NYC DATA SCIENCE ACADEMY

Data Analysis in Python Pandas Part 2

NYC Data Science Academy

Review: Pandas

- We are at the halfway point in our discussion of pandas. So far we have covered:
  - Creating Series/DataFrame: with dictionary or list (array).
  - Data manipulation:
    - Combining DataFrames: concat and merge.
    - Analysis: arithmetic, drop, apply and describe.
    - Subsetting: selection and filtering.
- We will continue our discussion on Pandas. We start with missing values.

Review: Pandas

- Again we import the packages we need:

```
import numpy as np
import pandas as pd
```

- We shared a folder of lecture code and the data files. All the material we need are included.

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - groupby
  - agg
- Time series
- Interacting with Database

Handling missing data

- Missing - or, what amounts to the same thing, corrupt - data is an unavoidable fact of life in dealing with large quantities of data. There are many ways of dealing with it, depending upon the circumstances:
  - Discard it, and all related data.
  - Interpolate values from surrounding data
  - Isolate it and analyze it separately
- Whatever approach is chosen - and this is a scientific, not a computational, question - pandas has methods to make it simpler to carry out.

Handling missing data

- First, let's read a csv file that contains NaNs. Note here we set *index_col* to 0 which means we are using the first column as the index.

```
df = pd.read_csv('missing.csv', index_col = 0)
df
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	NaN	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
d	NaN	NaN	NaN	NaN
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	NaN	NaN	-0.160499

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - groupby
  - agg
- Time series
- Interacting with Database

Handling missing data: isnull

- If we have no idea about what the dataset looks like, the first thing we want to do is to figure out where are the missing data. We can use the *isnull* method.

```
df.isnull()
```

	one	two	three	four
a	False	False	False	False
b	True	False	False	False
c	False	False	False	False
d	True	True	True	True
e	False	False	False	False
f	False	True	True	False

Handling missing data: isnull

- Also we can sum up the boolean array to see how many missing values each column has:

```
np.sum(df.isnull())
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	NaN	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
d	NaN	NaN	NaN	NaN
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	NaN	NaN	-0.160499

```
one      2
two      2
three    2
four     1
dtype: int64
```

Handling missing data: isnull

- We may do the same computation for rows

```
np.sum(df.isnull(), axis=1)
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	NaN	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
d	NaN	NaN	NaN	NaN
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	NaN	NaN	-0.160499

```
a    0
b    1
c    0
d    4
e    0
f    2
dtype: int64
```

Handling missing data: isnull

- Sometimes we need a close look at those NaNs, so we want to see only the rows that contain NaNs. To do that, we aggregate the data frame with boolean values, `df.isnull()`, by the function any. `axis=1` indicates rows.

```
df.isnull().any(axis=1)
```

```
a    False
b     True
c    False
d     True
e    False
f     True
dtype: bool
```

Handling missing data: isnull

- Fancy indexing:

```
df.loc[df.isnull().any(axis=1),:]
```

	one	two	three	four
b	NaN	-0.217766	0.655179	1.379276
d	NaN	NaN	NaN	NaN
f	-1.14906	NaN	NaN	-0.160499

Exercise 1

We showed how NaNs can be handled. However, not all missing values are NaNs. Consider the example below:

- Download the `Employee` data frame from `Employee.csv`.

```
Employee = pd.read_csv('Employee_continue.csv')
```

  - Print the data frame, looking for missing values by inspection. How many missing values do we have? Some of the missing values might not be NaNs.

Exercise 1

- We saw '?' in the data frame. For a small data frame like this we may replace '?' by np.nan manually. However, if we deal with a large data frame, we might need the function `replace`. Use `replace` to replace '?' by np.nan.

Remark

  - This problem is actually very common!
  - If you don't know the replace function, search for 'pandas replace'!
  - Make sure you update the data frame.

Exercise 1

- Write code to find how many missing values we have in each row of `Employee`. Then in each column.
- Print the rows with missing values.
- Print the columns with missing values.

OVERVIEW

- Handling Missing Data
  - isnull
  - **dropna, fillna and interpolate**
- Grouping and aggregation
  - groupby
  - agg
- Time series
- Interacting with Database

Handling missing data: dropna

- We may simply discard the rows with missing values. Below the arguments axis=0 and how='any' indicates dropping *rows* with a NaN in *any* position.

```
df.dropna(axis=0, how='any')
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
c	-0.860359	-1.313747	0.676174	1.034417
e	0.079169	0.029138	0.239183	-0.492039

Handling missing data: dropna

- We could also drop rows full of NaNs. This can be done with how='all'.

```
df.dropna(axis=0, how='all')
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	NaN	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	NaN	NaN	-0.160499

Handling missing data: dropna

- From the data frame above we can apply dropna once more. This time we drop a column with the argument axis=1.

```
df.dropna(axis=0, how='all').dropna(axis=1, how='any')
```

	four
a	-1.015682
b	1.379276
c	1.034417
e	-0.492039
f	-0.160499

Handling missing data: fillna

- Instead of discarding information we may also **impute** the data. This can be done with fillna function with value to be imputed as the argument.

```
df.fillna(0)
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	0.000000	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
d	0.000000	0.000000	0.000000	0.000000
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	0.000000	0.000000	-0.160499

Handling missing data: fillna

- Another common way to impute is by the mean of the column.

```
df['one'].fillna(df['one'].mean())
```

```
a   -1.250699
b   -0.795237
c   -0.860359
d   -0.795237
e    0.079169
f   -1.149060
Name: one, dtype: float64
```

Handling missing data: interpolate

- Interpolation means inserting between fixed points. `method='linear'` indicates equally spacing the difference between the values bordering the missing values.

```
df.interpolate(method='linear')
```

	one	two	three	four
a	-1.250699	-0.573801	0.705961	-1.015682
b	-1.055529	-0.217766	0.655179	1.379276
c	-0.860359	-1.313747	0.676174	1.034417
d	-0.390595	-0.642305	0.457679	0.271189
e	0.079169	0.029138	0.239183	-0.492039
f	-1.149060	0.029138	0.239183	-0.160499

Handling missing data: interpolate

- Since `df.loc['b','one']` is a NaN between `df.loc['a','one']` and `df.loc['c','one']`, the value inserted is the mean of them.

```
(df.loc['a', 'one'] + df.loc['c', 'one'])/2
-1.0555289364489999
```

Exercise 2

- We now create a pandas Series with NaNs and then impute with linear interpolation. What do you expect the values imputed would be? Confirm your guess by actually imputing with the `interpolate` function.

```
series = pd.Series([1, np.nan, np.nan, 4])
series
```

```
0    1
1  NaN
2  NaN
3    4
dtype: float64
```

- Linearly interpolate the data frame, `Employee`. Which columns did we impute?

Exercise 2

- Drop the column with a missing value.

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- **Grouping and aggregation**
  - groupby
  - agg
- Time series
- Interacting with Database

Grouping and aggregation

- Aggregation is often a critical component of a data analysis workflow. It involves one or more of the following steps:
  - **Splitting** the data into groups based on some features.
  - **Applying** a function to each group independently.
  - **Combining** the result into a data structure.

Grouping and aggregation

- Let's create a data frame.

```
np.random.seed(10)
df = pd.DataFrame({'key1':['a','a','b','b','a'],
                   'key2':['one','two','one','two','one'],
                   'data1': np.random.randn(5),
                   'data2': np.random.randn(5)})
df
```

	data1	data2	key1	key2
0	1.331587	-0.720086	a	one
1	0.715279	0.265512	a	two
2	-1.545400	0.108549	b	one
3	-0.008384	0.004291	b	two
4	0.621336	-0.174600	a	one

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - **groupby**
  - agg
- Time series
- Interacting with Database

Grouping and aggregation: groupby

- A natural question is: How many 'a's do we have in key1? One way to answer this is to group the data frame by the value in key1. That is:

```
group = df.groupby('key1')
```

- group is assigned the value returned by the groupby function, whose type is:

```
print type(group)
<class 'pandas.core.groupby.DataFrameGroupBy'>
```

Grouping and aggregation: groupby

- Here we introduce an important feature of the object. A *DataFrameGroupBy* object is an iterable. That says we can iterate over the object:

```
for item in group:
    print item
```

```
('a',       data1     data2 key1 key2
0  1.331587 -0.720086     a  one
1  0.715279  0.265512     a  two
4  0.621336 -0.174600     a  one)
('b',       data1     data2 key1 key2
2 -1.545400  0.108549     b  one
3 -0.008384  0.004291     b  two)
```

Grouping and aggregation: groupby

- With a careful inspection we see that each item we printed is a tuple with two components. In Python, there is an alternative way of iteration:

```
for key, values in group:
        print key
        print '-'*55
        print values
        print '\n'
```

Grouping and aggregation: groupby

- The result:

```
a
---------------------------------------------
      data1     data2 key1 key2
0  1.331587 -0.720086    a  one
1  0.715279  0.265512    a  two
4  0.621336 -0.174600    a  one

b
---------------------------------------------
      data1     data2 key1 key2
2 -1.545400  0.108549    b  one
3 -0.008384  0.004291    b  two
```

- In this way we can print and inspect a DataFrameGroupBy object. We also see how **splitting** is done.

Grouping and aggregation: groupby

- **Applying** and **combining** are often done together with a single function. For example:

```
group.size()
key1
a    3
b    2
dtype: int64
```

- The function `size` counts the number of observations in each group and then combines the result into a pandas series. This answers our question in the beginning of this section.

Grouping and aggregation

- A careful audience might notice that boolean operators and `sum` provide an easier way:

```
np.sum(df.key1 == 'a')
3
```

- However, what if we want to know the number of *b* as well? What if we have even more classes in the column, and we want to know the number of each of them?
- Grouping and aggregation applies a function (counting in this particular case) to **ALL** the groups, so users don't need to iterate manually.

Grouping and aggregation

- Inspect the code below. Consider:
  - where does **splitting** happens?
  - which function was **applied** to each group?
  - what type of object does the code **combine** the result into?

```
group_ = df.groupby('key2')
group_.mean()
```

	data1	data2
key2		
one	0.135841	-0.262046
two	0.353448	0.134902

Grouping and aggregation

- Instead of using the method `.mean()`, we may use agg.

```
group_.agg('mean')
```

	data1	data2
key2		
one	0.135841	-0.262046
two	0.353448	0.134902

- agg is more useful when applying multiple aggregation functions, as we will see in the next session.

Exercise 3

- Recall that we had a `Salary` data frame last time:

	Title	Salary
0	VP	250
1	associate	120
2	analyst	90

  - Recall also that we merged this `Salary` data frame with our `Employee` data frame and then gave the VPs 5% raise -- this basically resulted in the `Employee` data frame we have right now. Recover the Salary data frame from our `Employee` data frame, but with the updated salary.

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - groupby
  - agg
- Time series
- Interacting with Database

Grouping and aggregation: agg

- We may group the data by multiple keys:

```
group2 = df.groupby(['key1', 'key2'])
group2.mean()
```

		data1	data2
key1	key2		
a	one	0.976461	-0.447343
	two	0.715279	0.265512
b	one	-1.545400	0.108549
	two	-0.008384	0.004291

Grouping and aggregation: agg

- We may apply multiple functions to each group with the method agg:

```
group2.agg(['count', 'sum', 'min', 'max', 'mean', 'std'])
```

- Part of the result look like:

		data1					
		count	sum	min	max	mean	std
key1	key2						
a	one	2	1.952922	0.621336	1.331587	0.976461	0.502223
	two	1	0.715279	0.715279	0.715279	0.715279	NaN
b	one	1	-1.545400	-1.545400	-1.545400	-1.545400	NaN
	two	1	-0.008384	-0.008384	-0.008384	-0.008384	NaN

Grouping and aggregation: agg

- The other part of the result looks like:

		data2					
		count	sum	min	max	mean	std
key1	key2						
a	one	2	-0.894686	-0.720086	-0.174600	-0.447343	0.385716
	two	1	0.265512	0.265512	0.265512	0.265512	NaN
b	one	1	0.108549	0.108549	0.108549	0.108549	NaN
	two	1	0.004291	0.004291	0.004291	0.004291	NaN

- The column std misses three values because each of `['a', 'one']`, `['b', 'one']` and `['b', 'two']` includes only one row in `df`.

Grouping and aggregation: agg

- We may apply different aggregating functions to different columns. This can be done with a dictionary.

```
colFun = {'data1': ['min', 'max'],
          'data2': ['mean','std']}
group.agg(colFun)
```

	data1		data2	
	min	max	mean	std
key1				
a	0.621336	1.331587	-0.209725	0.493737
b	-1.545400	-0.008384	0.056420	0.073721

Exercise 4

- Find the minimum, mean, and maximum of the 'Year' for each Department.
- Find the minimum, mean, and maximum of all the numeric features for each Department.
- How many female or male are in each department? For each sex in each department, what are the minimum, mean, and maximum of their salaries?
- Does the company pay males and females differently? (We don't have "correct" answer for this question, you may design any method of analysis to approach this question.)

Grouping and aggregation

- We may apply custom aggregation functions. We observe in the previous examples that aggregation functions were applied to each **column** in a data frame. We need to keep this in mind when defining a custom function. For example, we might want to compute mean after removing maxima (truncated mean).

```
def trunc_mean(x):
    return np.mean(x[x!=x.max()])
df.groupby('key1').agg(trunc_mean)
```

	data1	data2
key1		
a	0.668307	-0.447343
b	-1.545400	0.004291

Grouping and aggregation

- We may create a dictionary to indicate how indices should be grouped.

```
dictionary = {0:'even', 1:'odd', 2:'even', 3:'odd', 4:'even'}
```

- Grouping the data according to the dictionary and aggregating in the usual way, we obtain:

```
dic_group = df.groupby(dictionary)
dic_group.size()
even    3
odd     2
dtype: int64
```

Exercise 5

- For each department, compute the difference between the maximum salary and the minimum salary.
- You suspect the salary is related to the first letter of one's name. To check that, let's group the Employee data frame by the first letter of its index and then find the average salary among each group.
- So far, we have been aggregating only one column in a data frame. If we need multiple columns involved, aggregating functions might not be enough. For example, find out the departments in which a female has the highest salary.

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - groupby
  - agg
- **Time series**
- Interacting with Database

Time series

- A common kind of data series - especially in financial analysis - is datetimes. For example, transaction or pricing data is almost always time-stamped.
- Dates are problematic because not only are they sometimes missing (as with all kinds of data), but they are inherently complex: dates are not sequential in that the next business day is different from the next day; markets may close unexpectedly; different countries have market vacations on different days. This makes it difficult to reconcile even non-corrupt data sources that contain dates.
- Again, pandas does not solve the problem in any fundamental way, but it makes it much easier to carry out whatever solution you choose.

Time series

- Creating a datetime series is simple with the date_range function.

```
# 72 hours starting with midnight Jan 1st, 2011
rng = pd.date_range('1/1/2011', periods=72, freq='H')
rng
DatetimeIndex(['2011-01-01 00:00:00', '2011-01-01 01:00:00',
               '2011-01-01 02:00:00', '2011-01-01 03:00:00',
               '2011-01-01 04:00:00', '2011-01-01 05:00:00',
               '2011-01-01 06:00:00', '2011-01-01 07:00:00',
               '2011-01-01 08:00:00', '2011-01-01 09:00:00',
                ...
               '2011-01-03 16:00:00', '2011-01-03 17:00:00',
               '2011-01-03 18:00:00', '2011-01-03 19:00:00',
               '2011-01-03 20:00:00', '2011-01-03 21:00:00',
               '2011-01-03 22:00:00', '2011-01-03 23:00:00'],
               dtype='datetime64[ns]', freq='H')
```

Time series

- The datatype of rng is

```
print type(rng)
<class 'pandas.tseries.index.DatetimeIndex'>
```

- And the datatype of each element in rng is

```
print type(rng[0])
<class 'pandas.tslib.Timestamp'>
```

Time series

- There are a couple of parameters to tweak. For example, you may change freq to 'M' which will give you the next 72 months instead of 72 hours. To see what you can do further, check the documentation here: http://pandas.pydata.org/pandas-docs/stable/timeseries.html

Time series

- Datetime data can be stored in different formats. Pandas provides a simple function to convert them. For example:

```
print pd.to_datetime('Jul 31, 2009')
print pd.to_datetime('2010-01-10')
2009-07-31 00:00:00
2010-01-10 00:00:00
```

- A missing datetime is recorded as a special value NaT (just like NaN in Numpy).

```
print pd.to_datetime(np.nan)
print type(pd.to_datetime(np.nan))
NaT
<class 'pandas.tslib.NaTType'>
```

Time series

- `to_datetime` can convert lists of datetimes:

```
pd.to_datetime(['2005/11/23', '2010.12.31'])
DatetimeIndex(['2005-11-23', '2010-12-31'],
              dtype='datetime64[ns]', freq=None)
```

- Another example:

```
print pd.to_datetime(pd.Series(['2010-10-10',
                                'Jan 1, 2008']))
```

```
0   2010-10-10
1   2008-01-01
dtype: datetime64[ns]
```

Time series

- Sometimes the dataset we have uses Unix time (also known as POSIX time and Epoch time). It is widely used in Unix-like and many other operating systems and file formats.
  - The Unix epoch is the time 00:00:00 UTC on 1 January 1970. The Unix time number is zero at the Unix epoch, and increases by exactly 86400 per day since the epoch. Thus 2004-09-16 00:00:00, 12677 days after the epoch, is represented by the Unix time number 12677 × 86400 = 1095292800.
  - You can find the detailed definition here: https://en.wikipedia.org/wiki/Unix_time

Time series

- Again, to_datetime can convert Unix time as well. We need to specify the unit is **seconds**:

```
pd.to_datetime(1095303803, unit='s')
Timestamp('2004-09-16 03:03:23')
```

- List of unix time can be converted as well:

```
pd.to_datetime([1349720105, 1349806505, 1349892905,
                1349979305, 1350065705], unit='s')
```

```
DatetimeIndex(['2012-10-08 18:15:05', '2012-10-09 18:15:05',
               '2012-10-10 18:15:05', '2012-10-11 18:15:05',
               '2012-10-12 18:15:05'],
              dtype='datetime64[ns]', freq=None)
```

Time series

- Once a datetime object is created, we can access the attributes.

```
my_datetime = pd.to_datetime(1095304828, unit='s')

print 'my_datetime is %s' % my_datetime
print '\n'
print 'The year of my_datetime %s' % my_datetime.year
print 'The month of my_datetime %s' %my_datetime.month
print 'The day of my_datetime %s' % my_datetime.day
print 'The hour of my_datetime %s' % my_datetime.hour
print 'The minute of my_datetime %s' % my_datetime.minute
print 'The second of my_datetime %s' % my_datetime.second
```

Time series

- The result:

```
my_datetime is 2004-09-16 03:20:28


The year of my_datetime 2004
The month of my_datetime 9
The day of my_datetime 16
The hour of my_datetime 3
The minute of my_datetime 20
The second of my_datetime 28
```

Exercise 6

- Load the OHLC time series from the Date_Price.csv file (OHLC records the open, high, low and close price of a commodity in a time interval). Use the date as the index for the data frame; make sure the datatype of the date is appropriate.
- Sort the data frame by the date. Make sure you update your data frame.
- Convert the data to monthly OHLC.

OVERVIEW

- Handling Missing Data
  - isnull
  - dropna, fillna and interpolate
- Grouping and aggregation
  - groupby
  - agg
- Time series
- **Interacting with Database**

Interacting with Databases

- In many applications the data comes from SQL-based relational databases (such as SQL Server, PostgreSQL, and MySQL).
- SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine.
- SQLite is the most widely deployed SQL database engine in the world.

Interacting with Databases

- Python has a built-in sqlite3 driver, which can be used to connect to the in-memory SQLite database:

```
import sqlite3
```

- ":memory:" below open a database connection to a database that resides in RAM instead of on disk.

```
con = sqlite3.connect(':memory:')
type(con)
sqlite3.Connection
```

Interacting with Databases

- We can also connect to an existing database by passing the name of the database to the function `connect`. If there is no database with the name passed, a new database will be created.

```
con = sqlite3.connect('./my_db')
```

- Pandas DataFrame can be easily saved into a SQL database using the function `to_sql()`. Let's create a data frame first.

```
data = {'state': ['Ohio', 'Ohio', 'Ohio',
                  'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
df = pd.DataFrame(data)
df
```

Interacting with Databases

- The data frame created:

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

Interacting with Databases: Save data frame to DataBase

- We then save the data frame to the database. The parameter index=None prevents saving the index `[0, 1, 2, 3, 4]`.

```
df.to_sql('my_df', con, if_exists='replace', index=None)
```

Interacting with Databases: query data from DataBase

- We also want to read a table into a pandas data frame. We often want to list the names of all the tables we have in a database. We may apply the `read_sql_query` function

```
pd.read_sql_query("SELECT name from sqlite_master WHERE
type='table';", con)
```

	name
0	my_df

Interacting with Databases: query data from DataBase

- Then we may apply the `read_sql_query` function to query a table.

```
df_from_db = pd.read_sql("SELECT pop FROM my_df;", con)
df_from_db
```

	pop
0	1.5
1	1.7
2	3.6
3	2.4
4	2.9

Interacting with Databases: query data from DataBase

- To select all the columns in a table, use *.

```
df_from_db = pd.read_sql("SELECT * FROM my_df;", con)
df_from_db
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002

Interacting with Databases: query data from DataBase

- We didn't explain the argument `if_exist='replace'` in the code:

```
df.to_sql('my_df', con, if_exists='replace', index=None)
```

- It simply says if there is already a table 'my_df' in the database connected by con, it will be replaced (overwritten) by `to_sql`. You may run the code below multiple times, every time the old table would be erased.

```
df.to_sql('my_df', con, if_exists='replace', index=None)
df_from_db = pd.read_sql("SELECT * FROM my_df;", con)
df_from_db
```

Interacting with Databases: query data from DataBase

- If we pass `if_exists='append'` to the function, the old table will no longer be erased, and the new data frame will be appended to the old one. Run the code below multiple times, you will see that the data frame is growing.

```
df.to_sql('my_df', con, if_exists='append', index=None)
df_from_db = pd.read_sql("SELECT * FROM my_df;", con)
df_from_db
```

Disconnect database

- Once we finish querying, we disconnect the database by:

```
con.close()
```