Introduction to Python: Part 2

NumPy package

The NumPy package is the core library for scientific computing in Python. Documentation (https://docs.scipy.org/doc/numpy/index.html)

In [1]:
```
# Import the numpy package
import numpy as np
```

Arrays

NumPy's main object is a multidimensional array. The elements of an array must be of the same type.

Creating arrays

Arrays can be created from lists using the array() function

In [2]:
```
# Create a 1-dimensional array (vector)
x = np.array([1, 2, 3, 4])
x
```

Out[2]:
```
array([1, 2, 3, 4])
```

In [3]:
```
# Create a 2-dimensional array (matrix)
C = np.array([[1, 2, 3, 4],
              [5, 6, 7, 8],
              [9, 10, 11, 12]])
C
```

Out[3]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Arrays can be created with specific numpy functions

In [4]:
```
# Create a 1-dimensional array of length 10 filled with values from 0 to 9.
# The arange() function works analogously to the range() function (see Part 1)
x = np.arange(10)
x
```

Out[4]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [5]:
```
# Create a 3x3 matrix of all zeros
np.zeros((3, 3))
```

Out[5]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [6]:
```
# Create a 4x10 matrix of all ones
np.ones((4, 10))
```

Out[6]:
```
array([[1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1., 1., 1., 1.]])
```

In [7]:
```
# Create a 2x2 matrix filled with the constant 24
np.full((2, 2), 24)
```

Out[7]:
```
array([[24, 24],
       [24, 24]])
```

In [8]:
```
# Create a 10x10 identity matrix
np.eye(10)
```

Out[8]:
```
array([[1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 1.]])
```

In [9]:
```
# Create a 5x3 matrix filled with random values drawn from uniform distribution
# over the interval [0.0, 1.0)
np.random.seed(12345)
np.random.random((5, 3))
```

Out[9]:
```
array([[0.92961609, 0.31637555, 0.18391881],
       [0.20456028, 0.56772503, 0.5955447 ],
       [0.96451452, 0.6531771 , 0.74890664],
       [0.65356987, 0.74771481, 0.96130674],
       [0.0083883 , 0.10644438, 0.29870371]])
```

In [10]:
```
# Create a 3-dimensional array of ones
np.ones((3, 4, 5))
```

Out[10]:
```
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]])
```

The shape of arrays can be changed with the reshape() function

In [11]:
```
# Create a vector of length 10
x = np.arange(10)
x
```

Out[11]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [12]:
```
# Turn vector into a 5x2 matrix
x = x.reshape((5, 2))
x
```

Out[12]:
```
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
```

Get attributes of arrays

In [13]:
```
# Get number of dimensions (axes)
x.ndim
```

Out[13]:
```
2
```

In [14]:
```
# Get the size of the array in each dimension
x.shape
```

Out[14]:
```
(5, 2)
```

In [15]:
```
# Get number of elements in array
x.size
```

Out[15]:
```
10
```

Accessing arrays

Arrays can be sliced. Since arrays may be multidimensional, you must specify a slice for each dimension of the array.

In [16]:
```
# Create a 1-dimensional array of length 50 filled with values from 0 to 49
x = np.arange(50)
x
```

Out[16]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

In [17]:
```
# Get the first 5 values
x[:5]
```

Out[17]:
```
array([0, 1, 2, 3, 4])
```

In [18]:
```
# Get the values from position 2 up to position 7 (not including 7)
x[2:7]
```

Out[18]:
```
array([2, 3, 4, 5, 6])
```

In [19]:
```
# Get the last five values
x[-5:]
```

Out[19]:
```
array([45, 46, 47, 48, 49])
```

In [20]:
```
# Give the array a new shape
x = x.reshape((5, 10))
x
```

Out[20]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

In [21]:
```
# Get value in second row and sixth column
x[1, 5]
```

Out[21]:
```
15
```

In [22]:
```
# Get values from first two rows and first five columns
x[:2, :5]
```

Out[22]:
```
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14]])
```

In [23]:
```
# Get values from last two rows and last two columns
x[-2:, -2:]
```

Out[23]:
```
array([[38, 39],
       [48, 49]])
```

In [24]:
```
# Get all rows and every second column
x[:, 0:x.shape[1]:2]
```

Out[24]:
```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28],
       [30, 32, 34, 36, 38],
       [40, 42, 44, 46, 48]])
```

Integer array indexing allows you to construct arbitrary arrays

In [25]:
```
# Get values from specific rows and columns
rows = [1, 4, 4]
cols = [0, -1, -2]
x[rows, cols]
```

Out[25]:
```
array([10, 49, 48])
```

Boolean array indexing allows to retrieve values based on logical conditions

In [26]:
```
# Get all values greather than 20 (Note that values are returned as 1-dimensional array)
x[x > 20]
```

Out[26]:
```
array([21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
       38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49])
```

In [27]:
```
# Get all values greater than 20 and smaller than 30
x[(x > 20) & (x < 30)]
```

Out[27]:
```
array([21, 22, 23, 24, 25, 26, 27, 28, 29])
```

In [28]:
```
# Get all values smaller than 20 or larger than 30
x[(x < 20) | (x > 30)]
```

Out[28]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
       45, 46, 47, 48, 49])
```

Changing values in arrays

Similar to Python lists, values of arrays can be changed

In [29]:
```
# Set top left value to -1
x[0, 0] = -1
x
```

Out[29]:
```
array([[-1,  1,  2,  3,  4,  5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

In [30]:
```
# Set values in the top left 3x3 submatrix to -1
x[:3, :3] = -1
x
```

Out[30]:
```
array([[-1, -1, -1,  3,  4,  5,  6,  7,  8,  9],
       [-1, -1, -1, 13, 14, 15, 16, 17, 18, 19],
       [-1, -1, -1, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

Apply mathematical operations to arrays

In [31]:
```
# Display matrix x
x
```

Out[31]:
```
array([[-1, -1, -1,  3,  4,  5,  6,  7,  8,  9],
       [-1, -1, -1, 13, 14, 15, 16, 17, 18, 19],
       [-1, -1, -1, 23, 24, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47, 48, 49]])
```

In [32]:
```
# Compute sum of all array elements
x.sum()
```

Out[32]:
```
1117
```

In [33]:
```
# Compute sum over each column
x.sum(axis=0)
```

Out[33]:
```
array([ 67,  69,  71, 115, 120, 125, 130, 135, 140, 145])
```

In [34]:
```
# Compute sum over each row
x.sum(axis=1)
```

Out[34]:
```
array([ 39, 109, 179, 345, 445])
```

In [35]:
```
# Compute arithmetic mean of all elements
x.mean()
```

Out[35]:
```
22.34
```

In [36]:
```
# Compute arithmetic mean of each column
x.mean(axis=0)
```

Out[36]:
```
array([13.4, 13.8, 14.2, 23. , 24. , 25. , 26. , 27. , 28. , 29. ])
```

In [37]:
```
# Create 3x10 matrix of stock returns
stock_returns = np.array([[5, 4, -1, 2, 7, 3, 2, 0, -2, -8],
                          [1, -1, 4, 0, -1, -2, 0, 4, 5, 8],
                          [3, -1, 3, -2, 4, -3, 7, 0, -3, 3]])
```

In [38]:
```
# Compute covariance matrix (each row represents a variable)
np.cov(stock_returns)
```

Out[38]:
```
array([[ 17.95555556, -12.4       ,   0.64444444],
       [-12.4       ,  10.62222222,   0.8       ],
       [  0.64444444,   0.8       ,  11.43333333]])
```

In [39]:
```
# Compute correlation matrix (each row represents a variable)
np.corrcoef(stock_returns)
```

Out[39]:
```
array([[ 1.        , -0.89787229,  0.04497795],
       [-0.89787229,  1.        ,  0.07259319],
       [ 0.04497795,  0.07259319,  1.        ]])
```

In [40]:
```
# Create a 3x5 matrix filled with values from 0 to 14
x = np.array(range(15)).reshape(3, 5)
# Create a 5x3 matrix filled with values from 0 to 14
y = np.array(range(15)).reshape(5, 3)
# Perform a matrix multiplication
x.dot(y)
```

Out[40]:
```
array([[ 90, 100, 110],
       [240, 275, 310],
       [390, 450, 510]])
```

In [41]:
```
# Multiply two arrays elementwise
prices = np.array([23.37, 52.85, 15.265, 452.20, 2374.00, 51.04])
shares = np.array([100, 375, 480, 75, 90, 120])
values = prices * shares
values
```

Out[41]:
```
array([  2337.  ,  19818.75,   7327.2 ,  33915.  , 213660.  ,   6124.8 ])
```

In [42]:
```
# Transpose matrix
print(x)
print()
print(x.T)
```

```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]]

[[ 0  5 10]
 [ 1  6 11]
 [ 2  7 12]
 [ 3  8 13]
 [ 4  9 14]]
```

Pandas package

The Pandas package is the core library for data manipulation in Python. [Documentation (https://pandas.pydata.org/pandas-docs/stable/genindex.html)](https://pandas.pydata.org/pandas-docs/stable/genindex.html)

In [43]:
```
# Import the pandas package
import pandas as pd
```

Series

- One-dimensional numpy array with axis labels (index)
- All elements must have the same type
- Labels must be immutable

Creating a Series

Series can be created from lists

In [44]:
```
# Creating a Series without a predefined index
s = pd.Series([1, 3, 5, 6, 8])
s
```

Out[44]:
```
0    1
1    3
2    5
3    6
4    8
dtype: int64
```

In [45]:
```
# Creating a Series with predefined index
population = pd.Series([130, 391, 63, 194, 81], index=['Bern', 'Zurich', 'Lugano', 'Gene
population
```

Out[45]:
```
Bern       130
Zurich     391
Lugano      63
Geneva     194
Lucerne     81
dtype: int64
```

Get attributes of a Series

In [46]:
```
# Get the number of elements in the Series
population.size
```

Out[46]:
```
5
```

Accessing data from a Series

In [47]:
```
# Get value for index 'Bern'
population['Bern']
```

Out[47]:
```
130
```

In [48]:
```
# Get value for all indices between 'Bern' and 'Geneva' (including Geneva)
population['Bern':'Geneva']
```

Out[48]:
```
Bern      130
Zurich    391
Lugano     63
Geneva    194
dtype: int64
```

In [49]:
```
# Get which elements are above 100
idx = population > 100
idx
```

Out[49]:
```
Bern        True
Zurich      True
Lugano     False
Geneva      True
Lucerne    False
dtype: bool
```

In [50]:
```
# Get values above 100
population[idx]
```

Out[50]:
```
Bern      130
Zurich    391
Geneva    194
dtype: int64
```

In [51]:
```
# Get values above 100 (short version)
population[population > 100]
```

Out[51]:
```
Bern      130
Zurich    391
Geneva    194
dtype: int64
```

In [52]:
```
# Get first two values
population[:2]
```

Out[52]:
```
Bern      130
Zurich    391
dtype: int64
```

In [53]:
```
# Get first value of Series
population[0]
```

Out[53]:
```
130
```

In [54]:
```
# Get first and fourth value of Series
population[[0, 3]]
```

Out[54]:
```
Bern      130
Geneva    194
dtype: int64
```

Apply mathematical operations on a Series

In [55]:
```
# Get sum of all elements of Series
population.sum()
```

Out[55]:
```
859
```

In [56]:
```
# Get mean of all elements of Series
population.mean()
```

Out[56]:
```
171.8
```

DataFrame

- Two-dimensional numpy array with labeled axes (rows and columns)
- Can be thought of as multiple Series with a shared index

Creating a DataFrame

A DataFrame can be created from a dictionary

In [57]:
```
# Create a dictionary
cities = {'name': ['Bern', 'Zurich', 'Lugano', 'Geneva', 'Lucerne'],
          'population': [130, 391, 63, 194, 81],
          'area': [51.6, 87.88, 75.98, 15.93, 29.04],
          'language': ['German', 'German', 'Italian', 'French', 'German']}
cities
```

Out[57]:
```
{'name': ['Bern', 'Zurich', 'Lugano', 'Geneva', 'Lucerne'],
 'population': [130, 391, 63, 194, 81],
 'area': [51.6, 87.88, 75.98, 15.93, 29.04],
 'language': ['German', 'German', 'Italian', 'French', 'German']}
```

In [58]:
```
# Create a DataFrame from a dictionary
df = pd.DataFrame(cities)
df
```

Out[58]:

	name	population	area	language
0	Bern	130	51.60	German
1	Zurich	391	87.88	German
2	Lugano	63	75.98	Italian
3	Geneva	194	15.93	French
4	Lucerne	81	29.04	German

In [59]:
```
# Create DataFrame from dictionary with a prespecified index
df = pd.DataFrame(cities, index=cities['name'])
df
```

Out[59]:

	name	population	area	language
Bern	Bern	130	51.60	German
Zurich	Zurich	391	87.88	German
Lugano	Lugano	63	75.98	Italian
Geneva	Geneva	194	15.93	French
Lucerne	Lucerne	81	29.04	German

A DataFrame can be imported from a file

In [60]:
```
# Create a DataFrame from a CSV-file
sales = pd.read_csv('sales.csv', index_col='customer_id')
sales.head()
```

Out[60]:

customer_id	total_sales	num_orders	gender	spender_type
100001	800.64	3	F	big
100002	217.53	3	F	medium
100003	74.58	2	M	small
100004	498.60	3	M	medium
100005	723.11	4	F	big

In [61]:
```
# Create a DataFrame from an EXCEL-file
sales = pd.read_excel('sales.xlsx', sheet='customer_data', index_col='customer_id')
sales.head()
```

Out[61]:

customer_id	total_sales	num_orders	gender	spender_type
100001	800.64	3	F	big
100002	217.53	3	F	medium
100003	74.58	2	M	small
100004	498.60	3	M	medium
100005	723.11	4	F	big

Get attributes of a DataFrame

In [62]:
```
# Get the number of rows and columns
sales.shape
```

Out[62]:
```
(10000, 4)
```

In [63]:
```
# Get the column labels
sales.columns
```

Out[63]:
```
Index(['total_sales', 'num_orders', 'gender', 'spender_type'], dtype='object')
```

In [64]:
```
# Get the index (row labels)
sales.index
```

Out[64]:
```
Int64Index([100001, 100002, 100003, 100004, 100005, 100006, 100007, 100008,
            100009, 100010,
            ...
            109991, 109992, 109993, 109994, 109995, 109996, 109997, 109998,
            109999, 110000],
           dtype='int64', name='customer_id', length=10000)
```

Accessing data from a DataFrame

In [65]:
```
# Get the row with index 'Bern'
df.loc['Bern']
```

Out[65]:
```
name            Bern
population       130
area            51.6
language      German
Name: Bern, dtype: object
```

In [66]:
```
# Get the rows with indices 'Bern' to 'Lugano' (including Lugano)
df.loc['Bern':'Lugano']
```

Out[66]:

	name	population	area	language
Bern	Bern	130	51.60	German
Zurich	Zurich	391	87.88	German
Lugano	Lugano	63	75.98	Italian

In [67]:
```
# Get the column 'population' for all rows
df.loc[:, 'population']
```

Out[67]:
```
Bern       130
Zurich     391
Lugano      63
Geneva     194
Lucerne     81
Name: population, dtype: int64
```

In [68]:
```
# Get the first two rows of the first column
df.iloc[0:2, 0]
```

Out[68]:
```
Bern        Bern
Zurich    Zurich
Name: name, dtype: object
```

In [69]:
```
# Get the second to last row of the last column
df.iloc[-2:, -1]
```

Out[69]:
```
Geneva     French
Lucerne    German
Name: language, dtype: object
```

In [70]:
```
# Get the column 'population'
df['population']
```

Out[70]:
```
Bern       130
Zurich     391
Lugano      63
Geneva     194
Lucerne     81
Name: population, dtype: int64
```

In [71]:
```
# Get the column 'population'
df.population
```

Out[71]:
```
Bern       130
Zurich     391
Lugano      63
Geneva     194
Lucerne     81
Name: population, dtype: int64
```

Apply methods on a DataFrame

In [72]:
```
# Get the first five rows of a DataFrame
df.head()
```

Out[72]:

	name	population	area	language
Bern	Bern	130	51.60	German
Zurich	Zurich	391	87.88	German
Lugano	Lugano	63	75.98	Italian
Geneva	Geneva	194	15.93	French
Lucerne	Lucerne	81	29.04	German

In [73]:
```
# Generate descriptiv statistics of numeric columns
df.describe()
```

Out[73]:

	population	area
count	5.000000	5.000000
mean	171.800000	52.086000
std	132.637476	30.375748
min	63.000000	15.930000
25%	81.000000	29.040000
50%	130.000000	51.600000
75%	194.000000	75.980000
max	391.000000	87.880000

In [74]:
```
# Get counts of unique values in the column language
df.language.value_counts()
```

Out[74]:
```
German     3
French     1
Italian    1
Name: language, dtype: int64
```

In [75]:
```
# Compute pairwise covariance of numeric columns
df.cov()
```

Out[75]:

	population	area
population	17592.7000	1639.15150
area	1639.1515	922.68608

In [76]:
```
# Compute pairwise correlation of columns
df.corr()
```

Out[76]:

	population	area
population	1.000000	0.406842
area	0.406842	1.000000

In [77]:
```
# Group Series of DataFrame using the column language and
# compute the sum of the remaining numeric columns for each group
df.groupby('language').sum()
```

Out[77]:

language	population	area
French	194	15.93
German	602	168.52
Italian	63	75.98

In [78]:
```
# Group Series of DataFrame using the column language and
# compute the mean of the remaining numeric columns for each group
df.groupby('language').mean()
```

Out[78]:

language	population	area
French	194.000000	15.930000
German	200.666667	56.173333
Italian	63.000000	75.980000

In [79]:
```
# Create a spreadsheet-style pivot table as a DataFrame
# Here: get the maximum population for groups of areas and languages
table = df.pivot_table(values=['population', 'area'], index=['language'], aggfunc=np.max
table
```

Out[79]:

language	area	population
French	15.93	194
German	87.88	391
Italian	75.98	63

Exporting a DataFrame to a file

In [80]:
```
# Export the DataFrame to a CSV-File
sales.to_csv('sales_out.csv')
```

In [81]:
```
# Export the DataFrame to an EXCEL-File
sales.to_excel('sales_out.xlsx')
```

Visualizations with pandas

In [82]:
```
# Display the DataFrame
sales.head()
```

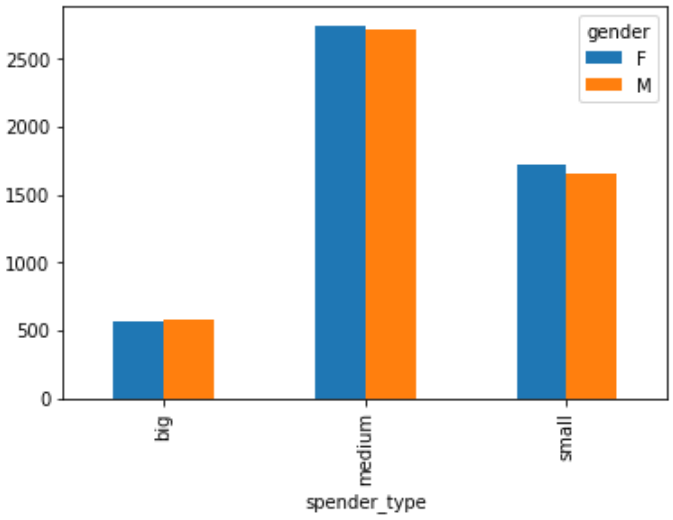
Out[82]:

customer_id	total_sales	num_orders	gender	spender_type
100001	800.64	3	F	big
100002	217.53	3	F	medium
100003	74.58	2	M	small
100004	498.60	3	M	medium
100005	723.11	4	F	big

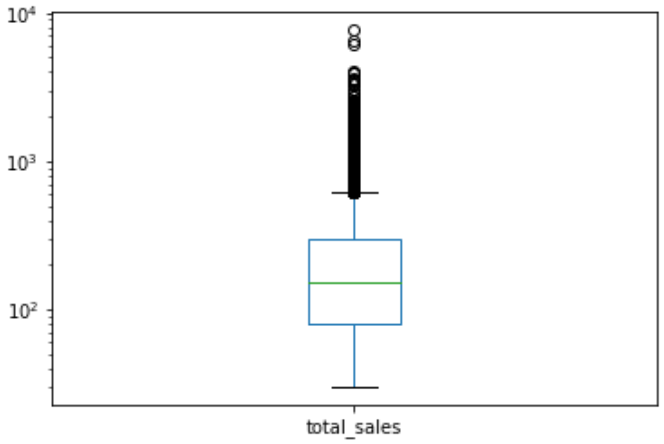
Generate simple plots

In [83]:
```
# Barplot
ax = sales.groupby('gender').size().plot.bar()
```

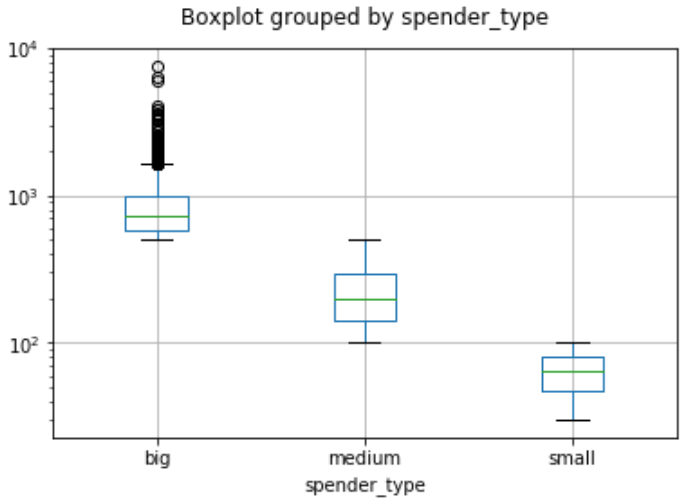
In [84]:
```
# Multiple barplots
sales.groupby(['spender_type', 'gender']).size().unstack().plot.bar();
```



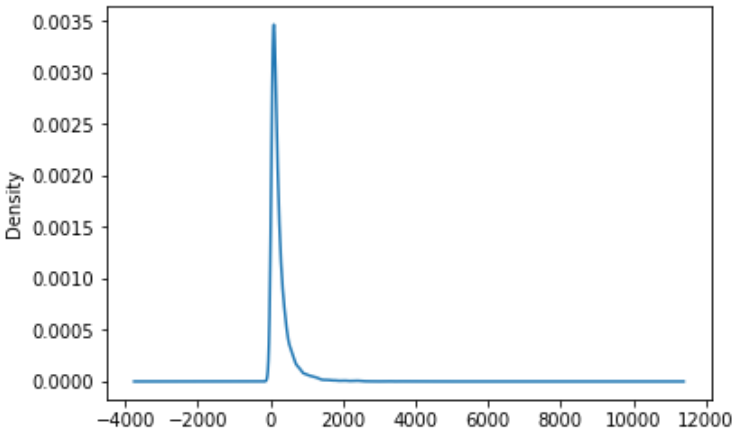
In [85]:
```
# Boxplot
sales.total_sales.plot.box().set_yscale('log')
```



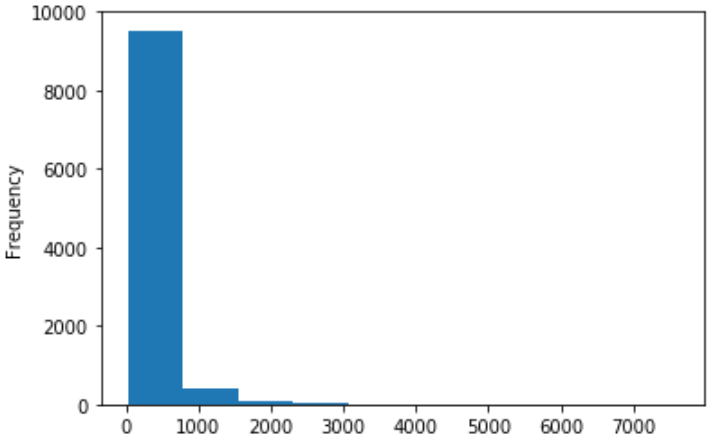
In [86]:
```
# Multiple boxplots
sales.boxplot(column='total_sales', by='spender_type').set(title='', yscale='log');
```



In [87]:
```
# Density plot
sales.total_sales.plot.density();
```

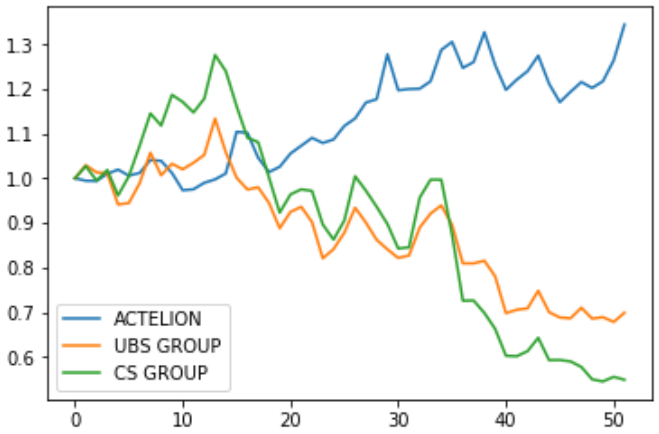


In [88]:
```
# Histogram
sales.total_sales.plot.hist();
```



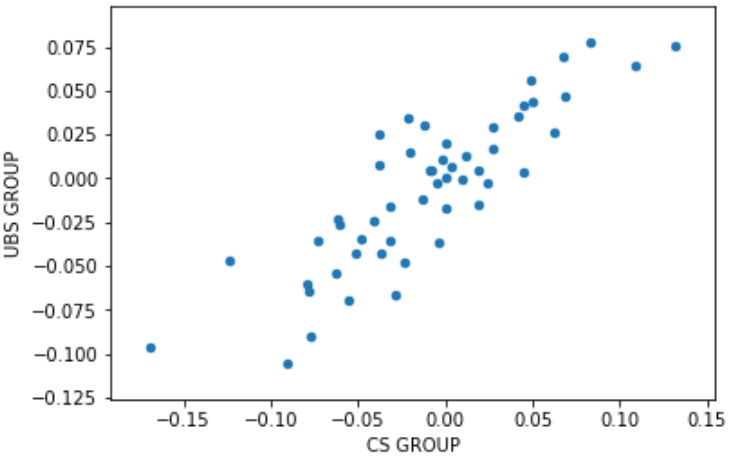
In [89]:
```
# Load data
df = pd.read_csv('smi.csv')
df = df + 1
df.loc[0] = 1
df = df.cumprod()
df = df[['ACTELION', 'UBS GROUP', 'CS GROUP']]
```

In [90]:
```
# Line plot
df.plot.line();
```

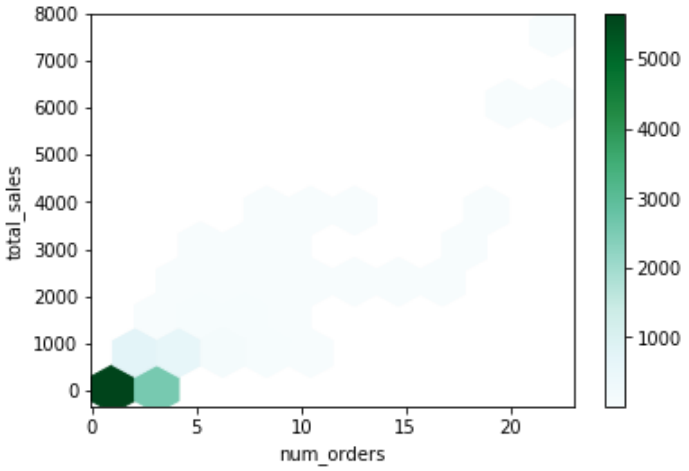


In [91]:
```
# Load data
df = pd.read_csv('smi.csv')
```

In [92]:
```
# Scatter plot
df.plot.scatter(x='CS GROUP', y='UBS GROUP');
```



In [93]:
```
# Hexbinplot
sales.plot.hexbin(x='num_orders', y='total_sales', gridsize=10, mincnt=1, sharex=False);
```



Generate customized plots

In [94]:
```
# Import the matplotlib package
import matplotlib.pyplot as plt
```

In [95]:
```
# Generate customized scatter plot
ax = df.plot.scatter(x='CS GROUP',
                     y='UBS GROUP',
                     s=30,
                     c='blue',
                     fontsize=12,
                     figsize=(8, 8))

# Change formatting of tick-labels
ax.set_yticklabels(['{:.2%}'.format(x) for x in ax.get_yticks()])
ax.set_xticklabels(['{:.2%}'.format(x) for x in ax.get_yticks()])

# Use equal scaling for the X- and Y-axis
ax.set_aspect('equal')

# Set label of X-axis
plt.xlabel('CSG', fontsize=15, labelpad=20)

# Set label of Y-axis
plt.ylabel('UBSN', fontsize=15, labelpad=20)

# Set title
plt.title('Scatterplot', fontsize=20, pad=10)

# Save figure using 100 dots per inch
plt.savefig("scatterplot.pdf", dpi=100)
```