP  | 15  |
| 2 | SNP_FB_0942385 | SNP  | 7   |
| 3 | CH01f09        | SSR  | 9   |

### Right merge (IDS from DFs2)

|   | id             | type | chr |
|---|----------------|------|-----|
| 0 | SNP_FB_0411211 | SNP  | 1   |
| 1 | SNP_FB_0412425 | SNP  | 15  |
| 2 | SNP_FB_0942385 | SNP  | 7   |
| 3 | CH01f09        | SSR  | 9   |
| 4 | SNP_FB_0428218 | NaN  | 1   |

```
leftJ = pd.merge(DFs1, DFs2, on = "id", how = "left")  
print(leftJ)
```

### Left merge (IDS from DFs1)

|   | id             | type | chr |
|---|----------------|------|-----|
| 0 | SNP_FB_0411211 | SNP  | 1   |
| 1 | SNP_FB_0412425 | SNP  | 15  |
| 2 | SNP_FB_0942385 | SNP  | 7   |
| 3 | CH01f09        | SSR  | 9   |
| 4 | Hi05f12x       | SSR  | NaN |
| 5 | SNP_FB_0942712 | SNP  | NaN |

### Outer merge (IDS from both)

|   | id             | type | chr |
|---|----------------|------|-----|
| 0 | SNP_FB_0411211 | SNP  | 1   |
| 1 | SNP_FB_0412425 | SNP  | 15  |
| 2 | SNP_FB_0942385 | SNP  | 7   |
| 3 | CH01f09        | SSR  | 9   |
| 4 | Hi05f12x       | SSR  | NaN |
| 5 | SNP_FB_0942712 | SNP  | NaN |
| 6 | SNP_FB_0428218 | NaN  | 1   |

# Merging DataFrames

The columns we merge on do not necessarily need to be the same, we can specify a correspondence between the row of the first dataframe (the one on the left) and the second dataframe (the one on the right) specifying which columns must have the same values to perform the merge.

This can be done by using the parameters `right_on = column_name` and `left_on = column_name`

```
import pandas as pd
```

```
d = dict({"A" : [1,2,3,4], "B" : [3,4,73,13]})
d2 = dict({"E" : [1,4,3,13], "F" : [3,1,71,1]})
```

```
DF = pd.DataFrame(d)
DF2 = pd.DataFrame(d2)
```

```
merged_onBE = DF.merge(DF2, left_on = 'B', right_on = 'E', how = "inner")
merged_onAF = DF.merge(DF2, right_on = "F", left_on = "A", how = "outer")
print("DF:")
print(DF)
print("DF2:")
print(DF2)
print("\ninner merge on BE")
print(merged_onBE)
print("\nouter merge on AF:")
print(merged_onAF)
```

DF:

|   | A | B  |
|---|---|----|
| 0 | 1 | 3  |
| 1 | 2 | 4  |
| 2 | 3 | 73 |
| 3 | 4 | 13 |

DF2:

|   | E  | F  |
|---|----|----|
| 0 | 1  | 3  |
| 1 | 4  | 1  |
| 2 | 3  | 71 |
| 3 | 13 | 1  |

inner merge on BE

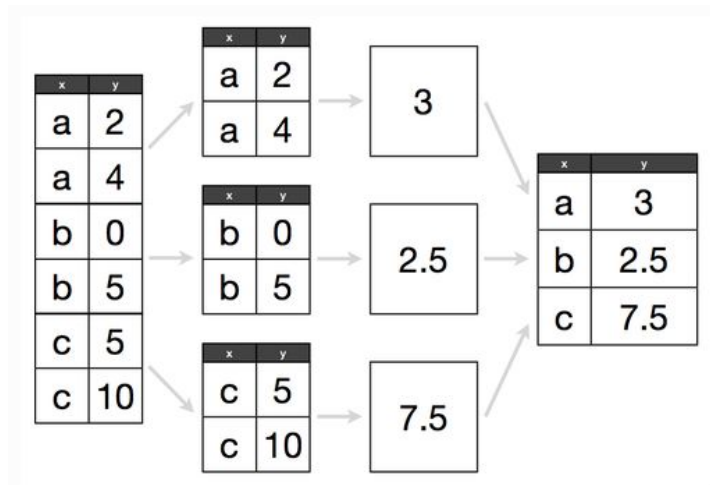
|   | A | B  | E  | F  |
|---|---|----|----|----|
| 0 | 1 | 3  | 3  | 71 |
| 1 | 2 | 4  | 4  | 1  |
| 2 | 4 | 13 | 13 | 1  |

outer merge on AF:

|   | A   | B    | E    | F    |
|---|-----|------|------|------|
| 0 | 1.0 | 3.0  | 4.0  | 1.0  |
| 1 | 1.0 | 3.0  | 13.0 | 1.0  |
| 2 | 2.0 | 4.0  | NaN  | NaN  |
| 3 | 3.0 | 73.0 | 1.0  | 3.0  |
| 4 | 4.0 | 13.0 | NaN  | NaN  |
| 5 | NaN | NaN  | 3.0  | 71.0 |

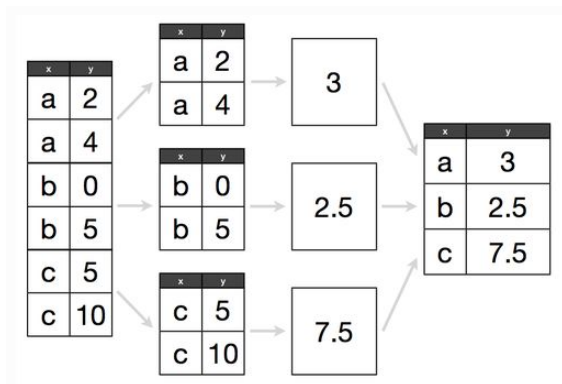
# Grouping DataFrames

The split-apply-aggregate  
paradigm



# Grouping DataFrames

## The split-apply-aggregate paradigm



```
import pandas as pd
```

```
test = {"x": ["a","a", "b", "b", "c", "c"],  
        "y": [2,4,0,5,5,10]}
```

```
DF = pd.DataFrame(test)
```

```
print(DF)
```

```
print("")
```

```
gDF = DF.groupby("x")
```

```
for i,g in gDF:
```

```
    print("Group: ", i)
```

```
    print(g)
```

```
    print(type(g))
```

```
aggDF = gDF.agg(pd.DataFrame.mean)
```

```
print(aggDF)
```

```
x  y  
0  a  2  
1  a  4  
2  b  0  
3  b  5  
4  c  5  
5  c 10
```

```
Group: a
```

```
x  y  
0  a  2  
1  a  4
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Group: b
```

```
x  y  
2  b  0  
3  b  5
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Group: c
```

```
x  y  
4  c  5  
5  c 10
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
y
```

```
x  
a  3.0  
b  2.5  
c  7.5
```

# Grouping DataFrames

## The split-apply-aggregate paradigm

```
import pandas as pd

test = {"x": ["a", "a", "b", "b", "c", "c"],
        "y": [2, 4, 0, 5, 5, 10]}

DF = pd.DataFrame(test)
print(DF)
print("")
gDF = DF.groupby("x")
for i, g in gDF:
    print("Group: ", i)
    print(g)
    print(type(g))

aggDF = gDF.agg(pd.DataFrame.mean)
print(aggDF)
```

```
#without looping through the groups...
print("\nThe 'a' group:")
print(gDF.get_group('a'))
print("\nThe 'c' group:")
print(gDF.get_group('c'))
```

The 'a' group:

|   | x | y |
|---|---|---|
| 0 | a | 2 |
| 1 | a | 4 |

The 'c' group:

|   | x | y  |
|---|---|----|
| 4 | c | 5  |
| 5 | c | 10 |

# Grouping DataFrames

| Row ID | Sales      | Profit  | Product Category |
|--------|------------|---------|------------------|
| 1      | 261.5400   | -213.25 | Office Supplies  |
| 49     | 10123.0200 | 457.81  | Office Supplies  |
| 50     | 244.5700   | 46.71   | Office Supplies  |
| 80     | 4965.7595  | 1198.97 | Technology       |
| 85     | 394.2700   | 30.94   | Office Supplies  |

Group: Furniture  
Group: Office Supplies  
Group: Technology

Count elements per category:

|                 |      |
|-----------------|------|
| Office Supplies | 4610 |
| Technology      | 2065 |
| Furniture       | 1724 |

Name: Product Category, dtype: int64

Total values:

| Product Category | Sales       | Profit    |
|------------------|-------------|-----------|
| Furniture        | 5178590.542 | 117433.03 |
| Office Supplies  | 3752762.100 | 518021.42 |
| Technology       | 5984248.182 | 886313.52 |

Mean values (sorted by profit):

| Product Category | Sales       | Profit     |
|------------------|-------------|------------|
| Furniture        | 3003.822820 | 68.116607  |
| Office Supplies  | 814.048178  | 112.369072 |
| Technology       | 2897.941008 | 429.207516 |

The most profitable is Technology

## Questions:

How many Product categories?

Total sales and profits per category?

What is the most profitable category?

```
import pandas as pd
import matplotlib.pyplot as plt

orders = pd.read_csv("file_samples/sampledata_orders.csv", sep=";",
                     index_col=0, header=0)

SPC = orders[["Sales", "Profit", "Product Category"]]
print(SPC.head())


SPC.plot(kind="hist", bins=10)
plt.show()

print("")
grouped = SPC.groupby("Product Category")
for i, g in grouped:
    print("Group: ", i)

print("")
print("Count elements per category:") #get the series corresponding to the column
                                     #and apply the value_counts() method
print(orders["Product Category"].value_counts())
print("")
print("Total values:")
print(grouped.aggreate(pd.DataFrame.sum))

print("Mean values (sorted by profit):")
mv_sorted = grouped.aggreate(pd.DataFrame.mean).sort_values(by="Profit")
print(mv_sorted)
print("")
print("The most profitable is {}".format(mv_sorted.index[-1]))
```

# https://pandas.pydata.org/pandas-docs/stable/reference/index.html

 pandas

Getting started User Guide **API reference** Development Release notes

Q Search the docs ...

Input/output

General functions

Series

DataFrame

pandas arrays

Panel

Index objects

Date offsets

Window

GroupBy

Resampling

Style

Plotting

General utility functions

Extensions

## API reference

This page gives an overview of all public pandas objects, functions and methods. All classes and functions exposed in `pandas.*` namespace are public.

Some subpackages are public which include `pandas.errors`, `pandas.plotting`, and `pandas.testing`. Public functions in `pandas.io` and `pandas.tseries` submodules are mentioned in the documentation. `pandas.api.types` subpackage holds some public functions related to data types in pandas.

**Warning**  
The `pandas.core`, `pandas.compat`, and `pandas.util` top-level modules are PRIVATE. Stable functionality in such modules is not guaranteed.

- Input/output
  - Pickling
  - Flat file
  - Clipboard
  - Excel
  - JSON
  - HTML
  - HDFStore: PyTables (HDF5)
  - Feather
  - Parquet
  - ORC
  - SAS
  - SPSS
  - SQL
  - Google BigQuery
  - STATA
- General functions
  - Data manipulations
  - Top-level missing data
  - Top-level conversions
  - Top-level dealing with datetimelike
  - Top-level dealing with intervals



# First things first

We are going to need some libraries

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
```

In **Linux** you can install the libraries by typing in a terminal `sudo pip3 install matplotlib`, `sudo pip3 install pandas` and `sudo pip3 install numpy` (or `sudo python3.X -m pip install matplotlib`, `sudo python3.X -m pip install pandas` and `sudo python3.6 -m pip install numpy`), where X is your python version.

In **Windows** you can install the libraries by typing in the command prompt (to open it type `cmd` in the search) `pip3 install matplotlib`, `pip3 install pandas` and `pip3 install numpy`.



## Exercises

1. The file [top\\_3000\\_words.txt](#) is a one-column file representing the top 3000 English words. Read the file and for each letter, count how many words start with that letter. Store this information in a dictionary. Create a pandas series from the dictionary and plot an histogram of all initials counting more than 100 words starting with them.

Show/Hide Solution

2. The file [filt\\_aligns.tsv](#) is a tab separated value file representing alignments of paired-end reads on some apple chromosomes. Paired end reads have the property of being X bases apart from each other as they have been sequenced from the two ends of some size-selected DNA molecules.



Each line of the file has the following information

`readID\tChrPE1\tAlignmentPosition1\tChrPE2\tAlignmentPosition2`. The two ends of the same pair have the same readID. Load the read pairs aligning on the same chromosome into two dictionaries. The first ( `inserts` ) having readID as keys and the insert size (i.e. the absolute value of `AlignmentPosition1 - AlignmentPosition2`) as value. The second dictionary ( `chrs` ) will have readID as key and chromosome ID as value. Example:

```
readID Chr11 31120 Chr11 31472
readID1 Chr7 12000 Chr11 11680
```

will result in:

```
inserts = {"readID" : 352, "readID1" : 320}
chrs = {"readID" : "Chr11", "readID1" : "Chr7"}
```