Python

Exploratory Data Analysis

DEI/ISEP
Fátima Rodrigues
mfc@dei.isep.ipp.pt

Python

- High-level interpreted programming language
- Supports multiple programming paradigms: imperative, object-oriented and functional
- Dynamically typed language with automatic memory management

- Is one of the most popular programming languages today
  - used primarily for Web Development, Data Science and Automation/Scripting
  - easy syntax
  - active community that creates new libraries and provides ongoing support for language improvement

Python Libraries for Data Science

Many popular Python toolboxes/libraries:

- NumPy
- SciPy
- Pandas
- SciKit-Learn

Visualization libraries

- matplotlib
- Seaborn

and many more …

Python Libraries for Data Science

NumPy:

- introduces objects for multidimensional arrays and matrices, as well as functions that allow to easily perform advanced mathematical and statistical operations on those objects
- provides vectorization of mathematical operations on arrays and matrices which significantly improves the performance
- many other python libraries are built on NumPy

Link: http://www.numpy.org/

Python Libraries for Data Science

SciPy:

- collection of algorithms for linear algebra, differential equations, numerical integration, optimization, statistics and more
- part of SciPy Stack
- built on NumPy

Link: https://www.scipy.org/scipylib/

Python Libraries for Data Science

Pandas:

- adds data structures and tools designed to work with table-like data (similar to Series and Data Frames in R)
- provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- allows handling missing data

Link: http://pandas.pydata.org/

Python Libraries for Data Science

SciKit-Learn:

- provides machine learning algorithms: classification, regression, clustering, model validation etc.

- built on NumPy, SciPy and matplotlib

Link: http://scikit-learn.org/

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Science

Seaborn:

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

Loading Python Libraries

```
In [ ]: #Import Python Libraries
        import numpy as np
        import scipy as sp
        import pandas as pd
```

Press `Shift+Enter` to execute the *jupyter* cell

Reading data using pandas

In [ ]:
```
#Read csv file
df =
pd.read_csv("http://isep/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None,
                                                    na_values=['NA'])

pd.read_stata('myfile.dta')

pd.read_sas('myfile.sas7bdat')

pd.read_hdf('myfile.h5','df')
```

Exploring dataFrames

In [3]:
```
#List first 5 records
df.head()
```

Out[3]:

	category	discipline	phd	service	sex	salary
0	Prof	B	56	49	Male	186960
1	Prof	A	12	6	Male	93000
2	Prof	A	23	20	Male	110515
3	Prof	A	40	31	Male	131205
4	Prof	B	20	18	Male	104800

- How to read the first 10, 20, 50 records?
- How to view the last few records?

DataFrame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

DataFrame data types

```
In [4]: #Check a particular column type
        df['salary'].dtype
```

```
Out[4]: dtype('int64')
```

```
In [5]: #Check types for all the columns
        df.dtypes
```

```
Out[5]:
category      object
discipline    object
phd            int64
service        int64
sex           object
salary         int64
dtype: object
```

DataFrame attributes

Python objects have *attributes* and *methods*.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

DataFrame attributes

- Find how many records this data frame has;
- How many elements are there?
- What are the column names?
- What types of columns we have in this data frame?

DataFrame attributes

In [5]:
```
#Check number of lines/columns
df.shape
```

Out[5]: (78, 6)

In [6]:
```
#Check number of elements
df.size
```

Out[6]: 468

In [7]:
```
#Check columns name
df.columns
```

Out[7]: Index([‘category', 'discipline', 'phd', 'service', 'sex', 'salary'], dtype='object')

DataFrames methods

Unlike attributes, python methods have *parenthesis.*
All attributes and methods can be listed with a *dir()* function: `dir(df)`

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

DataFrames methods

- Give the summary for the numeric columns in the dataset
- Calculate standard deviation for all numeric columns;
- The mean values of the first 50 records in the dataset?

Hint: use head() method to subset the first 50 records and then calculate the mean

DataFrames methods

In [8]:
```
# summary for the numeric columns
df.describe()
```

Out[8]:

	phd	service	salary
count	78.000000	78.000000	78.000000
mean	19.705128	15.051282	108023.782051
std	12.498425	12.139768	28293.661022
min	1.000000	0.000000	57800.000000
25%	10.250000	5.250000	88612.500000
50%	18.500000	14.500000	104671.000000
75%	27.750000	20.750000	126774.750000
max	56.000000	51.000000	186960.000000

DataFrames methods

```
In [9]: #Standard deviation for all numeric columns
        df[['phd','service','salary']].std()
```

```
Out[9]: phd         12.498425
        service     12.139768
        salary   28293.661022
        dtype: float64
```

```
In [10]: #Mean values of the first 50 records in the dataFrame
         df[['phd','service','salary']].head(50).mean()
```

```
Out[10]: phd 21.52
         service 17.60
         salary 113789.14
         dtype: float64
```

DataFrames methods

In [10]:
```
#basic statistics for the salary
df.salary.describe()
```

Out[10]:
```
count 78.000000
mean 108023.782051
std 28293.661022
min 57800.000000
25% 88612.500000
50% 104671.000000
75% 126774.750000
max 186960.000000
Name: salary, dtype: float64
```

In [11]:
```
# how many values in the salary column
df.salary.count()
```

Out [11]: 78

Selecting a column in a DataFrame

Method 1: Subset the dataFrame using column name:

```
df['sex']
```

Method 2: Use the column name as an attribute:

```
df.sex
```

Note: there is an attribute *rank* for pandas data frames, so to select a column with a name "rank" we should use method 1.

DataFrame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

In [ ]:
```
#Calculate mean salary for each professor rank:
df[ df['salary'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;

< less; <= less or equal;

== equal; != not equal;

In [ ]:
```
#Select only those rows that contain female professors:
df[ df['sex'] == 'Female' ]
```

DataFrames: Slicing

There are a number of ways to subset the Data Frame:

- one or more columns
- one or more rows
- a subset of rows and columns

Rows and columns can be selected by their position or label

DataFrames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a dataFrame):

In [ ]:
```
#Select column salary:
df['salary']
```

When we need to select more than one column and/or make the output to be a dataFrame, we should use double brackets:

In [ ]:
```
#Select column category and salary:
df[['category','salary']]
```

DataFrames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

```
In [ ]: #Select rows by their position:
        df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted:
So, for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

https://datacarpentry.org/python-ecology-lesson/03-index-slice-subset/

DataFrames: Selecting rows and columns

dataframe.loc[ ] datframe.iloc[ ] to select a single column or multiple columns from pandas DataFrame

loc[] is used with column labels/names and iloc[] is used with column index/position

```
# Using loc[] to take column slices
df2 = df.loc[:, ['rank','sex','salary']] # Select multiple columns
df2 = df.loc[:,'category':'salary'] # Select columns between two columns
df2 = df.loc[:,'category':]  # Select columns by range
df2 = df.loc[:,:'salary']    # Select columns by range
df2 = df.loc[:,::2]          # Select every alternate column

# Using iloc[] to select column by Index
df2 = df.iloc[:,[1,3,4]] # Select columns by Index
df2 = df.iloc[:,1:4] # Select between columns 1 and 4 (2,3,4)
df2 = df.iloc[:,2:] # Select From 3rd to end
df2 = df.iloc[:,:2] # Select First Two Columns

#Using get()
df2 = df.get(['rank','sex'])
```

DataFrames: method loc

If we need to select a range of rows, using their labels we can use method loc:

In[20]:
```
#Select rows by their labels:
df[['rank','sex','salary']][10:20]
```

Out[20]:

	rank	sex	salary
10	Prof	Male	128250
11	Prof	Male	134778
13	Prof	Male	162200
14	Prof	Male	153750
15	Prof	Male	150480
19	Prof	Male	150500

DataFrames: method iloc

If we need to select a range of rows and/or columns, using their positions we can use method iloc:

In[21]:
```
#Select rows/columns by their positions:
df.iloc[10:20,[0, 3, 4, 5]]
```

Out[21]:

	category	service	sex	salary
10	Prof	33	Male	128250
11	Prof	23	Male	134778
12	AsstProf	0	Male	88000
13	Prof	33	Male	162200
14	Prof	19	Male	153750
15	Prof	3	Male	150480
16	AsstProf	3	Male	75044
17	AsstProf	0	Male	92000
18	Prof	7	Male	107300
19	Prof	27	Male	150500

DataFrames: method iloc (summary)

```
df.iloc[0]  # First row of a data frame
df.iloc[i]  #(i+1)th row
df.iloc[-1] # Last row
```

```
df.iloc[:, 0]  # First column
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]        #First 7 rows
df.iloc[:, 0:2]     #First 2 columns
df.iloc[1:3, 0:2]   #Second through third rows and first 2 columns
df.iloc[[0,5], [1,3]]  #1st and 6th rows and 2nd and 4th columns
```

Aggregation Functions in Pandas

Aggregation - computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions:

min, max
count, sum, prod
mean, median, mode, mad
std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

```
In [25]: df[['phd','service','salary']].agg(['min','mean','max'])
```

Out[25]:

	phd	service	salary
min	1.000000	0.000000	57800.000000
mean	19.705128	15.051282	108023.782051
max	56.000000	51.000000	186960.000000

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

DataFrames *group by* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

In[12]:

```
#Group data by rank
df.groupby(['category']).mean()
```

Out[12]:

category	phd	service	salary
AssocProf	15.076923	11.307692	91786.230769
AsstProf	5.052632	2.210526	81362.789474
Prof	27.065217	21.413043	123624.804348

DataFrames *group by* method

Once group by object is created, we can calculate various statistics for each group:

In[13]:
```
#Calculate mean salary for each professor rank:
df.groupby('rank')[['salary']].mean()
```

Out[13]:

	salary
category	
AssocProf	91786.230769
AsstProf	81362.789474
Prof	123624.804348

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

DataFrames *group by* method

Group by performance notes:

- no grouping/splitting occurs until it's needed. Creating the group by object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the group by operation. You may want to pass **sort=False for potential speedup**:

In[14]:
```
#Calculate mean salary for each professor rank:
df.groupby(['category'], sort=False)[['salary']].mean()
```

Out[14]:

	salary
category	
Prof	123624.804348
AssocProf	91786.230769
AsstProf	81362.789474

DataFrames: Sorting

The data can be sorted by a value in the column. By default the sorting will occur in ascending order and a new dataFrame is return.

In[22]:
```
# Create a new data frame from the original sorted by the column service
df_sorted = df.sort_values( by ='service')
df_sorted.head()
```

Out[22]:

	category	discipline	phd	service	sex	salary
55	AsstProf	A	2	0	Female	72500
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000

DataFrames: Sorting

The data can be sorted using several columns:

```
In [23]: df_sorted = df.sort_values( by =['service', 'salary'], ascending = [True, False])
         df_sorted.head(10)
```

Out[23]:

	category	discipline	phd	service	sex	salary
52	Prof	A	12	0	Female	105000
17	AsstProf	B	4	0	Male	92000
12	AsstProf	B	1	0	Male	88000
23	AsstProf	A	2	0	Male	85000
43	AsstProf	B	5	0	Female	77000
55	AsstProf	A	2	0	Female	72500
57	AsstProf	A	3	1	Female	72500
28	AsstProf	B	7	2	Male	91300
42	AsstProf	B	4	2	Female	80225
68	AsstProf	A	4	2	Female	77500

Missing Values

Missing values are marked as NaN

In [24]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[24]:

	year	month	day	dep_time	dep_delay	arr_time	arr_delay	carrier	tailnum	flight	origin	dest	air_time	distance	hour	minute
330	2013	1	1	1807.0	29.0	2251.0	NaN	UA	N31412	1228	EWR	SAN	NaN	2425	18.0	7.0
403	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EHAA	791	LGA	DFW	NaN	1389	NaN	NaN
404	2013	1	1	NaN	NaN	NaN	NaN	AA	N3EVAA	1925	LGA	MIA	NaN	1096	NaN	NaN
855	2013	1	2	2145.0	16.0	NaN	NaN	UA	N12221	1299	EWR	RSW	NaN	1068	21.0	45.0
858	2013	1	2	NaN	NaN	NaN	NaN	AA	NaN	133	JFK	LAX	NaN	2475	NaN	NaN

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default (unlike R)

Iterate over the rows of a dataframe

Three different pandas functions to iterate through the dataframe rows and columns:

- Dataframe.iterrows()
- Dataframe.itertuples()
- Dataframe.items()

Iterating through a dataframe's rows and columns **should be the last resort** since **it’s slow and not worth it**.
It must only be used if we can’t achieve it using vectorization, loc, and apply functions.

Dataframe.iterrows()

It iterates over the dataframe rows as (Index, Series) pairs and returns a series for each row and itt does **not** preserve dtypes across the rows

```
for idx, row in df.iterrows():
    print(row.index)
```

```
Index(['category', 'discipline', 'phd', 'service', 'sex', 'salary'], dtype='object')
Index(['category', 'discipline', 'phd', 'service', 'sex', 'salary'], dtype='object')
Index(['category', 'discipline', 'phd', 'service', 'sex', 'salary'], dtype='object')
..
```

```
for idx, row in df.iterrows():
      print(row.values)
```

```
['Prof' 'B' 56 49 'Male' 186960]
['Prof' 'A' 12 6 'Male' 93000]
['Prof' 'A' 23 20 'Male' 110515]
['Prof' 'A' 40 31 'Male' 131205]
..
```

Dataframe.iterrows()

Access the column value of each row to check a condition and update the value of columns that meet condition

Increase salary by: Assist Professor 15%, AssocProf 10%, Prof 5%

```
for idx, row in df.iterrows():
    if (row.category == 'AsstProf'):
        df.at[idx,'salary'] = round(df.at[idx,'salary']*1.15,0)
    elif(row.category == ' AssocProf'):
        df.at[idx,'salary'] = round(df.at[idx,'salary']*1.10,0)
    else:
        df.at[idx,'salary'] = round(df.at[idx,'salary']*1.05,0)
```

Set Pandas Conditional Column Based on Values of Another Column

DataFrame Example

In [1]:
```
df = pd.DataFrame.from_dict(
    {
        'Name': ['Jane', 'Melissa', 'John', 'Matt'],
        'Age': [23, 45, 35, 64],
        'Birth City': ['London', 'Paris', 'Toronto', 'Atlanta'],
        'Gender': ['F', 'F', 'M', 'M']
    }
)
print(df)
```

Out [1]:

	Name	Age	Birth City	Gender
0	Jane	23	London	F
1	Melissa	45	Paris	F
2	John	35	Toronto	M
3	Matt	64	Atlanta	M

loc to Set Pandas Conditional Column

```
df.loc[df[‘column’] condition, ‘new column name’] = ‘value if condition is met’
```

In [2]:
```
df['Age Category'] = "Over 30"
df.loc[df['Age'] < 30, 'Age Category'] = "Under 30"
df
```

Out [2]:

	Name	Age	Birth City	Gender	Age Category
0	Jane	23	London	F	Under 30
1	Melissa	45	Paris	F	Over 30
2	John	35	Toronto	M	Over 30
3	Matt	64	Atlanta	M	Over 30

Set Values using Multiple Conditions

In [3]:
```
conditions = [
    (df['Age'] < 20),
    (df['Age'] >= 20) & (df['Age'] < 40),
    (df['Age'] >= 40) & (df['Age’] < 60),
    (df['Age'] >= 60)
]
values = ['<20 years old', '20-39 years old', '40-59 years old', '60+ years old']
df['Age Group'] = np.select(conditions, values)
df
```

Out [3]:

	Name	Age	Birth City	Gender	Age Category	Age Group
0	Jane	23	London	F	Under 30	20-39 years old
1	Melissa	45	Paris	F	Over 30	40-59 years old
2	John	35	Toronto	M	Over 30	20-39 years old
3	Matt	64	Atlanta	M	Over 30	60+ years old

Pandas Apply to Apply a function to a column

```
In [5]: df['Name Length'] = df['Name'].apply(len)
        df
```

Or using a custom function

```
In [6]: def age_groups(x):
            if x < 20:
                return '<20 years old'
            elif x < 40:
                return '20-39 years old'
            elif x < 60:
                return '40-59 years old'
            else:
                return '60+ years old’

        df['Age Group'] = df['Age'].apply(age_groups)
        df
```

Map to Set Values in Another Column

In [4]:
```
city_dict = {
    'Paris': 'France',
    'Toronto': 'Canada',
    'Atlanta': 'USA'
}
df['Country'] = df['Birth City'].map(city_dict).fillna('Other')
df
```

Out [4]:

	Name	Age	Birth City	Gender	Age Category	Age Group	Country
0	Jane	23	London	F	Under 30	20-39 years old	Other
1	Melissa	45	Paris	F	Over 30	40-59 years old	France
2	John	35	Toronto	M	Over 30	20-39 years old	Canada
3	Matt	64	Atlanta	M	Over 30	60+ years old	USA

Pandas Apply to Apply a function to a column

```
Out [6]:
```

	Name	Age	Birth City	Gender	Age Category	Age Group	Country	Name Length
0	Jane	23	London	F	Under 30	20-39 years old	Other	4
1	Melissa	45	Paris	F	Over 30	40-59 years old	France	7
2	John	35	Toronto	M	Over 30	20-39 years old	Canada	4
3	Matt	64	Atlanta	M	Over 30	60+ years old	USA	4

Data Visualization

Python for Data Visualization

With data visualization we try to understand data by placing it in a visual context so that patterns, trends and correlations that might not otherwise be detected can be exposed

Key motivations for data visualization include:

- To help to humans understand data
- To make use of humans' abilities to visually recognize patterns in data

Python Libraries for Data Science

matplotlib:

- python 2D plotting library which produces publication quality figures in a variety of hardcopy formats
- a set of functionalities similar to those of MATLAB
- line plots, scatter plots, barcharts, histograms, pie charts etc.
- relatively low-level; some effort needed to create advanced visualization

Link: https://matplotlib.org/

Python Libraries for Data Visualization

Seaborn:

- based on matplotlib
- provides high level interface for drawing attractive statistical graphics
- Similar (in style) to the popular ggplot2 library in R

Link: https://seaborn.pydata.org/

Loading Python Libraries

In [ ]:
```
#Import Python Libraries
import numpy as np
import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
```

Exploratory graphics

- To understand data properties
- To find patterns in data
- To suggest modelling strategies
- To "debug" analyses
- To communicate results

Characteristics of exploratory graphs

- They are made quickly
- A large number are made
- The goal is for personal understanding
- Axes/legends are generally cleaned up
- Colour/size are primarily used for information

Pandas Visualization

- Pandas is an open source high-performance, easy-to-use library providing data structures, such as dataframes, and data analysis tools
- Pandas Visualization makes it really easy to create plots out of a pandas dataframe and series
- It also has a higher level API than Matplotlib and therefore we need less code for the same results

Reading data using pandas

In [1]:
```
#Read csv file
iris = pd.read_csv('iris.csv', names=['sepal_length','sepal_width','petal_length',
                                      'petal_width','class'])
iris.head()
```

Out[1]:

	sepal_length	sepal_width	petal_length	petal_width	class
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

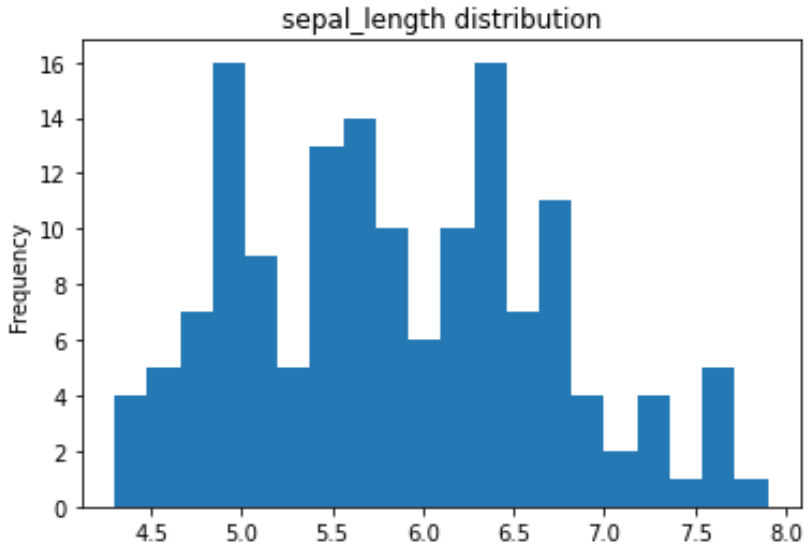
Histogram Plot

Histogram is suitable for continuous variables

- It is used to plot frequency distributions with or without classes
- Changing the classes intervals will change the underlying distribution

In [2]:
```
#pandas histogram
iris['sepal_length'].plot.hist(bins=20, title='sepal_length distribution')
plt.show()
```

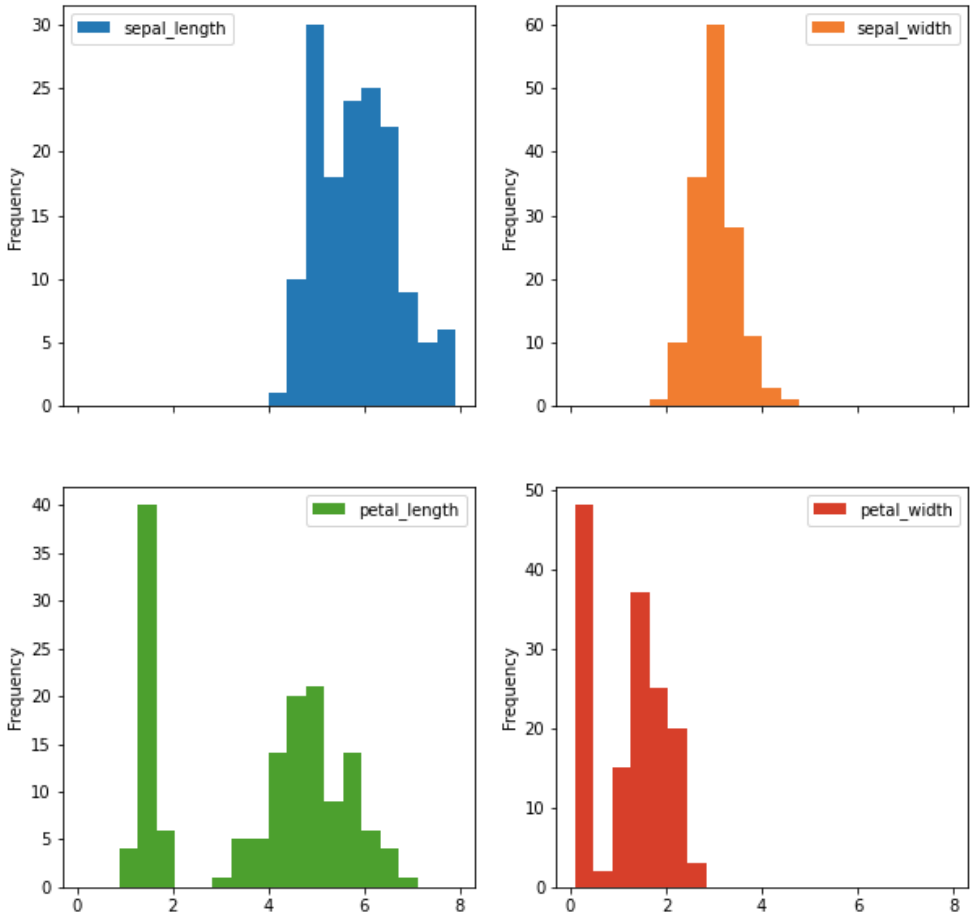
Out[2]:



Multiple Histograms

In [3]:
```
#multiple histograms
iris.plot.hist(subplots=True, layout=(2,2), figsize=(10, 10), bins=20)
```

Out[3]:



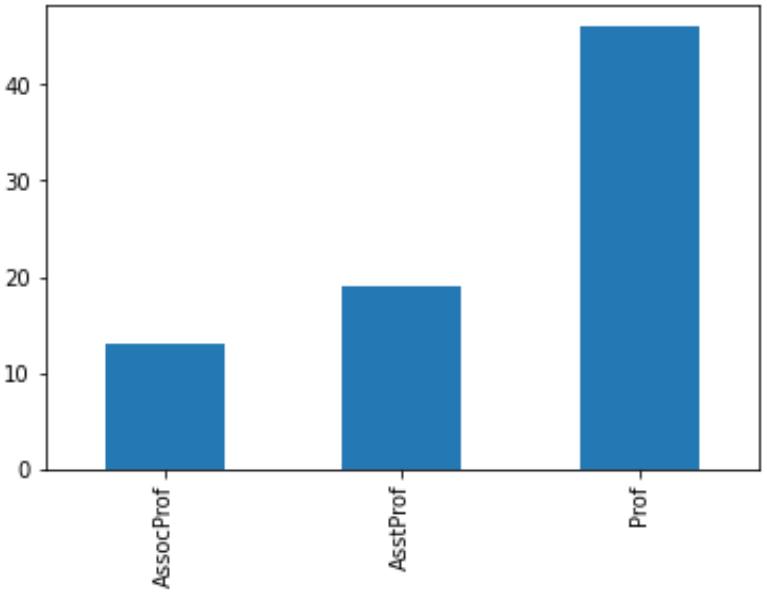
Bar plot

Bar graph is used to compare things between different groups or track changes over time (i.e. when changes are large)

Bar graph is suitable for categorical and interval variables

In [4]:
```
#bar graph
df['category'].value_counts().sort_index().plot.bar()
plt.show()
```

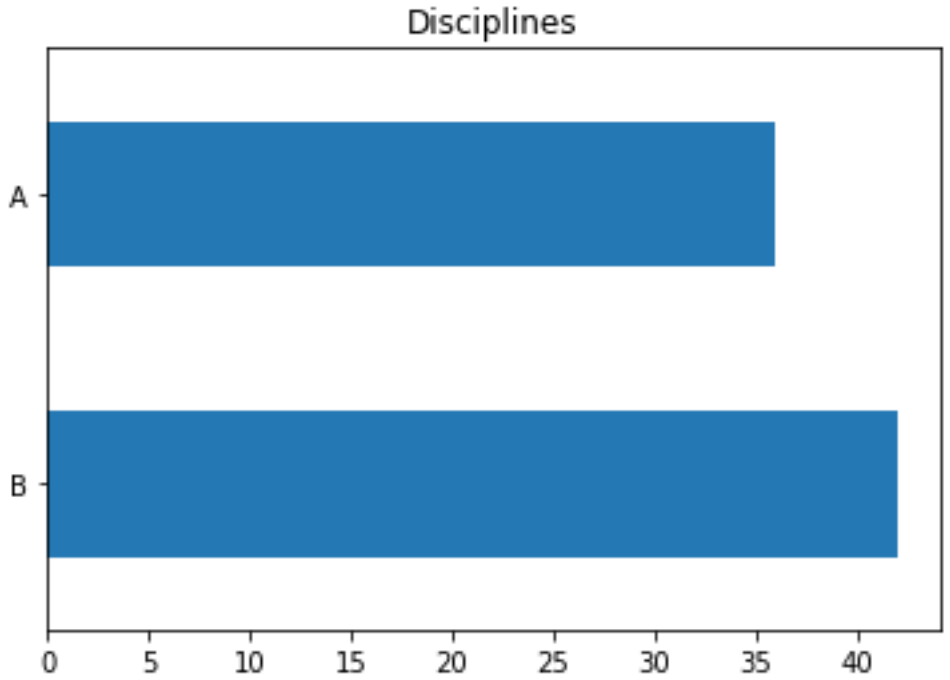
Out[4]:



Horizontal Bar plot

In[5]:
```
#horizontal bar chart
df['discipline'].value_counts().plot.barh(title="Disciplines")
plt.show()
```

Out[5]:



Plots with two variables

With two variables, typically the response variable on the y axis and the explanatory variable, on x axis, the kind of plot depends on the nature of the explanatory variable

If the explanatory variable is:

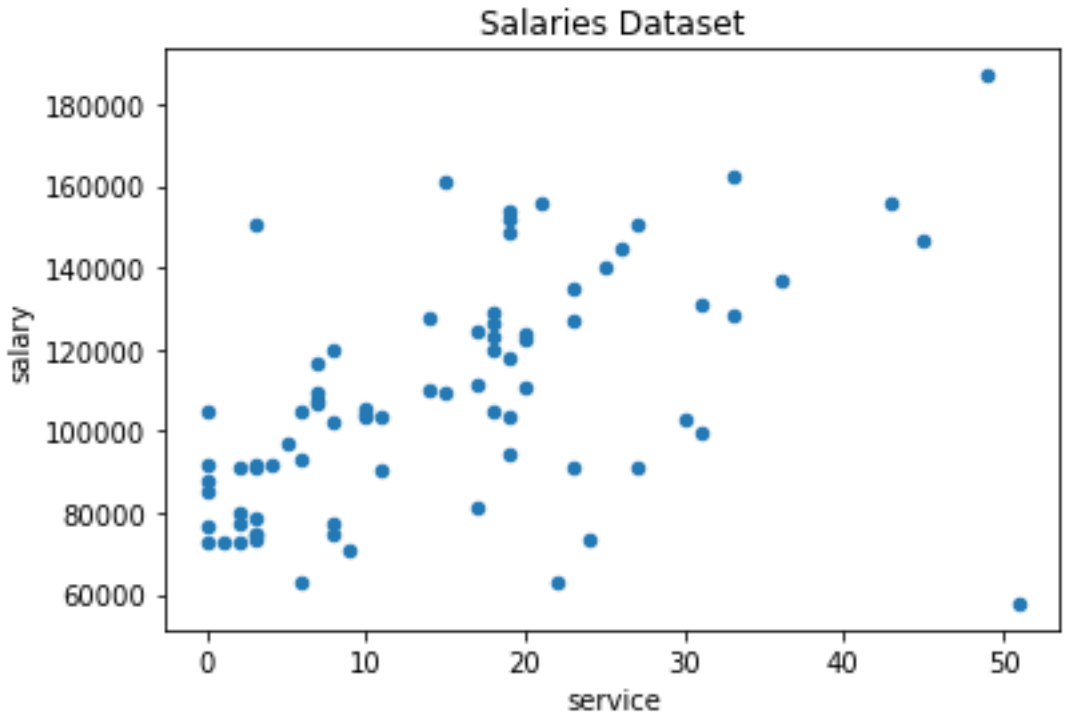
- Continuous, the appropriate plot is a scatter plot

- Categorical, the appropriate plot is a barplot or boxplot

Scatter plot

In [6]:
```
# scatter plot
df.plot.scatter(x='service', y='salary', title='Salaries Dataset')
plt.show()
```

Out[6]:

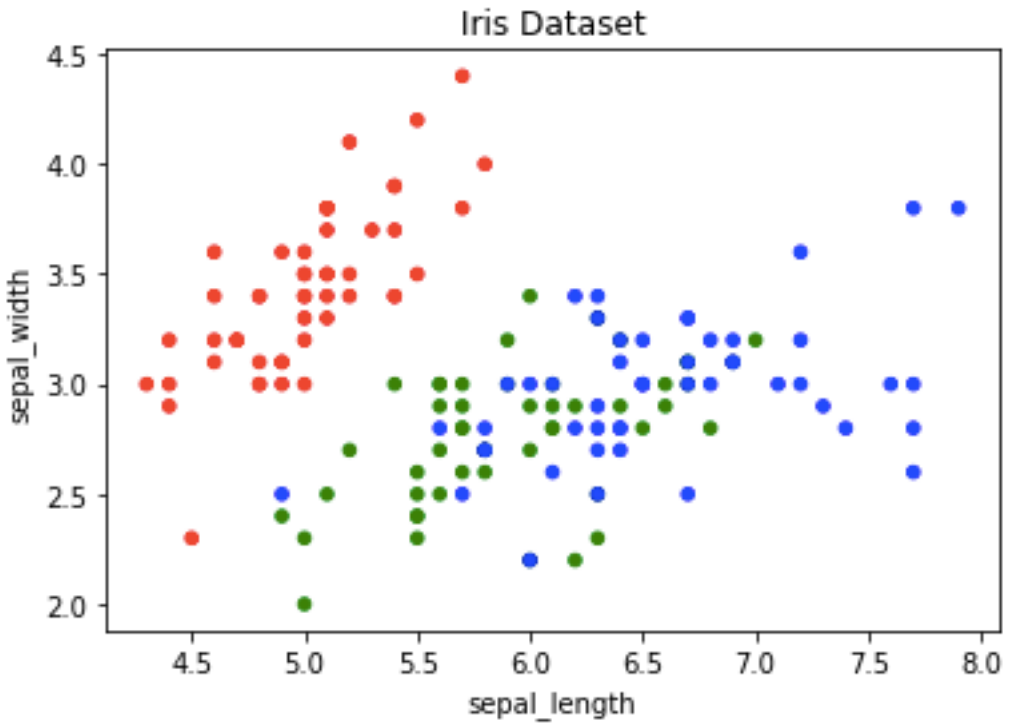


Scatter plot

Give the graph more meaning by colouring in each data-point by its class

In [7]:
```
# scatter plot
colors = {'setosa':'red', 'versicolor':'green', 'virginica':'blue'}
iris.plot.scatter(x='sepal_length', y='sepal_width',
                  c=iris['class'].map(colors), title='Iris Dataset')
plt.show()
```

Out[7]:



Bar plot with two categorical variables

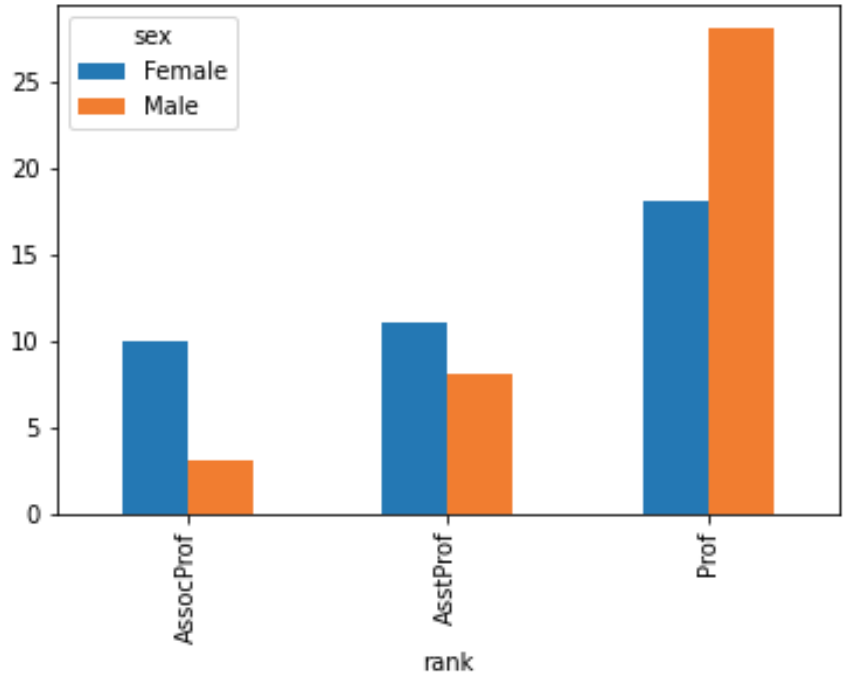
In [8]:
```
pd.crosstab(df['category'],df['sex'])
```

Out[8]:

category	sex	Female	Male
AssocProf		10	3
AsstProf		11	8
Prof		18	28

In [9]:
```
# Barplot: with two nominal variables
pd.crosstab(df['category'],df['sex']).plot(kind="bar")
plt.show()
```

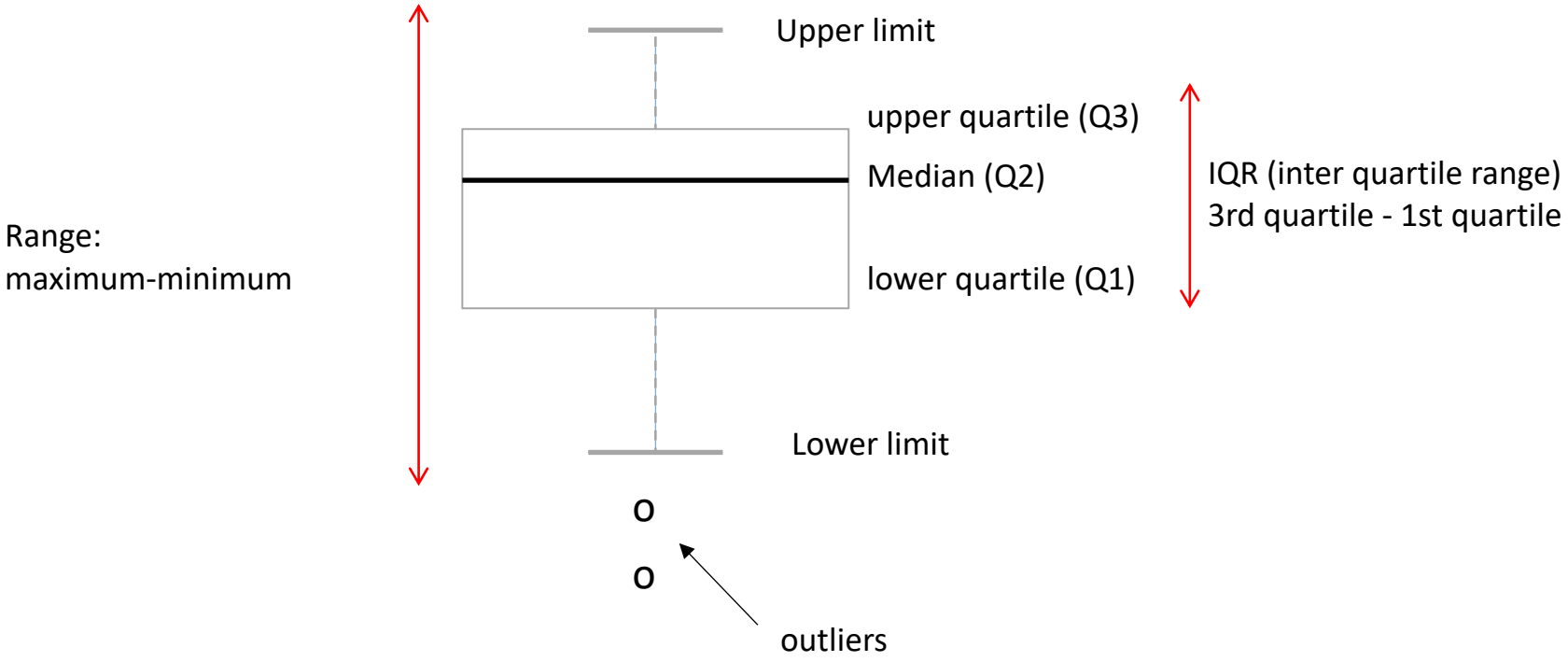
Out[9]:



Boxplot

- Box plots provide interesting summaries of a variable distribution
- They inform us of the interquartile range and of the outliers (if any)

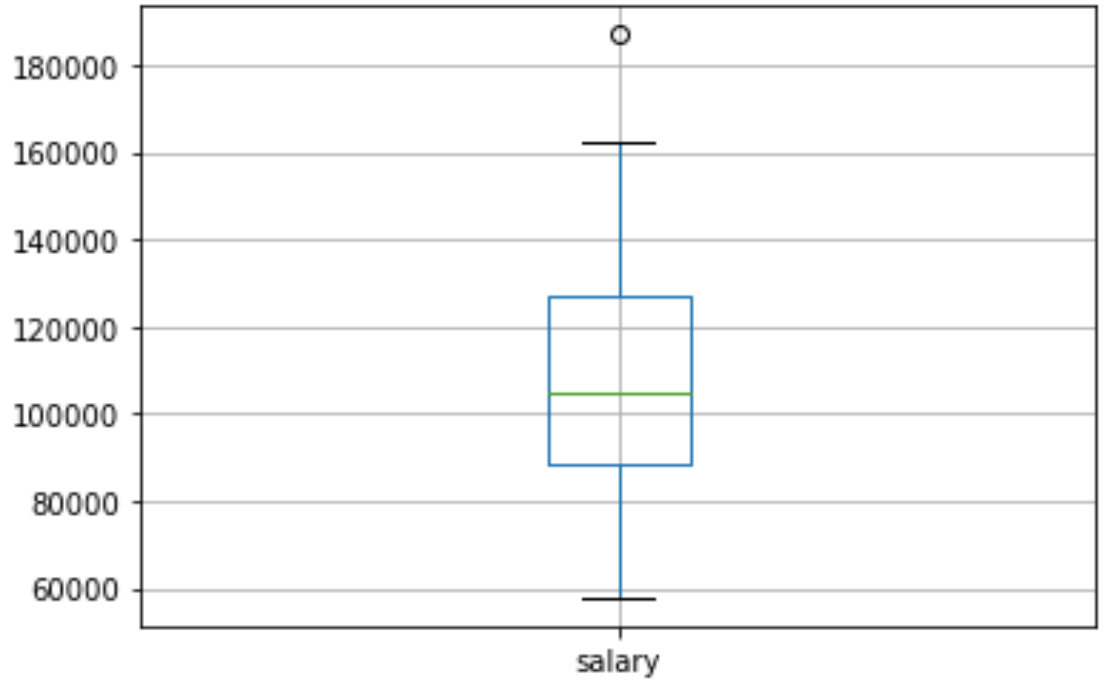
The Five number summary



Boxplot

In [10]:
```
#box plot
df.boxplot(column=['salary'])
plt.show()
```

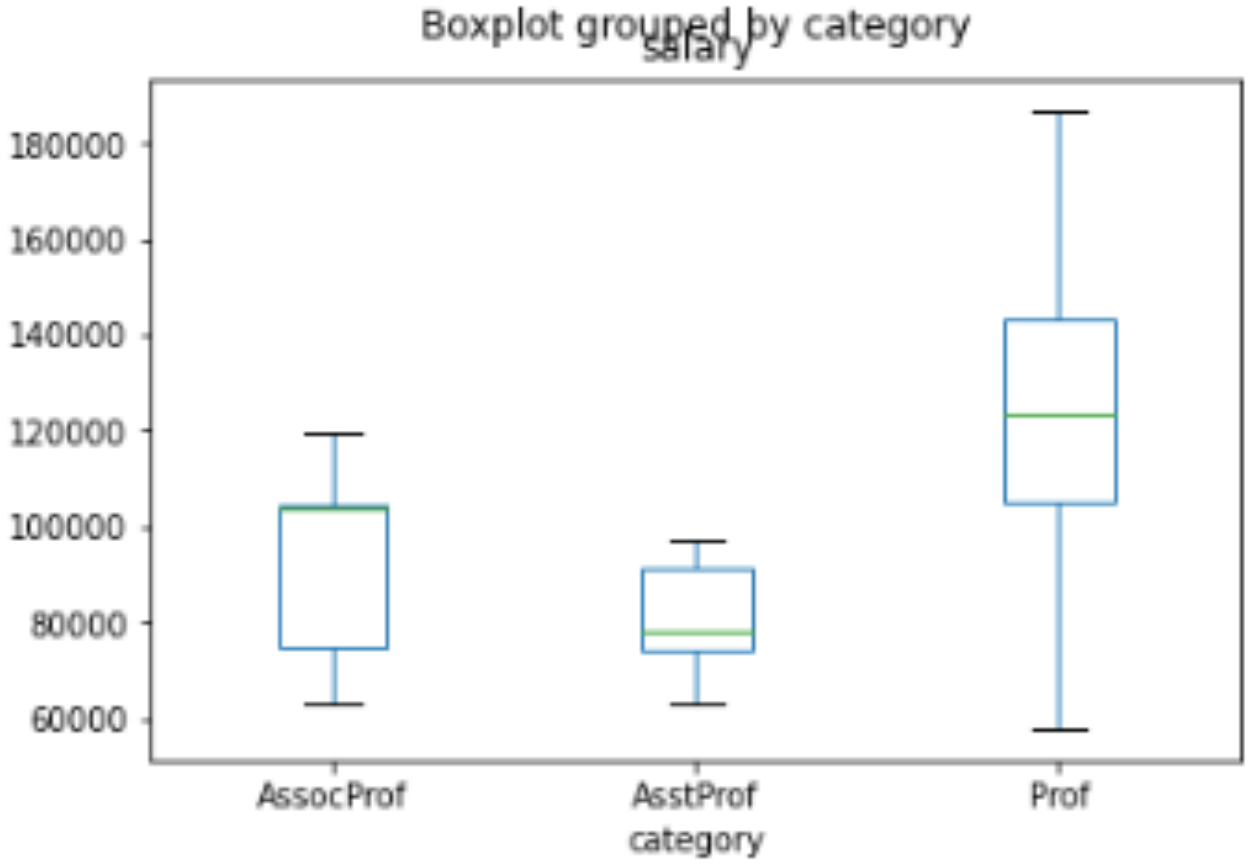
Out[10]:



Boxplot

In [11]:
```
# boxplot of salary by rank
df.boxplot(by='category', column=['salary'], grid=False)
plt.show()
```

Out[11]:



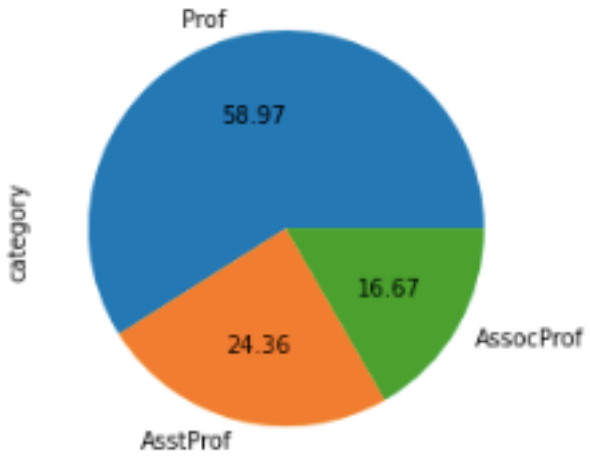
Pie Chart

- The pie chart is best to use when trying to compare parts of a whole
- It is suitable for analysing categorical variables

- It is the wrong choice as an exploratory device, and it must not be used to present complicated information

In [12]:
```
# Pie chart of salary by category
df['category'].value_counts().plot(kind='pie', autopct='%.2f')
plt.show()
```

Out[12]:



Basic Statistical Analysis

statsmodel and scikit-learn - both have several functions for statistical analysis

The first one is mostly used for regular analysis using R style formulas, while scikit-learn is more tailored for Machine Learning

statsmodels:

- linear regressions
- ANOVA tests
- hypothesis testings
- many more ...

scikit-learn:

- kmeans
- support vector machines
- random forests
- many more ...

Some Links

- Python Introduction
  https://www.w3schools.com/python/python_intro.asp